Many popular programming languages, including Ruby, JavaScript, and Python, feature dynamic type systems, in which types are not known until runtime. Dynamic typing provides the programmer with flexibility and allows for rapid program development. In contrast, static type systems, found in languages like C++ and Java, help catch errors early during development, enforce invariants as programs evolve, and provide useful documentation via type annotations. Many researchers have explored combining these contrasting paradigms, seeking to marry the flexibility of dynamic types with the correctness guarantees and documentation of static types.

However, many challenges remain in this pursuit: programmers using dynamic languages may wish to verify more expressive properties than basic type safety; operations for commonly used libraries, such as those for databases and heterogeneous data structures, are difficult to precisely type check; and type inference—the process of automatically deducing the types of methods and variables in a program—often produces type annotations that are complex and verbose, and thus less usable for the programmer. To address these issues, I present four pieces of work that aim to
increase the expressiveness and usability of static types for dynamic languages.

First, I present RTR, a system that adds refinement types to Ruby: basic types extended with expressive predicates. RTR uses assume-guarantee reasoning and a novel idea called just-in-time verification—in which verification is deferred until runtime—to handle dynamic program features such as mixins and metaprogramming. We found RTR was useful for verifying key methods in six Ruby programs.

Second, I present CompRDL, a Ruby type system that allows library method type signatures to include type-level computations (or comp types). Comp types can be used to precisely type check database queries, as well as operations over heterogeneous data structures like arrays and hashes. We used CompRDL to type check methods from six Ruby programs, enabling us to check more expressive properties, with fewer manually inserted type casts, than was possible without comp types.

Third, I present InferDL, a Ruby type inference system that aims to produce usable type annotations. Because the types inferred by standard, constraint-based inference are often complex and less useful to the programmer, InferDL complements constraints with configurable heuristics that aim to produce more usable types. We applied InferDL to four Ruby programs with existing type annotations and found that InferDL inferred 22% more types that matched the prior annotations compared to standard inference.

Finally, I present SimTyper, a system that builds on InferDL by using a novel machine learning-based technique called type equality prediction. When standard and heuristic inference produce a non-usable type for a position (argument/return/variable), we use a deep similarity network to compare that position to other posi-
tions with usable types. If the network predicts that two positions have the same type, we guess the usable type in place of the non-usable one, and check the guess against constraints to ensure soundness. We evaluated SimTyper on eight Ruby programs with prior annotations and found that, compared to standard inference, SimTyper finds 69% more types that match programmer-written annotations.

In sum, I claim that RTR, CompRDL, InferDL, and SimTyper represent promising steps towards more expressive and usable types for dynamic languages.
TOWARDS MORE EXPRESSIVE AND USABLE TYPES FOR DYNAMIC LANGUAGES

by

Milod Kazerounian

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 2021

Advisory Committee:
Professor Jeffrey S. Foster, Chair/Advisor
Professor David Van Horn
Professor Marine Carpuat
Professor Michael Hicks
Professor Donald Yeung
Acknowledgments

First and foremost, I’d like to thank my advisor, Jeff Foster, for his guidance over the past six years. When I felt lost, Jeff inspired new directions to explore. When I was excited about new ideas, he provided the support and insight I needed to pursue them. Most notably, Jeff was always patient, respectful, and kind. He has served as a model for the type of mentor I hope to be going forward.

I want to thank the members of the PLUM lab at UMD and the TuPL lab at Tufts, most especially Sankha Narayan Guria, Ian Sweet, Kris Micinski, and Niki Vazou. Over the years, they provided thoughtful conversation, camaraderie, and community, without which this would not have been possible.

Thanks to my friends—Puya, Becky, Sara, Matt, the Willowbrook crew, Charlie, Max, Navid, and all others I’ve inadvertently left out. I’m blessed that I could always escape into these friendships and the endless laughs these kids provide.

To my family—Jila, Kazem, Sohrob, Nico, Avi, Salmun, Darab, Azad, Efat, Haleh, Arman, and Yasmin (Iranian families get pretty big)—thank you for your boundless love and generosity, and your good humor. You’ve taught me how to be resilient in the face of adversity, and how to do it all with a smile on your face.

Finally, to Aurora. I like the work I did over these last few years, but never did I feel luckier than when I came home and got to be with you. Thank you for your love, perspective, patience, and support; I owe it all to you.
This work was supported in part by the NSF Graduate Research Fellowship Program under Grant No. DGE 1840340.
# Table of Contents

Acknowledgements ii

Table of Contents iv

List of Tables vii

List of Figures ix

Chapter 1: Introduction 1

1.1 RDL: A Ruby Type Checker 3
1.2 Novel Type Systems 5
  1.2.1 RTR: Refinement Types for Ruby 5
  1.2.2 CompRDL: Type-Level Computations for Ruby Libraries 6
  1.2.3 InferDL: Sound, Heuristic Type Annotation Inference for Ruby 7
  1.2.4 SimTyper: Sound Type Inference for Ruby using Type Equality Prediction 9

Chapter 2: Refinement Type for Ruby 11

2.1 Introduction 11
2.2 Overview 13
  2.2.1 Refinement Type Specifications 14
  2.2.2 Verification Using Rosette 14
  2.2.3 Encoding and Reasoning about Objects 15
  2.2.4 Method Calls 17
2.3 Just-In-Time Verification 19
  2.3.1 Mixins 19
  2.3.2 Metaprogramming 20
2.4 From Ruby to Rosette 22
  2.4.1 Core Ruby $\lambda^{RB}$ and Intermediate Representation $\lambda^I$ 22
  2.4.2 From $\lambda^{RB}$ to $\lambda^I$ 25
  2.4.3 From $\lambda^I$ to Rosette 30
  2.4.4 Primitive Types 31
  2.4.5 Verification of $\lambda^{RB}$ 32
2.5 Evaluation 34
2.6 Related Work 41
2.7 Conclusion 43
Chapter 3: Type-Level Computations for Ruby Libraries

3.1 Introduction

3.2 Overview

3.2.1 Typing Ruby Database Queries

3.2.2 Avoiding Casts using Comp Types

3.2.3 SQL Type Checking

3.2.4 Discussion

3.3 Soundness of Comp Types

3.3.1 Syntax and Semantics

3.3.2 Type Checking and Rewriting

3.3.3 Properties of $\lambda^C$

3.4 Implementation

3.5 Experiments

3.5.1 Library Types

3.5.2 Benchmarks

3.5.3 Results

3.6 Related Work

3.7 Conclusion

Chapter 4: Sound, Heuristic Type Annotation Inference for Ruby

4.1 Introduction

4.2 Overview

4.2.1 Standard Type Inference

4.2.2 Type Inference with Heuristics

4.3 Constraints, Solutions, and Heuristics

4.3.1 Solution Extraction

4.4 Implementation

4.5 Evaluation

4.5.1 Results

4.5.2 Case Studies

4.5.3 Testing Implementation Choices

4.6 Related Work

4.7 Conclusion

Chapter 5: SimTyper: Sound Type Inference for Ruby using Type Equality Prediction

5.1 Introduction

5.2 Overview

5.2.1 Standard Type Inference

5.2.2 Heuristic Type Inference

5.2.3 Predicting Type Equalities

5.3 Type Inference Algorithm

5.3.1 Standard and Heuristic Inference

5.3.2 Type Equality Prediction Algorithm

5.4 Implementing the DeepSim Network
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.1</td>
<td>Training the DeepSim Network</td>
<td>151</td>
</tr>
<tr>
<td>5.5</td>
<td>Evaluation</td>
<td>154</td>
</tr>
<tr>
<td>5.5.1</td>
<td>SimTyper Results</td>
<td>159</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Performance</td>
<td>166</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Comparing DeepSim and Heuristics</td>
<td>167</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Predicting Rare Types</td>
<td>168</td>
</tr>
<tr>
<td>5.5.5</td>
<td>SimTyper Design Choices</td>
<td>170</td>
</tr>
<tr>
<td>5.6</td>
<td>Related Work</td>
<td>171</td>
</tr>
<tr>
<td>5.7</td>
<td>Conclusion</td>
<td>173</td>
</tr>
</tbody>
</table>

Chapter 6: Conclusion  
6.1 Future Work 174

Appendix A: Soundness of CompRDL  
A.1 Soundness 181
A.2 Program Type Checking and Class Table Construction 203

Appendix B: SimTyper Evaluation Data 204

Bibliography 208
List of Tables

2.1 Method gives the class and name of the method verified. Ru-LoC and Ro-LoC give number of LoC for a Ruby method and the translated Rosette program. Spec is the number of method and variable type annotations we had to write. Verification Time is the median and semi-interquartile range of the time in seconds over 11 runs. App Total rows list the totals for an app, without double counting the same specs. ............................................ 37

3.1 Library methods with comp type definitions. .................... 74
3.2 CompRDL type checking results. ................................. 78

4.1 Type inference: number of targets and time performance. .... 111
4.2 Type inference: assessing inferred types. ...................... 112
4.3 Case Study Inference Results. Program name “AM” is short for Active Merchant, and “MM” is short for MiniMagick ............. 116
4.4 Testing Implementation Choices. ................................ 123

5.1 Benchmark statistics. .............................................. 156
5.2 Precision and recall of SimTyper. ............................... 164
5.3 Running time of SimTyper over nine runs. .................... 166
5.4 DeepSim’s ability to predict types also guessed by heuristics. H Matches is the number of matching (including up to parameter) types inferred by the heuristic in CH, and DS Matches counts the subset of those types also inferred by DeepSim in CD. Measurements with the DeepSim network were taken with the top-3 threshold. ........ 168
5.5 Numbers of Library, Training, and Program types guessed by DeepSim across all benchmarks, under CD with top-3 cutoff. .... 169
B.1 Data on the types that SimTyper inferred, but we did not have gold standards to compare against. For each program, we list the number of method types SimTyper inferred for which there was no gold standard, the lines of code those methods comprised, the number of argument types those methods include, and the number of instance, class, and global variables SimTyper inferred a type for. Note that we did not include a column for the number of return types in the methods, since this is equivalent to the number of methods. Then, of the total argument, return, and variable types SimTyper inferred, we list the number of these that were usable, overly general, and the number for which SimTyper failed to infer any type.

B.2 SimTyper evaluation results corresponding to the plots in Figure 5.8. For each benchmark, we list the number of matching, match up to parameter, and different inferred types measured under all configurations (C/CH/CD/CHD). For the different category, in parentheses we show the number of those types that were structural. Additionally, the No Type category indicates the algorithm could not find a more usable solution than giving a type variable for that position.

B.3 This table corresponds to the graphs in Figure 5.9. Measuring SimTyper’s performance for arguments, variables, and returns. For each category, we list the number of match, match up to parameter, and different inferred types measured under all configurations (C/CH/CD/CHD). For the different category, in parentheses we show the number of those types that were structural.

B.4 This table corresponds to the graphs in Figure 5.10a. Measuring SimTyper’s performance under top-1, -3, -5, and -7 configurations. Measurements were taken under the CD configuration. For the different category, in parentheses we show the number of those different types that were structural types.

B.5 This table corresponds to the graphs in Figure 5.10b. Measuring SimTyper’s performance under for two different methods of generating embeddings for arguments: Averaging vectors for all uses of an argument (All) or using just the vector for the argument in the method header (Head). Measurements were taken under the CD configuration. For the different category, in parentheses we show the number of those different types that were structural types.

B.6 This table corresponds to the graphs in Figure 5.10c. Measuring SimTyper’s performance under for two different methods of generating embeddings for returns: averaging method names and return sites (N+S), or using just method names (N). Measurements were taken under the CD configuration. For the different category, in parentheses we show the number of those different types that were structural types.
List of Figures

2.1 Syntax of the Ruby Subset $\lambda^{RB}$. .............................................. 23
2.2 Syntax of the Intermediate Language $\lambda^{I}$. .............................. 25
2.3 Expression translation from $\lambda^{RB}$ to $\lambda^{I}$. For simplicity rules T-PURE1 and T-IMPURE1 assume single argument methods. .......... 26
2.4 Program translation from $\lambda^{RB}$ to $\lambda^{I}$. .............................. 27

3.1 Discourse code that uses ActiveRecord to query a database. ............ 50
3.2 Comp type annotations for query methods. ................................. 50
3.3 Type Casts in a Method. .......................................................... 54
3.4 Type Checking SQL Strings in Discourse. .................................. 58
3.5 Syntax and Relations of $\lambda^{C}$. ............................................. 63
3.6 A subset of the type checking and rewriting rules for $\lambda^{C}$. .......... 66
3.7 Termination Checking with CompRDL. ......................................... 71

4.1 Method from the Discourse app, and the resulting constraints generated during inference. ......................................................... 90
4.2 Core types and constraints. ....................................................... 97
4.3 Standard constraint resolution rules. ......................................... 99
4.4 A code snippet from the TZInfo library. .................................... 119

5.1 Generating type constraints in TZInfo. ..................................... 133
5.2 An illustration of how DeepSim calculates the pairwise similarity scores for the set of input parameters \{year, month, day\} for the method from Figure 5.1a. ................................................. 138
5.3 A method defined in the code.org app and the resulting constraints. .... 141
5.4 Types, constraints, solutions, and inherited functions. ................ 143
5.5 Procedures used in SimTyper. ................................................... 145
5.6 SimTyper's Deep Similarity (DeepSim) Neural Network. ............... 148
5.7 A method * from the Money library which has YARD documentation. 151
5.8 Assessing the types inferred by SimTyper. We collected results under four configurations: constraint solving (C), constraint solving plus heuristics (CH), constraint solver plus the DeepSim network (CD), and all three (CHD). Results presented here use top-3 thresholding. Note the y-axis is scaled for each benchmark. .................. 160
5.9 Measuring SimTyper’s performance for arguments, returns, and instance, class, and global variables across all benchmarks. The plots use the same legend as Figure 5.8. Measurements were taken using top-3 thresholding. .................................................. 165

5.10 Evaluating design choices in SimTyper. Plots use the same legend as in Figure 5.8 and show data from CD across all benchmarks. Note y-axis starts at 700. Data for all plots appears in Appendix B. .......... 170

A.1 $\lambda^C$ and auxiliary definitions. ................................. 181
A.2 Dynamic semantics of $\lambda^C$ ........................................ 182
A.3 Type checking and check insertion rules for $\lambda^C$. ............. 183
A.4 Type checking rules for $\lambda^C$. .................................... 184
A.5 Program type checking rules. ........................................ 202
Chapter 1: Introduction

Dynamic type systems are featured in many popular programming languages including Ruby, Python, and JavaScript. The lack of static type enforcement means programmers can write code that is verbose and flexible, and that takes advantage of highly dynamic features such as metaprogramming. All of this allows for rapid prototyping and development of programs.

Yet there is a clear trade off. Static type systems are useful for catching bugs early in the development process, enforcing invariants, and providing useful documentation in the form of type annotations. These features become all the more beneficial as programs grow in size and are maintained by larger teams of developers. Thus, a growing area of research has examined ways to add static typing to dynamic languages [1, 2, 3, 4, 5, 6, 7]. The goal is to maintain the flexibility of dynamic type systems, while gaining some of the correctness guarantees of static types.

Such hybrid systems—typically deemed *gradual typing* when the system ensures soundness, and *optional typing* otherwise—have been explored for a growing number of languages, including Ruby [4, 5, 8], Racket [3, 9, 10], Python [11, 12, 13], JavaScript [14, 15, 16], and more. TypeScript, a typed superset of JavaScript which includes optional typing, was declared by GitHub to be the 4th most popular lan-
language in 2020 [17]. Moreover, recent versions of Ruby [18] and Python [19] include syntax for type annotations, which can serve as documentation and can be used in type-based analyses.

However, there are many common features and constructs behind programs written in dynamic languages that are challenging or not possible to capture with prior type systems. Programmers may wish to verify more expressive properties of programs than basic type correctness, such as the functional correctness properties of arithmetic operations. Moreover, the operations for many libraries popular in dynamic languages, such as libraries for database operations and heterogeneous data structures, cannot be precisely type checked using standard type systems. And finally, the process of annotating programs with static types itself can be quite burdensome for the programmer. But the standard solution to this problem—type inference based on constraint solving—often generates types that are verbose, confusing, and less useful to the programmer.

These challenges have motivated the research presented in this dissertation. In particular, I have explored these challenges through the design and implementation of new type systems for the Ruby language. However, the systems and methodologies I have contributed to are generalizable to other dynamic, and in some cases even static, languages.

I present the following thesis:

New mechanisms can be added to conventional type systems to make these systems more expressive and usable. Refinement types can be used to express and enforce stronger properties of programs, and combined with a just-in-time verification
approach, can be flexible enough to handle common dynamic features like mixins and metaprogramming. Type-level computations can express more precise properties for library methods, especially those for databases and heterogeneous data structures. And standard, constraint-based type inference can be complemented with heuristic rules and type equality prediction to produce more usable type annotations.

To substantiate this thesis, I present four novel type systems for Ruby: RTR, CompRDL, InferDL, and SimTyper. These four systems build on RDL, an existing Ruby type checker which I have contributed to. In the remainder of this chapter, I first provide an overview of RDL (§ 1.1), and then I briefly introduce each of the aforementioned novel type systems (§ 1.2).

1.1 RDL: A Ruby Type Checker

RDL [20] is an existing type and contract checking system for Ruby that I have extensively contributed to. We begin with a brief description of RDL, as it is foundational to the remainder of the work in this dissertation. Consider the following example:

```ruby
type '( Integer) → Integer', typecheck: :later
def incr_sec(x) if (x==59) then 0 else x+1 end; end
```

On the second line of this example, we define a simple Ruby method `incr_sec`, which takes a value `x` representing a second, and increments it by 1, wrapping back around to 0 when necessary. Above this method, we have written an RDL type annotation, which says that `incr_sec` should take and return an `Integer`. RDL will check `incr_sec` against this type and determine that the method is type safe.
Notably, RDL performs type checking by statically analyzing code, but it does so at runtime. Indeed, in the example, `type` is actually a method that we are calling, with a `String` argument representing the type of the subsequently defined method; in Ruby, methods can be called without enclosing parentheses around the arguments. This `String` argument will be parsed into RDL’s type system and stored in a global table that maintains a program’s type environment. We also pass the `type` method the label :later — in Ruby, strings prefixed by colons are `symbols`, which are interned strings. This label specifies the time that a method should be type checked. When the program subsequently calls `RDL.do_typecheck :later` (call omitted above), RDL will type check the source code of all methods whose type annotations are labeled :later.

This design allows RDL to support the metaprogramming that is common in Ruby and ubiquitous in Rails. For example, the programmer can perform type checking after metaprogramming code has run, when corresponding type definitions are available. See Ren and Foster [5] for more details.

RDL uses an expressive type language, including nominal, singleton, union, intersection, generic, variable, and structural types. It also includes special precise types for two of Ruby’s data structures: *tuple types* for fixed-sized arrays, which give the exact types for each element in an array; and *finite hash types*, which give the names of keys and types of values in a hash. In the coming chapters, we will see how we can piggyback off RDL’s expressive type language and unique design to implement refinement type checking, type-level computations, and expressive type inference.
1.2 Novel Type Systems

I now briefly introduce four novel type systems: RTR (§ 1.2.1), CompRDL (§ 1.2.2), InferDL (§ 1.2.3), and SimTyper (§ 1.2.4).

1.2.1 RTR: Refinement Types for Ruby

While RDL is useful for establishing the type safety of a program, its basic types alone cannot verify more expressive properties. To address this, we built RTR [21], a system that extends RDL with refinement types, which are types extended with expressive logical predicates. For example, recall the method incr_sec defined in Section 1.1. Below, we once again define the method, this time annotating it with refinement types:

```ruby
type '( Integer x { 0 ≤ x < 60 }) → Integer r { 0 ≤ r < 60 }'
def incr_sec(x) if (x==59) then 0 else x+1 end; end
```

In addition to enforcing basic type safety, the above refinement types enforce bounds on the method input and output: both the input (named x), and the output (named r) should be in the range $[0,60)$. Effectively, these refinement types encode pre- and postconditions on the method, which are more expressive than basic types alone. Checking such a specification is beyond the scope of traditional type checkers, and would typically have to be done using dynamic checks, which can be tedious and miss program paths. Using RTR, we can verify this specification statically for all paths through the method.

RTR works by encoding its verification problems into Rosette, a solver-aided
host language. **RTR** is able to achieve verification in the face of some of Ruby’s highly dynamic features, such as mixins and metaprogramming, using a combination of assume-guarantee reasoning and a novel approach we call *just-in-time verification*. We formalized **RTR** by showing a translation from a core, Ruby-like language with refinement types into **Rosette**. We evaluated **RTR** by applying it to six Ruby programs and using it to verify a range of functional correctness properties, and we found that **RTR** can successfully verify key methods in these programs, taking only a few minutes in the worst case to perform verification. We provide a full discussion of **RTR** in Chapter 2.

1.2.2 **CompRDL**: Type-Level Computations for Ruby Libraries

Because Ruby is commonly used for web development, Ruby programs frequently interact with a database. However, type checking database queries is a challenging problem. The types of database operations often depend on program values, such as the names of tables or columns being queried, in a way that traditional type systems cannot capture. We found similar issues arise in type checking other Ruby libraries as well, such as those for arrays and hashes.

To address this issue, we developed **CompRDL**, a type system for Ruby that allows library method type signatures to include *type-level computations* (or comp types for short). Combined with singleton types for table and column names, comp types let us give database query methods type signatures that compute a table’s schema to yield very precise type information. Comp types for hash, array, and
string libraries can also increase precision and thereby reduce the need for type casts.

We formalized CompRDL using a core, Ruby-like language, and proved its type system sound. Its soundness relies on both a lightweight termination checker and the use of dynamic checks to enforce the non-checked comp types. We evaluated CompRDL by writing annotations with type-level computations for several Ruby core libraries and database query APIs. We then used those annotations to type check two popular Ruby libraries and four Ruby on Rails web apps. We found the type annotations were relatively compact and we were able to successfully type check 132 methods across our subject programs. The use of type-level computations allowed us to check more expressive properties, with fewer manually inserted casts, than was possible without type-level computations. In the process, we found three errors that were confirmed by the developers. Chapter 3 gives a full presentation of CompRDL.

1.2.3 InferDL: Sound, Heuristic Type Annotation Inference for Ruby

The aforementioned type systems are useful systems for ensuring the correctness of programs, but they still require the programmer to manually write type annotations. This process can be burdensome and time-consuming, especially for larger, more complex code bases. The standard solution to this problem is type inference, which traditionally uses constraint solving to catch type errors in a program without needing type annotations. We can take this process a step further, not only using constraint solving to catch bugs, but also to generate the most-general
type annotations for program values. However, we have found that in practice, the resulting type annotations are often overly general, resulting in type annotations that are verbose, confusing, and less useful to the programmer.

Building on a prior proposal for heuristic Ruby type inference [22], we developed InferDL, a system for generating type annotations that uses heuristic rules to refine the overly general types produced by constraint solving. For example, InferDL can use a heuristic rule that guesses that a parameter whose name ends in “count” is an Integer. InferDL works by first running standard type inference, then applying heuristics to any positions for which standard type inference produced an overly general type. Heuristic guesses are added as constraints to the type inference problem to ensure they are consistent with the rest of the program and other heuristic guesses; inconsistent guesses are discarded. We formalized InferDL in a core type and constraint language. To evaluate InferDL, we applied it to four Ruby on Rails apps that had been previously type checked with RDL, and hence had type annotations. We found that, when using heuristics, InferDL inferred 22% more types that were as or more precise than the previous annotations, compared to standard type inference without heuristics. We also found one new type error. We further evaluated InferDL by applying it to six additional apps, finding five additional type errors. Chapter 4 presents InferDL in full.
1.2.4 SimTyper: Sound Type Inference for Ruby using Type Equality Prediction

With InferDL, we demonstrated that heuristics can be useful for refining constraint-based type inference. However, hard-coded heuristic rules, while useful for their target programs, may generalize poorly to new programs. While in theory a programmer can write new heuristics for each new program, in practice this can be a burdensome and time-consuming process.

To improve on heuristic-based inference, we introduce SimTyper, a system that builds on InferDL for the purpose of inferring usable annotations. The key novelty of SimTyper is type equality prediction, a new, machine learning-based technique that predicts when method arguments or returns are likely to have the same type. SimTyper finds pairs of positions that are predicted to have the same type yet one has a verbose, overly general solution and the other has a usable solution. It then guesses the two types are equal, keeping the guess if it is consistent with the rest of the program, and discarding it if not. In this way, like InferDL, types inferred by SimTyper are guaranteed to be sound. Type equality prediction is performed by a deep similarity (DeepSim) neural network, which follows the Siamese network architecture and uses CodeBERT, a pre-trained model, to embed source tokens into vectors that capture tokens and their contexts. DeepSim is trained on 100,000 pairs of arguments/returns labeled with type similarity information extracted from 371 Ruby programs with manually documented, but not checked, types. We evaluated SimTyper on eight Ruby programs and found that, compared to standard type
inference, SimTyper finds 69\% more types that match programmer-written type
information. Moreover, DeepSim can predict rare types that appear neither in the
Ruby standard library nor in the training data. Our results show that type equality
prediction can help type inference systems effectively produce more usable types.
Chapter 2: Refinement Type for Ruby

This chapter introduces \texttt{RTR}, a refinement type system for Ruby. Refinement types are a popular way to specify and reason about key program properties. \texttt{RTR} is built on top of \texttt{RDL}, and it works by encoding its verification problems into Rosette, a solver-aided host language. \texttt{RTR} is able to handle some of Ruby’s highly dynamic features, such as mixins and metaprogramming, through a combination of assume-guarantee reasoning, and a novel approach we call \textit{just-in-time verification}.

2.1 Introduction

Refinement types combine types with logical predicates to encode program invariants [23, 24]. Recall the method \texttt{incr\_sec} from § 1.1, which increments a second. We can write the following refinement type specification for this method:

\begin{verbatim}
  type :incr\_sec, '( Integer x { 0 \leq x < 60 }) \rightarrow Integer r { 0 \leq r < 60}'
\end{verbatim}

With this specification, \texttt{incr\_sec} can only be called with integers that are valid seconds (between 0 and 59) and the method will always return valid seconds.

Refinement types were introduced to reason about simple invariants, like safe array indexing [23], but since then they have been successfully used to verify sophisticated properties including termination [25], program equivalence [26], and correct-
ness of cryptographic protocols [27], in various languages (e.g., ML [28], Racket [10], and TypeScript [29]).

We introduce RTR, a tool that adds refinement types to RDL and verifies them via a translation into Rosette [30], a solver-aided host language. Since Rosette is not object-oriented, RTR encodes Ruby objects as Rosette structs that store object fields and an integer identifying the object’s class. At method calls, RTR uses RDL’s type information to statically overestimate the possible callees. When methods with refinement types are called, RTR can either translate the callee directly or treat it modularly by asserting the method preconditions and assuming the postcondition, using purity annotations to determine which fields (if any) the method may mutate. (§ 2.2)

In addition to standard object-oriented features, Ruby includes dynamic language features that increase flexibility and expressiveness. In practice, this introduces two key challenges in refinement type verification: mixins, which are Ruby code modules that extend other classes without direct inheritance, and metaprogramming, in which code is generated on-the-fly during runtime and used later during execution. The latter feature is particularly common in Ruby on Rails, a popular Ruby web development framework.

To meet these challenges, RTR uses two key ideas. First, RTR incorporates assume-guarantee checking [31] to reason about mixins. RTR verifies definitions of methods in mixins by assuming refinement type specifications for all undefined, external methods. Then, by dynamically intercepting the call that includes a mixin in a class, RTR verifies the appropriate class methods satisfy the assumed refine-
ment types (§ 2.3.1). Second, RTR uses *just-in-time verification* to reason about metaprogramming, following RDL’s just-in-time type checking [5]. In this approach, (refinement) types are maintained at run-time, and methods are checked against their types after metaprogramming code has executed but before the methods have been called (§ 2.3.2).

We formalized RTR by showing how to translate $\lambda^{RB}$, a core Ruby-like language with refinement types, into $\lambda^I$, a core verification-oriented language. We then discuss how to map the latter into Rosette, which simply requires encoding $\lambda^I$’s primitive object construct into Rosette structs and translating some control-flow constructs such as return (§ 2.4).

We evaluated RTR by using it to check a range of functional correctness properties on six Ruby and Rails applications. In total, we verified 31 methods, comprising 271 lines of Ruby, by encoding them as 1,061 lines of Rosette. We needed 73 type annotations. Verification took a total median time (over multiple trials) of 506 seconds (§ 2.5).

Thus, we believe RTR is a promising first step toward verification for Ruby.

2.2 Overview

We begin with an overview of RTR, which extends RDL (§ 1.1) with refinement types. In RTR, program invariants are specified with refinement types (§ 2.2.1) and checked by translation to Rosette (§ 2.2.2). We translate Ruby objects to Rosette structs (§ 2.2.3) and method calls to function calls (§ 2.2.4).
2.2.1 Refinement Type Specifications

Refinement types in RTR are RDL types extended with logical predicates. For example, consider the incr_sec method first introduced in § 1.1. We can tweak the type annotation for this method to include refinement types:

```
type '( Integer x \{ 0 \leq x < 60 \}) \to Integer r \{ 0 \leq r < 60\}'
def incr_sec(x) if (x==59) then 0 else x+1 end ; end
```

This type indicates the argument and result of incr_sec are integers in the range from 0 to 59. In general, refinements (in curly braces) may be arbitrary Ruby expressions that are treated as booleans, and they should be pure, i.e., have no side effects, since effectful predicates make verification either complicated or imprecise [32]. As in RDL, the type annotation, which is a string, is parsed and stored in a global table which maintains the program’s type environment.

2.2.2 Verification Using Rosette

RTR checks method specifications by encoding their verification into Rosette [30], a solver-aided host language built on top of Racket. Among other features, Rosette can perform verification by using symbolic execution to generate logical constraints, which are discharged using Z3 [33].

For example, to check incr_sec, RTR creates the equivalent Rosette program:

```
( define (incr_sec x) ( if ( = x 59) 0 (+ x 1)))
( define --symbolic x_in integer?)
( verify #:assume (assert 0 \leq x < 60)
  #:guarantee (assert (let ([r (incr_sec x)]) 0 \leq r < 60)))
```

Here x_in is an integer symbolic constant representing an unknown, arbitrary integer
argument. Rosette symbolic constants can range over the solvable types integers, booleans, bitvectors, reals, and uninterpreted functions. We use Rosette’s verify function with assumptions and assertions to encode pre- and postconditions, respectively. When this program is run, Rosette searches for an x_in such that the assertion fails. If no such value exists, then the assertion is verified.

2.2.3 Encoding and Reasoning about Objects

We encode Ruby objects in Rosette using a struct type, i.e., a record. More specifically, we create a struct type object that contains an integer classid identifying the object’s class, an integer objectid identifying the object itself, and a field for each instance variable of all objects encountered in the source Ruby program (similar to prior work [34, 35]).

For example, consider a Ruby class Time with three instance variables @sec, @min, and @hour, and a method is_valid that checks all three variables are valid:

```ruby
class Time
  attr_accessor :sec, :min, :hour

  def initialize (s, m, h) @sec = s; @min = m; @hour = h; end

  type '(' → bool'
  def is_valid 0 ≤ @sec < 60 ∧ 0 ≤ @min < 60 ∧ 0 ≤ @hour < 24; end end

RTR observes three fields in this program, and thus it defines:

( struct object ([ classid ][ objectid ]
  [@sec #:mutable] [@min #:mutable] [@hour #:mutable]))

Here object includes fields for the class ID, object ID, and the three instance vari-
ables. Note since `object`'s fields are statically defined, our encoding cannot handle dynamically generated instance variables, which we leave as future work.

RTR then translates Ruby field reads or writes as getting or setting, respectively, `object`'s fields in Rosette. For example, suppose we add a method `mix` to the `Time` class and specify it is only called with and returns valid times:

```ruby
type :mix, '(Time t1 { t1.is_valid }, Time t2 { t2.is_valid },
            Time t3 { t3.is_valid }) → Time r { r.is_valid }'
def mix(t1, t2, t3) @sec = t1.sec; @min = t2.min; @hour = t3.hour; self; end
```

Initially, type checking fails because the getters' and setters' (e.g., `sec` and `sec=`) types are unknown. Thus, we add those types:

```ruby
type :sec, '() → Integer i { i == @sec }'
type :sec=, '(Integer i) → Integer out { i == @sec }'
```

(Note these annotations can be generated automatically using our approach to meta-programming, described in § 2.3.2.) This allows RTR to proceed to the translation stage, which generates the following Rosette function:

```ruby
(define (mix self t1 t2 t3)
    (set-object-@sec! self (sec t1))
    (set-object-@min! self (min t2))
    (set-object-@hour! self (hour t3))
    self)
```

(Asserts, assumes, and verify call omitted.) Here `(set-object-x! y w)` writes `w` to the `x` field of `y` and the field selectors `sec`, `min`, and `hour` are uninterpreted functions. Note that `self` turns into an explicit additional argument in the Rosette definition. Rosette then verifies this program, thus verifying the original Ruby `mix` method.
2.2.4 Method Calls

To translate a Ruby method call e.m(e1, .., en), RTR needs to know the callee, which depends on the runtime type of the receiver e. RTR uses RDL’s type information to overapproximate the set of possible receivers. For example, if e has type A in RDL, then RTR translates the above as a call to A.m. If e has a union type, RTR emits Rosette code that branches on the potential types of the receiver using object class IDs and dispatches to the appropriate method in each branch. This is similar to class hierarchy analysis [36], which also uses types to determine the set of possible method receivers and construct a call graph.

Once the method being called is determined, we translate the call into Rosette. As an example, consider a method to_sec that converts a Time object to seconds, after it calls the method incr_sec from § 2.2.1.

\[
\text{type '}(\text{Time } t \{ \text{t.is_valid } \}) \rightarrow \text{Integer } r \{ 0 \leq r < 90060 \}\'
\]
\[
\text{def to_sec(t) incr_sec(t.sec) + 60 * t.min + 3600 * t.hour; end}
\]

RTR’s translation of to_sec could simply call directly into incr_sec’s translation. This is equivalent to inlining incr_sec’s code. However, inlining is not always possible or desirable. A method’s code may not be available because the method comes from a library, is external to the environment (§ 2.3.1), or has not been defined yet (§ 2.3.2). The method might also contain constructs that are difficult to verify, like diverging loops.

Instead, RTR can model the method call using the programmer provided method specification. To precisely reason with only a method’s specification, RTR follows
Dafny [37] and treats pure and impure methods differently.

**Pure methods.** Pure methods have no side effects and return the same result for the same inputs, satisfying the congruence property $\forall x, y. x = y \Rightarrow m(x) = m(y)$ for a given method $m$. Thus, pure methods can be encoded using Rosette’s uninterpreted functions. The method `incr_sec` is indeed pure, so we can label it as such:

```plaintext
type :incr_sec, `( Integer x { 0\leq x<60 }) \rightarrow Integer r { 0\leq r<60 }', :pure
```

With the `pure` label, the translation of `to_sec` treats `incr_sec` as an uninterpreted function. Furthermore, it asserts the precondition $0 \leq x < 60$ and assumes the postcondition $0 \leq r < 60$, which is enough to verify `to_sec`.

**Impure methods.** Most Ruby methods have side effects and thus are not pure. For example, consider `incr_min`, a mutator method that adds a minute to a `Time`:

```plaintext
type '(Time t { t.is_valid ∧ t.min < 59 }) \rightarrow Time r { r.is_valid }',
    modifies: { t: @min, t: @sec }
def incr_min(t)
    if t.sec<59 then t.sec=incr_sec(t.sec) else t.min+=1; t.sec=0 end
    return t
end
```

A translated call to `incr_min` generates a fresh symbolic value as the method’s output and assumes the method’s postcondition on that value. Because the method may have side effects, the `modifies` label is used to list all fields of inputs which may be modified by the method. Here, a translated call to `incr_min` will `havoc` (set to fresh symbolic values) t’s `@min` and `@sec` fields.

We leave support for other kinds of modifications (e.g., global variables, tran-
sively reachable fields), as well as enforcing the pure and modifies labels, as future work.

2.3 Just-In-Time Verification

Next, we show how RTR handles code with dynamic bindings via mixins (§ 2.3.1) and metaprogramming (§ 2.3.2).

2.3.1 Mixins

Ruby implements mixins via its module system. A Ruby module is a collection of method definitions that are added to any class that includes the module at runtime. Interestingly, modules may refer to methods that are not defined in the module but will ultimately be defined in the including class. Such incomplete environments pose a challenge for verification.

Consider the following method that has been extracted and simplified from the Money library, used in our evaluations (§ 2.5).

```ruby
module Arithmetic
  type '( Integer x)→ Float r { r == x/value }'
  def div_by_val(x) x/value; end
end
```

The module method div_by_val divides its input x by value. RTR’s specification for / requires that value cannot be 0.

Notice that value is not defined in Arithmetic. Rather, it must be defined wherever Arithmetic is included. Therefore, to proceed with verification in RTR, the programmer must provide an annotation for value:
Using this annotation, RTR can verify \texttt{div\_by\_value}. Then when \texttt{Arithmetic} is included in another class, RTR verifies \texttt{value}'s refinement type. For example, consider the following code:

```ruby
class Money
  include Arithmetic
  def value()
    if (@val > 0) then (return @val) else (return 0.01) end
  end
end
```

RTR dynamically intercepts the call to \texttt{include} and then applies the type annotations for methods not defined in the included module, in this case verifying \texttt{value} against the annotation in \texttt{Arithmetic}. Thus, RTR follows an assume-guarantee paradigm [31]: it assumes \texttt{value}'s annotation while verifying \texttt{div\_by\_val} and then guarantees the annotation once \texttt{value} is defined.

### 2.3.2 Metaprogramming

Metaprogramming in Ruby allows new methods to be created and existing methods to be redefined on the fly, posing a challenge for verification. RTR addresses this challenge using just-in-time checking [5], in which, in addition to code, method annotations can also be generated dynamically.

We illustrate the just-in-time approach using an example from \textit{Boxroom}, a Rails app for managing and sharing files in a web browser (§ 2.5). The app defines a class \texttt{UserFile} that is a Rails \textit{model} corresponding to a database table:

```ruby
class UserFile < ActiveRecord::Base
```
belongs_to :folder
...
  type '(Folder target) → Bool b { folder == target }'
  def move(target) folder = target; save!; end
...
end

Here calling `belongs_to` tells Rails that every UserFile is associated with a folder (another model). The `move` method updates the associated folder of a UserFile and saves the result to the database. We annotate `move` to specify that the UserFile's new folder should be the same as `move`'s argument.

This method and its annotation are seemingly simple, but there is a problem. To verify `move`, RTR needs an annotation for the `folder=` method, which is not statically defined. Rather, it is dynamically generated by the call to `belongs_to`.

To solve this problem in RTR, we instrument `belongs_to` to generate type annotations for the setter (and getter) method, as follows:

```ruby
module ActiveRecord::Associations::ClassMethods
  pre(:belongs_to) do |*args|
    name = args[0].to_s
    cname = name.camelize
    type "#{name}" , "() → #{cname} c", :pure
    type "#{name}=" , "(#{cname} i) → #{cname} o { #{name} == i }"
    true
  end
end
```

We call `pre`, an RDL method, to define a precondition code block (i.e., an anonymous function) which will be executed on each call to `belongs_to`. First, the block sets `name` and `cname` to be the string version of the first argument passed to `belongs_to` and its camelized representation, respectively. Then, we create types for the `name` and `name=` methods. Finally, we return `true` so the contract will succeed. In our
example, this code generates the following two type annotations upon the call to

```
begins{helbongs_to}:
```

```
type 'folder', '()' -> Folder c', :pure
```

```
type 'folder=', '(Folder i) -> Folder o { folder == i }'
```

These annotations will be generated when `belongs_to` is invoked with the `:folder` argument, which happens exactly after the `UserFile` class is loaded. Thus, not only will the call to `belongs_to` generate getter and setter methods, but it also generates useful annotations for these methods. With these annotations, verification of the initial `move` method succeeds.

2.4 From Ruby to Rosette

In this section, we formally describe our verifier and the translation from Ruby to Rosette. We start (§ 2.4.1) by defining $\lambda^{RB}$, a Ruby subset that is extended with refinement type specifications. We give (§ 2.4.2) a translation from $\lambda^{RB}$ to an intermediate language $\lambda^I$, and then (§ 2.4.3) we discuss how $\lambda^I$ maps to a Rosette program. Finally (§ 2.4.5), we use this translation to construct a verifier for Ruby.

2.4.1 Core Ruby $\lambda^{RB}$ and Intermediate Representation $\lambda^I$

2.4.1.1 $\lambda^{RB}$

Figure 2.1 defines $\lambda^{RB}$, a core Ruby-like language with refinement types. Constants consist of `nil`, booleans, and integers. Expressions include constants, variables, assignment, conditionals, sequences, and the reserved variable `self`, which
Constants  \( c ::= \text{nil} \mid \text{true} \mid \text{false} \mid 0, 1, -1, \ldots \)

Expressions  \( e ::= c \mid x \mid x:=e \mid \text{if } e \text{ then } e \text{ else } e \mid e \; e \)
\( \mid \text{self} \mid f \mid f:=e \mid e.m(\overline{e}) \mid \text{A.new} \mid \text{return}(e) \)

Refined Types  \( t ::= \{ x : A \mid e \} \)

Program  \( P ::= \cdot \mid d, P \mid a, P \)

Definition  \( d ::= \text{def } A.m(\overline{l})::t;l = e \)

Annotation  \( a ::= A.m :: (\overline{l}) \to t ; \ell \) \text{ with } l \neq \text{exact} \)

Labels  \( l ::= \text{exact} \mid \text{pure} \mid \text{modifies}[x.f] \)

\( x \in \text{var ids}, \; f \in \text{field ids}, \; m \in \text{meth ids}, \; A \in \text{class ids} \)

Figure 2.1: Syntax of the Ruby Subset \( \lambda^{RB} \).

refers to a method’s receiver. Also included are references to an instance variable \( f \) and instance variable assignment; we note that in actual Ruby, field names are preceded by a “@”. Finally, expressions include method calls, constructor calls \( \text{A.new} \) which create a new instance of class \( A \), and return statements.

Refined types \( \{ x : A \mid e \} \) refine the basic type \( A \) with the predicate \( e \). The basic type \( A \) is used to represent both user defined and built-in classes including nil, booleans, integers, floats, etc. The refinement \( e \) is a pure, boolean valued expression that may refer to the refinement variable \( x \). In the interest of greater simplicity for the translation, we require that \( \text{self} \) does not appear in refinements \( e \); however, extending the translation to handle this is natural, and our implementation allows for it. Sometimes we simplify the trivially refined type \( \{ x : A \mid \text{true} \} \) to just \( A \).

A program is a sequence of method definitions and type annotations over methods. A method definition \( \text{def } A.m(\{x_1 : A_1 \mid e_1 \}, \ldots, \{x_n : A_n \mid e_n \}):t;l = e \) defines the method \( A.m \) with arguments \( x_1, \ldots, x_n \) and body \( e \). The type specification of the definition is a dependent function type: each argument binder \( x_i \) can appear
inside the arguments’ refinements types $e_j$ for all $1 \leq j \leq n$, and can also appear in the refinement of the result type $t$. A method type annotation $A.m :: (\bar{t}) \rightarrow t ; l$ binds the method named $A.m$ with the dependent function type $(\bar{t}) \rightarrow t$. $\lambda^{RB}$ includes both method annotations and method definitions because annotations are used when a method’s code is not available, e.g., in the cases of library methods, mixins, or metaprogramming.

A label $l$ can appear in both method definitions and annotations to direct the method’s translation into Rosette as described in § 2.2.4. The label exact states that a called method will be exactly translated by using the translation of the body of the method. Since method type annotations do not have a body, they cannot be assigned the exact label. The pure label indicates that a method is pure and thus can be translated using an uninterpreted function. Finally, the modifies $[x.f]$ label is used when a method is impure, i.e., it may modify its inputs. As we saw earlier, the list $x.f$ captures all the argument fields which the method may modify.

2.4.1.2 $\lambda^I$.

Figure 2.2 defines $\lambda^I$, a core verification-oriented language that easily translates to Rosette. $\lambda^{RB}$ methods map to $\lambda^I$ functions, and $\lambda^{RB}$ objects map to a special object struct type. $\lambda^I$ provides primitives for creating, altering, and referencing instances of this type. Values in $\lambda^I$ consist of constants $c$ (defined identically to $\lambda^{RB}$) and object$(i_1, i_2, [f_1 w_1] \ldots [f_n w_n])$, an instantiation of an object type with class ID $i_1$, object ID $i_2$, and where each field $f_i$ of the object is bound to value
Values \( w ::= c \mid \text{object}(i, i, \overline{f w}) \)

Expressions \( u ::= w \mid x \mid x := u \mid \text{if } u \text{ then } u \text{ else } u \mid u ; u \)
\[ \mid \text{let } ([x u]) \text{ in } u \mid x(u) \mid \text{assert}(u) \]
\[ \mid \text{assume}(u) \mid \text{return}(u) \mid \text{havoc}(x.f) \mid x.f := u \mid x.f \]

Program \( Q ::= \cdot \mid d, Q \mid v, Q \)

Definition \( d ::= \text{define } x(x) = u \mid \text{define-sym}(x, A) \)

Verification Query \( v ::= \text{verify}(u \Rightarrow u) \)

\( x \in \text{var ids}, f \in \text{field ids}, A \in \text{types}, i \in \text{integers} \)

Figure 2.2: Syntax of the Intermediate Language \( \lambda^I \).

\( w_i \). Expressions include let bindings (let \((x_i u_i)\) in \(u\)) where each \(x_i\) may appear free in \(u_j\) if \(i < j\). They also include function calls, assert, assume, and return statements, as well as havoc\((x.f)\), which mutates \(x\)'s field \(f\) to a fresh symbolic value. Finally, they include field assignment \(x.f := u\) and field reads \(x.f\).

A program is a series of definitions and verification queries. A definition is a function definition or a symbolic definition \(\text{define-sym}(x, A)\), which binds \(x\) to either a fresh symbolic value if \(A\) is a solvable type (e.g., boolean, integer; see § 2.2.2) or a new object with symbolic fields defined depending on the type of \(A\). Finally, a verification query \(\text{verify}(\pi \Rightarrow u)\) checks the validity of \(u\) assuming \(\pi\).

2.4.2 From \(\lambda^{RB}\) to \(\lambda^I\)

Figures 2.3 and 2.4 define the translation function \(e \leadsto u\) that maps expressions and programs from \(\lambda^{RB}\) to \(\lambda^I\).
Expression Translation

\[
\begin{array}{ccl}
    c & \leadsto & c & \text{T-CONST} \\
    x & \leadsto & x & \text{T-VAR} \\
    e_1 & \leadsto & u_1 & e_2 & \leadsto & u_2 & \text{T-SEQ} \\
    \text{if } e_1 \text{ then } e_2 \text{ else } e_3 & \leadsto & \text{if } u_1 \text{ then } u_2 \text{ else } u_3 & \text{T-IF} \\
    e & \leadsto & u & \text{T-VAARassn} \\
    \text{return}(e) & \leadsto & \text{return}(u) & \text{T-RET} \\
    f \in \mathcal{F} & \leadsto & \text{self}.f & \text{T-Inst} \\
    f := e & \leadsto & \text{self}.f := u & \text{T-InstAssn} \\
    \text{classId}(A) = i_c & \leadsto & \text{freshID}(i_o) & f_i \in \mathcal{F} & \text{T-New} \\
    A.\text{new} & \leadsto & \text{object}(i_c, i_o, [f_i \text{ nil}] \ldots [f_{|\mathcal{F}|} \text{ nil}]) \\
    \text{typeof}(e_F) = A & \leadsto & \text{exact} = \text{labelOf}(A.m) \\
    A.m \in \mathcal{M} & \leadsto & e_F \leadsto u_F & e_i \leadsto u_i & \text{T-Exact} \\
    e_F.m(\pi) & \leadsto & A.m(u_F, \pi) \\
    e_F.m(e) & \leadsto & \text{let } ([x u]|r \ A.m(u_F, a)] \text{ in assert}(u_x); \text{ assume}(u_r); r & \text{T-Pure1} \\
    \text{typeof}(e_F) = A & \leadsto & \text{pure} = \text{labelOf}(A.m) \\
    A.m \in \mathcal{U} & \text{freshVar}(x, r) & \text{specOf}(A.m) = (\{x : A_x | e_x, k\}) \rightarrow \{r : A_r | e_r\} \\
    e_F \leadsto u_F & e \leadsto u & e_x \leadsto u_x & e_r \leadsto u_r & \text{T-Impure1} \\
    e_F.m(e) & \leadsto & \text{let } ([x u]) \text{ in define-sym}(r, A_r); \text{ assert}(u_x); \text{ havoc}(h_F \cup h_x); \text{ assume}(u_r); r & \text{T-Impure1} \\
\end{array}
\]

Figure 2.3: Expression translation from $\lambda^{RB}$ to $\lambda^l$. For simplicity rules T-Pure1 and T-Impure1 assume single argument methods.
Program Translation

\[
\begin{array}{c|c|c|c|c}
\text{T-EMP} & \text{T-ANN} \\
\vdash & \vdash \\
\end{array}
\]

\[
P \rightsquigarrow Q
\]

\[
A.m :: (x_1:t_1, \ldots, x_n:t_n) \rightarrow t; \ l, P \rightsquigarrow Q
\]

\[
\begin{array}{c|c|c|c}
\text{T-DEF} \\
t_i = \{x_i : A_{x_i} | e_{x_i}\} & t = \{r : A_r | e_r\} & P \rightsquigarrow Q & 1 \leq i \leq n \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
def A.m(t_1, \ldots, t_n)::t; \ l = e, P \rightsquigarrow & \text{define } A_m(self, x_1, \ldots, x_n) = u; \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
define-sym(self, A); & \\
define-sym(x_i, A_{x_i}); & \\
verify(u_{x_1}, \ldots, u_{x_n} \Rightarrow u_r); Q & \\
\end{array}
\]

Figure 2.4: Program translation from \(\lambda^{RB}\) to \(\lambda^I\).

Global States. The translation uses sets \(M, U, F\), to ensure all the methods, uninterpreted functions, and fields are well-defined in the generated \(\lambda^I\) term:

\[
\begin{align*}
M & ::= A_1.m_1, \ldots, A_n.m_n \\
U & ::= A_1.m_1, \ldots, A_n.m_n \\
F & ::= f_1, \ldots, f_n
\end{align*}
\]

In the translation rules, we use the standard set operations \(x \in \mathcal{X}\) and \(|\mathcal{X}|\) to check membership and size of the set \(\mathcal{X}\). Thus, the translation relation is defined over these sets: \(\mathcal{M}, \mathcal{U}, \mathcal{F} \vdash e \rightsquigarrow u\). Since the rules do not modify these environments, in Figures 2.3 and 2.4 we simplify the rules to \(e \rightsquigarrow u\). Note that even though the rules “guess” these environments by making assumptions about which elements are members of the sets, in an algorithmic definition the rules can be used to construct the sets.

Expressions. Figure 2.3 defines the translation function for expressions. The rules T-CONST and T-VAR are identity while the rules T-IF, T-SEQ, T-RET, and T-VARAASSN are trivially inductively defined. The rule T-SELF translates \texttt{self} into
the special variable named \textit{self} in $\lambda^I$. The \textit{self} variable is always in scope, since each $\lambda^{RB}$ method translates to a $\lambda^I$ function with an explicit first argument named \textit{self}.

The rules T-\text{Inst} and T-\text{InstAssn} translate a reference from and an assignment to the instance variable $f$, to a read from and write to, respectively, the field $f$ of the variable \textit{self}. Moreover, both the rules assume the field $f$ to be in global field state $\mathcal{F}$. The rule T-New translates from a constructor call $A$.\texttt{new} to an \texttt{object} instance. The \texttt{classId}(A) function in the premise of this rule returns the class ID of $A$. The \texttt{freshID}(i_o) predicate ensures the new \texttt{object} instance has a fresh object ID. Each field of the new \texttt{object}, $f_1, \ldots, f_{|\mathcal{F}|}$, is initially bound to \texttt{nil}.

\textbf{Method Calls.} To translate the $\lambda^{RB}$ method call $e_F.m(e)$, we first use the function \texttt{typeOf}(e_F) to type $e_F$ via RDL type checking 1.1. If $e_F$ is of type $A$, we split cases of the method call translation based on the value of \texttt{labelOf}(A.m), the label specified in the annotation of $A.m$ (as informally described in § 2.2.4).

The rule T-\text{Exact} is used when the label is \texttt{exact}. The receiver $e_F$ is translated to $u_F$ which becomes the first (i.e., the \textit{self}) argument of the function call to $A.m$. Moreover, $A.m$ is assumed to be in the global method name set $\mathcal{M}$ since it belongs to the transitive closure of the translation.

We note that for the sake of clarity, in the T-Pure1 and T-Impure1 rules, we assume that the method $A.m$ takes just one argument; the rules can be extended in a natural way to account for more arguments. The rule T-Pure1 is used when the label is \texttt{pure}. In this case, the call is translated as an invocation to the uninterpreted function $A.m$, so $A.m$ should be in the global set of uninterpreted
functions $\mathcal{U}$. The specification $\text{specOf}(A.m)$ of the method is also enforced. Let $$\{x : A_x \mid e_x\} \rightarrow \{r : A_r \mid e_r\}$$ be the specification. We assume that the binders in the specification are $\alpha$-renamed so that the binders $x$ and $r$ are fresh. We use $x$ and $r$ to bind the argument and the result, respectively, to ensure, via $\lambda$-normal form conversion [38], that they will be evaluated exactly once, even though $x$ and $r$ may appear many times in the refinements. To enforce the specification, we assert the method’s precondition $e_x$ and assume the postcondition $e_r$.

If a method is labeled with $\text{modifies}[p]$ then the rule $\text{T-IMPURE1}$ is applied. We locally define a new symbolic object as the return value, and we havoc the fields of all arguments (including $\text{self}$) specified in the $\text{modifies}$ label, thereby assigning these fields to new symbolic values. Since we do not translate the called method at all, no global state assumptions are made.

**Programs.** Finally, we use the translation relation to translate programs from $\lambda^{RB}$ to $\lambda^I$, i.e., $P \leadsto Q$. This is shown in Figure 2.4. The rule $\text{T-ANN}$ discards type annotations. The rule $\text{T-DEF}$ translates a method definition for $A.m$ to the function definition $A_.m$ that takes the additional first argument $\text{self}$. The rule also considers the declared type of $A.m$ and instantiates a symbolic value for every input argument. Finally, all refinements from the inputs and output of the method type are translated and the derived verification query is made.
2.4.3 From $\lambda^I$ to Rosette

We write $Q \rightarrow R$ to encode the translation of the $\lambda^I$ program $Q$ to the Rosette program $R$. This translation is straightforward, since $\lambda^I$ consists of Rosette extended with some macros to encode Ruby-verification specific operators, like `define-sym` and `return`. In fact, in the implementation of the translation (§ 2.5), we used Racket’s macro expansion system to achieve this final transformation.

Handling objects. $\lambda^I$ contains multiple constructs for defining and altering objects, which are expanded in Rosette to perform the associated operations over `object` structs. The expressions `object($i_c$, $i_o$, $[f\ w]$)` and `havoc($x.f$)`, and the definition `define-sym($x$, $A$)`, all described in § 2.4.1, are expanded to perform the corresponding operations over values of the `object` struct type.

Control Flow. Macro expansion is used to translate `return` and `assume` statements, and exceptions into Rosette, since those forms are not built-in to the language. To encode `return`, we expand every function definition in $\lambda^I$ to keep track of a local variable `ret`, which is initialized to a special `undefined` value and returned at the end of the function. We transform every statement `return(e)` to update the value of `ret` to $e$. Then, we expand every expression $u$ in a function to `unless-done(u)`, which checks the value of `ret`, proceeding with $u$ if `ret` is `undefined` or skipping $u$ if there is a return value.

We used the encoding of `return` to encode more operators. For example, `assume` is encoded in Rosette as a macro that returns a special `fail` value when
assumptions do not hold. The verification query then needs to be updated with
the condition that fail is not returned. A similar expansion is used to encode and
propagate exceptions.

2.4.4 Primitive Types

$\lambda^{RB}$ provides constructs for functions, assignments, control flow, etc, but does
not provide the theories required to encode interesting verification properties that,
for example, reason about booleans and numbers. On the other hand, Rosette
is a verification oriented language with special support for common theories over
built-in datatypes, including booleans, numeric types, and vectors. To bridge this
gap, we encode certain Ruby expressions, such as constants $c$ in $\lambda^{RB}$, into Rosette’s
Corresponding built-in datatypes.

Equality and Booleans. To precisely reason about equality, we encode Ruby’s
$==$ method over arbitrary objects using the object class’ $==$ method if one is defined.
If the class inherits this method from Ruby’s top class, BasicObject, then we encode
$==$ using Rosette’s equality operator equal? to check equality of object IDs. We
encode Ruby’s booleans and operations over them as Rosette’s respective booleans
and their operators.

Integers and Floats. By default, we encode Ruby’s infinite-precision Integer
and Float objects as Rosette’s built-in infinite-precision integer and real datatypes,
respectively. The infinite-precision encoding is efficient and precise, but it may re-
sult in undecidable queries involving non-linear arithmetic or loops. To perform (bounded) verification in such cases, we provide, via a configuration flag, the option of encoding Ruby’s integers as Rosette’s built-in finite sized bitvectors.

**Arrays.** Finally, we provide a special encoding for Ruby’s arrays, which are commonly used both for storing arbitrarily large random-access data and to represent mixed-type tuples, stacks, queues, etc. We encode Ruby’s arrays as a Rosette struct composed of a fixed-size vector and an integer representing the current size of the Ruby array. Because we used fixed-size vectors, we can only perform bounded verification over arrays. On the other hand, we avoid the need for loop invariants for iterators and reasoning over array operations can be more efficient.

### 2.4.5 Verification of $\lambda^{RB}$

We define a verification algorithm $\text{RTR}^\lambda$ that, given a $\lambda^{RB}$ program $P$, checks if all the definitions satisfy their specifications. The pseudo-code for this algorithm is shown below:

```python
def RTR^\lambda(P)
    (F, U, M) := guess(P)
    for (f ∈ F): add field $f$ to object struct
    for (u ∈ U): define uninterpreted function $u$
    $P \rightarrow Q \rightarrow R$
    return if (valid(R)) then SAFE else UNSAFE
end
```

First, we **guess** the proper translation environments. In practice (as discussed in § 2.4.2), we use the translation of $P$ to generate the minimum environments for which translation of $P$ succeeds. We define an object struct in Rosette containing
one field for each member of $\mathcal{F}$, and we define an uninterpreted function for each method in $\mathcal{U}$. Next, we translate $P$ to a $\lambda^I$ program $Q$ via $P \rightsquigarrow Q$ (§ 2.4.2) and $Q$ to a the Rosette program $R$, via $Q \rightarrow R$ (§ 2.4.3). Finally, we run the Rosette program $R$. The initial program $P$ is safe, i.e., no refinement type specifications are violated, if and only if the Rosette program $R$ is valid, i.e., all the verify queries are valid.

We conclude this section with a discussion of the RTR$^\lambda$ verifier.

RTR$^\lambda$ is Partial. There exist expressions of $\lambda^{RB}$ that fail to translate into a $\lambda^I$ expression. The translation requires at each method call $e.F.m(\overline{e})$ that the receiver has a class type $A$. There are two cases where this requirement fails: (1) $e.F$ has a union type or (2) type checking fails and so $e.F$ has no type. In our implementation (§ 2.5), we extend the translation to handle the first two cases. Handling for (1) is outlined in § 2.2.4. Case (2) can be caused by either a type error in the program or a lack of typing information for the type checker. Translation cannot proceed in either case.

RTR$^\lambda$ may Diverge. The translation to Rosette always terminates. All translation rules are inductively defined: they only recurse on syntactically smaller expressions or programs. Also, since the input program is finite, the minimum global environments required for translation are also finite. Finally, all the helper functions (including the type checking typeOf($\cdot$)) do terminate.

Yet, verification may diverge, as the execution of the Rosette program may
diverge. Specifications can encode arbitrary expressions, thus it is possible to encode undecidable verification queries. Consider, for instance, the following contrived Rosette program in which we attempt to verify an assertion over a recursive method:

\[
\begin{align*}
\text{(define (rec x) (rec x))} \\
\text{(define --symbolic b boolean?)} \\
\text{(verify (rec b))}
\end{align*}
\]

Rosette attempts to symbolically evaluate this program, and thus diverges.

**RTR\(^\lambda\) is Incomplete.** Verification is incomplete and its precision relies on the precision of the specifications. For instance, if a pure method \(A.m\) is marked as impure, the verifier will not prove the congruence axiom.

**RTR\(^\lambda\) is Sound.** If the verifier decides that the input program is safe, then all definitions satisfy their specifications, assuming that (1) all the refinements are pure boolean expressions and (2) all the labels are sound (i.e., methods match the specifications implied by the labels). The assumption (1) is required since verification under diverging (let alone effectful) specifications is difficult [32]. The assumption (2) is required since our translation encodes pure methods as uninterpreted functions, while for the impure methods it havoc only the unprotected arguments.

### 2.5 Evaluation

We implemented the Ruby refinement type checker RTR\(^\lambda\) by extending RDL with refinement types. Table 2.1 summarizes the evaluation of RTR.

\[^{1}\text{Code available at: https://github.com/mckaz/vmcai-rdl}\]
Benchmarks. We evaluate RTR on six popular Ruby libraries:

- *Money* [39] performs currency conversions over monetary quantities and relies on mixin methods,

- *BusinessTime* [40] performs time calculations in business hours and days,

- *Unitwise* [41] performs various unit conversions,

- *Geokit* [42] performs calculations over locations on Earth,

- *Boxroom* [43] is a Rails app for sharing files in a web browser and uses metaprogramming, and

- *Matrix* [44] is a Ruby standard library for matrix operations.

For verification, we forked the original Ruby libraries and provided manually written method specifications in the form of refinement types. The forked repositories are publicly available [45]. Experiments were conducted on a machine with a 3 GHz Intel Core i7 processor and 16 GB of memory.

We chose these libraries because they combine Ruby-specific features challenging for verification, like metaprogramming and mixins, with arithmetic-heavy operations. In all libraries we verify both (1) *functional correctness of arithmetic operations* (e.g., no division-by-zero, the absolute value of a number should not be negative) and (2) *data-specific arithmetic invariants* (e.g., integers representing months should always be in the range from 1 to 12 and a *data* value added to an aggregate should always fall between maintained @min and @max fields). In the *Matrix* library, we verify a matrix multiplication method, checking that multiplying a
matrix with \( r \) rows by a matrix with \( c \) columns yields a matrix of size \( r \times c \). Note this method makes extensive use of array operations, since matrices are implemented as an array of arrays.

**Quantitative Evaluation.** Table 2.1 summarizes our evaluation quantitatively. For each application, we list every verified Method. In our experiment, we focused on methods with interesting arithmetic properties.

The Ruby LoC column gives the size of the verified Ruby program. This metric includes the lines of all methods and annotations that were used to verify the method in question. For each verified method, RTR generates a separate Rosette program. We give the sizes of these resulting programs in the Rosette LoC column. Unsurprisingly, the LoC of the Rosette program increases with the size of the source Ruby program.

We present the median (Time(s)) and semi-interquartile range (SIQR) of the Verification Time required to verify all methods for an application over 11 runs. For each verified method, the SIQR was at most 2% of the verification time, indicating relatively little variance in the verification time. Overall, verification was fast, as might be expected for relatively small methods. The one exception was matrix multiplication. In this case, the slowdown was due to the extensive use of array operations mentioned above. We bounded array size (see § 2.4.4) at 10 for the evaluations. For symbolic arrays, this means Rosette must reason about every possible size of an array up to 10. This burden is exacerbated by matrix multiplication’s use of two symbolic two-dimensional arrays.
<table>
<thead>
<tr>
<th>Method</th>
<th>Ru-LoC</th>
<th>Ro-LoC</th>
<th>Spec</th>
<th>Verification Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Time(s)</td>
</tr>
<tr>
<td><strong>Money</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Money::Arithmetic#-@</td>
<td>7</td>
<td>29</td>
<td>4</td>
<td>5.69</td>
</tr>
<tr>
<td>Money::Arithmetic#eql?</td>
<td>11</td>
<td>40</td>
<td>3</td>
<td>5.74</td>
</tr>
<tr>
<td>Money::Arithmetic#positive?</td>
<td>5</td>
<td>24</td>
<td>3</td>
<td>5.40</td>
</tr>
<tr>
<td>Money::Arithmetic#negative?</td>
<td>5</td>
<td>24</td>
<td>2</td>
<td>5.42</td>
</tr>
<tr>
<td>Money::Arithmetic#abs</td>
<td>5</td>
<td>30</td>
<td>4</td>
<td>5.49</td>
</tr>
<tr>
<td>Money::Arithmetic#zero?</td>
<td>5</td>
<td>26</td>
<td>2</td>
<td>5.38</td>
</tr>
<tr>
<td>Money::Arithmetic#nonzero?</td>
<td>5</td>
<td>24</td>
<td>2</td>
<td>5.43</td>
</tr>
<tr>
<td><strong>App Total</strong></td>
<td>43</td>
<td>197</td>
<td>10</td>
<td>38.56</td>
</tr>
<tr>
<td><strong>BusinessTime</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParsedTime#-</td>
<td>10</td>
<td>58</td>
<td>8</td>
<td>6.28</td>
</tr>
<tr>
<td>BusinessHours#initialize</td>
<td>5</td>
<td>26</td>
<td>2</td>
<td>5.36</td>
</tr>
<tr>
<td>BusinessHours#non_negative_hours?</td>
<td>5</td>
<td>26</td>
<td>2</td>
<td>5.4</td>
</tr>
<tr>
<td>Date#week</td>
<td>7</td>
<td>32</td>
<td>2</td>
<td>5.53</td>
</tr>
<tr>
<td>Date#quarter</td>
<td>5</td>
<td>28</td>
<td>2</td>
<td>5.47</td>
</tr>
<tr>
<td>Date#fiscal_month_offset</td>
<td>5</td>
<td>25</td>
<td>2</td>
<td>5.41</td>
</tr>
<tr>
<td>Date#fiscal_year_week</td>
<td>7</td>
<td>33</td>
<td>2</td>
<td>5.53</td>
</tr>
<tr>
<td>Date#fiscal_year_month</td>
<td>12</td>
<td>35</td>
<td>3</td>
<td>5.65</td>
</tr>
<tr>
<td>Date#fiscal_year_quarter</td>
<td>9</td>
<td>42</td>
<td>2</td>
<td>5.72</td>
</tr>
<tr>
<td>Date#fiscal_year</td>
<td>11</td>
<td>32</td>
<td>4</td>
<td>5.81</td>
</tr>
<tr>
<td><strong>App Total</strong></td>
<td>76</td>
<td>337</td>
<td>24</td>
<td>56.15</td>
</tr>
<tr>
<td><strong>Unitwise</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unitwise::Functional.to_cel</td>
<td>4</td>
<td>25</td>
<td>2</td>
<td>5.42</td>
</tr>
<tr>
<td>Unitwise::Functional.from_cel</td>
<td>4</td>
<td>25</td>
<td>2</td>
<td>5.44</td>
</tr>
<tr>
<td>Unitwise::Functional.to_degf</td>
<td>4</td>
<td>22</td>
<td>1</td>
<td>5.41</td>
</tr>
<tr>
<td>Unitwise::Functional.from_degf</td>
<td>4</td>
<td>27</td>
<td>2</td>
<td>5.44</td>
</tr>
<tr>
<td>Unitwise::Functional.to_degre</td>
<td>4</td>
<td>27</td>
<td>2</td>
<td>5.44</td>
</tr>
<tr>
<td>Unitwise::Functional.from_degre</td>
<td>4</td>
<td>27</td>
<td>2</td>
<td>5.42</td>
</tr>
<tr>
<td><strong>App Total</strong></td>
<td>24</td>
<td>153</td>
<td>6</td>
<td>32.55</td>
</tr>
<tr>
<td><strong>Geokit</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geokit::Bounds#center</td>
<td>7</td>
<td>31</td>
<td>4</td>
<td>5.4</td>
</tr>
<tr>
<td>Geokit::Bounds#crosses_meridian?</td>
<td>7</td>
<td>35</td>
<td>6</td>
<td>5.59</td>
</tr>
<tr>
<td>Geokit::Bounds#==</td>
<td>9</td>
<td>60</td>
<td>5</td>
<td>5.97</td>
</tr>
<tr>
<td>Geokit::GeoLoc#province</td>
<td>5</td>
<td>26</td>
<td>2</td>
<td>5.52</td>
</tr>
<tr>
<td>Geokit::GeoLoc#success?</td>
<td>5</td>
<td>26</td>
<td>2</td>
<td>5.51</td>
</tr>
<tr>
<td>Geokit::Polygon#contains?</td>
<td>26</td>
<td>68</td>
<td>10</td>
<td>10.8</td>
</tr>
<tr>
<td><strong>App Total</strong></td>
<td>59</td>
<td>246</td>
<td>21</td>
<td>38.80</td>
</tr>
<tr>
<td><strong>Boxroom</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UserFile#move</td>
<td>12</td>
<td>34</td>
<td>3</td>
<td>5.57</td>
</tr>
<tr>
<td><strong>Matrix</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix.*</td>
<td>57</td>
<td>94</td>
<td>9</td>
<td>334.35</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>271</td>
<td>1061</td>
<td>73</td>
<td>505.98</td>
</tr>
</tbody>
</table>

Table 2.1: Method gives the class and name of the method verified. Ru-LoC and Ro-LoC give number of LoC for a Ruby method and the translated Rosette program. Spec is the number of method and variable type annotations we had to write. Verification Time is the median and semi-interquartile range of the time in seconds over 11 runs. App Total rows list the totals for an app, without double counting the same specs.
Finally, Table 2.1 lists the number of type specifications required to verify each method. These are comprised of method type annotations, including the refinement type annotations for the verified methods themselves, and variable type annotations for instance variables. Note that we do not quantify the number of type annotations used for Ruby’s core and standard libraries, since these are included in RDL.

We observe that there is significant variation in the number of annotations required for each application. For example, Unitwise required 6 annotations to verify 6 methods, while Geokit required 21 annotations for 6 methods. The differences are due to code variations: To verify a method, the programmer needs to give a refinement type for the method plus a type for each instance variable used by the method and for each (non-standard/core library) method called by the method.

**Case Study.** Next we illustrate the RTR verification process by presenting the exact steps required to specify and check the properties of a method from an existing Ruby library. For this example, we chose to verify the « method of the Aggregate library [46], a Ruby library for aggregating and performing statistical computations over some numeric data. The method « takes one input, data, and adds it to the aggregate by updating (1) the minimum @min and maximum @max of the aggregate, (2) the count @count, sum @sum, and sum of squares @sum2 of the aggregate, and finally (3) the correct bucket in @buckets.

```ruby
def «(data)
    if 0 == @count
        @min = data ; @max = data
    else
```
\begin{verbatim}
@max = data if data > @max ; @min = data if data < @min
end
@count += 1 ; @sum += data ; @sum2 += (data * data)
@buckets[to_index(data)] += 1 unless outlier?(data)
end

We specify functional correctness of the method \( \text{«} \) by providing a refinement type specification that declares that after the method is executed, the input \text{data} will fall between \text{@min} and \text{@max}.

type :«, \( \langle \text{Integer data} \rangle \to \text{Integer} \{ \text{@min} \leq \text{data} \leq \text{@max} \} \), verify: :bind

Here, the symbol :bind is an arbitrary label. To verify the specification, we load the library and call the verifier with this label:

\text{rdl\_do\_verify :bind}

RTR proceeds with verification in three steps:

- first use \text{RDL} to type check the basic types of the method,
- then translate the method to \text{Rosette} (using the translation of § 2.4), and
- finally run the \text{Rosette} program to check the validity of the specification.

Initially, verification fails in the first step with the error

error: no type for instance variable \text{`@count'}

To fix this error, the user needs to provide the correct types for the instance variables using the following type annotations.

\text{var\_type :@count, `Integer'}
\text{var\_type :@min, :@max, :@sum, :@sum2, `Float'}
\text{var\_type :@buckets, `Array<Integer>}'
The « method also calls two methods that are not from Ruby’s standard and core libraries: to_index, which takes a numeric input and determines the index of the bucket the input falls in, and outlier?, which determines if the given data is an outlier based on provided specifications from the programmer. These methods are challenging to verify. For example, the to_index method makes use of non-linear arithmetic in the form of logarithms, and it includes a loop. Yet, neither of the calls to_index or outlier? should affect verification of the specification of «. So, it suffices to provide type annotations with a pure label, indicating we want to use uninterpreted functions to represent them:

```plaintext
type :outlier?, '(Float i) → Bool b', :pure
type :to_index, '(Float i) → Integer out', :pure
```

Given these annotations, the verifier has enough information to prove the postcondition on «, and it will return the following message to the user:

Aggregate instance method « is safe.

When verification fails, an unsafe message is provided, combined with a counterexample consisting of bindings to symbolic values that causes the postcondition to fail. For instance, if the programmer incorrectly specified that data is less than the @min, i.e.,

```plaintext
type :«, '(Integer data) → Integer { data < @min }'
```

Then RTR would return the following message:

Aggregate instance method « is unsafe.
Counterexample: (model [real_data 0][real_@min 0] ...)

This gives a binding to symbolic values in the translated Rosette program which
would cause the specification to fail. We only show the bindings relevant to the specification here: when \texttt{real\_data} and \texttt{real\_omin}, the symbolic values corresponding to \texttt{data} and \texttt{omin} respectively, are both 0, the specification fails.

2.6 Related Work

**Verification for Ruby on Rails.** Several prior systems can verify properties of Rails apps. *Space* [47] detects security bugs in Rails apps by using symbolic execution to generate a model of data exposures in the app and reporting a bug if the model does not match common access control patterns. Bocić and Bultan propose *symbolic model extraction* [48], which extracts models from Rails apps at runtime, to handle metaprogramming. The generated models are then used to verify data integrity and access control properties. *Rubicon* [49] allows programmers to write specifications using a domain-specific language that looks similar to Rails tests, but with the ability to quantify over objects, and then checks such specifications with bounded verification. *Rubyx* [50] likewise allows programmers to write their own specifications over Rails apps and uses symbolic execution to verify these specifications.

In contrast to RTR, all of these tools are specific to Rails and do not apply to general Ruby programs, and the first two systems do not allow programmers to specify their own properties to be verified.

**Rosette.** Rosette has been used to help establish the security and reliability of several real-world software systems. Pernsteiner et al. [51] use Rosette to build a
verifier to study the safety of the software on a radiotherapy machine. *Bagpipe* [52] builds a verifier using Rosette to analyze the routing protocols used by Internet Service Providers (ISPs). These results show that Rosette can be applied in a variety of domains.

**Types For Dynamic Languages.** There have been a number of efforts to bring type systems to dynamic languages including Python [11, 12], Racket [3, 9], and JavaScript [16, 53, 54], among others. However, these systems do not support refinement types.

Some systems have been developed to introduce refinement types to scripting and dynamic languages. *Refined TypeScript* (RSC) [29] introduces refinement types to TypeScript [55, 56], a superset of JavaScript that includes optional static typing. RSC uses the framework of Liquid Types [57] to achieve refinement inference. Refinement types have been introduced [10] to Typed Racket as well. As far as we are aware, these systems do not support mixins or metaprogramming.

**General Purpose Verification.** Dafny [37] is an object-oriented language with built-in constructs for high-level specification and verification. While it does not explicitly include refinement types, the ability to specify a method’s type and pre- and postconditions effectively achieves the same level of expressiveness. Dafny also performs modular verification by using a method’s pre- and postconditions and labels indicating its purity or arguments mutated, an approach RTR largely emulates. However, unlike Dafny, RTR leaves this modular treatment of methods as an option.
for the programmer. Furthermore, unlike RTR, Dafny does not include features such as mixins and metaprogramming.

2.7 Conclusion

We formalized and implemented RTR, a refinement type checker for Ruby programs using assume-guarantee reasoning and the just-in-time checking technique of RDL. Verification at runtime naturally adjusts standard refinement types to handle Ruby’s dynamic features, such as metaprogramming and mixins. To evaluate our technique, we used RTR to verify numeric properties on six commonly used Ruby and Ruby on Rails applications, by adding refinement type specifications to the existing method definitions. We found that verifying these methods took a reasonable runtime and annotation burden, and thus we believe RTR is a promising first step towards bringing verification to Ruby.

Our work opens new directions for further Ruby verification. One potential area for future work is verification of purity and immutability labels, which are currently trusted by RTR. Another area would be refinement type inference by adapting Hindley-Milner and liquid typing [57] to the RDL setting, and by exploring whether Rosette’s synthesis constructs could be used for refinement inference. There is also potential for extending the expressiveness of RTR by adding support for loop invariants and dynamically defined instance variables, among other Ruby constructs. Finally, as Ruby is commonly used in the Ruby on Rails framework, it would be interesting extend RTR with modeling for web-specific constructs such as access con-
trol protocols and database operations to further support verification in the domain of web applications.
Chapter 3: Type-Level Computations for Ruby Libraries

The previous chapter introduced RTR, a system that extends RDL with refinement types. While RTR is useful for enforcing strong correctness properties of programs, there are still many commonly used constructs in Ruby programs that are out of RDL’s reach, even for basic type checking. For example, database queries, as well as operations over heterogeneous data structures such as arrays and hashes, can be difficult to tame with static types. In this chapter, we introduce CompRDL, a system that adds type-level computations to RDL. This greatly increases the expressiveness of RDL’s types, and it allows us to type check many common operations that were previously out of reach.

3.1 Introduction

While there is a large body of research on adding static typing to dynamic languages [3, 4, 5, 8, 9, 11, 12, 16, 53, 54], existing systems have limited support for the case when types depend on values. Yet this case occurs surprisingly often, especially in Ruby libraries. For example, consider the following database query, written for a hypothetical Rails app:

```ruby
Person.joins(:apartments).where({'name': 'Alice', age: 30, apartments: {bedrooms: 2}})
```
This query uses the *ActiveRecord* DSL to join two database tables, `people`\(^1\) and `apartments`, and then filter on the values of various columns (name, age, bedrooms) in the result.

We would like to type check such code, e.g., to ensure the columns exist and the values being matched are of the right types. But we face an important problem: what type signature do we give `joins`? Its return type—which should describe the joined table—depends on the value of its argument. Moreover, for \(n\) tables, there are \(n^2\) ways to join two of them, \(n^3\) ways to join three of them, etc. Enumerating all these combinations is impractical.

To address this problem, in this chapter we introduce *CompRDL*, which extends *RDL* to include method types with *type-level computations*, henceforth referred to as *comp types*. More specifically, in *CompRDL* we can annotate library methods with type signatures in which Ruby expressions can appear as types. During type checking, those expressions are evaluated to produce the actual type signature, and then typing proceeds as usual. For example, for the call to `Person.joins`, by using a singleton type for `:apartments`, a type-level computation can look up the database schemas for the receiver and argument and then construct an appropriate return type.\(^2\)

Moreover, the same type signature can work for any model class and any combination of joins. And, because *CompRDL* allows arbitrary computation in types, *CompRDL* type signatures have access to the full, highly dynamic Ruby environment.

\(^1\)Rails knows the plural of person is people.
\(^2\)The use of type-level computations and singleton types could be considered dependent typing, but as our type system is much more restricted we introduce new terminology to avoid confusion (see § 3.2.4 for discussion).
This allows us to provide very precise types for the large set of Rails database query methods. It also lets us give precise types to methods of finite hash types (heterogeneous hashes), tuple types (heterogeneous arrays), and const string types (immutable strings), which can help eliminate type casts that would otherwise be required.

Note that in all these cases, we apply comp types to library methods whose bodies we do not type check, in part to avoid complex, potentially undecidable reasoning about whether a method body matches a comp type, but more practically because those library methods are either implemented in native code (hashes, arrays, strings) or are complex (database queries). This design choice makes CompRDL a particularly practical system which we can apply to real-world programs. To maintain soundness, we insert dynamic checks to ensure that these methods abide by their computed types at runtime. (§ 2.2 gives an overview of typing in CompRDL.)

We introduce $\lambda^C$, a core, object-oriented language that formalizes CompRDL type checking. In $\lambda^C$, library methods can be declared with signatures of the form $(a:e_1/A_1) \rightarrow e_2/A_2$, where $A_1$ and $A_2$ are the conventional (likely overapproximate) argument and return types of the method. The precise argument and return types are determined by evaluating $e_1$ and $e_2$, respectively, and that evaluation may refer to the type of the receiver and the type $a$ of the argument. $\lambda^C$ also performs type checking on $e_1$ and $e_2$, to ensure they do not go wrong. To avoid potential infinite recursion, $\lambda^C$ does not use type-level computations during this type checking process, instead using the conventional types for library methods. Finally, $\lambda^C$ includes a rewriting step to insert dynamic checks to ensure library methods abide by their
computed types. We prove λC’s type system is sound. (See § 2.4 for our formalism.)

We implemented CompRDL on top of RDL (§ 1.1). Since CompRDL can include type-level computation that relies on mutable values, CompRDL inserts additional runtime checks to ensure such computations evaluate to the same result at method call time as they did at type checking time. Additionally, CompRDL uses a lightweight analysis to check that type-level computations (and thus type checking) terminate. The termination analysis uses purity effects to check that calls that invoke iterator methods—the main source of looping in Ruby, in our experience—do not mutate the receiver, which could introduce non-termination. Finally, we found that several kinds of comp types we developed needed to include weak type updates to handle mutation in Ruby programs. (§ 3.4 describes our implementation in more detail.)

We evaluated CompRDL by first using it to write type annotations for 482 Ruby core library methods and 104 Rails database query methods. We found that by using helper methods, we could write very precise type annotations for all 586 methods with just a few lines of code on average. Then, we used those annotations to type check 132 methods across two Ruby APIs and four Ruby on Rails web apps. We were able to successfully type check all these methods in approximately 15 seconds total. In doing so, we also found two type errors and a documentation error, which we confirmed with the developers. We also found that, with comp types, type checking these benchmarks required 4.75× fewer type cast annotations compared to standard types, demonstrating comp types’ increased precision. (§ 3.5 contains the results of our evaluation.)

Our results suggest that using type-level computations provides a powerful,
practical, and precise way to statically type check code written in dynamic languages.

3.2 Overview

We begin with a demonstration of comp types, by showing how they can be used to check the type correctness of Rails database queries (§ 3.2.1). We then show how comp types can reduce the need for type casts (§ 3.2.2) and how they can be used to type check some queries that incorporate raw SQL (§ 3.2.3). We conclude with a brief discussion (§ 3.2.4), including its use of dynamic checks, its lightweight termination checker, and its ability to encode constant folding.

3.2.1 Typing Ruby Database Queries

While RDL’s type system is powerful enough to type check Rails apps in general, it is actually very imprecise when reasoning about database (DB) queries. For example, consider Figure 3.1, which shows some code from the Discourse app. Among others, this app uses two tables, users and emails, whose schemas are shown on lines 2 and 3. Each user has an id, a username, and a flag indicating whether the account was staged. Such staged accounts were created automatically by Discourse and can be claimed by the email address owner. An email has an id, the email address, and the user_id of the user who owns the email address.

Next in the figure, we show code for the class User, which is a model, i.e., instances of the class correspond to rows in the users table. This class has one
Figure 3.1: *Discourse* code that uses *ActiveRecord* to query a database.

Figure 3.2: Comp type annotations for query methods.
method, available?, which returns a boolean indicating whether the username and email address passed as arguments are available. The method first checks whether the username was already reserved (line 8, note the postfix if). If not, it uses the database query method exists? to see if the username was already taken (line 9). (Note that in Ruby, {a: b} is a hash that maps the symbol :a, which is suffixed with a colon when used as a key, to the value b.) Otherwise, line 11 uses a more complex query to check whether an account was staged. More specifically, this code joins the users and emails table and then looks for a match across the joined tables.

We would like to type check the exists? calls in this code to ensure they are type correct, meaning that the columns they refer to exist and the values being matched are of the right type. The call on line 9 is easy to check, as RDL can type the receiver User as having an exists? method that takes a particular finite hash type {c1: t1, ..., cn: tn} as an argument, where the ci are singleton types for symbols naming the columns, and the ti are the corresponding column types.

Unfortunately, the exists? call on line 11 is another story. Notice that this query calls exists? on the result of User.joins(:emails). Thus, to give exists? a type with the right column information, we need to have that information reflected in the return type of joins. Unfortunately, there is no reasonable way to do this in RDL, because the set of columns in the table returned by joins depends on both the receiver and the value of the argument. We could in theory overload joins with different return types depending on the argument type—e.g., we could say that User.joins returns a certain type when the argument has singleton type :emails. However, we would need to generate such signatures for every possible way of joining two tables.
together, three tables together, etc., which quickly blows up. Thus, currently, RDL
types this particular `exists?` call as taking a `Hash<Symbol, Object>`, which would
allow type-incorrect arguments.

**Comp types for DB Queries.** To address this problem, CompRDL allows method
type signatures to include computations that can, on-the-fly, determine the method’s
type. Figure 3.2 gives comp type signatures for `exists?` and `joins`. It also shows the
definition of a helper method, `schema_type`, that is called from the comp types.
The comp types also make use of a new generic type `Table<T>` to type a DB table
whose columns are described by `T`, which should be a finite hash type.

Line 1 gives the type of `exists?`. Its argument is a comp type, which is a Ruby
expression, delimited by « », that evaluates to a standard type. When type checking
a call to `exists?` (including those in the body of `available?`), CompRDL runs the comp
type code to yield a standard type, and then proceeds with type checking as usual
with that type.

In this case, to compute the argument type for `exists?`, we call the helper
method `schema_type` with `tself`, which is a reserved variable naming the type of
the receiver. The `schema_type` method has a few different behaviors depending
on its argument. When given a type `Table<T>`, it returns `T`, i.e., the finite hash
type describing the columns. When given a singleton type representing a class or a
symbol, it uses another helper method `RDL.db_schema` (not shown) to look up the
corresponding table’s schema and return an appropriate finite hash type. Given any
other type, `schema_type` falls back to returning the type `Hash<Symbol, Object>`.
This type signature already allows us to type check the \texttt{exists?} call on line 9. On this line, the receiver has the singleton type for the \texttt{User} class, so \texttt{schema\_type} will use the second arm of the conditional and look up the schema for \texttt{User} in the DB.

Line 2 shows the comp type signature for \texttt{joins}. The signature’s input type binds \texttt{t} to the actual argument type, and requires it to be a subtype of \texttt{Symbol}. For example, for the call on line 11, \texttt{t} will be bound to the singleton type for \texttt{:emails}. The return comp type can then refer to \texttt{t}. Here, if \texttt{t} is a singleton type, \texttt{joins} returns a new \texttt{Table} type that merges the schemas of the receiver and the argument tables using \texttt{schema\_type}. Otherwise, it falls back to producing a \texttt{Table} with no schema information. Thus, the \texttt{joins} call on line 11 returns type

\[
\text{Table\<\{staged:%bool, username:String, id: Integer,}
\]
\[
\text{emails: \{email:String, user\_id: Integer \}}\>
\]

That is, the type reflects the schemas of both the \texttt{users} and \texttt{emails} tables. Given this type, we can now type check the \texttt{exists?} call on line 11 precisely. On this line, the receiver has the table type given above, so when called by \texttt{exists?} the helper \texttt{schema\_type} will use the first arm of the conditional and return the \texttt{Table} column types, ensuring the query is type checked precisely.

Though we have only shown types for two query methods in the figure, we note that comp types are easily extensible to other kinds of queries. Indeed, we have applied them to 104 methods across two DB query frameworks (§ 3.5). Furthermore, we can also use comp types to encode sophisticated invariants. For example, in
Rails, database tables can only be joined if the corresponding classes have a declared association. We can write a comp type for joins that enforces this. (We omitted this in Figure 3.2 for brevity.)

Finally, we note that while we include a “fallback” case that allows comp types to default to less precise types when necessary, in practice this is rarely necessary for DB queries. That is, parameters that are important for type checking, such as the name of tables being queried or joined, or the names of columns be queried, are almost always provided statically in the code.

3.2.2 Avoiding Casts using Comp Types

In addition to letting us find type errors in code we could not previously type check precisely enough, the increased precision of comp types can also help eliminate type casts.

For example, consider the code in Figure 3.3. The first line gives the type signature for a method of Hash, which is parameterized by a key type k and a value type v (declarations of the parameters not shown). The specific method is
which, given a key, returns the corresponding value. Notably, the form \( x[k] \) is desugared to \( x.[]((k)) \), and thus hash lookup, array index, and so forth are methods rather than built-in language constructs.

The second line similarly gives a type for \texttt{Array\#first}, which returns the first element of the array. Here type variable \( a \) is the array’s contents type (declaration also not shown). The third line gives a type for a method \texttt{page} of the current class, which takes no arguments and returns a hash in which \texttt{:info} is mapped to an \texttt{Array<String>} and \texttt{:title} is mapped to a \texttt{String}.

Now consider type checking the \texttt{image_url} method defined at the bottom of the figure. This code is extracted and simplified from a \textit{Wikipedia} client library used in our experiments (§ 3.5). Here, since \texttt{page} is a no-argument method, it can be invoked without any parentheses. We then invoke \texttt{Hash\#[]} on the result.

Unfortunately, at this point type checking loses precision. The problem is that whenever a method is invoked on a finite hash type \( \{c1: t1, ..., cn: tn\} \), RDL (retroactively) gives up tracking the type precisely and promotes it to \texttt{Hash<Symbol, t1 or...or tn>} [20]. In this case, \texttt{page}’s return type is promoted to \texttt{Hash<Symbol, Array<String> or String>}.

Now the type checker gets stuck. It reasons that \texttt{first} could be invoked on an array or a string, but \texttt{first} is defined only for the former and not the latter. The only currently available fix is to insert a type cast, as shown in the comment on line 8.

One possible solution would be to add special-case support for \texttt{[]} on finite hash types. However, this is only one of 54 methods of \texttt{Hash}, which is a lot of behavior.

\footnote{Here we use the Ruby idiom that \texttt{A#m} refers to the instance method \texttt{m} of class \texttt{A}.}
to special-case. Moreover, Ruby programs can *monkey patch* any class, including *Hash*, to change library methods’ behaviors. This makes building special support for those methods inelegant and potentially brittle since the programmer would have no way to adjust the typing of those methods.

In CompRDL, we can solve this problem with a comp type annotation. More specifically, we can give `Hash[]` the following type:

```ruby
type Hash, :[], "(t<:Object) → «if tself.is_a? (FiniteHash) ∧ t.is_a?(Singleton) then tself.elts[t.val] else tself.value_type end»"
```

This comp type specifies that if the receiver has a finite hash type and the key has a singleton type, then `Hash[]` returns the type corresponding to the key, otherwise it returns a value type covering all possible values (computed by `value_type`, definition not shown).

Notice that this signature allows `image_url` to type check without any additional casts. The same idea can be applied to many other *Hash* methods to give them more precise types.

**Tuple Types.** In addition to finite hash types, RDL has a special *tuple type* to model heterogeneous *Arrays*. As with finite hash types, RDL does not special-case the *Array* methods for tuples, since there are 124 of them. This leads to a loss of precision when invoking methods on values with tuple types. However, analogously to finite hash tables, comp types can be used to recover precision. As examples, the *Array#first* method can be given a comp type which returns the type of the first
element of a tuple, and the comp type for `Array[]` has essentially the same logic as `Hash[]`.

**Const String Types.** As another example, Ruby strings are mutable, hence RDL does not give them singleton types. (In contrast, Ruby symbols are immutable.) This is problematic, because types might depend on string values. In particular, in the next section we explore reasoning about string values during type checking raw SQL queries.

Using comp types, we can assign singleton types to strings wherever possible. We introduce a new `const string` type representing strings that are never written to. CompRDL treats const strings as singletons, and methods on `String` are given comp types that perform precise operations on const strings and fall back to the `String` type as needed. We discuss handling mutation for const strings, finite hashes, and tuples in Section 3.4.

3.2.3 SQL Type Checking

As we saw in Figure 3.1, **ActiveRecord** uses a DSL that makes it easier to construct queries inside of Ruby. However, sometimes programmers need to include raw SQL in their queries, either to access a feature not supported by the DSL or to improve performance compared to the DSL-generated query.

Figure 3.4 gives one such example, extracted and simplified from **Discourse**, one of our subject programs. Here there are three relevant tables: `posts`, which stores posted messages; `topics`, which stores the topics of posts; and `topic_allowed_groups`,
### Table Schema

- posts table { id: Integer, topic_id: Integer, ... }
- topics table { id: Integer, title: String, ... }
- topic_allowed_groups table { group_id: Integer, topic_id: Integer }

### Query with SQL strings

```ruby
Post.includes(:topic)
    .where('topics.title IN (SELECT topic_id FROM topic_allowed_groups WHERE 'group_id' = ?)', self.id)
```

Figure 3.4: Type Checking SQL Strings in Discourse.

which is used to limit the topics allowed by certain user groups.

Line 7 shows a query that includes raw SQL. First, the posts and topics tables are joined via the includes method. (This method does eager loading whereas joins does lazy loading.) Then where filters the resulting table based on some conditions. In this case, the conditions involve a nested SQL query, which cannot be expressed except using raw SQL that will be inserted into the final generated query.

This example also shows another feature: any ?'s that appear in raw SQL are replaced by additional arguments to where. In this case, the ? will be replaced by self.id.

We would like to extend type checking to also reason about the raw SQL strings in queries, since they may have errors. In this particular example, we have injected a bug. The inner SELECT returns a set of integers, but topics.title is a string, and it is a type error to search for a string in an integer set.

To find this bug, we developed a simple type checker for a subset of SQL, and
we wrote a comp type for where that invokes it as shown on line 10. In particular, if the type of the argument to where, here referred to by t, is a const string, then we type check that string as raw SQL, and otherwise we compute the valid parameters of where using the schema_type method from Figure 3.2. The result of where has the same type as the receiver.

The sql_typecheck method (not shown) takes the receiver type, which will be a Table with a type parameter describing the schema, and the SQL string. One challenge that arises in type checking the SQL string is that it is actually only a fragment of a query, which therefore cannot be directly parsed using a standard SQL parser. We solve this problem by creating a complete, but artificial, SQL query into which we inject the fragment. This query is never run, but it is syntactically correct so it can be parsed. Then, we replace any ?’s with placeholder AST nodes that store the types of the corresponding arguments.

For example, the raw SQL in Figure 3.4 gets translated to the following SQL query:

```sql
SELECT * FROM 'posts' INNER JOIN 'topics'
ON a.id = b.a_id
WHERE topics.title IN (SELECT topic_id FROM topic_allowed_groups WHERE 'group_id' = [Integer])
```

Notice the table names (posts, topics) occur on the first line and the ? has been replaced by a placeholder indicating the type Integer of the argument. Also note that the column names to join on (which are arbitrary here) are ignored by our type checker, which currently only looks for errors in the where clause.

Once we have a query that can be parsed, we can type check it using the DB
schema. In this case, the type mismatch between `topics.title` and the inner query will be reported.

In § 3.2.1, comp types were evaluated to produce a normal type signature. However, we use comp types in a slightly different way for checking SQL strings. The `sql_typecheck` method will itself perform type checking and provide a detailed message when an error is found. If no error is found, `sql_typecheck` will simply return the type `String`, allowing type checking to proceed.

### 3.2.4 Discussion

Now that we have seen `CompRDL` in some detail, we can discuss several parts of its design.

**Dynamic Checks.** In type systems with type-level computations, or more generally dependent type systems, comparing two types for equality is often undecidable, since it requires checking if computations are equivalent.

To avoid this problem, `CompRDL` only uses comp types for methods which themselves are not type checked. For example, `Hash#[]` is implemented in native code, and we have not attempted to type check `ActiveRecord`’s `joins` method, which is part of a very complex system.

As a result, type checking in `CompRDL` is decidable. Comp types are only used to type check method calls, meaning we will always have access to the types of the receiver and arguments in a method call. Additionally, in all cases we have encountered in practice, the types of the receiver and arguments are ground types (meaning
they do not contain type variables). Thus, comp types can be fully evaluated to non-comp types before proceeding to type checking.

For soundness, since we do not type check the bodies of comp type-annotated methods, CompRDL inserts dynamic checks at calls to such methods to ensure they match their computed types. For example, in Figure 3.3, CompRDL inserts a check that page[:info] returns an Array. This follows the approach of gradual [58] and hybrid [59] typing, in which dynamic checks guard statically unchecked code.

We should also note that although our focus is on applying comp types to libraries, they can be applied to any method at the cost of dynamic checks for that method rather than static checks. For example, they could be applied to a user-defined library wrapper.

**Termination.** A second issue for the decidability of comp types is that type-level computations could potentially not terminate. To avoid this possibility, we implement a termination checker for comp types. At a high level, CompRDL ensures termination by checking that iterators used by type-level code do not mutate their receivers and by forbidding type-level code from using looping constructs. We also assume there are no recursive method calls in type-level code. We discuss termination checking in more detail in § 3.4.

**Value Dependency.** We note that, unlike dependent types (e.g., Coq [60], Agda [61], F* [62]) where types depend directly on terms, in CompRDL types depend on the *types* of terms. For instance, in a comp type (t<:Object) -> tres the result type tres can
depend on the type \( t \) of the argument. Yet, since singleton types lift expressions into types, we could still use \texttt{CompRDL} to express some value dependencies in types in the style of dependent typing.

**Constant Folding.** Finally, in \texttt{RDL}, integers and floats have singleton types. Thus, we can use comp types to lift some arithmetic computations to the type level. For example, \texttt{CompRDL} can assign the expression \( 1+1 \) the type \texttt{Singleton(2)} instead of \texttt{Integer}. This effectively incorporates constant folding into the type checker.

While we did write such comp types for \texttt{Integer} and \texttt{Float} (see Table 3.1), we found that this precision was not useful, at least in our subject programs. The reason is that \texttt{RDL} only assigns singleton types to constants, and typically arithmetic methods are not applied to constant values. Thus, though we have written comp types for the \texttt{Integer} and \texttt{Float} libraries, we have yet to find a useful application for them in practice. We leave further exploration of this topic to future work.

### 3.3 Soundness of Comp Types

In this section we formalize \texttt{CompRDL} as \( \lambda^C \), a core object-oriented calculus that includes comp types for library methods. We first define the syntax and semantics of \( \lambda^C \) (§ 3.3.1), and then we formalize type checking (§ 3.3.2). The type checking process includes a rewriting step to insert dynamic checks to ensure library methods satisfy their type signatures. Finally, we prove type soundness (§ 3.3.3). For clarity of presentation, we leave the full formalism and proofs to Appendix A. Here we provide only the key details.
3.3.1 Syntax and Semantics

Figure 3.5 gives the syntax of $\lambda^C$. Values $v$ include $\text{nil}$, $\text{true}$, and $\text{false}$. To support comp types, class IDs $A$, which are the base types in $\lambda^C$, are also values.

We assume the set of class IDs includes several built-in classes: $\text{Nil}$, the class of $\text{nil}$; $\text{Obj}$, which is the root superclass; $\text{True}$ and $\text{False}$, which are the classes of $\text{true}$ and $\text{false}$, respectively, as well as their superclass $\text{Bool}$; and $\text{Type}$, the class of base types $A$.

Expressions $e$ include values $v$ and variables $x$ and $a$. By convention, we use the former in regular program expressions and the latter in comp types. The special variable $\text{self}$ names the receiver of a method call, and the special variable $\text{tself}$ names the type of the receiver in a comp type. New object instances are created with $A$.\text{new}$. Expressions also include sequences $e;e$, conditionals $\text{if } e \text{ then } e \text{ else } e$, and method calls $e.m(e)$, where, to simplify the formalism, methods take one argument.

Finally, our type system translates calls to library methods into checked method calls.
\( [A]e.m(e) \), which checks at run-time that the value returned from the call has type \( A \). We assume this form does not appear in the surface syntax.

We assume the classes form a lattice with \( \text{Nil} \) as the bottom and \( \text{Obj} \) as the top. We write the least upper bound of \( A_1 \) and \( A_2 \) as \( A_1 \sqcup A_2 \). For simplicity, we assume the lattice correctly models the program’s classes, i.e., if \( A \leq A' \), then \( A \) is a subclass of \( A' \) by the usual definition. Lastly, three of the built-in classes, \( \text{Nil}, \text{True}, \) and \( \text{False} \), are singleton types, i.e., they contain only the values \( \text{nil}, \text{true}, \) and \( \text{false} \), respectively. Extending \( \lambda^C \) with support for more kinds of singleton types is straightforward.

\textit{Method Types } \( \sigma \) are of the form \( A' \to A \) where \( A' \) and \( A \) are the domain and range types, respectively. \textit{Library Method Types} \( \delta \) are either method types or have the form \( (a<:e'\langle A' \rangle) \to e/A \), where \( e' \) and \( e \) are expressions that evaluate to types and that can refer to the variables \( a \) and \( \text{tself} \). The base types \( A' \) and \( A \) provide an upper bound on the respective expression types, i.e., for any \( a \), expressions \( e' \) and \( e \) should evaluate to subtypes of \( A' \) and \( A \), respectively. These upper bounds are used for type checking comp types (§ 3.3.2).

Finally, \textit{programs} are sequences of method definitions and library method declarations.

\textbf{Dynamic Semantics.} The dynamic semantics of \( \lambda^C \) are the small-step semantics of Ren and Foster [5], modified to throw blame (§ 3.3.3) when a checked method call fails. They use \textit{dynamic environments} \( E \), defined in Figure 3.5, which map variables to values. We define the relation \( \llangle E, e \rrangle \downarrow e' \), meaning the expression \( e \) evaluates
to $e'$ under dynamic environment $E$. The full evaluation rules use a stack as well ($\S$ A), but we omit the stack here for simplicity.

**Example.** As an example comp type in the formalism, consider type checking the expression $\text{true} \land (\text{true})$, where the $\land$ method returns the logical conjunction of the receiver and argument. Standard type checking would assign this expression the type $\text{Bool}$. However, with comp types we can do better.

Recall that $\text{true}$ and $\text{false}$ are members of the singleton types $\text{True}$ and $\text{False}$. Thus, we can write a comp type for the $\land$ method that yields a singleton return type when the arguments are singletons, and $\text{Bool}$ in the fallback case:

\[
\text{lib}\text{ Bool}. \land (x) : (a < : \text{Bool} / \text{Bool}) \rightarrow ( \\
\text{if} (tself == \text{True}). \land (a == \text{True}) \text{ then True} \\
\text{else if} (tself == \text{False}). \lor (a == \text{False}) \text{ then False} \\
\text{else} \text{ Bool} / \text{Bool}
\]

The first two lines of the condition handle the singleton cases, and the last line is the fallback case.

### 3.3.2 Type Checking and Rewriting

Figure 3.6 gives a subset of the rules for type checking $\lambda^C$ and rewriting $\lambda^C$ to insert dynamic checks at library calls. The remaining rules, which are straightforward, can be found in Appendix A. These rules use two additional definitions from Figure 3.5. *Type environments* $\Gamma$ map variables to base types, and the *class table* $CT$
Type Checking and Rewriting Rules

\[ \Gamma \vdash \text{CT} \quad \text{C-TYPE} \]

\[
\begin{align*}
\Gamma \vdash \text{CT} A &\rightarrow A : \text{Type} \\
\Gamma \vdash \text{CT} e \leftarrow e' : A &\quad CT(A.m) = A_1 \rightarrow A_2 \\
\Gamma \vdash \text{CT} e_x \leftarrow e'_x : A_x &\quad A.m \in \mathcal{U} \\
\Gamma \vdash \text{CT} e.m(e_x) \leftarrow e'.m(e'_x) : A_2 &\quad A_x \leq A_1
\end{align*}
\]

\[ \Gamma \vdash \text{CT} \quad \text{C-APPUD} \]

\[
\begin{align*}
\Gamma \vdash \text{CT} e \leftarrow e' : A &\quad CT(A.m) = A_1 \rightarrow A_2 \\
\Gamma \vdash \text{CT} e_x \leftarrow e'_x : A_x &\quad A.m \in \mathcal{L} \\
\Gamma \vdash \text{CT} e.m(e_x) \leftarrow [A_2]e'.m(e'_x) : A_2 &\quad A_x \leq A_1
\end{align*}
\]

\[ \Gamma \vdash \text{CT} \quad \text{C-APP-LIB} \]

\[
\begin{align*}
\Gamma \vdash \text{CT} e \leftarrow e' : A &\quad CT(A.m) = (a+e_{l1}/A_{t1}) \rightarrow e_{l2}/A_{t2} \\
\Gamma \vdash \text{CT} e_x \leftarrow e'_x : A_x &\quad A.m \in \mathcal{L} \\
a: \text{Type}, \text{tself}: \text{Type} \vdash \text{CT} e_{l1} \leftarrow e'_{l1} : \text{Type} \\
a: \text{Type}, \text{tself}: \text{Type} \vdash \text{CT} e_{l2} \leftarrow e'_{l2} : \text{Type} \\
\langle [a \mapsto A_x][\text{tself} \mapsto A], e'_{l1} \rangle \Downarrow A_1 &\quad A_x \leq A_1 \\
\langle [a \mapsto A_x][\text{tself} \mapsto A], e'_{l2} \rangle \Downarrow A_2 &\quad A_x \leq A_1
\end{align*}
\]

\[ \Gamma \vdash \text{CT} e.m(e_x) \leftarrow [A_2]e'.m(e'_x) : A_2 \]

\[ \Gamma \vdash \text{CT} \quad \text{C-APP-COMP} \]

Figure 3.6: A subset of the type checking and rewriting rules for \( \lambda^C \).

maps methods to their type signatures. We omit the construction of class tables, which is standard. We also use disjoint sets \( \mathcal{U} \) and \( \mathcal{L} \) to refer to the user-defined and library methods, respectively.

The rules in Figure 3.6 prove judgments of the form \( \Gamma \vdash \text{CT} e \leftarrow e' : A \), meaning under type environment \( \Gamma \) and class table \( CT \), source expression \( e \) is rewritten to target expression \( e' \), which has type \( A \).

Rule (C-TYPE) is straightforward: any class ID \( A \) that is used as a value is rewritten to itself, and it has type \( Type \). We include this rule to emphasize that types are values in \( \lambda^C \).

Rule (C-APPUD) finds the receiver type \( A \), then looks up \( A.m \) in the class table. This rule only applies when \( A.m \) is user-defined and thus has a (standard)
method type $A_1 \to A_2$. Then, as is standard, the rule checks that the argument’s type $A_x$ is a subtype of $A_1$, and the type of the whole call is $A_2$. This rule Rewrites the subexpressions $e$ and $e_x$, but it does not itself insert any new checks, since user-defined methods are statically checked against their type signatures (rule not shown).

Rule (C-APPLIB) is similar to Rule (C-APPUD), except it applies when the callee is a library method. In this case, the rule inserts a check to ensure that, at run-time, the library method abides by its specified type.

Rule (C-APP-COMP) is the crux of $\lambda^C$’s type checking system. It applies at a call to a library method $A.m$ that uses a type-level computation, i.e., with a type signature $(a <: e_{t1}/A_{t1}) \to e_{t2}/A_{t2}$. The rule first type checks and rewrites $e_{t1}$ and $e_{t2}$ to ensure they will evaluate to a type (i.e., have type $Type$). These expressions may refer to $a$ and $tself$, which themselves have type $Type$. The rule then evaluates the rewritten $e_{t1}$ and $e_{t2}$ using the dynamic semantics mentioned above to yield types $A_1$ and $A_2$, respectively. Finally, the rule ensures that the argument $e_x$ has a subtype of $A_1$; sets the return type of the whole call to $A_2$; and inserts a dynamic check that the call returns an $A_2$ at runtime. For instance, the earlier example of the use of logical conjunction would be rewritten to $\left\lceil True \right\rceil \text{true} \land (\text{true})$.

There is one additional subtlety in Rule (C-APP-COMP). Recall the example above that gives a type to $Bool.\land$. Notice that the type-level computation itself uses $Bool.\land$. This could potentially lead to infinite recursion, where calling $Bool.\land$ requires checking that $Bool.\land$ produces a type, which requires recursively checking that $Bool.\land$ produces a type etc.
To avoid this problem, we introduce a function $\|CT\|$ that rewrites class table $CT$ to drop all annotations with type-level expressions. More precisely, any comp type $(a:e_1/A_1) \rightarrow e_2/A_2$ is rewritten to $A_1 \rightarrow A_2$. Then type checking type-level computations in the premise of (C-App-Comp) is done under the rewritten class table.

Note that, while this prevents the type checking rules from infinitely recursing, it does not prevent type-level expressions from themselves diverging. In $\lambda^C$, we assume this does not happen, but in our implementation, we include a simple termination checker that is effective in practice (§ 3.4).

3.3.3 Properties of $\lambda^C$.

Finally, we prove type soundness for $\lambda^C$. For brevity, we provide only the high-level description of the proof. The details can be found in Appendix A.

Blame. The type system of $\lambda^C$ does not prevent null-pointer errors, i.e., nil has no methods yet we allow it to appear wherever any other type of object is expected. We encode such errors as blame. We also reduce to blame when a dynamic check of the form $\mathsf{⌈A⌉}A.m(v)$ fails.

Program Checking and $CT$. In the Appendix A we provide type checking rules not just for $\lambda^C$ expressions but also for programs $P$. These rules are where we actually check user-defined methods against their types. We also define a notion of validity for a class table $CT$ with respect to $P$, which enforces that $CT$’s types
for methods and fields match the declared types in $P$, and that appropriate sub-
typing relationships hold among subclasses. Given a well typed program $P$, it is
straightforward to construct a valid $CT$.

**Type Checking Rules.** In addition to the type checking and rewriting rules
of Figure 3.6, we define a separate judgment $\Gamma \vdash_{CT} e : A$ that is identical to
$\Gamma \vdash_{CT} e \leftrightarrow e : A$ except it omits the rewriting step, i.e., only performs type
checking.

We can then prove soundness of the judgment $\Gamma \vdash_{CT} e : A$ using preservation
and progress, and finally prove soundness of the type checking and rewriting rules
as a corollary:

**Theorem 1** (Soundness). For any expressions $e$ and $e'$, type $A$, class table $CT$, and
program $P$ such that $CT$ is valid with respect to $P$, if $\emptyset \vdash_{CT} e \leftrightarrow e' : A$ then $e'$ either
reduces to a value, reduces to blame, or does not terminate.

### 3.4 Implementation

We implemented CompRDL as an extension to RDL, a type checking system for
Ruby [5, 8, 20, 63]. In total, CompRDL comprises approximately 1,170 lines of code
added to RDL.

RDL’s design made it straightforward to add comp types, since types are already
runtime values (§ 1.1). We extended RDL so that, when type checking method calls,
type-level computations are first type checked to ensure they produce a value of
type $Type$ and then are executed to produce concrete types, which are then used
in subsequent type checking. Comp types use RDL’s contract mechanism to insert
dynamic checks for comp types.

**Heap Mutation.** For simplicity, \(\lambda^C\) does not include a heap. By contrast, CompRDL
allows arbitrary Ruby code to appear in comp types. This allows great flexibility,
but it means such code might depend on mutable state that could change between
type checking and the execution of a method call. For example, in Figure 3.2, type-
level code uses the global table RDL.db_schema. If, after type checking the method
available?, the program (pathologically) changed the schema of User to drop the
username column, then available? would fail at runtime even though it had type
checked. The dynamic checks discussed in § 3.2 and § 3.3 are insufficient to catch
this issue, because they only check a method call against the initial result of evalu-
ating a comp type; they do not consider that the same comp type might yield a
new result at runtime.

To address this issue, CompRDL extends dynamic checks to ensure types remain
the same between type checking and execution. If a method call is type checked
using a comp type, then prior to that call at runtime, CompRDL will reevaluate that
same comp type on the same inputs. If it evaluates to a different type, CompRDL will
raise an exception to signal a potential type error. An alternative approach would
be to re-check the method under the new type.

Of course, the evaluation of a comp type may itself alter mutable state. Cur-
rently, CompRDL assumes that comp type specifications are correct, including any
mutable computations they may perform. If a comp type does have any erroneous
Termination of Comp Types. A standard property of type checkers is that they terminate. However, because comp types allow arbitrary Ruby code, CompRDL could potentially lose this property. To address this issue, CompRDL includes a lightweight termination checker for comp types.

Figure 3.7 illustrates the ideas behind termination checking. In CompRDL, methods can be annotated with termination effects :+ for methods that always terminate (e.g., m1 and m2) and :- for methods that might diverge (e.g., m3).
CompRDL allows terminating methods to call other terminating methods (Line 9) but not potentially non-terminating methods (Line 10). Additionally, terminating methods may not use loops (Line 11). CompRDL assumes that type-level code does not use recursion, and leave checking of recursion to future work.

We believe it is reasonable to forbid the use of built-in loop constructs, and to assume no recursion, because in practice most iteration in Ruby occurs via methods that iterate over a structure. For instance, `array.map { block }` returns a new array in which the `block`, a `code block` or lambda, has been applied to each element of `array`. Since arrays are by definition finite, this call terminates as long as `block` terminates and does not mutate the `array`. A similar argument holds other iterators of `Array`, `Hash`, etc.

Thus, CompRDL checks termination of iterators as follows. Iterator methods can be annotated with the special termination effect `:blockdep` (Line 5), indicating the method terminates if its block terminates and is pure. CompRDL also includes purity effect annotations indicating whether methods are pure (`:+`) or impure (`:-`). A pure method may not write to any instance variable, class variable, or global variable, or call an impure method. CompRDL determines that a `:blockdep` method terminates as long as its block argument is pure, and otherwise it may diverge. Using this approach, CompRDL will allow Line 14 but reject reject Line 15.

**Type Mutations and Weak Updates.** Finally, to handle aliasing, our type annotations for `Array`, `Hash`, and `String` need to perform weak updates to type information when tuple, finite hash, and const string types, respectively, are mutated.
For example, consider the following code:

```ruby
a = [1, 'foo']; if...then b = a else...end ; a[0] = 'one'
```

Here (ignoring singleton types for simplicity), `a` initially has the type `t = [Integer, String]`, where `t` is a Ruby object, specifically an instance of RDL’s `TupleType` class. At the join point after the conditional, the type of `b` will be a union of `t` and its previous type.

We could potentially forbid the assignment to `a[0]` because the right-hand side does not have the type `Integer`. However, this is likely too restrictive in practice. Instead, we would like to mutate `t` after the write. However, `b` shares this type. Thus we perform a *weak update*: after the assignment we mutate `t` to be `[Integer \cup String, String]`, to handle the cases when `a` may or may not have been assigned to `b`.

For soundness, we need to retroactively assume `t` was always this type. Fortunately, for all tuple, finite hash, and const string types \( \tau \), RDL already records all asserted constraints \( \tau' \leq \tau \) and \( \tau \leq \tau' \) to support promotion of tuples, finite hashes, and const strings to types `Array`, `Hash`, and `String`, respectively [20]. We use this same mechanism to replay previous constraints on these types whenever they are mutated. For example, if previously we had a constraint \( \alpha \leq [\text{Integer}, \text{String}] \), and subsequently we mutated the latter type to \( [\text{Integer} \cup \text{String}, \text{String}] \), we would “replay” the original constraint as \( \alpha \leq [\text{Integer} \cup \text{String}, \text{String}] \).
### 3.5 Experiments

We evaluated CompRDL by writing comp type annotations for a number of Ruby core and third party libraries (§ 3.5.1) and using these types to type check real-world Ruby applications (§ 3.5.2). We discuss the results of type checking these benchmarks, including the type errors we found in the process (§ 3.5.3). In all, we wrote 586 comp type annotations for Ruby library methods, used them to type check 132 methods across six Ruby apps, found three bugs in the process, and used significantly fewer manually inserted type casts than are needed using RDL.

**3.5.1 Library Types**

Table 3.1 details the library type annotations we wrote.

We chose to define comp types for these libraries due to their popularity and because, as discussed in § 3.2, they are amenable to precise typing with comp types. These types were written based on the libraries’ documentation as well as manual...
testing to ensure type specifications matched associated method semantics.

- **Ruby core libraries:** These are libraries that are written in C and automatically loaded in all Ruby programs. We annotate the methods from the `Array`, `Hash`, `String`, `Integer`, and `Float` classes.

- **ActiveRecord:** `ActiveRecord` is the most used object-relational model (ORM) DSL of the Ruby on Rails web framework. We wrote comp types for `ActiveRecord` database query methods.

- **Sequel:** `Sequel` is an alternative database ORM DSL. It offers some more expressive queries than are available in ActiveRecord.

Table 3.1 lists the number of methods for which we defined comp types in each library and the number of Ruby lines of code (LoC) implementing the type computation logic. The LoC count was calculated with `sloccount` [66] and does not include the line of the type annotation itself.

In developing comp types for these libraries, we discovered that many methods have the same type checking logic. This helped us write comp types for entire libraries using a few common helper methods. In total, we wrote comp type annotations for 586 methods across these libraries, comprising 1447 lines of type-level code and using 83 helper methods. Once written, these comp types can be used to type check as many of the libraries’ clients as we would like, making the effort of writing them potentially very worthwhile.
3.5.2 Benchmarks

We evaluated CompRDL by type checking methods from two popular Ruby libraries and four Rails web apps:

- *Wikipedia Client* [67] is a Ruby wrapper library for the Wikipedia API.
- *Twitter Gem* [68] is a Ruby wrapper library for the Twitter API.
- *Discourse* [69] is an open-source discussion platform built on Rails. It uses ActiveRecord.
- *Huginn* [70] is a Rails app for setting up agents that monitor the web for events and perform automated tasks in response. It uses ActiveRecord.
- *Code.org* [71] is a Ruby app that powers code.org, a site that encourages people, particularly students, to learn programming. It uses a combination of ActiveRecord and Sequel.
- *Journey* [72] is a web application that provides a graphical interface to create surveys and collect responses from participants. It uses a combination of ActiveRecord and Sequel.

We selected these benchmarks because they are popular, well-maintained, and make extensive use of the libraries noted in § 3.5.1. More specifically, the APIs often work with hashes representing JSON objects received over HTTP, and the Rails apps rely heavily on database queries.
Since CompRDL performs type checking, we must provide a type annotation for any method we wish to type check. Our subject programs are very large, and hence annotating all of the programs’ methods is infeasible. Instead, we focused on methods for which comp types would be most useful.

In Wikipedia, we annotated the entire Page API. To simplify type checking slightly, we changed the code to replace string hash keys with symbols, since RDL’s finite hash types do not currently support string keys. In Twitter, we annotated all the methods of stream API bindings that made use of methods with comp types.

In Discourse and Huginn, we chose several larger Rails model classes, such as a User class that represents database rows storing user information. In Code.org and Journey, we type checked all methods that used Sequel to query the database. Within the selected classes for these four Rails apps, we annotated a subset of the methods that query the database using features that CompRDL supports. The features CompRDL does not currently support include the use of Rails scopes, which are essentially macros for queries, and the use of SQL strings for methods other than where.

Finally, because CompRDL performs type checking at runtime (similar to RDL—see § 1.1), we must first load each benchmark before type checking it. We ran the type checker immediately after loading a program and its associated type annotations.
### Table 3.2: CompRDL type checking results.

<table>
<thead>
<tr>
<th>Program</th>
<th># Meth</th>
<th>LoC</th>
<th>Extra Anns.</th>
<th>Casts</th>
<th>RDL Casts</th>
<th>Time (s) Median ± SIQR</th>
<th>Test Time No Chk (s)</th>
<th>Test Time w/Chk. (s)</th>
<th>Err</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>API client libraries</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wiki</td>
<td>16</td>
<td>47</td>
<td>3</td>
<td>1</td>
<td>13</td>
<td>0.06 ± 0.00</td>
<td>6.3 ± 0.13</td>
<td>6.32 ± 0.11</td>
<td>0</td>
</tr>
<tr>
<td>Twitter</td>
<td>3</td>
<td>29</td>
<td>11</td>
<td>3</td>
<td>8</td>
<td>0.02 ± 0.00</td>
<td>0.07 ± 0.00</td>
<td>0.08 ± 0.00</td>
<td>0</td>
</tr>
<tr>
<td><strong>Rails Applications</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disc.</td>
<td>36</td>
<td>261</td>
<td>32</td>
<td>13</td>
<td>22</td>
<td>7.77 ± 0.39</td>
<td>80.24 ± 0.63</td>
<td>81.04 ± 0.34</td>
<td>0</td>
</tr>
<tr>
<td>Huginn</td>
<td>7</td>
<td>54</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>2.46 ± 0.29</td>
<td>4.30 ± 0.21</td>
<td>4.59 ± 0.48</td>
<td>0</td>
</tr>
<tr>
<td>Code.org</td>
<td>49</td>
<td>530</td>
<td>53</td>
<td>3</td>
<td>68</td>
<td>0.49 ± 0.01</td>
<td>2.49 ± 0.13</td>
<td>2.74 ± 0.02</td>
<td>1</td>
</tr>
<tr>
<td>Journey</td>
<td>21</td>
<td>419</td>
<td>78</td>
<td>14</td>
<td>59</td>
<td>4.12 ± 0.08</td>
<td>4.52 ± 0.22</td>
<td>4.76 ± 0.24</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>132</td>
<td>1340</td>
<td>183</td>
<td>37</td>
<td>176</td>
<td>14.93 ± 0.77</td>
<td>97.93 ± 1.31</td>
<td>99.53 ± 1.20</td>
<td>3</td>
</tr>
</tbody>
</table>

3.5.3 Results

Table 3.2 summarizes our type checking results. In the first group of columns, we list the number of type checked methods and the total lines of code (computed with `sloccount`) of these methods. The third column lists the number of additional annotations we wrote for any global and instance variables referenced in the method, as well as any methods called that were not themselves selected for type checking.

The last column in this group lists the number of type casts we added. Many of these type casts were to the result of `JSON.parse`, which returns a nested `Hash/Array` data structure depending on its string input. Most of the remaining casts are to refine types after a conditional test; it may be possible to remove these casts by adding support for occurrence typing [10]. We further discuss type casts, in particular the reduced type casting burden afforded by comp types, below.

**Increased Type Checking Precision.** Recall from § 3.2.2 that comp types can potentially reduce the need for programmer-inserted type casts. The next column reports how many casts were needed using normal RDL (i.e., no comp types). As
shown, approximately $4.75 \times$ fewer casts were needed when using comp types. This reflects the significantly increased precision afforded by comp types, which greatly reduces the programmer’s annotation burden.

**Performance.** The next group of columns report performance. First we give the type checking time as the median and semi-interquartile range (SIQR) of 11 runs on a 2017 MacBook Pro with a 2.3GHz i5 processor and 8GB RAM. In total, we type checked 132 methods in approximately 15 seconds, which we believe to be reasonable. *Discourse* took most of the total time (8 out of 15 seconds). The reason turned out to be a quirk of *Discourse*’s design: it creates a large number of methods on-the-fly when certain constants are accessed. Type checking accessed those constants, hence the method creation was included in the type checking time.

The next two columns show the performance overhead of the dynamic checks inserted by *CompRDL*. We selected a subset of each app’s test suite that directly tested the type checked methods, and ran these tests without (“No Chk”) and with (“w/Chk”) the dynamic checks. In aggregate (last row), checks add about 1.6% overhead, which is minimal.

**Errors Found.** Finally, the last column lists the number of errors found in each program. We were somewhat surprised to find any errors in large, well-tested applications. We found three errors. In *Code.org*, the current_user method was documented as returning a *User*. We wrote a matching *CompRDL* annotation, and *CompRDL* found that the returned expression—whose typing involved a comp type—has a hash
In Journey, CompRDL found two errors. First, it found a method that referenced an undefined constant Field. We notified the developers, who fixed the bug by changing the constant to Question::Field. This bug had arisen due to namespace changes. Second, it found a method that included a call with an argument `{:action => prompt, ...}` which is a hash mapping key :action to prompt. The value prompt is supposed to be a string or symbol, but as it has neither quotes nor begins with a colon, it is actually a call to the prompt method, which returns an array. The developers confirmed this bug.

When type checking the aforementioned methods in RDL (i.e., without comp types), two out of three of the bugs are hidden by other type errors which are actually false positives. These errors can be removed by adding four type casts, which would then allow us to catch the true errors. With CompRDL, however, we do not need any casts to find the errors.

3.6 Related Work

Types For Dynamic Languages. There is a large body of research on adding static type systems to dynamic languages, including Ruby [4, 5, 8], Racket [3, 9], JavaScript [16, 53, 54], and Python [11, 12]. To the best of our knowledge, this research does not use type-level computations.

Dependent typing systems for dynamic languages have been explored as well.
Ou et al. [73] formally model type-level computation along with effects for a dynamic language. Other projects have sought to bring dependent types to existing dynamic languages, primarily in the form of refinement types [28], which are base types that are refined with expressive logical predicates. As discussed in Chapter 2, refinement types have been applied to Ruby [21], Racket [10], and JavaScript [15, 29]. In contrast to CompRDL, these systems focus on type checking methods which themselves have dependent types. On the other hand, CompRDL uses type-level computations only for non-type checked library methods, allowing us to avoid checking comp types for equality or subtyping (§ 3.2.4). While sacrificing some expressiveness, this makes CompRDL especially practical for real-world programs.

Turnstile [74] is a metalmanguage, hosted in Racket, for creating typed embedded languages. It lets an embedded DSL author write their DSL’s type system using the host language macro system. There is some similarity to CompRDL, where comp types manipulate standard RDL types. However, CompRDL types are not executed as macros (which do not exist in Ruby), but rather in standard Ruby so they have full access to the environment, e.g., so the joins type signature can look up the DB schema.

**Types For Database Queries.** There have been a number of prior efforts to check the type safety of database queries. All of these target statically typed languages, an important distinction from CompRDL.

Chlipala [75] presents Ur, a web-specific functional programming language. Ur uses type-level computations over record types [76] to type check programs that con-
struct and run SQL queries. Indeed, CompRDL similarly uses type-level computations over finite hash types (analogous to record types) to type check queries. To the best of our knowledge, Ur focuses on computations over records. In contrast, CompRDL supports arbitrary type-level computations targeting unchecked library methods, making comp types more easily extensible to checking new properties and new libraries. As discussed in § 3.2, for example, comp types can not only compute the schema of a joined table, but also check properties like two joined tables having a declared Rails association. Further, comp types can be usefully applied to many libraries beyond database queries (§ 3.5).

Similar to Ur, Baltopoulos et al. [77] make use of record types over embedded SQL tables. Using SMT-checked refinement types, they can statically verify expressive data integrity constraints, such as the uniqueness of primary keys in a table and the validation of data inserted into a table. In addition to the contrast we draw with Ur regarding extensibility of types, to the best of our knowledge, this work does not include more intricate queries like joins, which are supported in CompRDL.

**New Languages for Database Queries.** Domain-specific languages have long been used to write programs with correct-by-construction, type safe queries. Lei-jen and Meijer [78] implement Haskell/DB, an embedded DSL that dynamically generates SQL in Haskell. Karakoidas et al. [79] introduce J%, a Java extension for embedding DSLs into Java in an extensible, type-safe, and syntax-checked way. Fowler and Brady [80] use dependent types in the language Idris to enforce safety protocols associated with common web program features including database queries.
written in a DSL.

Language-integrated query is featured in languages like LINQ [81] and Links [82, 83]. This approach allows programmers to write database queries directly within a statically-typed, general purpose language.

In contrast to new DSLs and language-integrated query, our focus is on bringing type safety to an existing language and framework rather than developing a new one.

**Dependent Types.** Traditional dependent type systems are exemplified by languages such as Coq [60], Agda [61], and F* [62]. These languages provide powerful type systems that allow programmers to prove expressive properties. However, such expressive types may be too heavyweight for a dynamic language like Ruby. As discussed in § 3.2.4, our work has focused on applying a limited form of dependent types, where types depend on argument types and not arbitrary program values, resulting in a system that is practical for real-world Ruby programs.

Haskell allows for light dependent typing using the combination of singleton types [84] and type families [85]. CompRDL’s singleton types are similar to Haskell’s, i.e., both lift expressions to types, and comp types are analogous to anonymous type families. However, unlike Haskell, CompRDL supports runtime evaluation during type checking, and thus does not require user-provided proofs.

Scala supports *path dependent types*, a limited form of type/term dependency in which types can depend on variables, but, as of Scala version 2, does not allow dependency on general terms [86]. This allows for reasoning about database queries.
For example, the Scala library *Slick* [87], much like our approach, allows users to write database queries in a domain specific language (a lifted embedding) and uses the query’s AST to type check the query using Scala’s path dependent types. Unlike *CompRDL*, Scala’s path dependent types do not allow the execution of the full host language during type computations.

3.7 Conclusion

We presented *CompRDL*, a system for adding type signatures with type-level computations, which we refer to as comp types, to Ruby library methods. *CompRDL* makes it possible to write comp types for database queries, enabling us to type check such queries precisely. Comp type signatures can also be used for libraries over heterogeneous hashes and arrays, and to treat strings as immutable when possible. The increased precision of comp types can reduce the need for manually inserted type casts, thereby reducing the programmer’s burden when type checking. Since comp type-annotated method bodies are not themselves type checked, *CompRDL* inserts run-time checks to ensure those methods return their computed types. We formalized *CompRDL* as a core language $\lambda^C$ and proved its type system sound.

We implemented *CompRDL* on top of *RDL*, an existing type system for Ruby. In addition to the features of $\lambda^C$, our implementation includes run-time checks to ensure comp types that depend on mutable state yield consistent types. Our implementation also includes a termination checker for type-level code, and the type signatures we developed perform weak updates to type certain mutable methods.
Finally, we used CompRDL to write comp types for several Ruby libraries and two database query DSLs. Using these type signatures, we were able to type check six popular Ruby apps and APIs, in the process discovering three errors in our subject programs. We also found that type checking with comp types required $4.75 \times$ fewer type casts, due to the increased precision. Thus, we believe that CompRDL represents a practical approach to precisely type checking programs written in dynamic languages.
Chapter 4: Sound, Heuristic Type Annotation Inference for Ruby

The previous two chapters showed how to make RDL more expressive by introducing refinement types and type-level computations. However, RDL still requires the programmer to manually write type annotations for any type checked methods, as well as any variables and methods referenced in type checked code. This process can be tedious and time-consuming, especially when introducing types to large programs.

This chapter introduces InferDL, a system for automatically generating type annotations for the methods and variables in Ruby programs. While the traditional approach to this problem is to use constraint-based type inference, this chapter discusses why this approach sometimes fails to generate usable type annotations that reflect a programmer's intent, and explains how InferDL complements constraint solving with configurable heuristic rules in order to generate more useful annotations.

Acknowledgments

The core idea of refining constraint-based type inference with heuristic rules in order to produce more usable type annotations belongs to Ren [22]. They also contributed the STRUCT-TO-NOMINAL heuristic rule presented in this chapter. I
reimplemented this work, came up with the central algorithm and the idea of using configurable (rather than baked-in) heuristics, wrote the other 7 heuristic rules presented in this chapter, and conducted the presented evaluations.

4.1 Introduction

While many researchers have explored ways to add static types to dynamic languages, one key challenge in using such retrofitted type systems is finding type annotations for code that was previously untyped. *Type inference*, which aims to type check programs with few or no type annotations, is an obvious solution, and indeed there are several type inference systems for dynamic languages [4, 11, 16, 88]. Beyond type checking, type inference can also be extended to generate type annotations for program values. These annotations provide a useful form of documentation, and can be used in other forms of program analysis such as code completion for IDEs.

However, type inference systems typically aim to find the most general type for every position, i.e., the least restrictive possible type. If the type language is rich—particularly if it includes structural types—the most general possible annotations might be large, hard to read, and unnatural for programmers. For example, An et al. [88] describe a type inference system for Ruby that infers that a certain position accepts any object with $>$, $<$, $>$>, $&$, and $^\wedge$ methods. In contrast, a programmer would most likely, and much more concisely, say that position takes an *Integer*. Moreover, even if we are not interested in producing annotations, large, complex types can lead to difficult-to-understand error messages.
In this chapter we present InferDL, a novel Ruby type inference system that aims to infer sound and useful type annotations. More specifically, InferDL allows the programmer to specify heuristics for guessing type annotations. For example, one simple but effective heuristic is to guess the type `Integer` for any variables whose names end with `id`, `num`, or `count`. InferDL runs such heuristics for positions for which standard type inference is overly-general (meaning, among others, positions with inferred structural types). Heuristic guesses are added as additional type constraints and checked for consistency with the rest of the program. Only consistent solutions are kept. In this way, InferDL maintains soundness while producing a less general, but potentially more useful, solution than standard type inference. (§ 4.2 gives an overview of InferDL.)

We describe InferDL more formally on a core type and constraint language. We present standard constraint resolution rules, which rewrite a set of constraints into solved form from which most general solutions can be extracted. We describe that solution extraction procedure in detail and then show how to incorporate heuristics. (See § 4.3 for our formal description.)

We implemented InferDL as an extension to RDL (§ 1.1). We modified RDL to generate and resolve type constraints, run heuristics, and extract solutions to produce annotations. InferDL currently includes eight heuristics: one that replaces structural types with nominal types that match the structure; six that look at variable names, such as the one mentioned above for names ending in `id`, `num`, or `count`; and one that produces precise hash types. We also extended RDL with choice types, an idea inspired by variational typing [89] that helps type inference work in the
presence of overloaded methods. (§ 4.4 describes the implementation of InferDL.)

We evaluated InferDL by applying it to four Ruby on Rails apps for which RDL type annotations already existed [5, 90]. We note that our results are preliminary, and further work is needed to affirm they generalize beyond our benchmarks. For these apps, InferDL inferred 496 type annotations. Of these, 399 exactly matched or were more precise than the programmer-supplied annotations, compared to only 290 such annotations when not using heuristics. InferDL also found one previously unknown type error. We also applied InferDL to six additional Ruby programs for which we did not have previously written annotations, and InferDL found five previously unknown type errors. (§ 4.5 discusses our evaluation.)

We believe that InferDL is an effective type inference system and represents a promising approach to generating useful, sound type annotations.

4.2 Overview

We begin by discussing standard type inference, which generates and solves type constraints to yield type annotations. We then discuss why this approach alone can be inadequate and give a high-level overview of how InferDL uses heuristics to infer more precise, useful types.

4.2.1 Standard Type Inference

Figure 4.1 shows a code snippet taken from Discourse, a Ruby on Rails web app used in our evaluation (§ 4.5). The code defines two methods, normalize_username
class User < ActiveRecord::Base

  # α → β
  def self.normalize_username(name)
    name.unicode_normalize.downcase if name.present?
  end

  # γ → δ
  def self.find_by_name(name)
    find_by(name_lower: normalize_username(name))
  end
end

<table>
<thead>
<tr>
<th>Constraints Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) α ≤ [unicode_normalize : ⊥ → ϵ]</td>
</tr>
<tr>
<td>(2) α ≤ [present? : ⊥ → ζ]</td>
</tr>
<tr>
<td>(3) ϵ ≤ [downcase : ⊥ → η]</td>
</tr>
<tr>
<td>(4) η ≤ β</td>
</tr>
<tr>
<td>(5) γ ≤ α</td>
</tr>
<tr>
<td>(6) User ≤ δ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resolved Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>(7) γ ≤ [unicode_normalize : ⊥ → ϵ]</td>
</tr>
<tr>
<td>(8) γ ≤ [present? : ⊥ → ζ]</td>
</tr>
</tbody>
</table>

Figure 4.1: Method from the Discourse app, and the resulting constraints generated during inference.

and find_by_name, in the class User. Because User is a subclass of ActiveRecord::Base, it is a Rails model, meaning instances of the class represent rows of a database table.

Suppose we wish to infer types for these two methods using the standard, constraint-based approach. We first generate a type variable for the method argument and return types, as shown in the comments on lines 2 and 6, e.g., normalize_username takes a value of type α and returns type β. Then we analyze the method body, generating constraints of the form x ≤ y, indicating that x must be a subtype of y. In this case we also say x is a lower bound on y and y is an upper bound on x.
The top portion of the table in Figure 4.1 shows the constraints generated from this example. Constraint (1) arises from the call `name.unicode_normalize`. In this constraint, the *structural type* \([\text{unicode Normalize} : \bot \rightarrow \epsilon]\) represents an object with a `unicode_normalize` method that takes no argument (here written \(\bot\)) and returns \(\epsilon\), a fresh type variable generated at the call. Hence, by standard subtyping rules, \(\alpha\) must be a type that contains at least this method with appropriate argument and return types. Constraint (2) is similar.

Constraint (3) arises from calling `downcase` on the result of `unicode_normalize`. Constraint (4) arises because the result of the call to `downcase` is returned. Note that `normalize_username` may also return `nil` (if the conditional guard is false), but `nil` is a subtype of all other types in `InferDL`, so we omit this constraint here. Finally, constraint (5) arises from the call to `normalize_username` on line 8, and constraint (6) arises because `User`’s `find_by` method returns a `User` (as indicated by `find_by`’s type annotation, omitted here).

After generating constraints, `InferDL` performs *constraint resolution*, which applies a series of rewriting rules to the constraints. For example, one resolution rule is transitive closure: If \(a \leq b\) and \(b \leq c\) then we add constraint \(a \leq c\) (see §4.3 for a complete list of constraint resolution rules). In our example, constraint resolution generates the two new constraints (7) and (8).

During constraint resolution, if `InferDL` generates any invalid constraints, the program is untypable, and `InferDL` signals a type error. Otherwise, if constraint resolution terminates without finding any inconsistencies, then the program is ty-

---

1 In Ruby, the parentheses in a method call are optional.
Solution Extraction. Many traditional type inference approaches have the singular goal of uncovering type errors, and hence they stop after propagating constraints. Since our goal is to also infer type annotations, we must go a step further by extracting a solution for all type variables from the constraints. The standard approach is to compute a most general solution. For a method type, this means computing the least solution for its return and the greatest solution for its arguments, which are the solutions that are least constraining on the method’s callers.

Fortunately, after constraint resolution, the constraints are in solved form [91], which means that to extract a variable’s solution we need only look at its lower and upper bounds. More specifically, for a return, we compute the union of its lower bounds, ignoring variables (since any transitive constraints from them have been propagated by resolution), and for an argument, we compute the intersection of its upper bounds. Thus, in our example, the solution for $\alpha$ and $\gamma$ is $[\text{unicode}\_\text{normalize} : \bot \rightarrow \epsilon, \text{present?} : \bot \rightarrow \zeta]$, i.e., an object that has those methods, and the solution for $\delta$ is User.

However, notice there are some problems with producing type annotations using this approach. First, the solution for $\alpha$ and $\gamma$ is in fact not fully expanded. Using the same approach, we could recursively compute a solution to $\epsilon$ to get the following solution for $\alpha$ and $\gamma$: $[\text{unicode}\_\text{normalize} : \bot \rightarrow [\text{downcase} : \bot \rightarrow \eta, \text{present?} : \bot \rightarrow \zeta]$. However, such nested structural types are difficult to read and comprehend, and worse, in the presence of recursion, the type may not be expressible in finite form.
without additional syntax.

Second, notice that $\eta$ is the most general solution for $\beta$, and there is no most general solution for $\eta$ and $\zeta$ that we can write down as ground terms (i.e., terms with no type variables). That is, in fact we cannot always ignore type variables in solutions, because they are needed to express relationships among different parts of the solution (here, $\eta$ is the return type of `downcase` and $\zeta$ is the return type of `present`). This makes understanding the most general solution even more complex and difficult.

4.2.2 Type Inference with Heuristics

InferDL aims to infer more useful, readable, and understandable type annotations by extending standard inference with heuristics that guess nominal types, or small unions of nominal types, as solutions. For example, so far $\delta$ has a nominal type as a solution, and we would like the same thing for other type variables. To ensure type annotations are consistent, InferDL adds any solutions found by heuristics to the constraints and runs constraint resolution afterward; if the result is a type error, the heuristic choice is rejected.

Using Heuristics. We illustrate the use of heuristics on our example with one particular rule, STRUCT-TO-NOMINAL, defined (in English) as follows:

When an argument type variable's upper bounds include structural types, search all classes to see which have the methods in those types. If there are ten or fewer such classes, guess the union of these classes as the type
variable’s solution.

Note that this rule matches by method name only and not by method type. We chose ten as a cutoff because in our experience, larger unions are less useful than the original structural type. In our running example, STRUCT-TO-NOMINAL can be applied to $\alpha$ and $\gamma$. It turns out that $\text{String}$ is the only class that defines both $\text{unicode Normalize}$ and $\text{present?}$, so the nominal type $\text{String}$ would be our heuristic guess.

To ensure this guess is sound, we add $\text{String}$ as a solution for variables $\alpha$ and $\gamma$ to our constraints. More specifically, we add the solution constraint $\alpha = \text{String}$ (and similarly for $\gamma$) to the constraints, where $a = b$ is shorthand for the pair of constraints $a \leq b$ and $b \leq a$. We then resolve these new constraints, which in this case does not lead to any inconsistency, so we accept $\text{String}$ as the solution.

Moreover, the additional constraints on $\alpha$ and $\gamma$ in turn yield better solutions elsewhere because:

$\Rightarrow \text{String} \leq \alpha$ is added as a constraint. Transitively propagating to $\alpha$’s upper bounds yields...

$\Rightarrow \text{String} \leq [\text{unicode Normalize} : \bot \rightarrow \epsilon]$. To check this constraint, we look up $\text{String}$’s $\text{unicode Normalize}$ method type and generate the constraint...

$\Rightarrow \bot \rightarrow \text{String} \leq \bot \rightarrow \epsilon$. Propagating to the methods’ return types yields...

$\Rightarrow \text{String} \leq \epsilon$. Transitively propagating through $\epsilon$ yields...

$\Rightarrow \text{String} \leq [\text{downcase} : \bot \rightarrow \eta]$. Looking up $\text{String}$’s type for $\text{downcase}$, we get
the constraint...

⇒ ⊥ → String ≤ ⊥ → η, and propagating to return types yields...

⇒ String ≤ η. Finally, propagating to η’s upper bound yields...

⇒ String ≤ β

Thus, now β has nominal type String as a solution. Putting this together with the (most general) solution for δ and the (heuristic) solutions for α and γ, InferDL has now inferred fully nominal type annotations for our example:

normalize_username: String → String

find_by_name: String → User

Implementing Heuristics. in InferDL, heuristics are not baked-in. Rather, they can be created by the programmer, allowing heuristics to be adapted if needed to the target program. As an example, consider the following method, taken from the Rails app Journey and slightly simplified:

def self.find_answer ( response, question )
    where(response: response.id, question: question.id ) .first
end

We wish to infer the type of find_answer. However, notice that only id is called on each argument. In Rails, the method id is typically defined for all model classes, e.g., in Journey, 48 different classes include an id method. This means STRUCT-TO-NOMINAL will fail to infer a precise annotation for the arguments in this case.
Instead, we develop another heuristic that takes advantage of a common practice: Ruby programmers often name a variable after the class of the value it will hold, especially for Rails models. Indeed, the arguments `response` and `question` are intended to take instances of the model classes `Response` and `Question`, respectively.

We define a new heuristic `is_model` that guesses types based on this convention:

```ruby
RDL::Heuristic.add :is_model { |var|
  if (var.base_name.camelize.is_rails_model?)
    then var.base_name.to_type end }
```

To define this heuristic, we call `RDL::Heuristic.add`, passing the name of the heuristic, in this case `:is_model`, and a `code block`. Code blocks are Ruby’s version of anonymous functions or lambdas. The `RDL::Heuristic.add` method expects a code block that takes a single argument, which is the type variable whose solution the heuristic should guess. Note that in `InferDL`, which is built on `RDL`, types are actual values we can compute with; we discuss this in greater detail in § 4.4. The code block returns either `nil`, if there is no guess, or the guessed type.

The `is_model` heuristic consists of a single if statement. The guard calls, in order, `var.base_name`, to return the name of the variable as a `String`; the Rails method `camelize` to camel-case this string; and finally `is_rails_model?`, a method we defined (code omitted) to determine if there exists a Rails model with the same name as the receiver. If this last condition is true, the code block calls `to_type` (code omitted) to return the nominal type for the model class. Otherwise, by standard Ruby semantics the conditional will return `nil`.

During type inference, `InferDL` runs all heuristics for each type variable, accepting the solution from the first heuristic that produces a consistent type. For
Types

\[ \tau ::= \alpha \mid A \mid [m : \tau_m] \mid \tau \cup \tau \mid \tau \cap \tau \mid \bot \mid \top \]

Method Types

\[ \tau_m ::= \tau \rightarrow \tau \]

Constraints

\[ C ::= \tau \leq \tau \mid C \cup C \]

\( A \in \text{class IDs}, m \in \text{meth IDs} \)

Figure 4.2: Core types and constraints.

\textbf{find\_answer}, the \texttt{IS\_MODEL} heuristic produces appropriate nominal types for the arguments, which then become the final type annotations for those positions.

### 4.3 Constraints, Solutions, and Heuristics

In this section, we describe \texttt{InferDL} more formally. For brevity, we do not define a core language, nor do we describe constraint generation in detail. Rather, we focus on the language of types and constraints, constraint resolution, solution extraction, and heuristics.

Figure 4.2 formally defines a core subset of the types and constraints in \texttt{InferDL}. Types \( \tau \) include type variables \( \alpha \) and nominal types \( A \), which is the set of class IDs. Structural types \([m : \tau_m]\) name a method \( m \) and its corresponding method type \( \tau_m \). For brevity, structural types can only comprise a single method, and method types may only take a single argument. Types also include union types \( \tau \cup \tau \), intersection types \( \tau \cap \tau \), the bottom type \( \bot \), and the top type \( \top \). Constraints \( C \) consist of subtyping constraints \( \tau_1 \leq \tau_2 \) and unions of constraints \( C_1 \cup C_2 \), which allow us to build up sets of constraints.
Generating Constraints. Constraint generation is a straightforward modification of standard type checking in which, instead of type rules checking constraints, we view them as generating constraints. For example, the rule for typing a method call is

\[
\frac{\Gamma \vdash e_1 : \tau_{rec} \quad \Gamma \vdash e_2 : \tau_{arg}}{\tau_{rec} \leq [m : \tau_{arg} \rightarrow \tau_{ret}]} \quad \tau_{ret} \text{ is fresh}
\]

\[
\Gamma \vdash e_1.m(e_2) : \tau_{ret}
\]

This rule types a method call \( e_1.m(e_2) \) (where each \( e \) is an expression) in type environment \( \Gamma \) (a map from local variables to types), yielding type \( \tau_{ret} \). To apply this rule, we recursively type the receiver and the argument, yielding types \( \tau_{rec} \) and \( \tau_{arg} \), respectively. We then generate a constraint \( \tau_{rec} \leq [m : \tau_{arg} \rightarrow \tau_{ret}] \), where \( \tau_{ret} \) is a fresh type variable. Then, the return type of the method call has type \( \tau_{ret} \).

By convention, we assume any constraint in the premise of a rule is automatically added to a global set of constraints \( C \).

Full details of type checking rules for a core Ruby language can be found in section 3.3, or in Ren and Foster [5] or Kazerounian et al. [90], all of which formalize Ruby in a core language and provide type checking rules. As with the above example, those rules can be turned into inference rules by viewing them as generating constraints and adjusting them as needed to make all constraints explicit, e.g., in the rule above, we specify the type of \( e_1 \) as \( \tau_{rec} \) and write an explicit constraint on \( \tau_{rec} \), rather than implicitly constraining \( e_1 \)'s type to have a particular shape in the rule.
Resolving Constraints. Figure 4.3 gives standard constraint resolution rules. Each rule has the form \( C \Rightarrow C' \), meaning a set of constraints matching \( C \) can be rewritten to \( C' \). The rules are applied exhaustively until they either yield \( error \) or no additional constraints can be generated.

Rule (1) adds transitive constraints, as discussed earlier. Rule (2) eliminates a constraint among two nominal types as long as the subtyping is valid. On the other hand, Rule (3) produces \( error \) if there is an inconsistent constraint among nominal types. Rule (4) handles constraints of the form \( A \leq [m : \tau_1 \rightarrow \tau_2] \). In this case, if \( A \) has a method \( m \) of some type \( \tau'_1 \rightarrow \tau'_2 \), we erase the constraint and add two new constraints on the argument and return types. If \( A \) does not have a method \( m \), Rule (5) yields \( error \). Finally, Rules (6) and (7) simplify unions on the left and intersections on the right of a constraint, respectively, and Rules (8) and (9) eliminate constraints with \( \bot \) on the left and \( \top \) of the right, respectively.
4.3.1 Solution Extraction

Recall from § 4.2.1 that after generating and resolving constraints, the next step in standard type inference is to extract solutions for type variables. More precisely, standard type inference produces solutions using the following procedure:

\[
\text{procedure STANDARD\_SOLUTION}(C, \alpha) \\
\text{if } (\alpha \text{ represents arg}) \text{ then} \\
\quad sol = \top \\
\quad \text{for each constraint } \alpha \leq \tau \in C \text{ do} \\
\quad \quad sol = sol \cap \tau \\
\text{else} \quad \triangleright \alpha \text{ represents return} \\
\quad sol = \bot \\
\quad \text{for each constraint } \tau \leq \alpha \in C \text{ do} \\
\quad \quad sol = sol \cup \tau \\
\text{return } sol
\]

As discussed earlier, to derive the most general solution, for each return position we compute the union of its lower bounds, and for each argument position we compute the intersection of its upper bounds.

\textbf{InferDL} uses the same procedure as a subroutine to its heuristic inference algorithm, described next.

\textbf{Heuristics.} Formally, we can model \textbf{InferDL}'s heuristics as a set of additional constraint resolution rules beyond those in Figure 4.3. For example, we can express
STRUCT-TO-NOMINAL as the following constraint rewriting rule:

\[
\text{STRUCT-TO-NOMINAL}(C, \alpha) = \\
C' \cup \alpha \leq [m_1 : \ldots] \cup \ldots \cup \alpha \leq [m_n : \ldots] \Rightarrow \\
C' \cup (\alpha = (A_1 \cup \ldots \cup A_k))
\]

if \( k \leq 10 \) and \( A_1, \ldots, A_k \) are all classes with \( m_1 \ldots m_n \).

This rule applies to a type variable \( \alpha \) that has one or more structural type upper bounds. If there are at most 10 classes \( A_1 \ldots A_k \) matching those structural types, then we replace the structural constraints with a solution \( A_1 \cup \ldots \cup A_k \) for \( \alpha \).

Recall from § 4.2 that a solution constraint of the form \( \tau_1 = \tau_2 \) is shorthand for the two constraints \( \tau_1 \leq \tau_2 \) and \( \tau_2 \leq \tau_1 \).

Given a set \( H \) of heuristic rules, InferDL uses the following procedure to try each rule until some rule succeeds or all rules fail:

\begin{algorithm}
\caption{HEURISTIC\_SOLUTION(H, C, \alpha)}
\begin{algorithmic}
\For {h \in H}
\State \( C', \ sol = h(C, \alpha) \)
\State \( C' = \text{resolve}(C') \)
\If {\( C' \neq \text{error} \text{ and } sol \neq \text{nil} \)}
\State \textbf{return} \( C', \ sol \)
\EndIf
\EndFor
\State \textbf{return} \( C, \ \text{nil} \)
\end{algorithmic}
\end{algorithm}

Here we abuse notation slightly and assume that heuristic rules return a pair \( C', \ sol \), where \( C' \) is the new set of constraints after running the rule, and \( sol \) is the solution found for \( \alpha \). If \( h \) does not match \( C \), then \( h \) returns the pair \( \text{nil}, \text{nil} \). When \( h \) does return a set of constraints \( C' \) and a solution, we perform constraint resolution on \( C' \) to propagate the new solution and detect any inconsistencies in the set of constraints.
constraints. This is done by calling the RESOLVE procedure, which simply invokes
the constraint resolution rules of Figure 4.3. If the resulting resolved constraint set
$C'$ is valid, then we return $C'$ and sol. If all heuristics run without finding a valid
solution, we return the original constraints $C$ and nil.

Notice that an important consequence of this formulation is that the order
in which heuristics are run matters, since only the first guessed solution will be
returned. Future work could examine how to overcome this reliance on ordering
and handle the case that heuristics return differing solutions.

**Extracting Solutions.** The next step is to combine standard and heuristic solu-
tion extraction, doing the latter when a standard solution is overly-general. Thus,
we need to define what *overly-general* means. Based on our prior experience, we
believe that, in most cases, nominal types are far easier for programmers to un-
derstand and use than structural types because they are smaller and simpler. The
developers of Sorbet [92], another Ruby type checker, feel the same way—they have
found that structural types can be less intuitive and more difficult to read when
used in error messages [93].

Generalizing this insight, we define an *overly-general* solution as any non-
nominal type, that is, any type of the form $\alpha$, $\tau \cup \tau$, $\tau \cap \tau$, $[m : \tau_m]$, $\bot$, and $\top$. Our
motivation for treating union and intersection types as overly-general is the same as
for structural types: they make types bigger and more complex. We treat the top
and bottom types as overly-general since they are only ever used as solutions when
a type variable has no constraints, and we consider type variables overly-general
since they represent unknown types. We do note that, in general, there is no single correct definition of overly-general, and we leave exploring alternative definitions to future work.

Next, we can provide the pseudocode for solution extraction of a type variable:

```
procedure EXTRACT_SOLUTION(H, C, \alpha)
    sol = STANDARD_SOLUTION(C, \alpha)
    if overly_general(sol) then
        C', sol_h = HEURISTIC_SOLUTION(H, C, \alpha)
        if sol_h \neq nil then
            sol = sol_h
    else
        C' = resolve(C \cup (\alpha = sol))
    return C', sol
```

This procedure first extracts a standard solution for the given \alpha and C. If the resulting solution is overly-general, it calls HEURISTIC_SOLUTION to possibly yield a better solution for \alpha. We only use the new solution if it is non-nil (note that, with the definition of HEURISTIC_SOLUTION, the set of constraints will be unchanged in the event that the heuristic solution is nil). Otherwise, if the standard solution was not too general, we add the new solution to the set of constraints and perform constraint resolution. We perform constraint resolution again because, just like heuristic solutions can lead to other solutions, so too can standard solutions.

Finally, as shown in § 4.2.2, one extracted solution may lead to the discovery of other solutions. Thus, we continue to extract solutions for type variables until no new constraints are generated. More precisely, the following procedure takes H, C, and a set of all type variables \mathcal{V} as input, and extracts solutions until no new constraints are generated.
procedure extract_all_solutions(H, C, V)
solutions_map = {}
repeat
    for each α ∈ V do
        C, sol = extract_solution(H, C, α)
        solutions_map[α] = sol
    until no new constraints are added to C
return solutions_map

4.4 Implementation

InferDL is built on top of the Ruby type checker RDL [20]. An overview of RDL can be found in § 1.1. In this section, we briefly describe how we extended RDL to perform type inference, and some of the other implementation challenges in InferDL.

Adding Standard Inference. InferDL extends RDL so that inferred methods are specified with a call to RDL.infer:

    RDL.infer User, 'self.normalize_username', time: :later

Then, when RDL.do_infer :later is called, InferDL runs type inference to produce type annotations for any method associated with :later.

RDL already included type variables to support parametric polymorphism. InferDL extends type variables to store constraints as a list of upper and lower bounds on each type variable. Then, to perform constraint generation, InferDL modifies RDL’s type checker so that, whenever two types are checked for subtyping, and at least one of the types is a type variable, we store the subtyping constraint. After constraint generation, InferDL performs constraint resolution and solution
extraction, as explained in § 4.3.

**Heuristics.** Though heuristics are not baked-in to InferDL and are thus configurable, we have written eight heuristics that we found useful in practice, listed below. Recall from § 4.3.1 that heuristics are applied in a specified order. We list heuristics in the order in which they are applied.

- **is_model**: See § 4.2.2 for a description. This rule is only used for Rails apps.

- **is_pluralized_model**: If a variable name is the pluralized version of the name of model X, then guess solution $\text{Array<X>} \cup \text{ActiveRecord\_Relation<X>}$.
  
  Note that `ActiveRecord\_Relation` is a data structure provided by Rails that extends common array operations with some database queries. This rule is only used for Rails apps.

- **struct-to-nominal**: See § 4.2.2 or § 4.3.1.

- **int_names**: If a variable name ends with `id`, `count`, or `num`, guess solution `Integer`.

- **int_array_name**: If a variable name ends with `ids`, `counts`, or `nums`, guess solution `Array<Integer>`.

- **predicate_method**: If a method name ends with `?`, guess solution `%bool` (RDL’s boolean type) for the method return type.

- **string_name**: If a variable name ends with `name`, guess solution `String`. 

105
• HASH_ACCESS: Like all other values, hashes are objects in Ruby, not built-in constructs. As such, they are accessed using the method [], and written to using the method []=. This rule states that, if all of an argument type variable’s upper bounds are structural types consisting of the methods [] and []=, and all of the keys given to these methods are symbols, then guess the solution that is the finite hash type consisting of the keys and the unions of corresponding assigned values for these methods. For example, if an argument type variable α had constraints \( \alpha \leq [\cdot][] = : \text{id} \rightarrow \text{Integer} \), \( \alpha \leq [\cdot][] = : \text{id} \rightarrow \text{String} \), and \( \alpha \leq [\cdot][] = : \text{name} \rightarrow \text{String} \), then the solution for \( \alpha \) would be the finite hash type \{ \text{id}: \text{String} \cup \text{Integer}, \text{name}: \text{String} \}. This type says that \( \alpha \) is a hash mapping the symbol \text{id} to a \text{String} or \text{Integer}, and \text{name} to a \text{String}.

As discussed in § 4.3.1, InferDL treats type variables, union, intersection, structural, and bottom and top types as overly-general. In our implementation, we also treat \text{Object} and \text{nil} (which were omitted from the formalism for brevity but are almost the same as the top and bottom types, respectively) as overly-general. In addition to nominal types, RDL also includes several additional kinds of types that we treat as sufficiently precise: generic types (which are parameterized nominal types), finite hash and tuple types (which are more precise versions of \text{Hash} and \text{Array} types), and singleton types (such a type has only one value as an inhabitant).

Choice Types. Ruby methods often have intersection types, which pose a challenge for type inference. Consider the \text{Array} indexing method [], which has the following type in RDL:
Here, t is the type parameter for the Array class. When given an Integer index, [] returns a single element, and when given a Range<Integer> corresponding to multiple indexes, [] returns the subarray of elements at those indexes. Now consider the following contrived code snippet:

```python
def foo(x) arr = [1,2,3]; return arr[x] + 1; end
```

Suppose that InferDL assigns x the type variable α. Then, when analyzing the call arr[x], we encounter a problem: During constraint generation, we do not know α’s solution. One choice would be to assume both arms of the intersection are possible. However, then the result of the method call would have type Integer ∪ Array<Integer>, which leads to a type error when analyzing the larger expression arr[x] + 1, since we cannot add an Array to an Integer.

To address this issue, we introduce choice types, a type system feature loosely inspired by variational type checking [89]. A choice type, written Choice_i(τ_1, …, τ_n), represents a choice among the types τ_j. Each choice type also has a label i. During inference, if one τ_j of a choice type would result in a type error, then arm j is eliminated from all choice types with the same index i.

In the example above, the call to arr[x] would result in the constraint

(1) $\alpha \leq \text{Choice}_1(\text{Integer}, \text{Range<Integer>})$

because InferDL reasons that it has a choice between the two input types of Array’s [] method. Additionally, the return type of arr[x] would be

(2) $\text{Choice}_1(\text{Integer}, \text{Array<Integer>})$
representing both possible returns. Both choice types have the label 1, indicating that they are decided together. Then, when type checking the call \texttt{arr[x] + 1}, \texttt{InferDL} would recognize that the \texttt{Array<Integer>} arm of type (2) results in a type error, and it would eliminate that arm from both (2) and (1). Effectively, this would retroactively make the return type of \texttt{arr[x]} be the sole type \texttt{Integer}, and it would allow us to infer \( \alpha \)’s solution as the sole type \texttt{Integer}. If \texttt{InferDL} ever eliminates all arms of a choice type, it raises a type error.

\textbf{Library Types.} \texttt{RDL} comes with type annotations for Ruby’s core and standard libraries, as well as for common Rails methods and methods from \texttt{Sequel}, a popular framework for database queries. However, it is common for Ruby programs to make extensive use of other third-party libraries as well. Typically, a type checker would require type annotations for any such methods used in the subject program. But writing these type annotations is burdensome and often requires knowledge of the library’s implementation. This task is all the more tedious in the context of type inference, where the programmer aims to infer type annotations, not write them.

\texttt{InferDL}’s approach to library types avoids this issue. During constraint generation, if \texttt{InferDL} encounters a call to a method that both lacks a type annotation and is not itself the target of inference, \texttt{InferDL} finds the method definition to determine the method’s arity. \texttt{InferDL} then creates a type signature for the method with fresh type variables for the return and each argument. If no method definition is found, \texttt{InferDL} raises a type error.

This approach is similar to type inference for a method, except we do not
generate constraints from the method body. Thus, type inference might be unsound, producing a solution that would be impossible if we knew the library method’s implementation. However, in practice, we found this approach was essential for allowing us to apply inference to each new benchmark, and it also helped us discover a previously unknown bug in one of our benchmarks (§ 4.5.1).

Variable Types. In addition to method types, InferDL can infer type annotations for the three non-local kinds of Ruby variables: global, class, and instance variables. Recall that standard_solution (§ 4.3.1), used as a subroutine in InferDL, aims to infer most general types for methods. However, observe that variables are both read from and written to, i.e., for a field @x, there is conceptually a getter of type ⊥ → α and a setter of type α → ⊥. Notice that the α appears both co- and contravariantly. Hence, unlike method arguments and returns, it is not the case that a least or greatest solution will always be most general.

Instead, to extend standard_solution to variables, we take an ad-hoc approach: we take the intersection of the upper bounds on a variable’s type when it has upper bounds and, if not, we take the union of its lower bounds. We found this approach works reasonably well in practice, and we apply the same heuristic rules, with the same definition of overly-general, as for method types.

4.5 Evaluation

We evaluated InferDL on four Ruby on Rails web apps:

- Journey [72] is a web app that provides a graphical interface to create surveys
and collect responses from participants.

- **Discourse** [69] is an open-source discussion platform built on Rails.

- **Code.org** [71] is a Rails app that powers code.org, a programming education website.

- **Talks** [94] is a Rails app written by one of the authors for sharing talk announcements.

We chose these apps because they have all been used as type checking benchmarks in prior work [5, 90]. Thus, we could use the previously written type annotations for these apps as "gold standards" to compare against. We inferred type annotations for all methods and global, class, and instance variables for which type annotations existed in prior work. This includes both methods (and the variables they used) that were type checked in prior work and those that were not checked, but were annotated to assist in the type checking of other methods.

The only additional type annotations used were for the special Rails `params` hash, which contains values that come from a user’s browser. The `params` hash always maps symbols to various types of objects. Without annotations, InferDL typically infers that `params` has type `Hash<K, V>`, where `V` was the union of all observed value types for the hash. This effectively treats all values from the hash as belonging to the same type, which causes false positive type errors during inference. Rather than add type casts for these cases, we instead used the type annotations for `params` from the prior type checking work [5, 90]. In the future, we plan to incorporate special handling of the Rails `params` hash to avoid this issue.
Table 4.1: Type inference: number of targets and time performance.

Below, we discuss the results of our evaluation. We note that our results are preliminary, and further work is needed to affirm they generalize beyond our benchmarks, in particular for detecting type errors in real-world programs.

4.5.1 Results

Table 4.1 summarizes some of our type inference results. The first column gives the number of methods we inferred types for, totaling 250 methods across the four apps. The subsequent group of three columns counts the number of types we inferred. The first of these columns, Meth Typs, counts the number of method argument and return types inferred. We count each argument and return type separately so that we can more precisely evaluate InferDL’s performance. The second of these columns, Var Typs, shows the number of global, class, and instance variable types inferred. Finally, the Total Typs column counts the total number of types inferred, i.e., Meth Typs + Var Typs.

The next column shows the number of type casts we had to write to run InferDL on each app, without which InferDL would raise false positive type errors. Almost all of these type casts were needed when handling heterogeneous data structures like arrays and hashes, because there are cases where InferDL cannot
determine the type of a value accessed from one of these data structures.

The subsequent column reports InferDL’s running time on a 2014 MacBook Pro with a 3GHz i7 processor and 16GB RAM. We give the time as the median and semi-interquartile range (SIQR) of 11 runs. For comparison, we provide InferDL’s runtime when using the heuristics presented in § 4.4 (shown under “heur”), and when not using any heuristics (shown under “std”). In total, InferDL took 31.91s to run on all benchmarks when using heuristics, with an SIQR of just 0.95s, indicating little variance across runs. By comparison, when not using any heuristics, InferDL took 8.68s to run on all benchmarks. Upon closer examination, we found that approximately 75% of InferDL’s runtime when using heuristics was spent on just one rule, STRUCT-TO-NOMINAL. The rule involves searching through the space of all existing classes, and for each one, searching through the names of all its methods. This can be quite expensive for larger programs. We found we could achieve speedups by caching search results, and by building a mapping from method names to the classes that implement them in advance of running the rule. Nevertheless, this remains an expensive operation.

Table 4.2 reports the remainder of our inference results. In particular, it

<table>
<thead>
<tr>
<th>Program</th>
<th>Correct Meths</th>
<th>Correct Vars</th>
<th>Correct Total</th>
<th>Heuristic Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>heur/std</td>
<td>heur/std</td>
<td>heur/std</td>
<td>STN/Name/Hash</td>
</tr>
<tr>
<td>Journey</td>
<td>33 / 30</td>
<td>19 / 13</td>
<td>52 / 43</td>
<td>0 / 13 / 0</td>
</tr>
<tr>
<td>Discourse</td>
<td>61 / 47</td>
<td>0 / 0</td>
<td>61 / 47</td>
<td>3 / 42 / 1</td>
</tr>
<tr>
<td>code.org</td>
<td>111 / 60</td>
<td>10 / 10</td>
<td>121 / 70</td>
<td>0 / 80 / 5</td>
</tr>
<tr>
<td>Talks</td>
<td>127 / 102</td>
<td>38 / 28</td>
<td>165 / 130</td>
<td>7 / 58 / 3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>332 / 239</strong></td>
<td><strong>67 / 51</strong></td>
<td><strong>399 / 290</strong></td>
<td><strong>10 / 193 / 9</strong></td>
</tr>
</tbody>
</table>

Table 4.2: Type inference: assessing inferred types.
counts the number of types InferDL inferred correctly—the same as or more precise than the original type annotation—both with and without the use of heuristics. To determine whether inference results were correct, we automatically counted those cases where an inferred type matched the original annotation exactly, and we used manual inspection when they differed. For example, if the annotation for a type was \%any (RDL’s top type), and InferDL inferred Integer, we would count this as a more specific type. In our experience, we didn’t find any case where InferDL predicted a more specific type that was not an accurate reflection of programmer intent.

The first of these columns gives the number of method argument and return types correctly inferred for heuristic and standard inference. For example, InferDL correctly inferred 332 out of 411 total argument and return types for all apps when using heuristics, compared to just 239 correct types when performing standard inference. The next column gives the number of variable types correctly inferred, and finally, the Correct Total column gives the number of total types inferred correctly. As shown, the use of heuristics enables InferDL to infer about 22% more correct type annotations, a significant improvement. We found this percentage was fairly consistent across the benchmarks, indicating that the heuristics we used were not specific to one app, but rather captured some more common, general properties. We also found this improvement was approximately the same for types of global, class, and instance variables, and types of method inputs/outputs, indicating our approach to variables (discussed in § 4.4) is effective.

Note that inferred types which do not fall under the “Correct Types” column are not necessarily “incorrect”—typically, these types are simply more general than
the original, programmer-written annotation. For example, across the apps there were a number of cases where \texttt{InferDL} inferred the type \texttt{Array}<\alpha> for some type variable \(\alpha\), when the programmer’s annotation was a variable-free type (e.g., \texttt{Array<String>}).

In our subjective experience, many types \texttt{InferDL} failed to infer (with or without heuristics) were for arrays and hashes. This is largely because \texttt{RD\textsubscript{L}} treats Array and Hash types as invariant in their type parameters. This means, e.g., the constraint \texttt{Hash<String, Integer> \leq Hash<String, Object>} is invalid, since the type parameters are not equivalent. This leads to many potentially correct types being rejected due to the conservatism of type invariance. We are interested in exploring better approaches to type inference for heterogeneous data structures as future work.

Finally, the last column shows the number of times a heuristic successfully found a type for each app, that is, the heuristic’s guess was actually used as a solution. For brevity, we present the counts for all of the six name-based heuristics (\texttt{is\_model}, \texttt{is\_pluralized\_model}, \texttt{int\_names}, \texttt{int\_array\_name}, \texttt{predicate\_method}, and \texttt{string\_name}) under a single column \texttt{Name}, while the \texttt{STN} column gives the count for \texttt{struct\_to\_nominal}, and the \texttt{Hash} column for \texttt{hash\_access}. It is clear that the name-based rules were by far the most useful heuristics for inferring types, comprising a total of 193 of the successful heuristic applications. Of those 193 applications, 103 were of the \texttt{predicate\_method} rule. Overall, this suggests that variable and method names are a strong indicator of intended types.
Error Caught. In the process of inferring types, we discovered a previously unknown bug in the Journey app. This was particularly surprising because the method it was found in was already type checked in prior work [90]. The bug existed in the following code, which creates and saves a new person:

```ruby
begin
  invitee = IllyanClient::Person.new (:person =>{:email =>email})
  invitee.save
  ...
rescue
  logger.error "Error during invite."
  ...
end
```

The bug arises because there is no `save` method for the `IllyanClient::Person` class, so the call `invitee.save` always raises an error. Moreover, because this error exists within a `begin...rescue` clause, the bug will never be directly seen at runtime since control will always pass to the `rescue` clause. The bug could potentially have been detected via manual programmer inspection of the error log, though it never was. We confirmed this bug with the Journey developer. This bug was not caught by type checking in prior work because the programmer who wrote type annotations in that work wrongly assumed that the `IllyanClient::Person#save` method did exist. Thanks to InferDL’s handling of library types (§ 4.4), the same mistake was not made here.

4.5.2 Case Studies

To further evaluate InferDL, we applied it to five additional Ruby libraries and one additional Ruby app:
<table>
<thead>
<tr>
<th>Program</th>
<th>Num Meths</th>
<th>Meth Types</th>
<th>Var Types</th>
<th>Total Types</th>
<th>Total Typs</th>
<th>Type Casts</th>
<th>Time (s) Median (\pm) SIQR</th>
<th>Heuristic Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM</td>
<td>148</td>
<td>275</td>
<td>62</td>
<td>337</td>
<td>5</td>
<td>1.71 ± 0.06 / 0.80 ± 0.07</td>
<td>29 / 103 / 0</td>
<td></td>
</tr>
<tr>
<td>Diff-LCS</td>
<td>80</td>
<td>187</td>
<td>40</td>
<td>227</td>
<td>23</td>
<td>2.65 ± 0.02 / 2.38 ± 0.06</td>
<td>20 / 14 / 0</td>
<td></td>
</tr>
<tr>
<td>MM</td>
<td>79</td>
<td>166</td>
<td>13</td>
<td>179</td>
<td>7</td>
<td>0.57 ± 0.15 / 0.26 ± 0.01</td>
<td>4 / 10 / 0</td>
<td></td>
</tr>
<tr>
<td>Optcarrot</td>
<td>430</td>
<td>763</td>
<td>367</td>
<td>1130</td>
<td>48</td>
<td>38.9 ± 2.64 / 78.6 ± 17.4</td>
<td>204 / 22 / 1</td>
<td></td>
</tr>
<tr>
<td>Sidekiq</td>
<td>344</td>
<td>623</td>
<td>96</td>
<td>719</td>
<td>12</td>
<td>3.28 ± 0.56 / 2.12 ± 0.13</td>
<td>37 / 63 / 3</td>
<td></td>
</tr>
<tr>
<td>TZInfo</td>
<td>251</td>
<td>511</td>
<td>57</td>
<td>568</td>
<td>9</td>
<td>3.15 ± 0.25 / 5.06 ± 0.07</td>
<td>118 / 42 / 0</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1332</strong></td>
<td><strong>2525</strong></td>
<td><strong>635</strong></td>
<td><strong>3160</strong></td>
<td><strong>104</strong></td>
<td><strong>50.22 ± 3.67 / 89.2 ± 17.8</strong></td>
<td><strong>412 / 254 / 4</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Case Study Inference Results. Program name “AM” is short for Active Merchant, and “MM” is short for MiniMagick

- **Active Merchant** [95], a payment abstraction library.

- **Diff-LCS** [96] a library for generating difference sets between Ruby sequences.

- **MiniMagick** [97], an image processing library.

- **Optcarrot** [98], a Nintendo Entertainment System (NES) emulator implemented in Ruby and intended as a benchmark for runtime performance evaluation.

- **Sidekiq** [99], a background job handler library.

- **TZInfo** [100], a time management library.

Because we do not have gold standard type annotations for these programs, we refer to these experiments as case studies. With the exception of Optcarrot, we picked these programs because they are all highly popular, well-maintained, and well-tested. We chose Optcarrot because of its intended use as a Ruby benchmark and because, as an emulator, it relies heavily on binary arithmetic, which distinguishes it from other Ruby programs we looked at.
We ran inference for all methods defined in these programs, excluding methods that use features not supported by RDL; the most common unsupported feature was mixin methods. We also excluded methods defined in the Active Merchant payment gateways, a set of 215 distinct payment gateways comprising over 60,000 lines of code. Running InferDL for this many lines of code would have required a significant manual effort to add type casts to circumvent false positive errors, so we decided to leave them out of our case study. We discuss the issue of type casts further below.

**Using InferDL.** It would be tedious and time-consuming to call InferDL’s infer method (§ 4.4) on every method in our subject programs. Instead, we used InferDL’s infer_file and infer_path methods, which take a file or path, respectively, as an argument and then call infer on every method statically defined in that file or path. We called these methods for all code in a program’s lib/ directory, which by convention holds the program’s implementation (and excludes testing code, code for handling dependencies, etc.).

The first time InferDL runs on a new subject program, it often reports type errors. We manually inspected and addressed each type error, iterating until none remained. Overall, the errors found by InferDL fell into three categories:

- True errors resulting from bugs in the program. We discuss these below.

- False positives due to InferDL’s conservatism. We inserted appropriate type casts to suppress these type errors. We discuss type casts below.

- Errors resulting from features unsupported by InferDL. As mentioned earlier,
we exclude such methods from future rounds of inference.

As an aside, we note that currently, it can sometimes be difficult to find the underlying cause of a type error reported by InferDL. If an invalid constraint is generated during resolution, InferDL reports the invalid constraint and the line number origins of the left- and right-hand sides of the constraint. But often these constraints were generated through a series of propagations resulting from many different places in the code, so their origins do not always reveal the underlying cause. In the future, we hope to incorporate ideas from prior work on diagnosing type inference errors [101, 102, 103].

Results. Table 4.3 contains the results of running InferDL on the case study apps. This table includes the same columns as Tables 4.1 and 4.2, excluding the “Correct” columns. In total, we inferred types for 1,332 methods constituting 2,525 individual arguments and returns, and 635 global, class, or instance variables, for a total of 3,160 individual types.

We wrote 104 total type casts to run inference for these programs, or approximately one type cast for every 30 types we inferred. In addition to the need for type casts when accessing values from heterogeneous data structures (discussed in § 4.5.1), we encountered many cases where type casts were necessary for path-sensitive typing. For example, consider the code snippet in Figure 4.4, simplified from the TZInfo library.

This snippet refers to the instance variable @transitions, which has type Array<TimezoneTransition>. On line 2, we enter a loop for values of i=index down
Figure 4.4: A code snippet from the TZInfo library.

to i=0. On line 3, we use Ruby’s ternary operator to conditionally assign the variable `start_transition` to either a `TimezoneTransition` or to `nil`. Then, on line 5, we use the ternary operator again, this time with the variable `start_transition` as our condition. In Ruby, the value `nil` is falsey. Thus, on line 5, we only evaluate the expression `start_transition.offset` if `start_transition` is non-nil. This call is safe, because `TimezoneTransition` has a method `offset` defined.

However, InferDL does not know that `start_transition` is non-nil because it has limited support for path-sensitive typing. It will conservatively reason that `start_transition` may be `nil` on line 5 and thus raise a type error. To avoid this issue, we insert the following type cast for the call to `offset`:

```
RDL.type_cast(start_transition, TimezoneTransition).offset
```

This notifies InferDL that `start_transition` is a `TimezoneTransition` when `offset` is called. Such path-sensitive logic enables libraries to be maximally flexible for clients. We leave handling these cases without type casts to future work.

The next column of Table 4.3 reports the median time and SIQR taken across 11 runs of InferDL, when using vs. not using heuristics. Interestingly, on these apps InferDL actually took less total time when using heuristics compared to not
using them. This was attributable to two apps in particular, *Optcarrot* and *TZInfo*, that took $2 \times$ and $1.6 \times$ as long, respectively, when not using heuristics. This can occur due to the way that *InferDL* performs type inference. As shown in § 4.3.1, *InferDL* will repeatedly perform constraint resolution and solution extraction until no new constraints are generated. In some cases, heuristics may lead to solutions for type variables earlier on in this process, thereby helping to reach a constraint set fixpoint sooner. For instance, *Optcarrot* performed just 6 rounds of solution extraction when using heuristics, compared with 15 rounds of solution extraction when not using heuristics; for *TZInfo*, the numbers were 3 and 9, respectively.

Finally, we report the number of successful applications of heuristics for inferring types. Notably, the *struct-to-nominal* heuristic is far more useful for our case study programs than for the Rails apps in Table 4.2. It was used 412 times when running inference for 3,160 total types in our case studies, versus just 10 times for 496 total types for the Rails apps. This disparity is at least partly attributable to the order in which heuristics are run (§ 4.4). *struct-to-nominal* is applied after the rules *is_model* and *is_pluralized_model*. But the latter two rules are only used for Rails apps, meaning *struct-to-nominal* is applied third for Rails apps, and first for non-Rails apps. We tried re-running *InferDL* on the Rails apps with *struct-to-nominal* ordered first, and found it was applied 28 times for the Rails apps, which at least partly closes the gap with non-Rails apps.

We also more closely examined the uses of *struct-to-nominal* across all 10 programs in § 4.5.1 and § 4.5.2, and we found that approximately 14% of the time the heuristic produced a union type, while the remainder of the time it produced
just a single, nominal type. The usefulness of the produced union types varied. Sometimes, the unions were quite sensible. For example, in the TZInfo program, STRUCT-TO-NOMINAL produced the type TZInfo::Timestamp ∪ Time as the solution for a number of variables. Both the TZInfo::Timestamp and Time classes represent time values, and many of TZInfo’s methods are implemented to handle objects from both of these classes, so this is a sensible solution. In other cases, we found that STRUCT-TO-NOMINAL produced unions of unrelated classes that happened to have some same-named methods, thereby producing a solution that is less useful.

Name-based heuristics were also useful for our case studies, having been applied 254 times to infer types. However, this clearly comprises a far smaller proportion of uses than for the Rails apps. This may be because Rails emphasizes the principle convention over configuration, making names more important than in regular Ruby programs. Moreover, libraries are very domain-specific, and the names used in these programs reflect their domain. For example, TZInfo features many variables with names like time, datetime, timezone, etc. It is challenging to write general-purpose heuristics that can capture such domain-specific naming.

Finally, note that the HASH_ACCESS rule was used only four times across the programs in Table 4.3, and only nine total times across the programs in Table 4.2. This is partly attributable to the invariance of hashes (as discussed in § 4.5.1). Though the rule was applied few times in practice, we still believe it was useful in the cases it was used for converting structural type solutions to a more readable finite hash type. For example, for one type variable in the code.org app, InferDL used the HASH_ACCESS rule to infer the finite hash type solution \{ id: Integer, email:
String, gender: α } (some key/value pairs omitted for brevity), rather than a much larger and more difficult to read intersection of structural types.

**Errors Found.** InferDL found five previously unknown bugs in the case study programs, all of which were confirmed with the developers:

- In *Active Merchant*, InferDL caught a reference to an undefined constant *Billing::Integrations*.

- In *Sidekiq*, InferDL caught a reference to an undefined identifier e inside a rescue clause that was as follows:

  ```ruby
  rescue Exception => ex
  ...
  raise e
  end
  ```

  The notation `rescue Exception => ex` catches exceptions of type `Exception` and binds the specific Ruby exception object to `ex`. This rescue clause was meant to perform some error handling (elided with the ... above) and then raise the original error, but it erroneously referred to an undefined `e` rather than `ex`.

- In *Sidekiq*, InferDL caught a reference to an undefined constant `UNKNOWN`.

- In *Diff-LCS*, InferDL caught two different calls to an undefined method `Diff::LCS.YieldingCallbacks`.

- In *Diff-LCS*, InferDL caught a reference to an undefined constant `Text::Format`.

  We do note that given the nature of the above bugs (undefined methods, variables, and constants), it is possible that they could be found through alternative
analyses. Nevertheless, InferDL’s ability to catch these errors in popular and well-tested libraries indicates it is useful not only for generating type annotations, but also for catching type errors.

### 4.5.3 Testing Implementation Choices

Finally, in § 4.4 we discussed two novel design features of InferDL: the use of choice types for resolving calls to overloaded methods and InferDL’s handling of calls to library methods for which we do not have types. To evaluate these choices, we provide some relevant data in Table 4.4 collected from all 10 of the programs discussed in § 4.5.1 and § 4.5.2.

For each program, the first column gives the number of choice types used while running InferDL on the program. As discussed in § 4.4, a choice type is used when type checking a call to an overloaded method, when InferDL is not able to determine which type of the method to use. They can help avoid false positive type errors (and thus reduce the need for type casts), and to infer more precise types.

<table>
<thead>
<tr>
<th>Program</th>
<th>Choice Type Uses</th>
<th>Unknown Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Journey</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>Discourse</td>
<td>1</td>
<td>33</td>
</tr>
<tr>
<td>code.org</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>Talks</td>
<td>2</td>
<td>56</td>
</tr>
<tr>
<td>Active Merchant</td>
<td>1</td>
<td>69</td>
</tr>
<tr>
<td>Diff-LCS</td>
<td>124</td>
<td>14</td>
</tr>
<tr>
<td>MiniMagick</td>
<td>1</td>
<td>33</td>
</tr>
<tr>
<td>Optcarrot</td>
<td>154</td>
<td>73</td>
</tr>
<tr>
<td>Sidekiq</td>
<td>7</td>
<td>82</td>
</tr>
<tr>
<td>TZInfo</td>
<td>14</td>
<td>41</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>307</strong></td>
<td><strong>451</strong></td>
</tr>
</tbody>
</table>

Table 4.4: Testing Implementation Choices.
total, we used 307 choice types across all benchmarks. The vast majority of these uses were in just two programs, *Diff-LCS* and *Optcarrot*. This is likely because these programs rely heavily on array manipulation, and therefore make frequent use of the Array accessing method [], which requires choice types to resolve its overloaded method type (see § 4.4 for an example). Thus, we found that the need for choice types commonly arises in programs, but they are especially useful for programs that make frequent use of overloaded methods.

The next column gives the number of uses of “Unknown Types”—this is the name we give to the method types composed entirely of type variables that we generate for library methods and other methods for which we do not have a type. In total, we used 451 unknown types across all programs. Without our approach of generating unknown types, we would have had to write a type annotation in every one of these cases so that InferDL could type check the programs. Thus, we believe InferDL’s approach to handling calls to methods without a type is effective for reducing the programmer’s annotation burden.

4.6 Related Work

Researchers have been studying type inference for many decades. Traditionally, the problem is posed as follows: given a program without type annotations, can we determine the most general type for each expression in the program, and rule out any type errors? The problem was first formulated and solved by Curry and Feys [104] for the simply typed lambda calculus. Perhaps most famously, Hindley [105],
Milner [106], and Damas and Milner [107] developed an approach known today as Hindley-Milner-Damas type inference. Its central algorithm, Algorithm W, works by generating constraints on type variables, then resolving those constraints through a process known as unification.

Researchers have extended type inference to subtyping systems [1, 6, 91, 108] as well, and this has enabled the development of practical type inference systems for dynamic languages such as Python [11], JavaScript [14, 16], and Ruby [4]. As discussed earlier, these systems are often aimed at catching type errors rather than displaying the results of type inference to the user, and in our experience, the types inferred by such systems can be quite hard to understand. In contrast, InferDL aims to infer usable types.

Furr et al. [4] present DRuby, a static type inference system for Ruby, which features an expressive type language including intersection, union, optional, and structural types. While DRuby also focuses exclusively on finding type errors in programs, many of the type system features it includes are part of RDL [20], on which InferDL is built.

Finally, there are a number of probabilistic type inference systems which aim to infer usable type annotations. We direct the reader to Section 5.6 for a discussion of these systems.
4.7 Conclusion

We presented InferDL, a novel type inference system for Ruby. In addition to uncovering type errors, InferDL aims to produce useful type annotations for methods and variables. Because the constraint-based approach to type inference often results in types that are overly-general, InferDL incorporates heuristics that guess a solution for type variables that better matches what a programmer would write. InferDL enforces the correctness of heuristic guesses by checking them against existing constraints. Moreover, heuristics are not baked-in to InferDL but rather provided as code blocks, making InferDL highly configurable.

We formalized the type and constraint language of InferDL and provided the rules and procedures for resolving type constraints, producing standard type solutions, and using heuristics to produce more useful, sound type annotations. We implemented InferDL on top of RDL, an existing Ruby type checker which we extended with support for constraint generation, heuristics, and choice types to handle overloaded methods. We also discussed the eight heuristics we found useful in applying InferDL to programs.

Finally, we evaluated InferDL by applying it to four Rails apps for which we already had type annotations. We found that, without using heuristics, we were able to correctly infer about 58% of all type annotations for these apps, and when using heuristics we were able to infer 80% of annotations. We also applied InferDL to six additional case study Ruby programs. Across the Rails apps and the case study apps, InferDL discovered six previously unknown bugs. Thus, we believe that
InferDL is an effective type inference system and represents a promising approach to generating useful, correct type annotations.
Chapter 5: SimTyper: Sound Type Inference for Ruby using Type Equality Prediction

The last chapter introduced InferDL, a type annotation inference system that complements constraint solving with heuristic rules. While the evaluations of InferDL showed that heuristics were useful for inferring more annotations that match what a programmer would write, there remains room for improvement. First, there were still methods and variables in the benchmarks for which InferDL failed to infer matching type annotations. Second, seven out of eight of the heuristic rules used by InferDL were written while applying it to the benchmarks. In this chapter, we show that, while these heuristics perform well on their initial target programs, they sometimes generalize poorly to new programs. While we could write new heuristic rules for each new target program, this process can be time-consuming.

In this chapter we introduce SimTyper, a type inference system that builds on InferDL by adding type equality prediction, a novel machine learning-based approach to inferring useful type annotations. We evaluated SimTyper on eight Ruby programs and found that, compared to standard constraint-based inference, SimTyper finds 69% more types that match programmer-written annotations. Moreover, we found that when using type equality prediction (without the use of heuristics),
SimTyper inferred 19% more matching type annotations than InferDL. When running exclusively on benchmarks that were not included in the previous chapter (and thus had not been incorporated in the development of InferDL’s heuristics), that number improved to 42%.

5.1 Introduction

Many researchers have explored ways to add static types to dynamic languages, aiming to provide the benefits of static typing while preserving the flexibility of the language [1, 2, 3, 5, 6, 13, 53, 58, 90]. In this setting, type inference [4, 11, 14, 16, 109] is potentially very attractive, as it can catch type errors without requiring users of these retrofitted type systems to provide many new type annotations. Typically, traditional static type inference (see § 5.6 for a discussion of machine learning-based type inference systems) aims to infer most general types, so as not to reject any program that would be statically typable. Unfortunately, as demonstrated in Chapter 4, in the presence of complex type features such as subtyping, structural types, and union types—which are often needed to type existing dynamic language code—most general types can be verbose, confusing, and difficult to understand. For example, when typing a Ruby method that performs arithmetic computations on an argument \( x \), the most general type of \( x \) might be a structural type like any object with +, -, *, and / methods. In contrast, a programmer would much more likely write the shorter, simpler, and more understandable nominal type Numeric, giving up some generality for the sake of usability.
In this chapter, we introduce SimTyper, a type inference system that aims to infer usable types for Ruby. SimTyper is built on top of InferDL (Chapter 4), a type inference system that combines constraint solving with manually written heuristics whose guesses are checked against the constraints, to ensure soundness \cite{110}. SimTyper uses the same basic infrastructure, but adds a novel *deep similarity* (Deep-Sim) network that performs *type equality prediction*. More precisely, DeepSim predicts type similarity scores among method arguments. If, after standard type inference, an argument has an *overly general* type (e.g., a structural type) and DeepSim predicts that argument is similar to another argument with a *usable* type (e.g., a nominal type), SimTyper guesses the overly general type can be replaced by the usable type. If that guess is consistent with the rest of the constraints, it is kept. Otherwise it is discarded, and further guesses are made up to some bound. Thus, even though it is probabilistic, SimTyper is still guaranteed to be sound. SimTyper applies the same idea to method returns and to instance, class, and global variables. (§ 5.2 shows how SimTyper integrates standard type inference, heuristics, and type equality prediction.)

We describe SimTyper’s algorithm on a core language of types and constraints. SimTyper begins by running the standard, constraint-based type inference algorithm which, at a high level, generates constraints among types; applies constraint resolution to check that the constraints are consistent; and then extracts solutions for the type variables. Next, for each type variable $\alpha$ with an overly general solution, SimTyper finds the type variable $\beta$ that is most similar to $\alpha$ and has a usable type solution. SimTyper then adds a constraint $\alpha = \tau$, where $\tau$ is $\beta$’s solution, and runs
constraint resolution again. If the constraints are still consistent, $\alpha$’s solution is set to $\tau$. If not, SimTyper retracts the $\alpha = \tau$ constraint and tries the next most similar usable type, and so on. This guessing-and-backtracking approach was first proposed for InferDL, and SimTyper uses the same machinery, enabling SimTyper to infer types with both DeepSim-based predictions and guesses based on InferDL heuristics. (§ 5.3 describes SimTyper’s inference algorithm.)

The DeepSim network itself takes as input the tokenized source code of two methods and the positions within that code of the arguments or method return sites of interest. The network then uses CodeBERT, a transformer-based pre-trained code embedding model [111], to transform each token at the given positions into a fixed-dimensional vector that captures both the token and its surrounding context. DeepSim then averages those vectors to produce one vector for each input. These vectors are then passed through a trained similarity function to predict whether they are similar or dissimilar. The network itself is trained on 371 Ruby programs that include YARD [112] documentation. We extract type information from YARD (which is not checked against code and hence might be noisy) to create a training data set with 100,000 pairs labeled as either similar (for two positions with the same YARD types) or dissimilar. (§ 5.4 describes the DeepSim network in detail.)

We evaluated SimTyper by applying it to eight Ruby programs with type information: four web apps written in Ruby on Rails (a popular web development framework) that InferDL was previously evaluated on, and four popular Ruby libraries with YARD documentation. We then compared the types inferred by SimTyper to existing, programmer-written annotations. Following prior work [113], we count
the number of cases where the inferred type matches the existing type, as well as the number of matches up to parameter—generic types where the base matches but the parameter is different (e.g., inferring `Array<Integer>` when the original was `Array<String>` would fall in this category). We found that, by combining constraint solving, InferDL heuristics, and type equality prediction from DeepSim, SimTyper generated 66% more type annotations that matched programmer-written types compared to using constraint solving alone. If we include matches up to parameter, this number improves to 69%. Moreover, SimTyper inferred 16% more matching type annotations when using DeepSim alone than when using heuristics alone. Including matches up to parameter improves this to 19%. DeepSim was also able to correctly predict rare types, including 16 types that did not appear in either the standard Ruby library or the training data. This number is the same whether or not we consider matches up to parameter. (§ 5.5 describes these results and several other experiments in more detail.)

In summary, we believe that by incorporating type equality prediction with DeepSim, SimTyper takes an important step forward towards type inference systems that produce more usable types.

5.2 Overview

We begin by illustrating how SimTyper is used to infer a type annotation for the method shown in Figure 5.1a, which is extracted and simplified from TZInfo, one of the benchmarks in our experiments (§ 5.5). The method, Timestamp.create,
class Timestamp
  # Assigned method type: (α, β, γ) → δ
  def self.create (year, month, day)
    raise ArgumentError, 'year must be an Integer' unless year.kind_of?(Integer)
    raise ArgumentError, 'month must be an Integer' unless month.kind_of?(Integer)
    raise ArgumentError, 'day must be an Integer' unless day.kind_of?(Integer)
    after_february = month > 2
    year = year - 1 unless after_february
    era = year / 400 # eras are 400 year periods
    day_of_year = day + (153 * (month + ... # additional computation
    value = ... * 24 * 60 * 60 # seconds since unix time
    new(value)
  end
end

(a) A method from the TZInfo library.

(b) Constraints generated on type variables.

Figure 5.1: Generating type constraints in TZInfo.
takes three integers representing a year, month, and day corresponding to a date, and returns a new \texttt{Timestamp} encoding that date as the number of seconds since the start of the Unix Epoch (midnight on 01/01/1970).

The first three lines of the method check that all parameters are integers.\footnote{Amusingly, we could in theory create a heuristic (§ 5.2.2) that uses this coding pattern to guess the argument types. That heuristic, however, would have to be hand-written and would be specific to this pattern. One strength of \texttt{SimTyper} is that it can discover useful types even without manually creating heuristics.}

The subsequent lines (lines 8–13) perform the computation. The details of the computation [114] are not important, but notice the parameters appear in various places in somewhat complex arithmetic expressions. The method returns the value of the expression on the last line, which constructs a new \texttt{Timestamp} (code omitted here) with the computed number of seconds.

5.2.1 Standard Type Inference

The first step of \texttt{SimTyper}, inherited from \texttt{InferDL}, is to perform standard, constraint-based type inference [4]. We refer the reader to Chapter 4 for a more detailed treatment of standard type inference; here, we quickly go over the details.

We begin by assigning a type variables $\alpha$, $\beta$, and $\gamma$ to the parameters (in that order) and $\delta$ to the return type. \texttt{SimTyper} then analyzes the method body to generate constraints on these type variables. Figure 5.1b shows several of the constraints generated for this example. Constraint (1), from line 4, states that $\alpha$, the type of \texttt{year}, must define a method \texttt{kind\_of?} that takes a \texttt{Class} and returns a fresh unknown type $\epsilon$. Or, more formally, $\alpha$ is a subtype of the structural type $[\texttt{kind\_of?} : \texttt{Class} \to \epsilon]$. Constraints (2) and (3) are similar.
Constraint (4), from line 8, states that `month` must define a `>` method that takes a `Number` (for simplicity, all Ruby numeric objects are typed as `Number` in SimTyper) and returns an unknown type. Note that in Ruby, binary arithmetic operations are actually method calls on the left-hand argument, e.g., `month > 2` is syntactic sugar for `month.(2)`. Constraints (5)–(8) are similar, with the first two arising from lines 9 and 10, respectively, and the last two from line 11. Finally, constraint (9) arises from returning the newly created `Timestamp` on line 15.

Next, SimTyper performs constraint resolution, which applies a set of constraint rewriting rules until reaching saturation. In this particular case, because `Timestamp.create` is relatively simple and is considered in isolation, resolution does not change the set of constraints.

After constraint resolution, the constraints are in solved form [91], meaning SimTyper can read off a type variable’s *most general solution*—any other solution would be more restrictive—by looking at its immediate bounds. As seen in Chapter 4, for (contravariant) method arguments, we compute the *greatest solution* by intersecting all of the argument type’s upper bounds, excluding type variables. For (covariant) method returns, we compute the *least solution* by unioning its non-variable lower bounds. We leave type variables out of solutions because their bounds will have already been transitively propagated during resolution.
For the constraints in Figure 5.1b, SimTyper finds the following solutions:

\[
\alpha = [\text{kind_of?} : \text{Class} \to \epsilon, - : \text{Number} \to \iota, / : \text{Number} \to \kappa]
\]

\[
\beta = [\text{kind_of?} : \text{Class} \to \zeta, > : \text{Number} \to \theta, + : \text{Number} \to \mu]
\]

\[
\gamma = [\text{kind_of?} : \text{Class} \to \eta, + : \text{Number} \to \lambda]
\]

\[
\delta = \text{Timestamp}
\]

Out of these four solutions, the only one that matches developer-provided documentation (not shown) is \(\delta\), which has a simple, easy-to-understand nominal type. In contrast, the solutions for \(\alpha, \beta,\) and \(\gamma\), while very precise, are also complex, verbose, and hard to read. Moreover, they are not even fully expanded, since they contain type variables—and adding solutions for the nested variables would only make the types more complex. Standard type inference with subtyping and structural types often produces such difficult-to-use types.

5.2.2 Heuristic Type Inference

To address this problem, SimTyper builds on an approach pioneered by InferDL and applies a series of heuristic rules that aim to guess more useful types, typically nominal or generic types, for positions for which standard type inference produces overly general types, like those for \(\alpha, \beta,\) and \(\gamma\) above. More precisely, any type that is not one of the following is considered overly general: nominal types, generic types, finite hash and tuple types (which are more precise types for hashes and arrays, described in detail in Chapter 3), singleton types (which represent just a single value),
and the boolean type.

In this case, SimTyper applies the STRUCT-TO-NOMINAL rule (inherited from InferDL), which converts structural types to nominal types; see Chapter 4.2 for a full definition of this rule. When SimTyper and InferDL apply STRUCT-TO-NOMINAL to our running example, the heuristic guesses that the solution for \( \alpha \) (corresponding to the parameter year) is \texttt{Number}, because the only existing classes that define methods \texttt{kind_of?}, \texttt{>, +} are Ruby’s numeric classes. This guess is consistent with the other constraints, so SimTyper and InferDL would both set \( \alpha = \text{Number} \) as the solution.

However, STRUCT-TO-NOMINAL fails to infer a new solution for \( \beta \), because more than ten classes define the set of methods \{\texttt{kind_of?}, \texttt{>, +}\}—e.g., possible classes include \texttt{String}, \texttt{Set}, \texttt{Time}, and others—and similarly for \( \gamma \). Thus, while heuristics are effective, there is still room for improvement. Moreover, while InferDL allows users to write new heuristics that apply specifically to their programs, doing so requires a lot of care and insight, and heuristics may not be portable across programs. For example, InferDL also includes a heuristic INT_NAMES that, among others, guesses that an argument named \texttt{id} has type \texttt{Number}. However, while this is an excellent guess for Rails code, in other codebases, e.g., in the Stripe codebase, \texttt{ids} are generally \texttt{Strings} \texttt{[93]}.

5.2.3 Predicting Type Equalities

To address the limitations of heuristics, SimTyper builds on InferDL’s approach by additionally using a machine learning-based approach to guess types.
Figure 5.2: An illustration of how DeepSim calculates the pairwise similarity scores for the set of input parameters \{year, month, day\} for the method from Figure 5.1a.

These guesses can either be used in place of or in addition to guesses from handwritten heuristics. Like \textit{InferDL}, these guesses are checked for consistency with the underlying constraints to ensure soundness, and any inconsistent guesses are discarded.

More specifically, \textit{SimTyper} uses a novel network based on the Siamese network architecture [115] which we call a \textit{deep similarity} (DeepSim) network. DeepSim is a deep neural network model that we use to guess when two positions have equal types. For example, notice that \textit{year}, \textit{month}, and \textit{day} are closely related words in English. Moreover, in \texttt{Timestamp.create}, they are used in similar contexts: First they appear in nearly-identical dynamic type checks (lines 4–6) and then in arithmetic expressions (lines 8–13). Thus, the DeepSim network guesses that all three variables have equal types—and since \textit{SimTyper} previously determined that \textit{year} has type \texttt{Number}, using DeepSim it will guess the same type for the other two arguments.

**Guessing Types with the DeepSim Neural Network.** DeepSim is based on the Siamese network architecture [115]. The network takes two inputs, which are
the source code for two methods with position markers indicating mentions of the relevant parameter/return that is being compared. The inputs are first run through identical encoders. The encoder uses a state-of-the-art model trained on programming and natural languages [111] to produce a fixed-dimension contextualized vector representation for the input. This is a numeric vector that encodes not only the name of an identifier (e.g., the name of an argument), but also the code context in which it occurs. The encoded vectors are such that parameters with similar names (e.g., “day” and “month”) and contexts (e.g., basic arithmetic expressions) will be encoded as vectors that are close together. Contextualized embedding models have recently been shown to achieve state-of-the-art performance on a range of natural language processing [116] and programming language [111, 117] tasks.

The encoded inputs are then run through a trained similarity function, which produces a similarity score between 0 and 1, indicating the network’s belief that two inputs have the same (1) or different (0) types. For example, Figure 5.2 illustrates how DeepSim calculates the pairwise similarity scores for the set of input parameters \{year, month, day\} for the method from Figure 5.1a. More details on the encoder and the similarity function are in Section 5.4.

SimTyper applies the DeepSim network after standard inference and handwritten heuristics have been run. SimTyper considers all remaining overly general type positions and uses DeepSim to compare them to all positions for which a “usable” (i.e., not overly general) solution was found. After eliminating positions with scores below 0.5—which indicates the network believes those positions have different types—SimTyper guesses type equivalence with the highest scoring position. If that
guess is consistent with the constraints, it is accepted. If that guess is inconsistent, SimTyper continues guessing equivalence with the next highest scoring position, and so on for the top \( N \) scores (§ 5.5 evaluates choices for \( N \)).

For example, picking up from the heuristic guesses in § 5.2.2, SimTyper asks DeepSim for the expected type similarity among \texttt{year}, which has a usable solution at this point, and \texttt{month} and \texttt{day}, which have overly general solutions. As shown in Figure 5.2, the network has a very high degree of confidence that \texttt{year} and \texttt{month} have the same type, so SimTyper guesses that \( \beta = \alpha = \text{Number} \). This guess is consistent, and so it is accepted. Next, \texttt{day} is predicted to be most likely similar to \texttt{month}, so SimTyper guesses \( \gamma = \beta = \text{Number} \), which is also accepted. Thus, after applying standard type inference, hand-written heuristics, and the DeepSim network, SimTyper has successfully inferred the type \((\text{Number}, \text{Number}, \text{Number}) \rightarrow \text{Timestamp})\) for \texttt{Timestamp.create}, which matches the hand-written documentation.

Cascading Type Predictions. Similar to heuristic guesses in InferDL, guesses made via DeepSim can cascade through the constraints, leading to further usable solutions. For example, consider the code snippet in Figure 5.3 extracted and simplified from \texttt{code.org}, one of our benchmarks (§ 5.5). This code defines a class method \texttt{initial} that, given a \texttt{String} called \texttt{name}, returns the first non-whitespace character in \texttt{name} as an upper-case letter.

SimTyper assigns \texttt{initial} the type \((\alpha) \rightarrow \beta\) and generates the constraints shown in the right of Figure 5.3. Constraints (1), (2), and (3) result from the calls to \texttt{strip}, [], and \texttt{upcase}, respectively, and constraint (4) results from the \texttt{return}. Using
Figure 5.3: A method defined in the code.org app and the resulting constraints.

standard type inference, SimTyper would generate the type \(([\text{strip} : () \rightarrow \gamma]) \rightarrow \epsilon\)
for this method.

However, DeepSim predicts that name has a similarity score of approximately 0.996 with another parameter, also called name, from a different method (code omitted for brevity). Because the other parameter’s type is determined to be String, SimTyper guesses String as a solution for the name parameter of self.initial, which is accepted as consistent.

But something interesting happens when String is added as the solution for name. The type String propagates further through the constraints:

1. String = α is added as the solution. Propagating this yields...

2. String ≤ [strip : () \rightarrow \gamma]. Looking up the type of String#strip (SimTyper includes types for Ruby’s core and standard libraries) yields...

3. String ≤ γ. In other words, strip returns a String. Propagating further yields...

4. (in several steps) String ≤ δ, i.e., [0] also returns a String. Propagating further yields...

5. (in several steps) String ≤ ϵ, i.e., upcase returns a String. Propagating to ϵ’s upper bound...
6. String $\leq \beta$.

Then, using the usual rules for computing the solution at a return position, we can set $\beta = \text{String}$ also. Thus, we have found that the method has type $(\text{String}) \rightarrow \text{String}$, which matches its documentation. This exemplifies another benefit of SimTyper: by integrating DeepSim within the constraint solver, the former can lead the latter to better type solutions, and vice versa.

As an aside, we note that the same cascading effect can happen with heuristics—and in fact, in this case that would occur, as InferDL includes the rule \texttt{STRING\_NAME} that guesses that arguments called \texttt{name} have type \texttt{String}. While there are many cases from our benchmarks where DeepSim alone leads to a cascading solution, we present the example of Figure 5.3 due to its relative brevity and simplicity.

**Discussion.** An alternative approach would be to use machine learning to directly predict types. However, prior work [113, 118, 119], as well as our own dataset (§ 5.4), has found that the distribution of types in programs is Zipfian: a small number of types occur very frequently, while most types occur rarely. Moreover, some types are program-specific and thus will not occur in a training dataset at all. This makes it challenging to train a direct prediction model for the many infrequent types. Moreover, DeepSim can perform one-shot type prediction, in which it predicts the correct type of an argument/return by knowing the type of just one other instance. Finally, DeepSim is tightly integrated with type inference, allowing it to propagate any usable types that standard inference infers.
5.3 Type Inference Algorithm

In this section we present SimTyper’s algorithm more formally. We first briefly discuss the elements of the algorithm that are inherited from InferDL (§ 5.3.1), then build on these to present the type equality prediction algorithm (§ 5.3.2).

5.3.1 Standard and Heuristic Inference

In Section 4.3, we provided a full discussion of InferDL’s algorithm, which combines constraint solving and configurable heuristics. Here, we briefly summarize the basic definitions and relevant aspects of this algorithm that are inherited by SimTyper.

The top part of Figure 5.4 defines a core language of types, constraints, and solutions. Types include variables \( \alpha \), nominal types \( A \), structural types \([m : \tau \rightarrow \tau]\) (which assume a single argument for simplicity), union types, intersection types, the bottom type \( \bot \), and the top type \( \top \). Constraints \( C \) take the form \( \tau_1 \leq \tau_2 \), meaning \( \tau_1 \) is a subtype of \( \tau_2 \). We use union to construct sets of constraints. Finally, a
solution $S$ is a mapping from variables to their types.

The bottom part of Figure 5.4 gives the types for the $\text{resolve}$ and $\text{extract_solution}$ functions, which we inherit from $\text{InferDL}$. Given a set of constraints, $\text{resolve}$ performs constraint resolution and returns either a new set of constraints or $\text{error}$ if the given set of constraints are inconsistent. The $\text{extract_solution}$ function takes a list of heuristics, a set of constraints, and a type variable $\alpha$, and uses a combination of constraint solving and heuristics to extract a solution for $\alpha$; it returns a new set of constraints $C$ which includes the new solution, as well as the solution $\tau$ itself. We will use both of these functions in the type equality prediction algorithm defined below, and we refer the reader to section 4.3 for their definitions.

Finally, we assume the existence of two functions. The first is the predicate $\text{og}(\tau)$, which returns true when $\tau$ is considered overly-general and false otherwise. Following Chapter 4, for this core language of types, $\text{og}(\tau)$ is true when $\tau$ is a non-nominal type. The second function, $\text{usable_sols}$, returns the set of solutions in a given mapping that are not overly general:

$$\text{usable_sols}(S) = \{ \alpha \mapsto \tau \in S \mid \neg \text{og}(\tau) \}$$

### 5.3.2 Type Equality Prediction Algorithm

We can now build on these definitions to introduce the type equality prediction algorithm. We first introduce the function $\text{sim}$ to represent the DeepSim neural
procedure DEEPSIM_SOL(α, S, C)
poss_sols ← \{ (τ_i, s) | α_i ↦ τ_i ∈ usable_sols(S), s = sim(α, α_i), s ≥ 0.5 \}
k ← 1
while \( k \leq \text{size}(\text{poss}_\text{sols}) \land k \leq N \) do
    pick \( k \)th highest scoring solution \( τ_k \) in \text{poss}_\text{sols}
    \( C_{\text{new}} \leftarrow \text{resolve}(C \cup τ_k ≤ α \cup α ≤ τ_k) \)
    if \( C_{\text{new}} \neq \text{error} \) then
        return \( C_{\text{new}}, τ_k \)
    \( k \leftarrow k + 1 \)
return \( C, S[α] \)
(a) DEEPSIM_SOL, which uses DeepSim to find a solution for a single type variable.

procedure TYPE_EQ_INFEERENCE(H, C, V)
S ← \{ \}
\( \text{first}_\text{round} \leftarrow \text{true} \)
repeat
    for each \( α ∈ V \) do
        \( C, \text{sol} \leftarrow \text{EXTRACT_SOLUTION}(H, C, α) \)
        \( S[α] \leftarrow \text{sol} \)
        if \(!\text{first}_\text{round} \land \text{og(sol)} \) then
            \( C, \text{sol} \leftarrow \text{DEEPSIM_SOL}(α, S, C) \)
            \( S[α] \leftarrow \text{sol} \)
    \( \text{first}_\text{round} \leftarrow \text{false} \)
until no new constraints are added to \( C \)
return \( S \)
(b) SimTyper’s overall algorithm.

Figure 5.5: Procedures used in SimTyper.

network. Given two type variables \( α_1 \) and \( α_2 \), \( \text{sim}(α_1, α_2) ∈ [0, 1] \) is a similarity score between \( α_1 \) and \( α_2 \), where scores closer to 1 indicate greater similarity and a score below 0.5 indicates dissimilarity. We also assume there is an \( N \) specifying the maximum number of similar variables to try for a solution.

Now we can introduce DEEPSIM_SOL, the procedure for finding a single DeepSim network solution, which we define at the top of Figure 5.5. Given a type variable \( α \), a solution \( S \), and a constraint set \( C \), the function DEEPSIM_SOL returns an updated constraint set \( C_{\text{new}} \) and a solution for \( α \); if DeepSim did not find any new
solution, then the returned constraints and solution are the same as the original ones provided to the function.

The first line of `deepsim_sol` defines `poss_sols` to be a set of pairs, where the first element of the pair is a possible type solution, and the second element is the corresponding similarity score. The set is constructed by comparing \( \alpha \) with every \( \alpha_i \in \text{dom}(\text{usable}_sols(S)) \), and keeping the corresponding type solution \( \tau_i \) when its similarity score is above 0.5. Then the function loops, picking the highest scoring solution \( \tau_k \) in `potential_sols`. The function then “tests” the solution by equating it to \( \alpha \) and running constraint resolution. If this succeeds, we have found a consistent guess, so the function returns the new set of constraints and the found solution \( \tau_k \). Otherwise, the loop continues with the next highest score, etc. If we exceed \( N \) iterations or explore all the potential matching solutions, then the function returns the original constraint set and the original solution for \( \alpha \) in \( S \), since no consistent guess was found.

Finally, we present `SimTyper`'s overall algorithm, the procedure `type_eq_inference` defined at the bottom of Figure 5.5. This procedure is very similar to the `extract_all_solutions` procedure used by `InferDL` (§ 4.3), but it adds the use of `deepsim_sol`. The procedure takes as input a set of heuristics \( H \), a set of constraints \( C \), and a set of type variables to find solutions for \( V \). It begins by assigning an empty solution map to \( S \), and by assigning the variable `first_round`, which keeps track of whether we are performing the first round of solution extraction, to `true`. Next, we iterate through each variable in \( V \) and perform standard and heuristic inference by calling `extract_solution`, updating the constraints \( C \) and
solutions $S$ as we go. We perform one full round of standard and heuristic inference over all type variables before ever using DeepSim; this gives us more usable solutions to compare against with DeepSim. After one round, we perform inference again, this time calling DEEPSIM_SOL for any type variables which still have overly general solutions. We update $C$ and $S$ with any new solutions. This process repeats until no new constraints are added to $C$; by looping until we reach a fixpoint in $C$, we can allow any new solutions found by DeepSim or constraint solving to cascade to new solutions, as discussed in Section 5.2.

Note that this algorithm is greedy, so the order order in which DeepSim network solutions are generated may matter. In particular, if the network generates incompatible solutions for two different type variables (i.e., the resulting constraints are inconsistent), then the solution that was generated earlier may effectively block the later solution. In our implementation, the order used is effectively arbitrary. Determining a way to pick among incompatible solutions is an interesting avenue for future work.

5.4 Implementing the DeepSim Network

Figure 5.6a shows DeepSim’s network architecture. DeepSim encodes a pair of inputs into a pair of fixed-dimensional embedding vectors (of the same dimensionality) via a weight-sharing encoder, and then runs them through a similarity function to predict the likelihood both inputs have the same type. Following the Siamese Network structure [115], the encoder used for both inputs is the same. Next, we
Figure 5.6: SimTyper’s Deep Similarity (DeepSim) Neural Network.

discuss the network in more detail.

**Network Input.** For each argument, the network takes as input the tokenized source code for the method containing the argument plus the positions at which the argument appears. More formally, suppose \( \text{arg}_1 \) and \( \text{arg}_2 \) are the method arguments to be compared, and that \( X_i \) is the tokenized source code of the method in which \( \text{arg}_i \) appears. Then the input to DeepSim is the two token sequences \( X_1 = \ldots, x_1^{i_1}, \ldots, x_1^{i_2}, \ldots, x_1^{i_m}, \ldots \) and \( X_2 = \ldots, x_2^{j_1}, \ldots, x_2^{j_2}, \ldots, x_2^{j_n}, \ldots \), where each \( x_1^{i_k} \) represents the mention of \( \text{arg}_1 \) at position \( i_k \) in the first sequence, and each \( x_2^{j_k} \) represents the mention of \( \text{arg}_2 \) at position \( j_k \) in the second sequence. The input also includes the sequences \( i_1, i_2, \ldots, i_m \) and \( j_1, j_2, \ldots, j_n \), i.e., the indices of the parameters within the source code tokens.
DeepSim uses an analogous approach when comparing method returns: The input is the method’s tokenized source code plus the positions of the method name itself and all the returns within its body. We use the method’s name because it is often used to describe the return value; § 5.5 includes an evaluation of different approaches to representing returns.

**Contextualized Vector Representations.** Next, DeepSim encodes each input token sequence into a sequence of *contextualized vector representations* (one vector per token) using CodeBERT [111], a transformer-based [120] pre-trained code embedding model.\(^2\) Contextualized vector representations can capture both the English-language meaning of tokens and the surrounding code context. The goal is for tokens with similar meanings (e.g., *year* and *month*) and usage (e.g., used inside basic arithmetic expressions) to map to nearby vectors in the vector space.

Continuing the formal notation just above, the output of this layer is the sequences of vectors \(CV_1^1, \ldots, CV_1^m\) and \(CV_2^1, \ldots, CV_2^n\), where \(CV_1^k\) is the contextualized vector representation of token \(x_1^{ik}\) for \(arg_1\), and \(CV_2^k\) is the representation of token \(x_2^{jk}\) for \(arg_2\). These vectors are of dimension \(d = 768\), the size of vectors produced by CodeBERT. Note that we keep only the contextualized vectors for the tokens of the relevant argument/return.

**Pooling.** Next, the encoder uses a pooling layer to combine the contextualized vector representations into a single, fixed-dimension vector. We use mean pool-\\(^2\)CodeBERT was trained on over 8.5 million methods and functions from programs written in Ruby, Java, JavaScript, Go, PHP, and Python.
ing, which is shown to be effective in Siamese Network models for text similarity tasks [121]. For example, as illustrated in Figure 5.6b for our running example, for the argument \texttt{day}, the pooling layer averages the vectors corresponding to its mentions in the method header and method body. For the return of \texttt{self.create}, the pooling layer averages the vectors for the method name in the method header and for the last line of the method, whose value is returned. Interestingly, our experiments suggest mean pooling is important for arguments but not for returns (§ 5.5.5).

Formally, the output of this layer is two fixed-dimensional vectors
\[ V_1 = \text{MeanPool}(CV_1^1, \ldots, CV_m^1) \quad \text{and} \quad V_2 = \text{MeanPool}(CV_1^2, \ldots, CV_n^2), \]
where \text{MeanPool} averages its vector arguments.

**Similarity Function.** The subsequent stage, the similarity function, produces a similarity score for the encoded inputs (now a pair of fixed-dimension vectors). First, the pair of vectors are joined to form a relational vector representing the pair as well as their interactive features. Then this relational vector is run through a sigmoid function to produce a similarity score in the range (0, 1), where a score closer to 1 indicates the inputs likely have the same type, and a score closer to 0 indicates the inputs likely have different types.

Formally, the similarity function begins by concatenating \( V_1 \) and \( V_2 \) with the element-wise difference \(|V_1 - V_2|\) to generate the pair representation \( V = (V_1, V_2, |V_1 - V_2|) \). This approach follows Reimers and Gurevych [121], who show it to be effective in capturing both input features and the interactive features between the pair. We then apply a fully-connected layer with trainable weight matrix \( W \in \mathbb{R}^{3d} \) (recall
def +(value) ... end

Figure 5.7: A method * from the Money library which has YARD documentation.

$d = 768$ is the dimensionality of the CodeBERT contextualized vectors) and bias term $b \in \mathbb{R}$, and then pass the result through the sigmoid function $\sigma$, which squashes values into the valid probability range $[0,1]$ to generate the probability of $X_1$ and $X_2$ being similar ($y = 1$):

$$P(y = 1|X_1, X_2) = \sigma(W \cdot V + b) = \frac{1}{1 + e^{-(W \cdot V + b)}}$$

Taking $N$ training pairs with labels 1 or 0 ($y_i = 1$ or 0 for the $i$th pair), DeepSim is trained with the Adam optimizer [122] by minimizing the binary cross-entropy loss, which is a common choice for binary classification tasks:

$$\mathcal{J} = -\sum_{i=1}^{N}(y_i \cdot \log(p(y_i))) + (1 - y_i) \cdot \log(1 - p(y_i))$$

5.4.1 Training the DeepSim Network.

SimTyper trains two different networks: one for comparing arguments, and one for comparing returns. To train the networks, we need type information for Ruby programs. However, Ruby is dynamically typed and does not include type annotations. Recently, several Ruby type systems have emerged, including RDL [20], Sorbet [92], and the new types available with Ruby 3.0. However, to the best of our
knowledge, the number of publicly available programs with type annotations is still very small and therefore is insufficient for training. One idea for generating training data would be run Ruby programs and document the observed runtime method types. Though Strickland et al. [123] found that such types are often specific to a single run of a program and thus may miss possible types, it is possible this approach would be sufficient for a training dataset and it is worth exploring in the future.

Instead, we collect type information from programs that use YARD, a popular Ruby documentation tool. Figure 5.7 shows an example of YARD documentation for method * of the Money library, one of our benchmarks. The first two comment lines provide a general description, and the last two lines give structured information including types of the parameter (of type Numeric) and the return (of type Money). Note that this documentation is noisy because it may not be accurate—there is no system that automatically checks YARD documentation against code to enforce its correctness—and the standard notation for types in YARD is not enforced. Nevertheless, for purposes of training DeepSim, it is still very effective, especially since DeepSim can tolerate some noise.

To build our training dataset, we looked at the top 1000 starred Ruby repositories on Github, and the top 1000 gems on rubygems.org, Ruby’s central package hosting service. After eliminating overlapping programs and removing programs without YARD type data, we were left with 371 Ruby programs, comprising over 285,000 methods with documented types, and over 417,000 individual data points, where each parameter and return type is counted as a separate data point.

However, for training, DeepSim expects pairs of inputs labeled with 1 for pairs
with the same type and 0 otherwise. The set of all possible pairs from our dataset of over 417,000 would be prohibitively large, so we restrict ourselves to a set of 100,000 randomly chosen pairs; the number of pairs was chosen through a tuning process discussed below. Moreover, we restrict training pairs to come from the same program, with the idea that the naming choices and coding patterns used within a program are more likely to be cohesive than the choices between different programs.

**Hyperparameters.** We tuned three hyperparameters for the DeepSim network: the number of data points, the number of training epochs, and the learning rate. We considered all data sizes from 25,000 to 200,000 in increments of 25,000, all numbers of epochs from 25 to 200 in increments of 25, and all learning rates in the set \{0.001, 0.0005, 0.0001, 0.0005\}. We used grid search, training networks using all possible combinations of values for the hyperparameters, then selecting the models that scored the highest accuracy on a validation dataset that was independent from the training data and test benchmarks used in our experiments (§ 5.5). Ultimately, we trained the argument model using 150 epochs and the return model using 100 epochs, and both models were trained with 100,000 data points and a learning rate of 0.001.

**Types for Instance, Class, and Global Variables.** In addition to predicting type similarity for arguments and returns, SimTyper also uses DeepSim to do the same for instance, class, and global variables. However, there are a few changes required. First, because variables can be both read and written, it is not the case
that a greatest or least solution will always be most general. Instead, SimTyper follows the approach of Kazerounian et al. [110] (also presented in Chapter 4), which was shown to work well in practice: when the variable’s type has upper bounds, SimTyper uses the intersection of the type’s upper bounds, and otherwise SimTyper uses the union of its lower bounds.

Second, for a given variable, pooling averages the vectors corresponding to all uses of that variable. However, unlike arguments and returns, instance, class, and global variables can be accessed in multiple methods. Hence, the encoding step must vectorize all the methods that refer to the variable. Note that although class, global, and (some) instance variables can be accessed outside of methods, SimTyper currently does not include such occurrences in its analysis.

Finally, YARD does not currently support documentation for instance, class, and global variables. Thus we could not collect training data for them. Instead, we use the network we trained for arguments to answer questions about variables, since we expect arguments may be named similarly to and used in similar contexts to non-local variables. In the future, we are also interested in exploring whether we could use just a single network for returns, arguments, and variables.

5.5 Evaluation

We evaluated SimTyper on a range of Ruby benchmarks with existing type information, which we treated as gold standard types we aim to infer. Our benchmarks came from two sources. First, we used the same four Rails web apps that InferDL
was evaluated on (Section 4.5). We refer to these as the InferDL Benchmarks:

- *code.org* [71] – the code.org website app
- *Discourse* [69] – online discussion platform
- *Journey* [72] – site for creating surveys and collecting responses
- *Talks* [94] – site for sharing talk announcements

For these apps, we use SimTyper to infer types for all methods and instance, class, and global variables for which manually written type annotations already existed in the InferDL study.

Second, we applied SimTyper to four popular, well-maintained libraries that have extensive YARD documentation that provides types for a majority of their methods. We refer to these as the YARD Benchmarks:

- *TZInfo* [100] – library for manipulating timezone data
- *MiniMagick* [97] – wrapper for the ImageMagick image manipulation platform
- *Ronin* [124] – platform for vulnerability research and exploit development
- *Money* [39] – library for currency arithmetic and conversion

We were particularly interested in libraries because, in our experience, they are more likely to have well-documented APIs compared to complete programs like web apps. For these libraries, we use SimTyper to generate types for all methods with YARD type documentation except those that use features that are not supported by InferDL.
The most common feature that blocked standard type inference was the presence of mixins, which are only partially supported by InferDL. Note that we excluded any measurements about instance, class, and global variables for these benchmarks because YARD does not include documentation about variable types. Finally, we withheld all the type data for these programs from the datasets we used for training and validation (§ 5.4.1).

Table 5.1 summarizes the benchmarks’ statistics. For each benchmark, the table lists the number of methods for which SimTyper generates a type annotation that we compare against a gold standard, followed by the number of lines of code comprised by those methods. Additionally, the subsequent column lists the number of non-local (instance, class, and global) variable annotations generated by SimTyper that we compare against gold standards. In total, we ran SimTyper on 844 methods comprising 5,853 lines of code, and generated types for 84 non-local variables. Note that the number of types for variables for the YARD Benchmarks was 0, since YARD does not include documentation for these variables that we can compare against.

Table 5.1: Benchmark statistics.

<table>
<thead>
<tr>
<th>Program</th>
<th># Meths</th>
<th>LoC</th>
<th># Vars</th>
<th># Casts</th>
</tr>
</thead>
<tbody>
<tr>
<td>code.org</td>
<td>74</td>
<td>689</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>Discourse</td>
<td>43</td>
<td>331</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Journey</td>
<td>23</td>
<td>375</td>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>Talks</td>
<td>110</td>
<td>878</td>
<td>47</td>
<td>8</td>
</tr>
<tr>
<td>MiniMagick</td>
<td>40</td>
<td>216</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Money</td>
<td>87</td>
<td>444</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>Ronin</td>
<td>226</td>
<td>1628</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>TZInfo</td>
<td>241</td>
<td>1292</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>844</strong></td>
<td><strong>5853</strong></td>
<td><strong>84</strong></td>
<td><strong>55</strong></td>
</tr>
</tbody>
</table>

We additionally note that, while we ran SimTyper for more methods in the YARD
Benchmarks, there were some which did not include gold standard types to compare against.

The table also shows the number of type casts required to run standard type inference on the benchmark’s methods. The first four benchmarks already came with type casts from InferDL; we wrote type casts for the last four benchmarks. In our experience, the most common reasons for needing type casts were to handle path-sensitive typing and to cast a value extracted from heterogeneous data structures like arrays and hashes.

Additionally, we note that there were some method and variable types that SimTyper inferred which we did not have a gold standard to compare against. More precisely, all of the YARD Benchmarks included some method types inferred by SimTyper without corresponding gold standards, and seven out of eight of all of the benchmarks included at least one instance, class, or global variable type inferred by SimTyper without a corresponding gold standard. While we could not compare these inferred types against gold standards, we found that of 418 non-compared argument, return, and variable types, 277 were usable types, 81 were overly-general, and SimTyper failed to infer types for 60 of these positions. The full results are shown in Table B.1 in the Appendix.

**Evaluation Methodology.** We ran SimTyper on the above benchmarks under four separate configurations: using constraint solving alone (C), constraint solving and InferDL’s built-in set of heuristics (CH), constraint solving and the DeepSim network (CD), and all three approaches together (CHD). We compare the results to
the original type annotations (InferDL Benchmarks) or YARD documentation (YARD Benchmarks). To provide finer-grained analysis, we make comparisons on a per-argument, per-return, and (for InferDL Benchmarks) per-variable basis, rather than, e.g., comparing whole method signatures at once. Inspired by prior work [113], we place each comparison in one of three categories:

- **Match.** SimTyper inferred a type that exactly matches the gold standard or is a subtype of the gold standard. We also consider a match exact if both the generated and gold standard types are from the set \{String, Symbol, String \cup Symbol\}. In Ruby, Symbol is a special kind of interned String, and the two types are often used interchangeably. Lastly, we also treat the types Array and ActiveRecord_Relation (Rails’ special array implementation, used for database queries) as interchangeable.

- **Match up to Parameter.** The gold standard type is a generic type, and SimTyper inferred the base of the generic type but not the parameter. For example, if the gold standard was Array<String>, then SimTyper generating either Array or Array<Integer> would fall in this category. This category provides a notion of partial matches.

- **Different Type.** SimTyper inferred a type that was consistent with the constraints—hence it is sound—but differs from the gold standard type in a way that does not fall into the above categories. For example, inferred structural types fall into this category because they very rarely occur in the gold standard types (out of 1,496 gold standard annotations in our benchmarks, just 8 use struc-
tural types). Note that it is possible for programmer-provided and SimTyper-inferred types to be incomparable, e.g., below we mention a case when one is `Integer` and the other is `String`, and both are sound because they share a common structural supertype.

5.5.1 SimTyper Results

Figure 5.8 shows the types inferred by SimTyper under the C, CH, CD, and CHD configurations, categorized as just described. We split the last category, Different Type, into structural and non-structural types, to aid the discussion below. These results were collected using the top-3 solutions suggested by DeepSim with a similarity score cutoff of 0.5 (see § 5.3). Below, unless we say otherwise, comparisons of matching sum both Match and Match up to Parameter. Table B.2 in Appendix B presents the same data as the figure.

Looking at the totals (lower right corner of the figure), we see that the CHD configuration outperformed all others in the number of matches to the gold standard. In total, SimTyper inferred 1,033 total matching annotations under CHD, compared to 610 under C alone, a 69% increase. This is still a 66% increase if we exclude matches up to parameter. CHD inferred the most matching types not only in total, but also for each individual benchmark, with the exception of Money, for which CD performed only slightly better. This suggests that combining constraint solving, heuristics, and the DeepSim network is an effective approach to inferring type annotations that match what a programmer would write.
Figure 5.8: Assessing the types inferred by SimTyper. We collected results under four configurations: constraint solving (C), constraint solving plus heuristics (CH), constraint solver plus the DeepSim network (CD), and all three (CHD). Results presented here use top-3 thresholding. Note the y-axis is scaled for each benchmark.
Comparing CD to CH, we see that in total, CD inferred 19% more matching types than CH (or 16% excluding matches up to parameter), indicating that DeepSim can outperform hand-written heuristics. Interestingly, while this was true overall, there is a contrast between the YARD Benchmarks and the InferDL Benchmarks. For the YARD Benchmarks, CD infers 44% more matches than CH, but for the InferDL Benchmarks, CD generates 7% fewer matches than CH. The biggest single contributor to the difference is code.org, where CD generates 20% fewer matches than CH. We believe the reason for this overall difference is that the heuristic rules of InferDL were developed while applying type inference to the InferDL Benchmarks. For example, InferDL includes a heuristic int_names that guesses that arguments ending in _id have type Number and arguments ending in _ids have type Array<Number>. Without this heuristic, in the code.org app, most such positions are inferred by standard type inference to be Number ∪ Array<Number>. DeepSim can at best propagate this union type—it has no particular mechanism to refine it—and so it cannot improve on standard type inference in these cases. Moreover, the int_names heuristic was applied more than 30 times for code.org, compared to just 5 times for all other benchmarks combined. Thus, we see the tradeoff between DeepSim and hand-written heuristics: the heuristics perform well on their initial target but do not necessarily generalize to other programs, while DeepSim generalizes well but does not fully cover all uses of heuristics. Thus, we think SimTyper’s architecture, which incorporates both approaches, is the right design choice.

Finally, we examined the Different Types category in more detail. We note that in the C configuration, structural types constitute the majority of different
types inferred. The one exception is code.org, where C infers union types in many positions as discussed just above. As we introduce heuristics and DeepSim, we see a clear trend where the number of structural different types decreases as they are replaced by nominal and generic types (indeed, by design, InferDL heuristics and DeepSim do not infer any new structural types), some of which become matches and some of which remain different types. The other category that decreases, not shown explicitly in the figure but included in the table in Appendix B, is positions where C could only infer a variable type, but heuristics or DeepSim could infer a nominal or generic type.

To get more insight into the non-structural different types, we manually examined their occurrence in Money under CHD. We found these types fall into roughly two categories. First, in some cases the gold standard type is a union and DeepSim’s inferred type was one arm of the union. For example, for one parameter new_currency, the gold standard type is Money::Currency ∪ String ∪ Symbol and the DeepSim inferred type is Money::Currency. Second, sometimes DeepSim infers a type that was unrelated to the gold standard but happened to be consistent with the program. For example, for one parameter named amount, the gold standard type is Number but DeepSim inferred the type String. The latter type is consistent because the only use of amount is to call to_d on it (to convert it to a BigDecimal), and that method is also defined on String. It is interesting future work to try to address both of these cases.
Failed Inference. There are also some arguments, returns, and variables for which SimTyper fails to infer any type. Table B.2 in the Appendix presents the specific number of positions for which this was the case under each configuration. Under CHD, across all benchmarks, SimTyper fails to infer a type for 6.8% of arguments, returns, and variables. These are cases where there are not enough constraints for standard inference to produce a solution: there are no non-type variable upper bounds (for arguments and variables) or lower bounds (for returns and variables). In other words, for arguments, no method is called on it (otherwise it would at least have a structural type upper bound); and for returns, typically the returned value comes from a method SimTyper does not analyze (e.g., a third-party library), and hence its signature has type variables that are not constrained by its method body.

Additionally, it must be that both heuristics and DeepSim either fail to guess a type, or they guess a type that is inconsistent with existing constraints. Of the 6.8% of positions (arguments/returns/variables) for which SimTyper failed to infer a type under CHD, the DeepSim network guessed a type for about 46% of these positions, but the guess was rejected due to inconsistency with constraints; none of the heuristics guessed a type for any of these positions. Interestingly, in 5 of these positions, the type guessed by the DeepSim network was actually a correct array or hash type, but the guess was rejected because the type checker conservatively uses invariant subtyping for arrays and hashes.

Precision and Recall. Another way of measuring SimTyper’s results is in terms of precision and recall. Following Pradel et al. [125], we compute precision as
<table>
<thead>
<tr>
<th>Config</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>59.6%</td>
<td>40.8%</td>
</tr>
<tr>
<td>CH</td>
<td>73.4%</td>
<td>54.3%</td>
</tr>
<tr>
<td>CD</td>
<td>69.6%</td>
<td>64.9%</td>
</tr>
<tr>
<td>CHD</td>
<td>74.1%</td>
<td>69.1%</td>
</tr>
</tbody>
</table>

Table 5.2: Precision and recall of SimTyper.

\( \frac{n_{\text{match}}}{n_{\text{type}}} \), where \( n_{\text{match}} \) is the number of type matches (including up to parameter) and \( n_{\text{type}} \) is the total number of positions (arguments, returns, and variables) for which SimTyper inferred any type. We compute recall as \( \frac{n_{\text{match}}}{n_{\text{all}}} \), where \( n_{\text{all}} \) is the total number of positions for which SimTyper attempted to infer a type (whether it did so or not).

From the table, we see that, consistent with the earlier interpretation of the data, CHD achieved the highest precision and recall. Additionally, CH outperformed CD on precision by 3.8%, while CD outperformed CH on recall by 10.6%. This means that heuristics alone are less likely to predict a matching type than DeepSim alone, but when they do predict a type, the type is slightly more likely to be a matching one. C was the worst performing configuration in both precision and recall.

We note that our notion of precision and recall is slightly different from Alamanis et al. [113]. They compute precision as \( \frac{n_{\text{neutral}}}{n_{\text{type}}} \), where \( n_{\text{neutral}} \) is the number of “neutral” types: predicted types that pass a type checker. By this measurement, under all four configurations, SimTyper’s precision would be 100%, since all of its predicted types are consistent with the program’s constraints. Moreover, they compute recall as \( \frac{n_{\text{type}}}{n_{\text{all}}} \), that is, the proportion of the dataset for which any type was inferred. By this metric, the recall for CHD would be 93.2%. Instead,
Figure 5.9: Measuring SimTyper’s performance for arguments, returns, and instance, class, and global variables across all benchmarks. The plots use the same legend as Figure 5.8. Measurements were taken using top-3 thresholding.

we focus on the number of matches inferred by SimTyper, since this measures types that reflect programmer intent.

**Arguments vs. Returns vs. Variables.** Recall that SimTyper uses separate networks for arguments and returns, and uses the argument network for instance, class, and global variables with some small adaptations (§ 5.4.1). Figure 5.9 measures SimTyper’s performance separately for these three groups. The data for these plots are included in Appendix B.

The figure shows that DeepSim improved performance the most on arguments. Under CD and CHD, SimTyper infers approximately 308% and 57% more matching (including up to parameter) types relative to the C and CH configurations, respectively. This is at least in part because arguments had the most room for improvement. For example, under C, SimTyper inferred matches for just 12% of all argument types, while for return and variable types, it inferred 60% and 73% matches, respectively.

However, it is also plausible that DeepSim is best tuned for arguments. First,
Table 5.3: Running time of SimTyper over nine runs.

recall that we could not train a network specifically on variables since we did not have this data (§ 5.4). Second, we found that incorporating return sites into return embeddings does not significantly improve performance (§ 5.5.5). We leave exploring other ways to incorporate method code into DeepSim’s predictions to future work.

5.5.2 Performance

Table 5.3 measures the performance of SimTyper in performing type inference. We report the median time and semi-interquartile range (SIQR) over nine runs under each configuration. Times were measured on a 2014 MacBook Pro with a 3GHz i7 processor and 16GB RAM. We can see clearly that DeepSim introduces overhead. CHD and CD are approximately $4.2 \times$ and $50 \times$ slower than CH and C, respectively.

Upon closer inspection, we found the biggest bottleneck was running CodeBERT; in the future, we plan to explore methods for speeding up this performance, such as alternative methods for batching inputs to CodeBERT.

Interestingly, there was just one benchmark, code.org, that took longer under
CH than under CD. For *Discourse*, CH and CD performance was nearly equal, and for all other benchmarks, CD took notably longer than CH. The slowdown for *code.org* and *Discourse* occurred primarily due to the STRUCT-TO-NOMINAL heuristic, which involves searching all methods defined for all classes in the program, which is a particularly large search space for these benchmarks.

### 5.5.3 Comparing DeepSim and Heuristics

Table 5.4 reports how often DeepSim predicted a matching (including up to parameter) type that a heuristic rule also guessed. For each heuristic rule (descriptions of the rules are in Chapter 4), the table lists how many types the heuristic matched in CH followed by how many of those matches DeepSim also predicted in CD.

From the table, we see that DeepSim performed best on types guessed by IS MODEL and STRING NAME, two name-based heuristics, predicting 87% of those types. DeepSim also inferred a majority of types for PREDICATE METHOD, another name-based heuristic. This makes sense as DeepSim’s embeddings reflect argument and method names. However, as discussed earlier, DeepSim did poorly on INT NAMES and INT ARRAY NAME, even though they are also name-based. This was primarily due to the aforementioned pattern in *code.org* that DeepSim failed to capture.

Overall, DeepSim predicted 43% of the types guessed by heuristics. We also examined the dual (not shown in the table): of the 267 matching types inferred by
Table 5.4: DeepSim’s ability to predict types also guessed by heuristics. *H Matches* is the number of matching (including up to parameter) types inferred by the heuristic in CH, and *DS Matches* counts the subset of those types also inferred by DeepSim in CD. Measurements with the DeepSim network were taken with the top-3 threshold.

DeepSim under CD, heuristics guessed 80 (about 30%) of them under CH. Because these two sets are largely non-overlapping, these results reinforce that using DeepSim alongside hand-written heuristics is an effective combination.

5.5.4 Predicting Rare Types

One potential benefit of SimTyper is that it can infer *rare* types, i.e., those that are relatively less common across programs. Such types could be inferred through standard constraint solving, e.g., in Figure 5.1b, standard type inference found *Timestamp* as a solution; through applying heuristics, e.g., the *struct-to-nominal* heuristic can guess a user-defined type that matches a structural type; and through DeepSim, which could predict a rare type by guessing two positions have the same type. In this section, we measure how often DeepSim can predict rare types, since rare types have historically posed a challenge for machine learning-based type inference (§ 5.6).
Table 5.5: Numbers of Library, Training, and Program types guessed by DeepSim across all benchmarks, under CD with top-3 cutoff.

Table 5.5 measures three different categories of types inferred specifically by DeepSim. First, the Library types are the 78 types in Ruby’s standard and core library as well as the core Rails classes. These types are “common” because we expect them to appear frequently in Ruby programs. DeepSim predicted 356 types in this category, of which 240 were matches (67% precision).

The Training types are “rare” types that occur in the training dataset but not in the Library types. In theory we could train a machine learning-based classifier to predict them directly. However, the training data has 9,149 distinct types, yet 71% of all argument and return types in the data are Library types. Hence in practice a classifier would not be very likely to predict non-Library types. From the table, we see that DeepSim predicted 13 Training types, of which 12 were matches (92% precision).

Finally, Program types only occur in the benchmarks and not in the standard Ruby libraries or in the training data. Thus, a machine learning-based direct classifier would have no ability to predict these at all. In contrast, DeepSim predicted 38 such types, of which 16 were matches (42% precision).

Thus, overall, we can see that while DeepSim often predicts common types (which is expected, since they are common), it can also effectively predict rare types.
Finally, we evaluate two design choices in SimTyper. First, Figure 5.10a compares top-1, top-3, top-5, and top-7 thresholds for solutions predicted by DeepSim running under CD, across all benchmarks. (We omit heuristics because we are specifically interested in DeepSim here.) The data for these plots is in Appendix B. From the figure, we can see an increase in matches (including up to parameter) from top-1 (927 matches) to top-3 (971 matches). However, the results for top-3, top-5, and top-7 are nearly identical: for each category (match, match up to parameter, etc), the numbers are within two of each other. Thus, we settled on top-3 for our experiments.

Next, we evaluate the use of mean pooling. Figure 5.10b compares pooling the vectors for all uses of an argument (AllA), as in § 5.4, with using just the argument in the method header (HeadA). Figure 5.10c similarly compares pooling method names and return sites (AllR), also as in § 5.4, to using just method names.

Figure 5.10: Evaluating design choices in SimTyper. Plots use the same legend as in Figure 5.8 and show data from CD across all benchmarks. Note y-axis starts at 700. Data for all plots appears in Appendix B.
We see that under AllA, SimTyper infers 72 more matches than under HeadA, while AllR yields just two more matches than HeadR. This suggests that for arguments, the context from the uses in the method body is important, while for returns the method name alone is likely sufficient.

5.6 Related Work

For a discussion of related work on standard type inference systems, we direct the reader to Section 4.6.

In recent years, researchers have proposed a number of probabilistic type inference systems that aim to address the shortcomings of standard type inference. To the best of our knowledge, SimTyper is the first system to apply this style of type inference to Ruby.

JSNice [126] was one of the earliest probabilistic inference systems. JSNice represents JavaScript source as a dependency graph and uses conditional random fields to predict program properties, including type annotations. JSNice is limited to predicting a small set of types seen in training data. DeepTyper [119] trains a bidirectional recurrent neural network (RNN) classifier on JavaScript source code to predict type annotations from over 11,000 types seen in its training data. NL2Type [118] similarly trains an RNN, but exclusively on natural language information (i.e., identifier names and comments) from JavaScript programs. NL2Type predicts type annotations from a set of 1,000 types. Unlike SimTyper, none of the above approaches are able to predict types outside their training dataset, nor are
they sound.

TypeWriter [125] trains a neural model to predict types based on identifier names, comments, and source code from Python programs. After ranking the model’s predictions, TypeWriter uses a gradual type checker to rule out any inconsistent predictions, similarly to SimTyper’s use of constraints to rule out inconsistent predictions from DeepSim. However, unlike SimTyper, all of TypeWriter’s types come from the neural model, whereas SimTyper uses standard type inference to produce an initial set of solutions, and to propagate DeepSim’s solutions. Moreover, TypeWriter is restricted to predicting types from its training dataset, while SimTyper is not (§ 5.5.4).

Typilus [113] uses a graph neural network model to map program values to an embedding in a type space. Types are then predicted based on the similarities of embeddings. Typilus also checks predicted types against an optional type checker to rule out invalid types. Because new types can be added to the type space, Typilus, like SimTyper, is able to predict rare types. However, such types must be manually added to the type space. In contrast, in SimTyper, rare or user-defined types can be inferred by standard type inference or heuristics and then propagated through use of DeepSim. And, like TypeWriter, all types in Typilus come from the neural network model, whereas SimTyper starts with standard type inference.
5.7 Conclusion

We presented SimTyper, a system that combines standard type inference via constraint solving, manually written heuristics, and type equality prediction via the DeepSim network, in order to generate usable types. SimTyper iterates through the overly general type variable solutions remaining after constraint solving and heuristics. For each such type variable $\alpha$, it finds the usable type $\tau$ from the position most likely similar to $\alpha$, and then guesses $\alpha = \tau$. Guesses that are consistent with the other constraints are kept, and inconsistent guesses are discarded. In this way, even though DeepSim is probabilistic, SimTyper itself always makes sound inferences.

The DeepSim network operates by using CodeBERT to encode source tokens into a vector space and then pooling vectors that represent occurrences of the same argument or, for returns, the return positions in the code. A pair of encoded inputs is then run through a trained similarity function, which predicts whether those arguments or returns are likely to have the same type. The network is trained on a set of Ruby programs that include manual type documentation.

We evaluated SimTyper on eight Ruby benchmarks and found that combining constraint solving, heuristics, and type equality prediction results in inferring significantly more types that match hand-written types, compared to constraint solving alone. Moreover, we found that the DeepSim network can help to infer rare and program-specific types. Our results show that type equality prediction can help type inference systems effectively produce more usable types. Chapter 5 presents SimTyper in full.
Chapter 6: Conclusion

In this dissertation, I presented several ideas for making static type systems for dynamic languages more expressive and usable.

First, in Chapter 2, I presented RTR, a system that adds refinement types, basic types extended with expressive logical predicates, to the Ruby type checker RDL. RTR works by encoding its verification problems into Rosette, a solver-aided host language. RTR is able to verify ruby programs that use highly dynamic features, such as metaprogramming and mixins, using a combination of assume-guarantee reasoning and a novel approach called just-in-time verification, whereby program verification is deferred until runtime. We evaluated RTR by using it to verify a range of functional correctness properties in six Ruby programs.

Though RDL is useful for type checking, it still has trouble handling some constructs that are common in Ruby programs, such as databases and heterogeneous data structures. To address this, in Chapter 3 I introduced CompRDL, a system that extends RDL with type-level computations; we call types that include such computations comp types. With comp types, we can check more precise properties in Ruby programs, e.g., we can check the correctness of the names of columns and types of values in complex database queries. Additionally, due to the precision
of comp types, fewer manually inserted type casts are needed for operations over heterogeneous data structures such as arrays and hashes. We evaluated \texttt{CompRDL} by writing 586 comp type annotations over six Ruby libraries, then using these to type check six Ruby programs. We found that comp types significantly reduced the need for type casts when type checking these programs. Moreover, comp types helped us discover three previously unknown errors.

One difficulty in using \texttt{RDL} (as well as \texttt{RTR} and \texttt{CompRDL}) is that it places the burden of writing type annotations on the programmer. To address this issue, in Chapter 4 I introduced \texttt{InferDL}, a system for inferring usable type annotations. While standard, constraint-based type inference generates types that are technically correct, they are often complex, verbose, and overly general. \texttt{InferDL} complements constraint-based inference with configurable heuristic rules that guess more usable type annotations, i.e., annotations that precisely capture programmer intent. Heuristic guesses are checked for consistency with a program’s constraints, and any inconsistent guesses are discarded, thereby ensuring the soundness of inferred types. We evaluated \texttt{InferDL} by writing eight heuristics and running it on four benchmarks and six case study programs. We found that, compared to constraint-based inference, \texttt{InferDL} inferred 22% more annotations that were as or more precise than programmer-written annotations, and \texttt{InferDL} also found six new type errors in the process.

While \texttt{InferDL} helps generate more usable type annotations, there is still room for improvement. While in theory, a programmer could continue writing new heuristics for each new target program, this process is time-consuming, and the heuristics
may not generalize well to new targets. Thus, to improve on InferDL, in Chapter 5 I introduced SimTyper, a system that builds on InferDL by introducing a novel technique called type equality prediction. This machine learning-based technique uses a deep similarity (DeepSim) neural network to predict when two arguments, variables, or returns have the same type. This allows SimTyper to assign usable types to positions with overly general types. Like InferDL, any inferred annotations are checked against existing constraints, and thus are guaranteed to be sound. Moreover, unlike prior work on probabilistic type inference, SimTyper is able to infer rare and program-specific type annotations. We evaluated SimTyper by applying it to eight Ruby programs, and we found that, when using a combination of constraint solving, heuristic, and the DeepSim network, SimTyper finds 69% more types that match programmer-written annotations compared to constraint solving alone. Moreover, we found that the DeepSim network was able to infer more matching annotations than the heuristics of InferDL.

**High Level Conclusion.** Based on the novel ideas and promising results presented in this dissertation, I conclude that the proposed type systems represent an important step forward in bringing more expressive and usable static types to dynamic languages.

6.1 Future Work

There are a number of interesting avenues for future work.
Exploring Choice Types. Chapter 4 introduced choice types, an idea loosely inspired by variational type checking [89]. Choice types are used in InferDL and SimTyper for the specific purpose of considering multiple possible types for inputs to/outputs from a method with an intersection type. Subsequent constraints in a program allow us to eliminate inconsistent arms of a choice type. Thus, they provide us with a way of considering multiple possible types for a value, and eliminating those which are not possible as our program analysis provides more information.

Though we used choice types for this one specific purpose in InferDL and SimTyper, it is likely they would prove useful for other purposes. For instance, choice types could effectively represent the types of values received from a heterogeneous data structure, e.g., a value taken from an array containing both integers and strings. As we discussed in Chapter 3, we often need manually written type casts for such values since it is difficult for a type checker to statically determine the value’s type, which can lead to false-positive type errors. However, a choice type would allow us to consider multiple possible types for the value, and later eliminate inconsistent types rather than raise a type error. Choice types may also prove useful for other purposes, e.g., for handling path-sensitive typing in cases where a value’s type may differ depending on which path through a program is taken. Thus, I believe greater study of choice types for dynamic languages could prove interesting and useful.

Type Casts. One challenge that affects all four of the type systems presented in this dissertation is the need for manually written type casts to suppress false positive type errors. Though CompRDL reduced the number of casts needed for many
programs, it did not eliminate this need entirely. The necessity of manually written
type casts negatively affects the usability of these type systems, and thus it would be
useful to explore ways to reduce this need even further. There are multiple possible
routes for this exploration. First, we could explore ways to further increase the
precision of these type systems, thereby reducing the occurrence of false positive type
errors. The choice type discussion above suggests one way for increasing precision.
Another possibility would be to use occurrence typing [3], which could reduce false
positive errors resulting from path-sensitive typing; occurrence typing is challenging
in a language like Ruby, which provides many possible ways for dynamically checking
the type of a value, and thus this could be an interesting subject to study.

A different way to reduce the need for type casts would be to develop a means
for inferring type casts. This would necessitate (1) determining which program
locations could be responsible for false positive type errors, and (2) generating the
appropriate casts for these locations.

**Error Messages.** One challenge for type inference systems generally, and **InferDL**
and **SimTyper** in particular, is that error messages are often difficult to understand.
This can occur because type errors are sometimes related to many different locations
in a program, and thus choosing which locations to report in an error message is
nontrivial. Some prior work has suggested ways to improve error messages in similar
cases [101, 102, 103]; we plan to explore these for our systems in the future. It is
possible that this would also be helpful for one challenge with type casting presented
above: how to determine which program locations result in false positive errors.
Conflicts in Inferred Types. One challenge for both InferDL and SimTyper is that it is possible for the system to infer two or more type annotations that are inconsistent; in InferDL, this can occur when heuristics guess conflicting annotations, and in SimTyper it can occur when the DeepSim network guesses conflicting annotations. In both cases, the system takes a greedy approach and uses whichever annotation is guessed first, thereby effectively blocking out subsequent annotations that are inconsistent. In the future, we would like to explore alternative ways for handling such conflicts. One way could be to assign a confidence score to each inferred annotation, and pick those annotations with higher scores. Another approach may be to use backtracking in cases where inconsistencies occur.

Return Embeddings. As discussed in Section 5.5, we found that incorporating return sites into the embeddings representing method returns did not have a significant effect on SimTyper’s results for the benchmarks. This suggests that we are not effectively incorporating clues from a method body’s code as to the type of the method return. In the future, we plan to explore alternative ways of producing return embeddings.
Appendix A: Soundness of CompRDL

This appendix contains the full definitions, static and dynamic semantics, and proof of soundness for $\lambda^C$ (Chapter 3).

Figure A.1 once again presents the syntax of $\lambda^C$, as well as some new auxiliary definitions to be used for defining the semantics and proving soundness. Here we see for the first time object instances $[A]$, which denote an instance of a class $A$. The dynamic semantics rules are shown in Figure A.2. Most of the rules are standard. The slightly more intricate rule is (E-CONTEXT), which takes a step within a subexpression, and it contains premises that differentiate it from (E-AppUD), the other context cases, and ensure that the context $C$ is the largest possible context for purposes of disambiguation. Figure A.3 once again defines the dynamic check insertion rules. Finally, Figure A.4 defines the type checking rules, which are largely a simplification of the check insertion rules; it is for these type checking rules that we prove preservation and progress. We prove soundness of the check insertion rules as a corollary of the soundness of the type checking rules.
values \( v \) ::= nil \mid [A] \mid A \mid true \mid false

expressions \( e \) ::= \( v \mid x \mid self \mid tself \mid e;e \)
\hspace{1em} \mid A\text{.new} \mid \text{if } e \text{ then } e \text{ else } e \mid e == e
\hspace{1em} \mid e.m(e) \mid [A]e.m(e)

method types \( \sigma \) ::= \( A \rightarrow A \)
TLC method types \( \delta \) ::= \( \sigma \mid (a\ll\leqslant e/A) \rightarrow e/A \)

programs \( P \) ::= def \( A.m(x) : \sigma = e \mid \text{lib} \ A.m(x) : \delta \mid P;P \)

\( x \in \text{var IDs}, \ m \in \text{meth IDs}, \ A \in \text{class IDs} \)

\( \text{dyn env} \quad E : \quad \text{var ids} \rightarrow \text{values} \)

\( \text{contexts} \quad C ::= \quad \Box \mid C.m(e) \mid v.m(C) \mid [A]C.m(e) \mid [A]v.m(C) \)
\hspace{1em} \mid C;e \mid \text{if } C \text{ then } e \text{ else } e \mid C == e \mid v == C

\( \text{stack} \quad S ::= \quad \cdot \mid (E,C) :: S \)

\( \text{type stack} \quad TS ::= \quad \cdot \mid (\Gamma[A],A) :: TS \)

\( \text{typ env} \quad \Gamma,\Delta : \quad \text{var ids} \rightarrow \text{base types} \)

\( \text{class table} \quad CT : \quad \text{class ids} \rightarrow \text{meth ids} \rightarrow \text{types} \)

\( \text{objects} \quad O : \quad \text{objects} \)

\( \text{object instance} \quad [] : \quad A \rightarrow O \)

\( \text{method sets} \quad \mathcal{U} : \quad \text{set of user-defined methods} \)
\( \quad \mathcal{L} : \quad \text{set of library methods} \)
\hspace{1em} \text{where } \mathcal{L} \cap \mathcal{U} = \emptyset \)

\( \text{Nil}, \ Obj, \ Bool, \ True, \ False, \ \text{and } Type \) are all presumed to be class IDs \( A \).

Subtyping is defined as \( \text{Nil} \leqslant A, \ A \leqslant A, \text{ and } A \leqslant A \sqcup A' \) for all \( A, A' \). \( A \sqcup A' \) is the

least upper bound of types \( A \) and \( A' \).

Figure A.1: \( \lambda^C \) and auxiliary definitions.

### A.1 Soundness

We prove soundness of the type system of \( \lambda^C \) by first proving preservation

and then progress. The proof of preservation is the more involved step here, and it

requires a number of preliminary definitions. First, we define a notion of consistency

between the type and dynamic environments:

**Definition 1** (Environmental consistency). Type environment \( \Gamma \) is consistent with
Dynamic semantics

\[
\begin{align*}
(E,v,S) & \rightsquigarrow (E,E(v),S) \\
(E,\text{self},S) & \rightsquigarrow (E,E(\text{self}),S) \\
(E,\text{tself},S) & \rightsquigarrow (E,E(\text{tself}),S) \\
(E,v;e,S) & \rightsquigarrow (E,\langle e,S \rangle) \\
(E,A.\text{new},S) & \rightsquigarrow (E,[A],S) \\
(E,\text{if}\, v \text{ then } e_2 \text{ else } e_3,S) & \rightsquigarrow (E,e_2,S) \\
& \text{if } v \text{ is not nil or false} \\
(E,\text{if}\, e_2 \text{ else } e_3,S) & \rightsquigarrow (E,e_3,S) \\
& \text{if } v = \text{nil or } v = \text{false} \\
(E,v_1 == v_2,S) & \rightsquigarrow (E,\text{true},S) \\
& \text{if } v_1 \text{ and } v_2 \text{ are equivalent} \\
(E,v_1 == v_2,S) & \rightsquigarrow (E,\text{false},S) \\
& \text{if } v_1 \text{ and } v_2 \text{ are not equivalent} \\
(E,C[v_r,m(v)],S) & \rightsquigarrow ([\text{self} \mapsto v_r, x \mapsto v],e,(E,C) :: S) \\
& \text{if } \text{type}_\text{of}(v_r) = A \text{ and } A.m \in U \text{ and } \text{def}_\text{of}(A.m) = x.e \\
(E,[A]v_r,m(v),S) & \rightsquigarrow (E,v',S) \\
& \text{if } \text{type}_\text{of}(v_r) = A \text{ and } A.m \in L \text{ and } v' = \text{call}(A.m,v_r,v) \text{ and } \text{type}_\text{of}(v') \leq A \\
(E',v,(E,C) :: S) & \rightsquigarrow (E,C[v],S) \\
\end{align*}
\]

where \text{def}_\text{of}(A.m) = x.e if there exists a definition \text{def} \ A.m(x) : \sigma = e.

Additionally, \text{call}(A.m,v,v) is our way of dispatching a library method, where \text{A.m}
identifies the method, the first \text{v} is the receiver of the method call, and the second \text{v}
is the argument of the method call. It returns a value \text{v}_r, the value returned by the
method call. Finally, we use \text{type}_\text{of}(v), where \text{type}_\text{of}(\text{nil}) = \text{Nil}, \text{type}_\text{of}(\text{A})
= \text{Type}, \text{type}_\text{of}(\text{true}) = \text{True}, \text{type}_\text{of}(\text{false}) = \text{False}, \text{and } \text{type}_\text{of}([A]) = A.

Figure A.2: Dynamic semantics of \(\lambda^C\)
Type Checking and Check Insertion Rules.

\[ \Gamma \vdash_{CT} e \leftrightarrow e : A \]

\[ \begin{array}{ll}
\Gamma \vdash_{CT} \text{nil} & \leftrightarrow \text{nil} : \text{Nil} \quad \text{C-NIL} \\
\Gamma \vdash_{CT} \text{new} & \leftrightarrow \text{new} : A \quad \text{C-NEW}
\end{array} \]

\[ \begin{array}{ll}
\Gamma \vdash_{CT} A \leftrightarrow A : \text{Type} \quad \text{C-TYPE} \\
\Gamma \vdash_{CT} \text{true} & \leftrightarrow \text{true} : \text{True} \quad \text{C-TRUE}
\end{array} \]

\[ \begin{array}{ll}
\Gamma \vdash_{CT} \text{false} & \leftrightarrow \text{false} : \text{False} \quad \text{C-FALSE} \\
\Gamma \vdash_{CT} \text{var} & \leftrightarrow \text{var} : A \quad \text{C-Obj}
\end{array} \]

\[ \begin{array}{ll}
\Gamma \vdash_{CT} e_1 & \leftrightarrow e'_1 : A_1 \\
\Gamma_1 \vdash_{CT} e_2 & \leftrightarrow e'_2 : A_2
\end{array} \]

\[ \Gamma \vdash_{CT} e_1 ; e_2 & \leftrightarrow e'_1 ; e'_2 : A_2 \quad \text{C-SEQ} \]

\[ \begin{array}{ll}
\Gamma \vdash_{CT} e & \leftrightarrow e' : A \\
\Gamma \vdash_{CT} e_x & \leftrightarrow e'_x : A_x
\end{array} \]

\[ \Gamma \vdash_{CT} e.m(e_x) & \leftrightarrow e'.m(e'_x) : A_2 \quad \text{C-AppUD} \]

\[ \begin{array}{ll}
\Gamma \vdash_{CT} e_1 & \leftrightarrow e'_1 : A_1 \\
\Gamma \vdash_{CT} e_2 & \leftrightarrow e'_2 : A_2 \\
\Gamma \vdash_{CT} e_3 & \leftrightarrow e'_3 : A_3
\end{array} \]

\[ \Gamma \vdash_{CT} \text{if } e_1 \text{ then } e_2 \text{ else } e_3 & \leftrightarrow \text{if } e'_1 \text{ then } e'_2 \text{ else } e'_3 : (A_2 \sqcup A_3) \quad \text{C-IF} \]

\[ \begin{array}{ll}
CT(A.m) = A_1 & \rightarrow A_2 \\
A.m & \in U \quad A_x \leq A_1
\end{array} \]

\[ \Gamma \vdash_{CT} e & \leftrightarrow e' : A \\
\Gamma \vdash_{CT} e_x & \leftrightarrow e'_x : A_x
\]

\[ \Gamma \vdash_{CT} e.m(e_x) & \leftrightarrow [A_2]e'.m(e'_x) : A_2 \quad \text{C-AppLib} \]

\[ \begin{array}{ll}
CT(A.m) = (a_{<e_1/A_1}) & \rightarrow e_2/A_2 \\
A.m & \in \mathcal{L} \quad A_x \leq A_1
\end{array} \]

\[ \begin{array}{ll}
\Gamma \vdash_{CT} e & \leftrightarrow e' : A \\
\Gamma \vdash_{CT} e_x & \leftrightarrow e'_x : A_x
\end{array} \]

\[ \begin{array}{ll}
\langle [x \mapsto A] | \text{tself} \mapsto A | e'_{t_1}, \cdot \rangle & \rightsquigarrow^* \langle E_1, A_1, \cdot \rangle \\
\langle [x \mapsto A] | \text{tself} \mapsto A | e'_{t_2}, \cdot \rangle & \rightsquigarrow^* \langle E_2, A_2, \cdot \rangle
\end{array} \]

\[ \Gamma \vdash_{CT} e.m(e_x) & \leftrightarrow [A_2]e'.m(e'_x) : A_2 \quad \text{C-App-Comp} \]

Figure A.3: Type checking and check insertion rules for $\lambda^C$.
Type checking rules

\[ \Gamma \vdash_{CT} \text{nil} : \text{Nil} \quad \text{T-NIL} \]

\[ \Gamma \vdash_{CT} \text{[A]} : A \quad \text{T-OBJ} \]

\[ \Gamma(\text{self}) = A \quad \Gamma \vdash_{CT} \text{self} : A \quad \text{T-SELF} \]

\[ \Gamma \vdash_{CT} \text{true} : \text{True} \quad \text{T-TRUE} \]

\[ \Gamma \vdash_{CT} \text{false} : \text{False} \quad \text{T-FALSE} \]

\[ \Gamma \vdash_{CT} A : \text{Type} \quad \text{T-TYPE} \]

\[ \Gamma \vdash_{CT} \text{let} x = A \quad \text{T-VAR} \]

\[ \Gamma \vdash_{CT} e_1 == e_2 : \text{Bool} \quad \text{T-EQ} \]

\[ \Gamma \vdash_{CT} e_1 ; e_2 : A_2 \quad \text{T-SEQ} \]

\[ \Gamma \vdash_{CT} A.\text{new} : A \quad \text{T-NEW} \]

\[ \Gamma \vdash_{CT} \text{if} e_1 \text{ then } e_2 \text{ else } e_3 : (A_2 \sqcup A_3) \quad \text{T-IF} \]

\[ \Gamma \vdash_{CT} e_0 : A \quad A.m \in \mathcal{U} \]

\[ \Gamma \vdash_{CT} e_0.m(e_1) : A \quad \text{T-APP} \]

\[ \Gamma \vdash_{CT} e_0 : A \quad A.m \in \mathcal{L} \]

\[ \Gamma \vdash_{CT} \text{let} x = A \quad \text{T-APP-LIB} \]

Figure A.4: Type checking rules for \(\lambda^C\).
dynamic environment $E$, written $\Gamma \sim E$, if for all variables $x$, $x \in \text{dom}(E)$ if and only if $x \in \text{dom}(\Gamma)$, and for all $x \in \text{dom}(E)$ there exists $A$ such that $\Gamma \vdash_{CT} E(x) : A$ and $A \leq \Gamma(x)$.

We will use the notation $\text{type}_o(v)$, where $\text{type}_o(\text{nil}) = \text{Nil}$, $\text{type}_o(\text{true}) = \text{True}$, $\text{type}_o(\text{false}) = \text{False}$, $\text{type}_o([A]) = A$, and $\text{type}_o(A) = \text{Type}$, for any $A$.

On blame: Our type system does not prevent invoking a method on a value that is $\text{nil}$. Additionally, runtime evaluation can fail if an inserted dynamic check fails. In order to retain soundness of our system, we add dynamic semantics rules which step to blame in these cases, where $\Gamma \vdash_{CT} \text{blame} : \text{Nil}$ for any $\Gamma$. Additionally, we add rules which take a step to blame whenever a subexpression takes a step to blame. We omit the rules here for brevity.

Because we make use of a stack in our dynamic semantics, a standard type preservation theorem which says that we always step to an expression which has the same type (or subtype) will not suffice. Rules (E-AppUD) and (E-Ret) push and pop from the stack. In these cases, an expression $e$ may have an entirely different type than the expression that it steps to, $e'$. To account for this we incorporate a notion of a type stack $TS$ to mirror the runtime stack, which is defined Figure A.1.

As an example, suppose we want to apply preservation to $C[v_1.m(v_2)]$. The type checking judgment is $\Gamma \vdash_{CT} C[v_1.m(v_2)] : A'$. Because the dynamic semantics rule (E-AppUD) pushes the current environment and context onto the stack, we will push the current typing judgment on to the type stack. Specifically, we will push
an element of the form \((\Gamma[A], A')\), where \(\Gamma\) is the environment of the current typing judgment; \(A'\) is the type of the surrounding context; and \(A\) is the type of expression \(v_1.m(v_2)\), i.e., the type that the method must return.

With this type stack, we can now define what it means for a type to be a subtype of the type stack, which is the crucial preservation invariant we will prove:

**Definition 2** (Stack subtyping). \(A_0 \leq (\Gamma[A], A') :: TS\) if \(A_0 \leq A\).

**Definition 3** (Stack consistency). Type stack element \((\Gamma[A], A')\) is consistent with dynamic stack element \((E, C)\), written \((\Gamma[A], A') \sim (E, C)\), if \(\Gamma \sim E\) and \(\Gamma[\square \mapsto A] \vdash_{CT} C : A'\) (Here we abuse notation and treat \(\square\) as if its a variable.)

Type stack \(TS\) is consistent with dynamic stack \(S\), written \(TS \sim S\), is defined inductively as

1. \(. \sim .\)

2. \((\Gamma[A], A') :: TS \sim (E, C) :: S\) if

   (a) \((\Gamma[A], A') \sim (E, C)\)

   (b) \(TS \sim S\)

   (c) \(A' \leq TS\) if \(TS \neq .\)

We will also make use of a notion of class table validity, which tells us that a class table maps fields and methods to the "correct" types:

**Definition 4** (Class table validity). Let \(A.m\) be an arbitrary method. We say \(\text{valid}(CT)\) if
1. if \( A.m \in \mathcal{U} \), then \( A.m \in \text{dom}(CT) \), \( CT(A.m) = A_1 \rightarrow A_2 \) for some \( A_1, A_2 \), and there exists a single method definition \( \text{def} \ A.m(x) : A_1 \rightarrow A_2 = e \) such that \([\text{self} \mapsto A, x \mapsto A_1] \vdash_{CT} e : A'_2 \) for some \( A'_2 \) where \( A'_2 \leq A_2 \).

2. if \( A.m \in \mathcal{L} \), then \( A.m \in \text{dom}(CT) \), \( CT(A.m) = \sigma \) for some \( A_m \), and there exists a single method declaration \( \text{lib} \ A.m(x) : \sigma \) such that \( \sigma = A_1 \rightarrow A_2 \) or \( \sigma = (a < e_{t1}/A_1) \rightarrow e_{t1}/A_2 \) for some \( A_1, A_2, e_{t1}, e_{t2} \).

3. For all \( A' \) such that \( A' \leq A \), if \( CT(A'.m) = A'_1 \rightarrow A'_2 \) and \( CT(A.m) = A_1 \rightarrow A_2 \) then \( A_1 \leq A'_1 \) and \( A'_2 \leq A_2 \).

See section A.2 for the programming type checking rules, and a discussion of how to construct and check the validity of a class table.

Finally, we make use of the following lemmas in our proofs of soundness:

**Lemma 1** (Contextual substitution). If

\[
\frac{
\Gamma \vdash_{CT} e : A'}{
\Gamma \vdash_{CT} C[e] : A_C}
\]

then \( \Gamma_C[\square \mapsto A'] \vdash_{CT} C : A_C \).

**Lemma 2** (Substitution). If

1. \( \Delta[\square \mapsto A_C] \vdash_{CT} C : A'_C \)

2. \( \Delta \vdash_{CT} e : A \)

3. \( A \leq A_C \)
then $\Delta \vdash_{CT} C[e] : A'_C$, where $A'_C \leq A'_C$.

With the above definitions and lemmas, we can finally state our preservation theorem:

**Theorem 2 (Preservation).** If

1. $\langle E, e, S \rangle \rightsquigarrow \langle E', e', S' \rangle$
2. $\Gamma \vdash_{CT} e : A$
3. $A \leq TS$
4. $\Gamma \sim E$
5. $TS \sim S$
6. $\text{valid}(CT)$

then there exists $\Delta, TS', A'$ such that

a. $\Delta \vdash_{CT} e' : A'$

b. $A' \leq TS'$

c. $\Delta \sim E'$

d. $TS' \sim S'$

e. If $S = S'$ then $A' \leq A$ and $\Delta = \Gamma, \Gamma'$ for some $\Gamma'$

(1) and (2) are standard: they say that some expression $e$ takes a step, and that $e$ is well typed. Conclusion (a) states that $e'$ is also well typed. (3) says that
the type of \( e \) is a subtype of the type stack, and (b) says the same of \( e' \); this is the crux of the type preservation proof, and as explained above, it must be phrased in this way in order to account for the use of a stack, which allows us to step to expressions with completely different types. Notice that (e) also tells us that if the stack goes unchanged, then \( A' \leq A \); this is much closer to the standard statement of preservation. (e) also gives us that if the stack goes unchanged, then the new type environment is an extension of the old one.

(4) gives us consistency between type and dynamic environments, and (5) gives us consistency between the type stack and stack, the corresponding conclusions (c) and (d) respectively give us the same for the new environments.

(6) gives us validity of the class table (a corresponding conclusion is unnecessary since the class table goes unchanged).

Finally, we proceed with the proof of preservation.

Proof. By induction on \( \langle E, e, S \rangle \leadsto \langle E', e', S' \rangle \)

- Case (E-Self). By assumption we have

  (1) \( \langle E, \text{self}, S \rangle \leadsto \langle E, E(\text{self}), S \rangle \) by (E-Self)

  (2) \( \Gamma \vdash_{CT} \text{self} : A \)

  (3) \( \Gamma(\text{self}) \leq TS \)

  (4) \( \Gamma \sim E \)

  (5) \( TS \sim S \)

  (6) \( valid(CT) \)
Since (2) must have been derived by (T-SELF), by inversion of this rule we have that $A = \Gamma(\text{self})$. Let $\Delta = \Gamma$ and $TS' = TS$. By equality of $\Delta$ and $\Gamma$, (2), (4), and the definition of environmental consistency, there exists $A'$ such that $\Delta \vdash_{CT} E(\text{self}) : A'$ and $A' \leq \Delta(\text{self})$. Then (a) holds since $E(\text{self})$ is well typed. (b) holds since $A' \leq \Delta(\text{self}) = \Gamma(\text{self}) \leq TS$ by (3). Because $A' \leq \Gamma(\text{self})$ and $\Delta = \Gamma, \emptyset$, we have (e). Finally, (c) holds by (4), and (d) holds by (5).

- Case (E-Var), (E-TSELF). Similar to (E-Self) case.

- Case (E-New). By assumption we have

  (1) $\langle E, A.\text{new}, S \rangle \rightsquigarrow \langle E, [A], S \rangle$ by (E-New)

  (2) $\Gamma \vdash_{CT} A.\text{new} : A$ by (T-New)

  (3) $A \leq TS$

  (4) $\Gamma \sim E$

  (5) $TS \sim S$

  (6) valid($CT$)

  Let $\Delta = \Gamma$, $TS' = TS$, and $A' = A$. Then we get (a) from (T-Obj), (b) from (3), (c) from (4), (d) from (5), and (e) immediately by definition of $A'$ and $\Delta$.

- Case (E-SEQ). Trivial.

- Case (E-EqTrue), (E-EqFalse). Trivial.

- Case (E-IfTrue). By assumption we have
(1) \( \langle E, \text{if } v \text{ then } e_1 \text{ else } e_2, S \rangle \rightsquigarrow \langle E, e_1, S \rangle \)

(2) \( \Gamma \vdash_{CT} \text{if } v \text{ then } e_1 \text{ else } e_2 : A \)

(3) \( A \leq TS \)

(4) \( \Gamma \sim E \)

(5) \( TS \sim S \)

(6) \( \text{valid}(CT) \)

By (2) and inversion of (T-If), there exits \( A_v, A_1, A_2 \) such that:

(7) \( \Gamma \vdash_{CT} v : A_v \)

(8) \( \Gamma \vdash_{CT} e_1 : A_1 \)

(9) \( \Gamma \vdash_{CT} e_2 : A_2 \)

(10) \( A = A_1 \cup A_2 \)

Let \( \Delta = \Gamma, \ TS' = TS, \) and \( A' = A_1 \). Then from (8) we trivially have (a).

Additionally, \( A' = A_1 \leq A_1 \sqcup A_2, \) and so \( A' \leq TS' \) giving us (b). Finally, we have (c) by (4), (d) by (5), and (e) by the fact that \( A' \leq A \) and \( \Delta = \Gamma, \emptyset \).

- Case (E-IfFalse). Similar to (E-IfTrue) case.

- Case (E-Context). By assumption we have:

(1) \( \langle E, C[e], S \rangle \rightsquigarrow \langle E', C'[e'], S' \rangle \) where

(1a) \( \langle E, e, S \rangle \rightsquigarrow \langle E', e', S' \rangle \)

(1b) \( \neg(e = v_r.m(v) \land \text{type}_o f(x_r) = A \land A.m \in U) \) for some \( v, v_r, A, m \)
(1c) \( e \neq v \) for some \( v \)

(1d) \( e \neq C'[e'] \) for some \( C', e' \) by (E-CONTEXT)

(2) \( \Gamma \vdash_{CT} C[e] : A \)

(3) \( A \leq TS \)

(4) \( \Gamma \sim E \)

(5) \( TS \sim S \)

(6) valid(CT)

First, note that we must have \( S' = S \), because the only cases where this would not happen would be if (E-AppUD) or (E-Ret) were used to derive (1a), and this cannot be the case given (1b) and because (E-Ret) only applies to top-level values and thus can’t apply to a context. Now note that since (2) gives us \( \Gamma \vdash_{CT} C[e] : A \), by inversion, there should exist \( A_e \) so that

(7) \( \Gamma \vdash_{CT} e : A_e \)

Let \( TS_e \) be a type stack such that \( TS_e \sim S, A_e \leq TS_e, \) and all type environments in \( TS_e \) are the same as those in \( TS \); it is straightforward to construct such a \( TS_e \) from the existing \( TS \). Then, by (1a), (7), \( A_e \leq TS_e, \) (4), \( TS_e \sim S, \) and (6), we satisfy the premises of the preservation theorem. Therefore, applying the inductive hypothesis, there exists \( \Delta_e, TS'_e, A'_e \) such that:

(a) \( \Delta_e \vdash_{CT} e' : A'_e \)

(b) \( A'_e \leq TS'_e \)
\[(c_i) \quad \Delta_e \sim E'\]

\[(d_i) \quad TS'_e \sim S'\]

\[(e_i) \quad \text{If } S = S' \text{ then } A'_e \leq A_e \text{ and } \Delta_e = \Gamma, \Gamma' \text{ for some } \Gamma'\]

Let $\Delta = \Delta_e$ and $TS' = TS$. Now, because $S' = S$, by $(g_i)$ we have $\Delta = \Gamma, \Gamma'$.

By Lemma 1 we have $\Gamma[\square \mapsto A_e] \vdash_{CT} C : A$, and by the weakening lemma, we also have $\Delta[\square \mapsto A_e] \vdash_{CT} C : A$. By $(e_i)$ we also have $A'_e \leq A_e$. These, along with $(a_i)$ and the substitution lemma 2, give us that $\Delta \vdash_{CT} C[e]' : A'$ for some $A'$ where $A' \leq A$. This immediately gives us (a), and because $A' \leq A$ and $A \leq TS$ by (3) we get (b). Because $\Delta = \Gamma, \Gamma'$ and $A' \leq A$, we get (e). We get (c) from $(c_i)$. (d) comes from (5) and the fact that $S' = S$.

- **Case (E-AppLib).** By assumption we have

\[(1) \quad \langle E, [A_{res}]v_r.m(v_1), S \rangle \leadsto \langle E, v, S \rangle \text{ where}

\[(1a) \quad \text{type}_o f(v_r) = A_{rec}\]

\[(1b) \quad A_{rec}.m \in \mathcal{L}\]

\[(1c) \quad v = \text{call}(A_{rec}.m, v_r, v_1)\]

\[(1d) \quad \text{type}_o f(v) \leq A_{res}\]

by (E-AppLib).

\[(2) \quad \Gamma \vdash_{CT} [A_{res}]v_r.m(v_1) : A\]

\[(3) \quad A \leq TS\]

\[(4) \quad \Gamma \sim E\]
Because of (1b), we know that (2) must have been derived by (T-App-Lib). Instantiating this rule with the obvious bindings, we get that $A = A_{\text{res}}$. Let $\Delta = \Gamma$, $TS' = TS$, and $A' = \text{type\_of}(v)$. If $v$ is $\text{nil}$, we get (a) immediately by (1d), definition of $\text{type\_of}$, and (T-NIL); a similar argument holds for all other potential values of $v$. In all cases, we get (b) from (1d) and (3). We get (c) from (4), (d) from (5), and (e) from (1d) and definition of $\Delta$.

• Case (E-AppUD). By assumption we have

$$(1) \langle E, C[v_r.m(v)], S \rangle \rightsquigarrow ([\text{self} \mapsto v_r, x \mapsto v], e, (E, C) :: S)$$ where

$$(1a) \text{type\_of}(v_r) = A_{\text{rec}}$$

$$(1b) A_{\text{rec}.m} \in U$$

$$(1c) \text{def\_of}(A_{\text{rec}.m}) = x.e$$

$$(2) \Gamma \vdash_{CT} C[v_r.m(v)] : A_C$$

$$(3) A_C \leq TS$$

$$(4) \Gamma \sim E$$

$$(5) TS \sim S$$

$$(6) \text{valid}(CT)$$

Noting the type checking rules for each context case, we know (2) must have been derived by some rule of the form:
\[ \Gamma \vdash_{CT} v.r.m(v) : A_m \]
\[ \vdots \]
\[ \Gamma \vdash_{CT} C[v_r.m(v)] : A_C \]

From this and the contextual substitution lemma 1, we know \( \Gamma[\square \mapsto A_m] \vdash_{CT} C : A_C \). Additionally, by inversion, we have

(7) \( \Gamma \vdash_{CT} v_r.m(v) : A_m \)

By (1a), definition of type of, and the value type checking rules, we know that \( \Gamma \vdash_{CT} v_r : A_{rec} \). Because by (1b) we know that \( A_{rec}.m \in \mathcal{U} \) and \( \mathcal{U} \) and \( \mathcal{L} \) are by definition disjoint, the type checking judgment (7) must have been derived from rule T-App.

By instantiation of rule T-App with \( A_m = A_2 \) we get

\[
\begin{array}{c}
\Gamma \vdash_{CT} v_r : A_{rec} \\
\Gamma \vdash_{CT} v : A_{arg} \\
CT(A.m) = A_1 \rightarrow A_2 \\
A_{arg} \leq A_1 \\
\hline
\Gamma \vdash_{CT} v_r.m(v) : A_2
\end{array}
\]  

T-App

Thus, by inverting T-App, there must exist \( A_{arg}, A_1, A_2 \) such that:

(8) \( \Gamma \vdash_{CT} v : A_{arg} \)

(9) \( CT(A.m) = A_1 \rightarrow A_2 \)

(10) \( A_{arg} \leq A_1 \)

Now, let \( \Delta = [x \mapsto A_1, \text{self} \mapsto A_{rec}] \). Also, let \( TS' = (\Gamma[A_2], A_C) :: TS \).

By (1b), (1c), (6), (8), and (9), we know there exists \( A' \) such that \( [\text{self} \mapsto A_{rec}, x \mapsto A_1] \vdash_{CT} e : A' \) where \( A' \leq A_2 \), which gives us (a); that is to say,
the body of the method type checks as expected. (b) holds by construction of $TS'$. (c) holds by construction of $\Delta$. (e) holds trivially since $S \neq S'$.

Finally, we show (d). By (4) we have $\Gamma \sim E$, and as noted above we have $\Gamma[\square \mapsto A_2] \vdash_{CT} C : A_C$. This gives us $(\Gamma[A_2], A_C) \sim (E, C)$. By (3) we have $A_C \leq TS$, and by (5) we have $TS \sim S$. Putting this all together, by the definition of stack consistency, this gives us $(\Gamma_C[A_2], A_C) :: TS \sim (E, C) :: S$, which is (d).

- Case (E-Ret). By assumption we have

  (1) $\langle E', v, (E, C) :: S \rangle \leadsto \langle E, C[v], S \rangle$

  (2) $\Gamma \vdash_{CT} v : A$

  (3) $A \leq (\Gamma_C[A_C], A'_C) :: TS$

  (4) $\Gamma \sim E'$

  (5) $(\Gamma_C[A_C], A'_C) :: TS \sim (E, C) :: S$

  (6) valid$(CT)$

Let $\Delta = \Gamma_C$ and $TS' = TS$. By (5) we have $\Gamma_C[\square \mapsto A_C] \vdash_{CT} C : A'_C$, and by (3) we have $A \leq A_C$. Then by these, (2), and the substitution lemma 2, we have that $\Gamma_C \vdash_{CT} C[v] : A''_C$ where $A''_C \leq A'_C$; this also means that $\Delta \vdash_{CT} C[v] : A''_C$.

Let $A' = A''_C$ and we have (a). By (5) and the definition of stack consistency we have $A'_C \leq TS$, and since $A''_C \leq A'_C$, we have (b). Finally, (c) and (d) hold by (5), and (e) holds trivially since $S \neq S'$. 

196
Before proving progress, we introduce one assumption and one lemma:

**Assumption 1** (Library Method Termination). For any $A$, $m$, $v_1$, and $v_2$ where $A.m \in \mathcal{L}$, call$(A.m, v_1, v_2)$ will terminate and return a value.

This assumption is necessary to prove progress since we do not have the mechanism to refer to the step-by-step evaluation of library methods. A straightforward expansion of $\lambda C$ would allow us to do so, but we omit such an expansion here for simplicity.

**Lemma 3** (Proper context). For any expression $e$, if $e = C[e']$ for some $C, e'$, then there exists a proper context $C_P$ and proper subexpression $e_P$ such that $e = C_P[e_P]$, $e_P \neq v$ for any value $v$, and $\not\exists C', e''$ such that $e_P = C'[e'']$.

Note that it is still possible in Lemma 3 that $C_P = C$ and $e_P = e$. The high-level idea behind the proof of Lemma 3 is that we construct the proper context by recursively pushing the hole $[]$ deeper into subcontexts while possible. With these, we can proceed with defining and proving progress:

**Theorem 3** (Progress). If

1. $\Gamma \vdash_{CT} e : A$
2. $A \leq TS$
3. $\Gamma \sim E$
4. $TS \sim S$
(5) valid(CT)

then one of the following holds

1. e is a value

2. There exists $E', e', S'$ such that $\langle E, e, S \rangle \rightsquigarrow \langle E', e', S' \rangle$

3. $\langle E, e, S \rangle \rightsquigarrow$ blame

Proof. By induction on $e$.

- Case nil, true, false, [A], or A. $e$ is a value.

- Case self. By assumption (1) and (T-Self), we know self $\in dom(\Gamma)$. By (3), this means self $\in dom(E)$. This means rule (ESelf) can be applied, so we can take a step.

- Cases x, tself. Similar to self.

- Case $e_1 == e_2$ We split into cases on whether or not $e_1, e_2$ are values, for any $v_1, v_2$:

  - $e_1 \equiv v_1$ and $e_2 \equiv v_2$. This case is trivial since one of (E-EQTRUE) or (E-EQFALSE) will always apply.

  - $e_1 \not\equiv v_1$. Then $e \equiv C[e_1]$ with $C \equiv [] == e_2$. By Lemma 3, there exists a proper context $C_P$ and proper subexpression $e_P$ such that $e \equiv C_P[e_P]$. If $e_P \not\equiv v_r.m(v)$ for any $v_r, A, m, v$ where $\text{type}_\_ of(v_r) = A_r \land A_r.m \in U$, then by the inductive hypothesis there exists $E', e'_P, S'$ such that $\langle E, e_P, S \rangle \rightsquigarrow$
\( \langle E', e'_P, S' \rangle \) (or such that we step to blame, in which case \( e \) steps to blame and we are done). See the (E-CONTEXT) case of the preservation proof for a discussion of how to satisfy the premises of the inductive hypothesis, since the premises here a subset of those of preservation. By construction of the proper subexpression as specified in Lemma 3, \( \forall C', e'' \) such that \( e_P = C'[e''] \), and \( e_P \neq v \) for any value \( v \). This satisfies all the premises of (E-CONTEXT), therefore this rule would apply to \( C_P[e_P] \) to take a step.

Otherwise, \( e_P \equiv v_r.m(v) \) with \( \text{type}_o f(v_r) = A_r \land A_r.m \in U \). We know that (1) must have been derived by rule (T-EQ). By inversion of this rule, this means \( \Gamma \vdash_{CT} v_r.m(v) : A_m \) for some \( A_m \). By definition of \( \text{type}_o f \), it must be that \( A_m = A_r \). Because \( A_r.m \in U \) and by definition \( U \cap L = \emptyset \), we therefore know \( \Gamma \vdash_{CT} v_r.m(v) : A_m \) must have been derived from (T-App). By inversion of this rule, we know \( CT(A_r.m) = A_1 \to A_2 \) for some \( A_1, A_2 \). By (5), this means \( \text{def}_o f(A_r.m) = x.e_m \) for some \( e_m \). If \( v_r \) is \text{nil}, then we return blame. Thus, we have satisfied all the premises of rule (E-APPUD), therefore we can apply this rule and we are done.

- \( e_1 \equiv v_1 \) and \( e_2 \neq v_2 \). Then \( e \equiv C[e_2] \) with \( C \equiv v_1 == [] \). By a similar argument to the previous case, either (E-CONTEXT) or (E-APPUD) must apply.

- Case \( e_1; e_2 \). We split cases on whether or not \( e_1 \) is a value:

  - \( e_1 \equiv v \). Then, the evaluation rule E-SEQ on the expression \( v; e_2 \) applies to take a step.

\[ 199 \]
Otherwise, $e \equiv C[e_1]$ with $C \equiv []; e_2$. By a similar argument to the $e_1 == e_2$ case, either (E-CONTEXT) or (E-AppUD) must apply.

- Case $A$.new. Trivial.

- Case $if \ e_0 \ then \ e_1 \ else \ e_2$. We split cases on the structure of $e_0$.
  - $e_0 \equiv \text{nil}$ or $e_0 \equiv \text{false}$. The rule E-IfFalse applies to take a step.
  - $e_0 \equiv v$ where $v$ is not nil or false. The rule E-IfTRUE applies to take a step.
  - Otherwise, $e \equiv C[e_0]$ with $C \equiv \text{if} \ [] \ \text{then} \ e_1 \ \text{else} \ e_2$. By a similar argument to the $e_1 == e_2$ case, either (E-CONTEXT) or (E-AppUD) must apply.

- Case $e_1.m(e_2)$. We split this case on the structure of $e_1, e_2$:
  - $e_1 \neq v$. Then $e \equiv C[e_1]$ with $C \equiv [] .m(e_2)$. By a similar argument to the $e_1 == e_2$ case, either (E-CONTEXT) or (E-AppUD) must apply.
  - $e_1 \equiv v_1$ and $e_2 \neq v_2$. Then $e \equiv C[e_2]$ with $C \equiv e_1.m([])$. By a similar argument to the $e_1 == e_2$ case, either (E-CONTEXT) or (E-AppUD) must apply.
  - $e_1 = v_1$ and $e_2 = v_2$. If $v_1$ is nil, we step to blame. Because $e$ is a non-checked method call, we know (1) must have been derived by (T-App). By inversion of this rule, $\Gamma \vdash_{CT} v_1 : A_1$ for some $A_1$, and that $A_1.m \in \mathcal{U}$ and $CT(A.m) = A_{in} \to A_{out}$. By (7), this means $\text{def}_o f (A.m) = x.e_m$ for
some $e_m$. This satisfies all the premises of rule (E-AppUD), therefore we can apply this rule.

- Case $[A_{res}]e_1.m(e_2)$. We split this case on the structure of $e_1, e_2$:

  - $e_1 \not\equiv v$. Then $e \equiv C[e_1]$ with $C \equiv []_.m(e_2)$. By a similar argument to the $e_1 == e_2$ case, either (E-CONTEXT) or (E-AppUD) must apply.

  - $e_1 \equiv v_1$ and $e_2 \not\equiv v_2$. Then $e \equiv C[e_2]$ with $C \equiv e_1.m([])$. By a similar argument to the $e_1 == e_2$ case, either (E-CONTEXT) or (E-AppUD)

  - $e_1 = v_1$ and $e_2 = v_2$. If $v_1$ is nil, we return blame. Because $e$ is a checked method call, we know (1) must have been derived by (T-AppLib). By inversion of this rule, we know $\Gamma \vdash_{CT} v_1 : A_1$ for some $A_1$, where $A_1.m \in L$. Let $v_{res} = call(A.m, v_1, v_2)$; by Assumption 1 we know that this call will terminate and return a value. Then, if $\text{type}_\circ(v_{res})$ is not a subtype of $A_{res}$, we will return blame. Otherwise, we will have satisfied all the preconditions of (E-AppLib), therefore we can apply this rule and take a step.

\[
\square
\]

We now introduce our theorem of soundness of the type checking judgment.

**Theorem 4** (Soundness of Type Checking). *If valid(CT) and $\emptyset \vdash_{CT} e \leftrightarrow e' : A_C$ and $\emptyset \vdash_{CT} e' : A$, then either $e'$ reduces to a value, $e'$ reduces to blame, or $e'$ does not terminate.*
Program type checking rules

\[ CT(A.m) = A_1 \rightarrow A_2 \quad [\text{self} \mapsto A, x \mapsto A_1] \vdash_{CT} e : A'_2 \]
\[ A'_2 \leq A_2 \]
\[ CT \vdash \text{def} A.m(x) : A_1 \rightarrow A_2 = e \]  
\[ \text{T-PDef} \]

\[ CT(A.m) = \sigma \]
\[ CT \vdash \text{lib} A.m(x) : \sigma \]  
\[ \text{T-PLib} \]

\[ CT \vdash P_1 \quad CT \vdash P_2 \]
\[ CT \vdash P_1, P_2 \]  
\[ \text{T-PSeq} \]

Figure A.5: Program type checking rules.

Proof. Let \( \Gamma = \emptyset, E = \emptyset, S = (\emptyset, \square) :: \cdot \), and \( TS = (\Gamma[A], A) :: \cdot \). By construction we have \( A \leq TS, \Gamma \sim E, \) and \( TS \sim S \). Thus, we satisfy the preconditions of progress and preservation, and soundness holds by standard argument.

With this soundness theorem, it is straightforward to extend soundness to the check insertion rules. We make use of a lemma that states that the type assigned by the check insertion rules will be equivalent to the type assigned by the type checking rules.

**Lemma 4.** \( \Gamma \vdash_{CT} e \leftrightarrow e' : A_C \) and \( \Gamma \vdash_{CT} e' : A \) if and only if \( \Gamma \vdash_{CT} e \leftrightarrow e' : A \).

Proof. Straightforward by induction on check insertion rules \( \Gamma \vdash_{CT} e \leftrightarrow e' : A \).

**Theorem 5** (Soundness of Check Insertion). If valid(CT) and \( \emptyset \vdash_{CT} e \leftrightarrow e' : A \) then either \( e' \) reduces to a value, \( e' \) reduces to blame, or \( e' \) does not terminate.

Proof. By Theorem 4 and Lemma 4.
A.2 Program Type Checking and Class Table Construction

The rules for type checking a program are given in Figure A.5. They rely on using a class table. We omit a formal definition of the class table construction here as it is straightforward. Informally: to construct a class table, traverse a program, adding type annotations from definitions and declarations as you go. We can then check that $CT \models P$ to ensure that the program $P$ type checks under the constructed class table. If $CT \models P$, and the appropriate subtyping relations hold among methods of subclasses, we can conclude that $\text{valid}(CT)$ according to definition 4.
Appendix B: SimTyper Evaluation Data

This appendix contains the data corresponding to the graphs in Chapter 5, as well as some additional data.

<table>
<thead>
<tr>
<th>Program</th>
<th># Meths</th>
<th>LoC</th>
<th># Args</th>
<th># Vars</th>
<th># Usable</th>
<th># Overly General</th>
<th># No Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>code.org</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Discourse</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Journey</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Talks</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MiniMagick</td>
<td>39</td>
<td>271</td>
<td>28</td>
<td>13</td>
<td>45</td>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>Money</td>
<td>54</td>
<td>350</td>
<td>36</td>
<td>38</td>
<td>83</td>
<td>29</td>
<td>16</td>
</tr>
<tr>
<td>Ronin</td>
<td>41</td>
<td>367</td>
<td>8</td>
<td>69</td>
<td>92</td>
<td>17</td>
<td>9</td>
</tr>
<tr>
<td>TZInfo</td>
<td>10</td>
<td>60</td>
<td>14</td>
<td>57</td>
<td>48</td>
<td>23</td>
<td>10</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>144</strong></td>
<td><strong>1048</strong></td>
<td><strong>86</strong></td>
<td><strong>188</strong></td>
<td><strong>277</strong></td>
<td><strong>81</strong></td>
<td><strong>60</strong></td>
</tr>
</tbody>
</table>

Table B.1: Data on the types that SimTyper inferred, but we did not have gold standards to compare against. For each program, we list the number of method types SimTyper inferred for which there was no gold standard, the lines of code those methods comprised, the number of argument types those methods include, and the number of instance, class, and global variables SimTyper inferred a type for. Note that we did not include a column for the number of return types in the methods, since this is equivalent to the number of methods. Then, of the total argument, return, and variable types SimTyper inferred, we list the number of these that were usable, overly general, and the number for which SimTyper failed to infer any type.
<table>
<thead>
<tr>
<th>Program</th>
<th>Match</th>
<th>Match up to Param.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C/CH/CD/CHD</td>
<td>C/CH/CD/CHD</td>
</tr>
<tr>
<td>code.org</td>
<td>59/106/87/119</td>
<td>14/19/14/19</td>
</tr>
<tr>
<td>Discourse</td>
<td>45/55/54/57</td>
<td>2/3/2/3</td>
</tr>
<tr>
<td>Journey</td>
<td>39/47/45/48</td>
<td>1/2/3/3</td>
</tr>
<tr>
<td>MiniMagick</td>
<td>25/29/31/35</td>
<td>1/1/2/2</td>
</tr>
<tr>
<td>Money</td>
<td>58/67/82/81</td>
<td>6/6/13/11</td>
</tr>
<tr>
<td>Ronin</td>
<td>102/134/161/167</td>
<td>144/126/16/15</td>
</tr>
<tr>
<td>Talks</td>
<td>105/136/138/140</td>
<td>178/159/42/45</td>
</tr>
<tr>
<td>TZInfo</td>
<td>117/166/261/266</td>
<td>472/388/101/101</td>
</tr>
<tr>
<td>Total</td>
<td>550/740/859/913</td>
<td>60/73/112/120</td>
</tr>
</tbody>
</table>

(a) Number of inferred matches and matches up to parameter.

<table>
<thead>
<tr>
<th>Program</th>
<th>Different (Structural)</th>
<th>No Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C/CH/CD/CHD</td>
<td>C/CH/CD/CHD</td>
</tr>
<tr>
<td>code.org</td>
<td>67(28) / 25(13) / 56(12) / 20(5)</td>
<td>23 / 13 / 6 / 5</td>
</tr>
<tr>
<td>Discourse</td>
<td>17(14) / 11(7) / 14(10) / 11(6)</td>
<td>13 / 8 / 7 / 6</td>
</tr>
<tr>
<td>Journey</td>
<td>7(7) / 2(2) / 6(3) / 4(0)</td>
<td>10 / 6 / 3 / 2</td>
</tr>
<tr>
<td>MiniMagick</td>
<td>8(5) / 8(5) / 12(2) / 10(2)</td>
<td>23 / 19 / 12 / 10</td>
</tr>
<tr>
<td>Money</td>
<td>47(41) / 41(26) / 47(25) / 47(12)</td>
<td>41 / 38 / 10 / 13</td>
</tr>
<tr>
<td>Ronin</td>
<td>81(53) / 68(29) / 123(23) / 119(14)</td>
<td>144/126/16/15</td>
</tr>
<tr>
<td>Talks</td>
<td>38(26) / 21(7) / 37(13) / 29(7)</td>
<td>40 / 19 / 5 / 5</td>
</tr>
<tr>
<td>TZInfo</td>
<td>149(127) / 119(78) / 129(59) / 122(26)</td>
<td>178/159/42/45</td>
</tr>
<tr>
<td>Total</td>
<td>414(301)/295(167)/424(147)/362(72)</td>
<td>472/388/101/101</td>
</tr>
</tbody>
</table>

(b) Number of inferred different types, and number of positions for which SimTyper did not infer any type.

Table B.2: SimTyper evaluation results corresponding to the plots in Figure 5.8. For each benchmark, we list the number of matching, match up to parameter, and different inferred types measured under all configurations (C/CH/CD/CHD). For the different category, in parentheses we show the number of those types that were structural. Additionally, the No Type category indicates the algorithm could not find a more usable solution than giving a type variable for that position.
Table B.3: This table corresponds to the graphs in Figure 5.9. Measuring SimTyper’s performance for arguments, variables, and returns. For each category, we list the number of match, match up to parameter, and different inferred types measured under all configurations (C/CH/CD/CHD). For the different category, in parentheses we show the number of those types that were structural.

<table>
<thead>
<tr>
<th>Cat.</th>
<th>Match</th>
<th>Match up to Param.</th>
<th>Different (Structural)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Args</td>
<td>69 / 208 / 279 / 322</td>
<td>5 / 17 / 23 / 32</td>
<td>339(296) / 222(166) / 268(146) / 219(71)</td>
</tr>
<tr>
<td>Vars</td>
<td>52 / 57 / 61 / 61</td>
<td>9 / 11 / 13 / 14</td>
<td>6(4) / 4(2) / 9(1) / 8(1)</td>
</tr>
<tr>
<td>Ret</td>
<td>429 / 475 / 519 / 530</td>
<td>46 / 45 / 76 / 74</td>
<td>69(0) / 69(0) / 147(0) / 135(0)</td>
</tr>
</tbody>
</table>

Table B.4: This table corresponds to the graphs in Figure 5.10a. Measuring SimTyper’s performance under top-1, -3, -5, and -7 configurations. Measurements were taken under the CD configuration. For the different category, in parentheses we show the number of those different types that were structural types.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Match</th>
<th>Match up to Param.</th>
<th>Different (Structural)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top-1</td>
<td>819</td>
<td>108</td>
<td>449(172)</td>
</tr>
<tr>
<td>top-3</td>
<td>859</td>
<td>112</td>
<td>424(147)</td>
</tr>
<tr>
<td>top-5</td>
<td>860</td>
<td>112</td>
<td>423(144)</td>
</tr>
<tr>
<td>top-7</td>
<td>860</td>
<td>113</td>
<td>422(143)</td>
</tr>
</tbody>
</table>

Table B.5: This table corresponds to the graphs in Figure 5.10b. Measuring SimTyper’s performance under for two different methods of generating embeddings for arguments: Averaging vectors for all uses of an argument (All) or using just the vector for the argument in the method header (Head). Measurements were taken under the CD configuration. For the different category, in parentheses we show the number of those different types that were structural types.

<table>
<thead>
<tr>
<th>Embedding Method</th>
<th>Match</th>
<th>Match up to Param.</th>
<th>Different (Structural)</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>859</td>
<td>112</td>
<td>424(147)</td>
</tr>
<tr>
<td>Head</td>
<td>792</td>
<td>107</td>
<td>463(176)</td>
</tr>
</tbody>
</table>
Table B.6: This table corresponds to the graphs in Figure 5.10c. Measuring SimTyper’s performance under for two different methods of generating embeddings for returns: averaging method names and return sites (N+S), or using just method names (N). Measurements were taken under the CD configuration. For the different category, in parentheses we show the number of those different types that were structural types.

<table>
<thead>
<tr>
<th>Embedding Method</th>
<th>Match</th>
<th>Match up to Param.</th>
<th>Different (Structural)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N+S</td>
<td>859</td>
<td>112</td>
<td>424(147)</td>
</tr>
<tr>
<td>N</td>
<td>866</td>
<td>103</td>
<td>432(147)</td>
</tr>
</tbody>
</table>
Bibliography


213


