SimTyper: Sound Type Inference for Ruby using Type Equality Prediction

MILOD KAZEROUNIAN, University of Maryland, USA
JEFFREY S. FOSTER, Tufts University, USA
BONAN MIN, Raytheon BBN Technologies, USA

Many researchers have explored type inference for dynamic languages. However, traditional type inference computes most general types which, for complex type systems—which are often needed to type dynamic languages—can be verbose, complex, and difficult to understand. In this paper, we introduce SimTyper, a Ruby type inference system that aims to infer *usable* types—specifically, nominal and generic types—that match the types programmers write. SimTyper builds on `InferDL`, a recent Ruby type inference system that soundly combines standard type inference with heuristics. 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, types inferred by SimTyper are guaranteed to be sound. To perform type equality prediction, we introduce the *deep similarity* (DeepSim) neural network. DeepSim is a novel machine learning classifier that 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 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.

CCS Concepts: • **Software and its engineering** → *Data types and structures.*

Additional Key Words and Phrases: type inference, dynamic languages, machine learning, Ruby

ACM Reference Format:
Milod Kazerounian, Jeffrey S. Foster, and Bonan Min. 2021. SimTyper: Sound Type Inference for Ruby using Type Equality Prediction. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 106 (October 2021), 27 pages. https://doi.org/10.1145/3485483

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 [Aiken et al. 1994; Cartwright and Fagan 1991; Flanagan and Felleisen 1997; Kazerounian et al. 2019; Lerner et al. 2013; Ren and Foster 2016; Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2008; Vitousek et al. 2014].

Authors' addresses: Milod Kazerounian, University of Maryland, College Park, Maryland, USA, milod@cs.umd.edu; Jeffrey S. Foster, Tufts University, Medford, Massachusetts, 02155, USA, jfoster@cs.tufts.edu; Bonan Min, Raytheon BBN Technologies, Cambridge, Massachusetts, 02138, USA, bonan.min@raytheon.com.

2475-1421/2021/10-ART106
https://doi.org/10.1145/3485483

In this setting, type inference [Aiken and Murphy 1991; Anderson et al. 2005; Aycock 2000; Furr et al. 2009; Hackett and Guo 2012] 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 § 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, 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 paper, we introduce SimTyper, a type inference system that aims to infer usable types for Ruby. SimTyper is built on top of `InferDL`, a recent Ruby type inference system that combines constraint solving with manually written heuristics whose guesses are checked against the constraints, to ensure soundness [Kazerounian et al. 2020]. SimTyper uses the same basic infrastructure, but adds a novel *deep similarity* (DeepSim) 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. (§ 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 α with an overly general solution, SimTyper finds the type variable β that is most similar to α and has a usable type solution. SimTyper then adds a constraint $\alpha = \tau$, where τ is β's solution, and runs constraint resolution again. If the constraints are still consistent, α's solution is set to τ. 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. (§ 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 [Feng et al. 2020], 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` [Segal 2020] 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. (§ 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 [Allamanis et al. 2020], we count the number of cases where the 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 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.

2 OVERVIEW

We begin by illustrating how SimTyper is used to infer a type annotation for the method shown in Figure 1a, which is extracted and simplified from *TZInfo*, one of the benchmarks in our experiments (§ 5). The method, Timestamp.create, takes three integers representing a year, month, and day corresponding to a date, and returns a new 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.[1] The subsequent lines (line 8–13) perform the computation. The details of the computation [Hinnant 2013] 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 Timestamp (code omitted here) with the computed number of seconds.

2.1 Standard Type Inference

The first step of SimTyper, inherited from InferDL, is to perform standard, constraint-based type inference [Furr et al. 2009], which begins by assigning a *type variable* to each unknown. Here, as shown on line 2, we assign α, β, and γ to the parameters (in that order) and δ to the return type. SimTyper then analyzes the method body, generating *constraints* of the form $a \leq b$, meaning that a is a subtype of b. We also say a is a *lower bound* on b and b is an *upper bound* on a.

Figure 1b shows several of the constraints generated for this example. Constraint (1), from line 4, states that α, the type of year, must define a method kind_of? that takes a Class and returns a fresh unknown type ϵ. Or, more formally, α is a subtype of the *structural type* [kind_of? : Class $\rightarrow \epsilon$]. 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

[1] Amusingly, we could in theory create a heuristic (§ 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 SimTyper is that it can discover useful types even without manually creating heuristics.

```
1  class Timestamp
2    # Assigned method type: (α, β, γ) → δ
3    def self.create (year, month, day)
4      raise ArgumentError, 'year must be an Integer' unless year.kind_of?(Integer)
5      raise ArgumentError, 'month must be an Integer' unless month.kind_of?(Integer)
6      raise ArgumentError, 'day must be an Integer' unless day.kind_of?(Integer)
7
8      after_february = month > 2
9      year = year - 1 unless after_february
10     era = year / 400 # eras are 400 year periods
11     day_of_year = day + (153 * (month + ...
12      ... # additional computation
13     value = ... * 24 * 60 * 60 # seconds since unix time
14
15     new(value)
16   end
17 end
```

(a) A method from the TZInfo library.

(1) $\alpha \leq$ [kind_of? : Class $\rightarrow \epsilon$]	(6) $\alpha \leq$ [/ : Number $\rightarrow \kappa$]
(2) $\beta \leq$ [kind_of? : Class $\rightarrow \zeta$]	(7) $\gamma \leq$ [+ : Number $\rightarrow \lambda$]
(3) $\gamma \leq$ [kind_of? : Class $\rightarrow \eta$]	(8) $\beta \leq$ [+ : Number $\rightarrow \mu$]
(4) $\beta \leq$ [> : Number $\rightarrow \theta$]	(9) Timestamp $\leq \delta$
(5) $\alpha \leq$ [- : Number $\rightarrow \iota$]	

(b) Constraints generated on type variables.

Fig. 1. Generating type constraints in *TZInfo*.

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. For example, if $a \leq b$ and $b \leq c$, then SimTyper applies transitive closure to generate a constraint $a \leq c$. In this particular case, because Timestamp.create is relatively simple and is considered in isolation, resolution does not change the set of constraints. See § 2.3 for more details of constraint resolution.

After constraint resolution, the constraints are in *solved form* [Pottier 1998], 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. 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. Full details of this *solution extraction* process can be found in prior work [Kazerounian et al. 2020].

For the constraints in Figure 1b, SimTyper finds the following solutions:

$$\begin{array}{lcl} \alpha & = & [\text{kind\_of?} : \text{Class} \to \epsilon,\ \text{-} : \text{Number} \to \iota,\ / : \text{Number} \to \kappa] \\ \beta & = & [\text{kind\_of?} : \text{Class} \to \zeta,\ > : \text{Number} \to \theta,\ + : \mathit{Number} \to \mu] \\ \gamma & = & [\text{kind\_of?} : \text{Class} \to \eta,\ + : \text{Number} \to \lambda] \\ \delta & = & \text{Timestamp} \end{array}$$

Out of these four solutions, the only one that matches developer-provided documentation (not shown) is δ, which has a simple, easy-to-understand nominal type. In contrast, the solutions for α, β, and γ, 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. In our experience, standard type inference with subtyping and structural types often produces such difficult-to-use types.

2.2 Heuristic Type Inference

To address this problem, SimTyper builds on an approach pioneered by `InferDL`, recent work that adds heuristic rules on top of standard type inference. The heuristics 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 α, β, and γ 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[2], singleton types (which represent just a single value), and the boolean type.

Critically, when a heuristic rule guesses a solution, that guess is added as an additional constraint to the type inference problem. Guesses that are consistent with the other constraints are kept; guesses that are inconsistent are retracted and alternate heuristics, if any remain, are applied. Thus, even though `InferDL` is heuristic, it is guaranteed to produce sound types.

For example, `InferDL` includes a rule STRUCT-TO-NOMINAL, which is defined 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." § 2.2, [Kazerounian et al. 2020]

When SimTyper and `InferDL` apply STRUCT-TO-NOMINAL to our running example, the heuristic guesses that the solution for α (corresponding to the parameter year) is Number, because the only existing classes that define methods kind_of?, -, and / are Ruby's numeric classes. This guess is consistent with the other constraints, so SimTyper and `InferDL` would both set α = Number as the solution.

However, STRUCT-TO-NOMINAL fails to infer a new solution for β, because more than ten classes define the set of methods {kind_of?, >, +}—e.g., possible classes include String, Set, Time, and others—and similarly for γ. 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 id has type Number. However, while this is an excellent guess for Rails code, in other codebases, e.g., in the the Stripe codebase, ids are generally Strings [Petrashko 2020].

[2] `InferDL`'s more precise types for hashes and arrays. Finite hash types give the exact type of each key and value in a hash, and tuple types give the type and position of each element in an array.

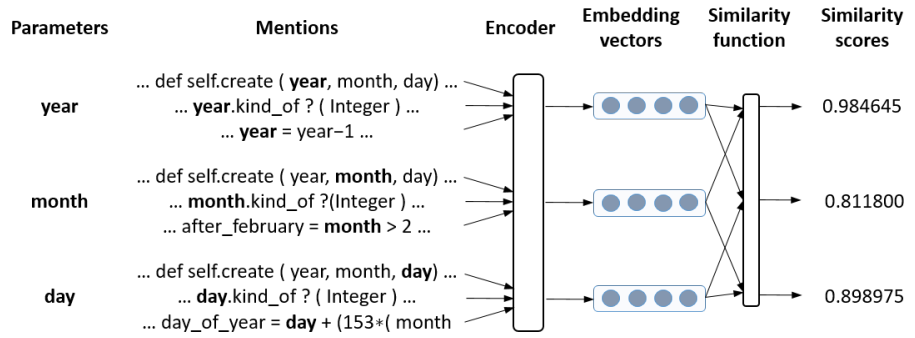


Fig. 2. An illustration of how DeepSim calculates the pairwise similarity scores for the set of input parameters {year, month, day} for the method from Figure 1a.

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. These guesses can either be used in place of or in addition to guesses from hand-written heuristics. Like `InferDL`, these guesses are checked for consistency with the underlying constraints to ensure soundness, and any inconsistent guesses are discarded.

More specifically, SimTyper uses a novel network based on the Siamese network architecture [Koch et al. 2015], which we call a *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 year, month, and day are closely related words in English. Moreover, in 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 SimTyper previously determined that year has type 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 [Koch et al. 2015]. 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 language model trained on programming and natural languages [Feng et al. 2020] to produce a fixed-dimension *contextualized vector representation* for the input. This is a numeric vector that encodes not only the 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 [Brown et al. 2020] and programming language [Feng et al. 2020; Kanade et al. 2020] 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 2 illustrates how DeepSim calculates the pairwise similarity scores for the set of input parameters {year, month, day} for the method from Figure 1a. More details on the encoder and the similarity function are in Section 4.

```
1 class UserHelpers
2   def self.initial(name) # (α) → β
3     return name.strip[0].upcase
4 end end
```

(1) $\alpha \leq [\text{strip} : () \rightarrow \gamma]$
(2) $\gamma \leq [[] : (\text{Number}) \rightarrow \delta]$
(3) $\delta \leq [\text{upcase} : () \rightarrow \epsilon]$
(4) $\epsilon \leq \beta$

Fig. 3. A method defined in the *code.org* app and the resulting constraints.

SimTyper applies the DeepSim network after standard inference and hand-written 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 evaluates choices for N).

For example, picking up from the heuristic guesses in § 2.2, SimTyper asks DeepSim for the expected type similarity among year, which has a usable solution at this point, and month and day, which have overly general solutions. As shown in Figure 2, the network has a very high degree of confidence that year and month have the same type, so SimTyper guesses that $\beta = \alpha = \text{Number}$. This guess is consistent, and so it is accepted. Next, day is predicted to be most likely similar to 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 (Number, Number, Number) → Timestamp for Timestamp.create, which matches the hand-written documentation.

Cascading Type Predictions. In some cases, guesses made via DeepSim can cascade through the constraints, leading to further usable solutions. For example, consider the code snippet in Figure 3 extracted and simplified from *code.org*, one of our benchmarks (§ 5). This code defines a class method initial that, given a String called name, returns the first non-whitespace character in name as an upper-case letter.

SimTyper assigns initial the type $(\alpha) \rightarrow \beta$ and generates the constraints shown in the right of Figure 3. Constraints (1), (2), and (3) result from the calls to strip, [], and upcase, respectively, and constraint (4) results from the return. Using 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 $= \alpha$ is added as the solution. Propagating this yields...
(2) String $\leq [\text{strip} : () \rightarrow \gamma]$. Looking up the type of String#strip (SimTyper includes types for Ruby's core and standard libraries) yields...
(3) String $\leq \gamma$. In other words, strip returns a String. Propagating further yields...
(4) (in several steps) String $\leq \delta$, i.e., [0] also returns a String. Propagating further yields...
(5) (in several steps) String $\leq \epsilon$, i.e., upcase returns a String. Propagating to ϵ's upper bound...
(6) String $\leq \beta$.

$$
\begin{array}{lrcl}
\textit{Types} & \tau & ::= & \alpha \mid A \mid [m : \tau \to \tau] \mid \tau \cup \tau \mid \tau \cap \tau \mid \bot \mid \top \\
\textit{Constraints} & C & ::= & \tau \leq \tau \mid C \cup C \\
\textit{Solutions} & S & ::= & \{\alpha_1 \mapsto \tau_1, ..., \alpha_n \mapsto \tau_n\}
\end{array}
$$

$$\alpha \in \text{type vars} \quad A \in \text{class names} \quad m \in \text{meth names}$$

$$\textit{generate} : \mathcal{P} \to C \times \textit{Vars} \qquad \textit{resolve} : C \to C \cup \{\textit{error}\} \qquad \textit{solve} : C \times \textit{Vars} \to S$$

$$\mathcal{P} \in \text{programs} \quad \textit{Vars} \in \text{list of type vars}$$

Fig. 4. Types, constraints, solutions, and type inference functions.

Then, using the usual rules for computing the solution at a return position, we can set $\beta = \texttt{String}$ also. Thus, we have found that the method has type $(\texttt{String}) \to \texttt{String}$, 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 a rule that guesses that arguments called `name` have type `String`. While there are many cases from our benchmarks where DeepSim alone leads to a cascading solution, we present the example of Figure 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 [Allamanis et al. 2020; Hellendoorn et al. 2018; Malik et al. 2019], as well as our own dataset (§ 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.

3 TYPE INFERENCE ALGORITHM

In this section, we discuss SimTyper's type inference algorithm more formally, and in the following section we discuss the implementation of SimTyper in detail.

3.1 Standard Type Inference

We begin by establishing some basic definitions and describing the standard type inference procedures that SimTyper inherits. The top part of Figure 4 defines a core language of types, constraints, and solutions. Types τ can either be a type variable α, a nominal type A, a structural type $[m : \tau \to \tau]$ (for simplicity we assume methods only take one argument), union types $\tau \cup \tau$, intersection types $\tau \cap \tau$, the bottom type $\bot$, or the top type $\top$. Constraints C take the form $\tau_1 \leq \tau_2$, meaning τ_1 is a subtype of τ_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 the figure gives the types for three functions that together implement standard type inference. We assume that these functions exist but do not discuss their implementation. We direct the reader to Kazerounian et al. [2020] for a more detailed treatment of the standard inference algorithm with integrated heuristics.

The *generate* function takes a program as input and produces a set of constraints generated from the program (e.g., the constraints in Figure 1b) and a list of type variables for which we want solutions

```
procedure STANDARD_INFERENCE(P)
    C, Vars ← generate(P)
    C_new ← resolve(C)
    if C_new = error then
        fail
    return solve(C_new, Vars), C_new
```

(a) Standard type inference.

```
procedure TYPE_EQ_INFERENCE(P)
    S, C ← STANDARD_INFERENCE(P)
    S, C ← HEURISTIC_INFERENCE(S, C)
    repeat
        for α ∈ dom(og_sols(S)) do
            C ← DEEPSIM_SOL(α, S, C)
            S ← solve(C, Vars)
    until C is unchanged
```

(c) Type equality-based inference.

```
procedure DEEPSIM_SOL(α, S, C)
    potential_sols ← {(τ_i, s) | α_i ↦ τ_i ∈ useable_sols(S), s = simscore(α, α_i), and s ≥ 0.5}
    k ← 1
    while k ≤ size(potential_sols) ∧ k ≤ N do
        pick kth highest scoring solution τ_k in potential_sols
        C_new ← resolve(C ∪ τ_k ≤ α ∪ α ≤ τ_k)
        if C_new ≠ error then
            return C_new
        k ← k + 1
    return C
```

(b) Finding a DeepSim solution.

Fig. 5. The procedures of SimTyper.

(i.e., types for method arguments and returns). The *resolve* function performs constraint resolution on its input set of constraints, applying rules like transitive closure ($a \le b \wedge b \le c \Rightarrow a \le c$). It produces either the solved form of those constraints or *error* if the constraints are inconsistent. For example, in Ruby, the constraint String $\le$ Integer would result in an error.

Finally, the *solve* function is given a set of constraints and a list of type variables, and it produces a solution for those type variables. This function follows the process described in § 2.1, taking the intersection of upper bounds for argument type variables, and the union of lower bounds for returns. An empty union produces $\bot$, and an empty intersection produces $\top$.

With these functions, we can define the STANDARD_INFERENCE procedure shown in Figure 5a. Given a program $\mathcal{P}$, standard inference generates constraints over the program, resolves those constraints, and then produces a solution over the relevant type variables. We also return the generated set of constraints from the program for use in the type equality prediction algorithm. Note that, if the constraints are inconsistent, standard inference will fail to produce a solution as the program is ill-typed. This could result from a true type error or due to a false positive. In the latter case, the programmer must provide a type cast to suppress the error; as discussed in § 5, we needed to do this in some cases in our experiments.

3.2 Type Equality Prediction Algorithm

Following Kazerounian et al. [2020], we say that a solution from standard type inference is *overly general*, denoted $og(\tau)$, if τ is one of α, $\tau_1 \cup \tau_2$, $\tau_1 \cap \tau_2$, $\bot$, or $\top$. That is, $og(\tau)$ holds whenever τ is

not a nominal type. Note that this definition is heuristic and thus may not be airtight in all cases, e.g., a programmer may want to use a union type in an annotation. We choose this definition of overly general since, in our experience, programmers typically write nominal types, but we are interested in exploring alternative definitions in the future.

With these definitions, we can partition a solution S into the usable solutions *useable_sols* and the overly general solutions *og_sols*:

$$useable\_sols(S) = \{\alpha \mapsto \tau \in S \mid \neg og(\tau)\} \qquad og\_sols(S) = \{\alpha \mapsto \tau \in S \mid og(\tau)\}$$

Next we introduce the function *simscore* to represent the DeepSim neural network. Given two type variables α_1 and α_2, $simscore(\alpha_1, \alpha_2) \in [0, 1]$ 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.

Now we can introduce DEEPSIM_SOL, the procedure for finding a single DeepSim network solution, which we define in Figure 5b. Given a type variable α, a solution S, and a constraint set C, the function DEEPSIM_SOL returns an updated constraint set C_{new}, which either includes a new solution for α if one was found, or is the same as C if no solution was found.

The first line of DEEPSIM_SOL defines *potential_sols* to be the set of pairs of potential type solutions and their corresponding similarity scores. The set is constructed by comparing α with every $\alpha_i \in dom(useable\_sols(S))$, and keeping the corresponding type solution τ_i when its similarity score is above 0.5. Then the function loops, picking the highest scoring solution τ_k in *potential_sols*. The function then "tests" the solution by equating it to α and running constraint resolution. If this succeeds, we have found a consistent guess, so the function returns the new set of constraints. 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, since no consistent guess was found.

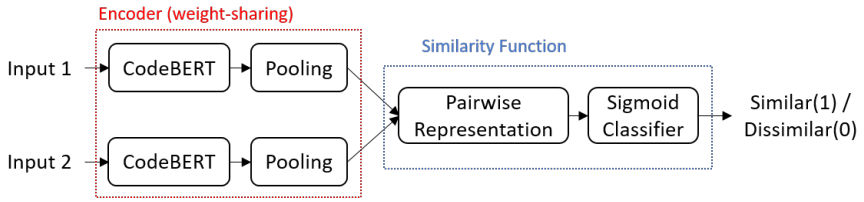
Finally, we can present SIMTYPER's overall algorithm, the TYPE_EQ_INFERENCE procedure defined in Figure 5c. TYPE_EQ_INFERENCE begins by executing standard inference. It then continues with heuristic inference (see Kazerounian et al. [2020] for details; for convenience, here we assume a procedure with slightly different inputs and outputs than in that presentation). Then, for all type variables α with overly general solutions, it invokes DEEPSIM_SOL to guess a usable solution, if possible. Since this additional solution could cascade (§ 2.3), the constraints are solved again after finding a usable solution. Notice this might remove an overly general solution from $og\_sols(S)$ (which strictly shrinks as only usable solutions are added by the algorithm). The overall process repeats until no new solutions are added.

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.

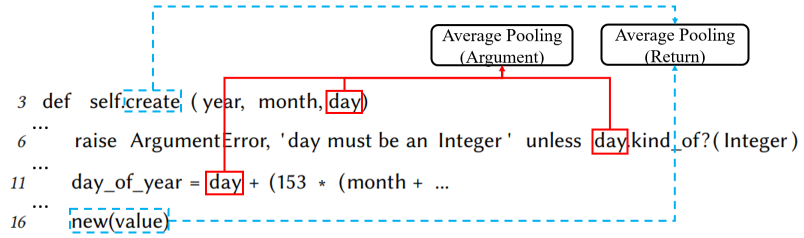
4 IMPLEMENTING THE DEEPSIM NETWORK

The DeepSim network is implemented, trained, and run in Python using the `TensorFlow` library. Because `InferDL` is implemented in Ruby, SIMTYPER uses a local web server to interface between `InferDL` and the DeepSim network.

Figure 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



(a) Diagram of the DeepSim network.



(b) Two types of pooling in DeepSim, demonstrated on the method from Figure 1a. After converting each position into contextualized vectors via CodeBERT, the blue dashed lines show the average pooling for the method return, and the red solid lines show the average pooling for the argument day.

Fig. 6. SimTyper's Deep Similarity (DeepSim) Neural Network.

same type. Following the Siamese Network structure [Koch et al. 2015], the encoder used for both inputs is the same. Next, we 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 arg_1 and arg_2 are the method arguments to be compared, and that X_i is the tokenized source code of the method in which 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 arg_1 at position i_k in the first sequence, and each $x_2^{j_k}$ represents the mention of 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 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 [Feng et al. 2020], a transformer-based [Vaswani et al. 2017] pre-trained code embedding model.[3] 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

[3] CodeBERT was trained on over 8.5 million methods and functions from programs written in Ruby, Java, JavaScript, Go, PHP, and Python.

```
# Multiplies the monetary value with the given number and returns a new
# +Money+ object with this monetary value and the same currency.
# @param [Numeric] value Number to multiply by.
# @return [Money] The resulting money.
def *(value) ... end
```

Fig. 7. A method * from the *Money* library which has YARD documentation.

$x_1^{i_k}$ for arg_1, and CV_2^k is the representation of token $x_2^{j_k}$ 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 pooling, which is shown to be effective in Siamese Network models for text similarity tasks [Reimers and Gurevych 2019]. For example, as illustrated in Figure 6b for our running example, for the argument day, the pooling layer averages the vectors corresponding to its mentions in the method header and method body. For the return of 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).

Formally, the output of this layer is two fixed-dimensional vectors $V_1 = MeanPool(CV_1^1, \ldots, CV_1^m)$ and $V_2 = MeanPool(CV_2^1, \ldots, CV_2^n)$, where $MeanPool$ 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 [2019], 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 $d = 768$ is the dimensionality of the CodeBERT contextualized vectors) and bias term $b \in \mathbb{R}$, and then pass the result through the sigmoid function σ, 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 ith pair), DeepSim is trained with the Adam optimizer [Kingma and Ba 2015] by minimizing the binary cross-entropy loss, which is a common choice for binary classification tasks:

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

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 [Foster et al. 2018], which InferDL is built on, Sorbet [Stripe 2020],

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. [2014] 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 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 ued 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). 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. [2020], which they showed 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 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 [Kazerounian et al. 2020]. We refer to these as the *InferDL Benchmarks*:

- *code.org* [Code.org 2021] – the code.org website app
- *Discourse* [Inc. 2021] – online discussion platform
- *Journey* [Budin 2021] – site for creating surveys and collecting responses
- *Talks* [Foster 2021] – 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* [Ross 2021] – library for manipulating timezone data
- *MiniMagick* [MiniMagick 2021] – wrapper for the ImageMagick image manipulation platform
- *Ronin* [Postmodern 2021] – platform for vulnerability research and exploit development
- *Money* [Emmons and Dmitriyev 2021] – 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 (§ 4.1).

Table 1. Benchmark statistics.

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

Table 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.

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 6 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 [Allamanis et al. 2020], 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 ∪ 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, just 8 use structural 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.

All results can be reproduced using the publicly available artifact [Kazerounian et al. 2021].

5.1 SimTyper Results

Figure 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 § 3). Below, unless we say otherwise,

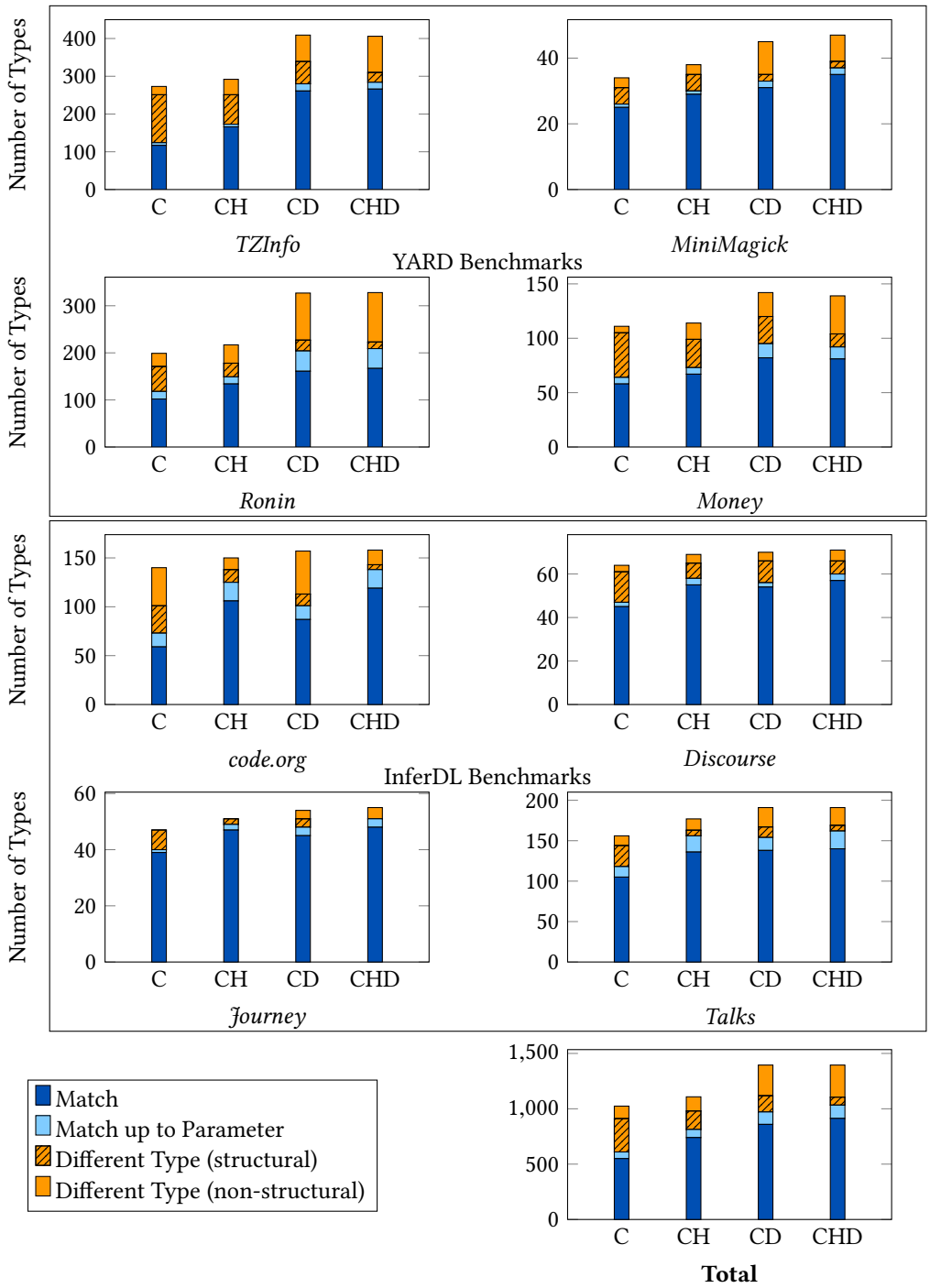


Fig. 8. Assessing the types inferred by SimTyper. We collected results under four configurations: constraint solving (C), constraint solving plus heuristics (CH), constraint solver plus the DeepSim network (CD), and all three (CHD). Results presented here use top-3 thresholding. Note the y-axis is scaled for each benchmark.

comparisons of *matching* sum both *Match* and *Match up to Parameter*. Table 7 in the Appendix 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.

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 [Kazerounian 2021]. 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, 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 A, 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 7 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. [2019], we compute precision as n_{match}/n_{type}, where n_{match} is the number of type matches (including up to parameter) and n_{type} is the total number of positions (arguments, returns, and variables) for which SimTyper inferred any type. We compute recall as n_{match}/n_{all}, where n_{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.

Table 2. Precision and recall of SimTyper.

Config	Precision	Recall
C	59.6%	40.8%
CH	73.4%	54.3%
CD	69.6%	64.9%
CHD	74.1%	69.1%

We note that our notion of precision and recall is slightly different from Allamanis et al. [2020]. They compute precision as $n_{neutral}/n_{type}$, where $n_{neutral}$ 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, the paper computes recall as n_{type}/n_{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, 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 (§ 4.1). Figure 9 measures SimTyper's performance separately for these three groups. The data for these plots are included in Appendix A.

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)

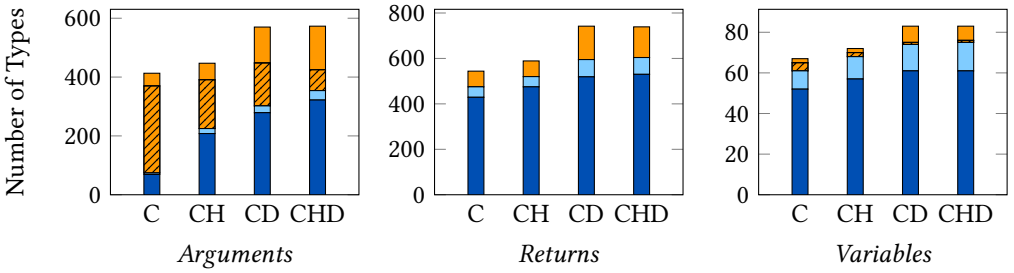


Fig. 9. Measuring SimTyper's performance for arguments, returns, and instance, class, and global variables across all benchmarks. The plots use the same legend as Figure 8. Measurements were taken using top-3 thresholding.

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, recall that we could not train a network specifically on variables since we did not have this data (§ 4). Second, we found that incorporating return sites into return embeddings does not significantly improve performance (§ 5.5). We leave exploring other ways to incorporate method code into DeepSim's predictions to future work.

5.2 Performance

Table 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× and 50× 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`.

Table 3. Running time of SimTyper over nine runs.

Program	Median Time (s) ± SIQR			
	C	CH	CD	CHD
code.org	4.0 ± 0.03	124 ± 0.36	72 ± 1.52	189 ± 0.27
Discourse	1.0 ± 0.10	32 ± 0.09	33 ± 0.33	62 ± 0.23
Journey	0.7 ± 0.06	3 ± 0.11	42 ± 0.31	41 ± 0.40
MiniMagick	0.5 ± 0.02	2 ± 0.02	39 ± 0.66	37 ± 0.37
Money	1.0 ± 0.03	5 ± 0.12	72 ± 1.91	71 ± 1.07
Ronin	2.0 ± 0.04	12 ± 0.21	172 ± 2.23	177 ± 2.22
Talks	2.0 ± 0.10	6 ± 0.21	68 ± 0.80	70 ± 1.35
TZInfo	2.0 ± 0.04	8 ± 0.21	156 ± 0.99	161 ± 0.75
Total	**13.2** ± 0.42	**192** ± 1.33	**654** ± 8.75	**808** ± 6.66

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.

5.3 Comparing DeepSim and Heuristics

Table 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 Appendix A and Kazerounian et al. [2020]), the table lists how many types the heuristic matched in CH followed by how many of those matches DeepSim also predicted in CD.

Table 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 DeepSim were taken using the top-3 threshold.

Heuristic Rule	H Matches	DS Matches
STRUCT-TO-NOMINAL	61	17 (28%)
IS_MODEL	21	18 (86%)
IS_PLURALIZED_MODEL	6	2 (33%)
INT_NAMES	32	5 (16%)
INT_ARRAY_NAME	3	0 (0%)
PREDICATE_METHOD	32	21 (66%)
STRING_NAME	18	16 (89%)
HASH_ACCESS	9	0 (0%)
Total	**182**	**79 (43%)**

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 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.

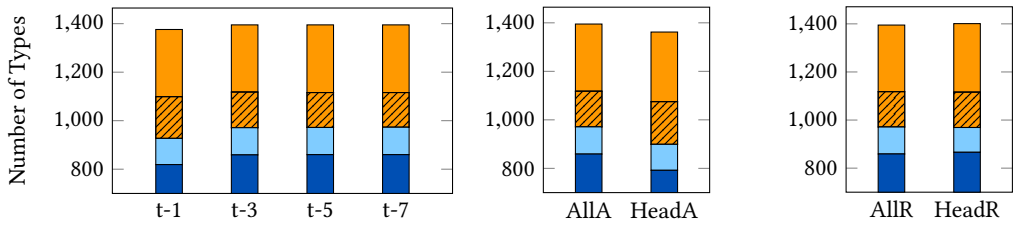
5.4 Predicting Rare Types

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 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 (§ 6).

Table 5. Numbers of Library, Training, and Program types guessed by DeepSim across all benchmarks, under CD with top-3 cutoff.

Library		Training		Program	
Match incl. param.	All	Match incl. param.	All	Match incl. param.	All
240	356	12	13	16	38

Table 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).



(a) Comparing top-1, -3, -5, and -7 thresholding.

(b) Comparing arg embeddings using all arg uses (AllA) vs. just the header (HeadA).

(c) Comparing return embeddings, using method names and return sites (AllR) vs. names only (HeadR).

Fig. 10. Evaluating design choices in SimTyper. Both plots use the same legend as in Figure 8 and show data from CD across all benchmarks. Note y-axis starts at 700. Data for all plots appears in Appendix A.

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, and 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. In total, 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 libraries or in the training data. Thus, a machine learning-based direct classifier would have no ability to predict these types. 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.

5.5 SimTyper Design Choices

Finally, we evaluate two design choices in SimTyper. First, Figure 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 A. 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 10b compares pooling the vectors for all uses of an argument (AllA), as in § 4, with using just the argument in the method header (HeadA). Figure 10c similarly compares pooling method names and return sites (AllR), also as in § 4, to using just method names (NameR). 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.

6 RELATED WORK

There are several threads of related work.

Standard Type Inference. Type inference has a long history [Curry and Feys 1958], including the well-known Hindley-Milner-Damas type inference algorithm [Damas and Milner 1982; Hindley

1969; Milner 1978]. In our experience, because the algorithm is based on unification, the most general types it infers can usually be presented to programmers without too much difficulty (though the addition of more complex features like module systems can hinder usability). Researchers have extended type inference to subtyping systems [Aiken et al. 1994; Cartwright and Fagan 1991; Pottier 1998; Wright and Cartwright 1997] as well, and this has enabled the development of practical type inference systems for dynamic languages such as Python [Aycock 2000], JavaScript [Anderson et al. 2005; Hackett and Guo 2012], and Ruby [Furr et al. 2009]. 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, SimTyper aims to infer usable types.

Probabilistic Type Inference. 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 [Raychev et al. 2015] 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 [Hellendoorn et al. 2018] trains a bidirectional recurrent neural network (RNN) on JavaScript source code to predict type annotations from over 11,000 types seen in its training data. NL2Type [Malik et al. 2019] 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 [Pradel et al. 2019] 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 (§ 5.4).

Typilus [Allamanis et al. 2020] 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.

Types for Ruby. Furr et al. [2009] introduced DRuby, an early static type inference system for Ruby. DRuby is focused exclusively on catching type errors in Ruby programs, rather than generating type annotations. Much of their expressive type system, which includes union, intersection, optional, and structural types, has been incorporated in to `InferDL`, on which SimTyper is built. Ren and Foster [2016] introduce Hummingbird, a Ruby type checking system that checks programs at runtime in order to support metaprogramming. This is a core idea behind RDL [Foster et al. 2018], a Ruby type checker on which `InferDL` and SimTyper are built. Finally, Kazerounian et al. [2019] introduced the idea of type-level computations, which are also incorporated into SimTyper and greatly reduce the need for type casts in the presence of arrays, hashes, and Rails database queries.

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 α, it finds the usable type τ from the position most likely similar to α, 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.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This research was supported in part by NSF grants CCF-1918233 and DGE-1840340, by the IARPA BETTER Program via Contract No.: 2019-19051600006.

A APPENDIX

Table 6. 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.

Program	# Meths	LoC	# Args	# Vars	# Usable	# Overly General	# No Type
code.org	0	0	0	1	0	1	0
Discourse	0	0	0	8	8	0	0
Journey	0	0	0	0	0	0	0
Talks	0	0	0	2	1	1	0
MiniMagick	39	271	28	13	45	10	25
Money	54	350	36	38	83	29	16
Ronin	41	367	8	69	92	17	9
TZInfo	10	60	14	57	48	23	10
Total	**144**	**1048**	**86**	**188**	**277**	**81**	**60**

Table 7. SimTyper evaluation results corresponding to the plots in Figure 8. For each benchmark, we list the number of matching, match up to parameter, and different inferred types measured under all for configurations (C/CH/CD/CHD). Unlike the plots in Figure 8, we also list the number of cases for which SimTyper didn't infer any type. For the different category, in parentheses we show the number of those different types that were structural types. The *No Type* category indicates the algorithm could not find a more usable solution than giving a type variable for that position.

Program	Match	Match up to Param.	Different (Structural)	No Type
	C /CH/CD/CHD	C /CH/CD/CHD	C / CH / CD / CHD	C /CH/CD/CHD
code.org	59 /106/ 87 / 119	14/ 19 / 14 / 19	67(28) / 25(13) / 56(12) / 20(5)	23 / 13 / 6 / 5
Discourse	45 / 55 / 54 / 57	2 / 3 / 2 / 3	17(14) / 11(7) / 14(10) / 11(6)	13 / 8 / 7 / 6
Journey	39 / 47 / 45 / 48	1 / 2 / 3 / 3	7(7) / 2(2) / 6(3) / 4(0)	10 / 6 / 3 / 2
MiniMagick	25 / 29 / 31 / 35	1 / 1 / 2 / 2	8(5) / 8(5) / 12(2) / 10(2)	23 / 19 / 12 / 10
Money	58 / 67 / 82 / 81	6 / 6 / 13 / 11	47(41) / 41(26) / 47(25) / 47(12)	41 / 38 / 10 / 13
Ronin	102/134/161/ 167	16/ 15 / 43 / 42	81(53) / 68(29) / 123(23) /119(14)	144/126/ 16 / 15
Talks	105/136/138/ 140	13/ 20 / 16 / 22	38(26) / 21(7) / 37(13) / 29(7)	40 / 19 / 5 / 5
TZInfo	117/166/261/ 266	7 / 7 / 19 / 18	149(127)/ 119(78) / 129(59) /122(26)	178/159/ 42 / 45
Total	550/740/859/ 913	60/ 73 /112/ 120	414(301)/295(167)/424(147)/362(72)	472/388/101/ 101

Table 8. This table corresponds to the graphs in Figure 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 for configurations (C/CH/CD/CHD). For the different category, in parentheses we show the number of those different types that were structural types.

Category	Match	Match up to Param.	Different (Structural)
	C /CH/CD/CHD	C /CH/CD/CHD	C / CH / CD / CHD
Args	69 /208/279/ 322	5 / 17 / 23 / 32	339(296)/222(166)/268(146)/219(71)
Vars	52 / 57 / 61 / 61	9 / 11 / 13 / 14	6(4) / 4(2) / 9(1) / 8(1)
Rets	429/475/519/ 530	46/ 45 / 76 / 74	69(0) / 69(0) / 147(0) / 135(0)

For easy reference, here are descriptions of the heuristics in `InferDL` [Kazerounian et al. 2020]. `RDL` [Foster et al. 2018], mentioned below, is the Ruby type checker `InferDL` (and hence SimTyper also) is built on.

- IS_MODEL: When an argument has the same name as a Rails *model* class, the variable is given the the nominal type of that class. A model is a special Rails class that maps to a table in the backend database. For example, if a model called User exists, an argument called user would be assigned the nominal type User.
- IS_PLURALIZED_MODEL: Similar to the above rule, when an argument is the pluralized version of a model class u, the argument is given the union type Array<u> ∪ ActiveRecord_Relation<u>, where ActiveRecord_Relation is a special type of array provided by Rails.
- INT_NAMES: If a variable (for arguments and non-local variables) or method (for returns) name ends with id, num, or count, guess the type Integer.
- INT_ARRAY_NAME: If a variable or method name ends with ids, nums, or counts, guess Array<Integer>.
- PREDICATE_METHOD: If a method name ends with a ?, guess %bool (RDL's boolean type) for the return type.

- STRING_NAME: If a variable or method name ends with name, guess String.
- HASH_ACCESS: A rule for generating finite hash types, RDL's precise version of a hash type. If all of a variable type's upper bounds are structural types for the methods [] and []=, use inputs/outputs for these structural types to guess a corresponding finite hash type.

Table 9. This table corresponds to the graphs in Figure 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.

Embedding Method	Match	Match up to Param.	Different (Structural)
All	859	112	424(147)
Head	792	107	463(176)

Table 10. This table corresponds to the graphs in Figure 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.

Embedding Method	Match	Match up to Param.	Different (Structural)
N+S	859	112	424(147)
N	866	103	432(147)

Table 11. This table corresponds to the graphs in Figure 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.

Configuration	Match	Match up to Param.	Different (Structural)
top-1	819	108	449(172)
top-3	859	112	424(147)
top-5	860	112	423(144)
top-7	860	113	422(143)

REFERENCES

Alex Aiken and Brian Murphy. 1991. Static Type Inference in a Dynamically Typed Language. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Orlando, Florida, USA) *(POPL '91)*. Association for Computing Machinery, New York, NY, USA, 279–290. https://doi.org/10.1145/99583.99621

Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. 1994. Soft Typing with Conditional Types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. ACM, New York, NY, USA, 163–173. https://doi.org/10.1145/174675.177847

Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 91–105. https://doi.org/10.1145/3385412.3385997

Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards Type Inference for Javascript. In *ECOOP 2005 - Object-Oriented Programming (ECOOP)*. Springer, Berlin, Heidelberg, 428–452. https://doi.org/10.1007/11531142_19

John Aycock. 2000. Aggressive Type Inference.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

Nat Budin. 2021. Journey: An online questionnaire application. https://github.com/nbudin/journey/.

Robert Cartwright and Mike Fagan. 1991. Soft Typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 278–292. https://doi.org/10.1145/113445.113469

Code.org. 2021. The code powering code.org and studio.code.org. https://github.com/code-dot-org/code-dot-org.

H. B. Curry and R. Feys. 1958. *Combinatory Logic, Volume I.* North-Holland, Amsterdam. Second printing 1968.

Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL '82)*. Association for Computing Machinery, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176

Shane Emmons and Anthony Dmitriyev. 2021. Money. https://github.com/RubyMoney/money.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Cormac Flanagan and Matthias Felleisen. 1997. Componential Set-Based Analysis. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation* (Las Vegas, Nevada, USA) *(PLDI '97)*. Association for Computing Machinery, New York, NY, USA, 235–248. https://doi.org/10.1145/258916.258937

Jeffrey Foster, Brianna Ren, Stephen Strickland, Alexander Yu, Milod Kazerounian, and Sankha Narayan Guria. 2018. RDL: Types, type checking, and contracts for Ruby. https://github.com/tupl-tufts/rdl.

Jeffrey S. Foster. 2021. Talks. https://github.com/jeffrey-s-foster/talks.

Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. 2009. Static Type Inference for Ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC '09)*. ACM, New York, NY, USA, 1859–1866. https://doi.org/10.1145/1529282.1529700

Brian Hackett and Shu-yu Guo. 2012. Fast and Precise Hybrid Type Inference for JavaScript. *SIGPLAN Not.* 47, 6 (June 2012), 239–250. https://doi.org/10.1145/2345156.2254094

Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/3236024.3236051

R. Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60. https://doi.org/10.1090/S0002-9947-1969-0253905-6

Howard Hinnant. 2013. chrono-Compatible Low-Level Date Algorithms. https://howardhinnant.github.io/date_algorithms.html.

Civilized Discourse Construction Kit Inc. 2021. Discourse: A platform for community discussion. https://github.com/discourse/discourse.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 5110–5121. http://proceedings.mlr.press/v119/kanade20a.html

Milod Kazerounian. 2021. Personal communication.

Milod Kazerounian, Jeffrey S. Foster, and Bonan Min. 2021. SimTyper Artifact. https://doi.org/10.5281/zenodo.5449078

Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S. Foster, and David Van Horn. 2019. Type-level Computations for Ruby Libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 966–979. https://doi.org/10.1145/3314221.3314630

Milod Kazerounian, Brianna M. Ren, and Jeffrey S. Foster. 2020. Sound, Heuristic Type Annotation Inference for Ruby. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages* (Virtual, USA) *(DLS 2020)*. Association for Computing Machinery, New York, NY, USA, 112–125. https://doi.org/10.1145/3426422.3426985

Diederik P Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR (Poster)*.

Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. 2015. Siamese neural networks for one-shot image recognition *(ICML deep learning workshop)*. Lille, JMLR.org, Online.

Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. 2013. TeJaS: Retrofitting Type Systems for JavaScript. In *Proceedings of the 9th Symposium on Dynamic Languages (DLS)*. ACM, New York, NY, USA, 1–16.

Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript Function Types from Natural Language Information. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Montreal, Quebec, Canada, 304–315. https://doi.org/10.1109/ICSE.2019.00045

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

MiniMagick. 2021. MiniMagick. https://github.com/minimagick/minimagick.

Dmitry Petrashko. 2020. Personal communication.

Postmodern. 2021. Ronin. https://github.com/ronin-rb/ronin.

François Pottier. 1998. A Framework for Type Inference with Subtyping. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 228–238. https://doi.org/10.1145/291251.289448

Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2019. TypeWriter: Neural Type Prediction with Search-based Validation. arXiv:1912.03768 [cs.SE]

Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 111–124. https://doi.org/10.1145/2775051.2677009

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics, Hong Kong, China, 3982–3992. https://doi.org/10.18653/v1/D19-1410

Brianna M. Ren and Jeffrey S. Foster. 2016. Just-in-time Static Type Checking for Dynamic Languages. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 462–476.

Phil Ross. 2021. TZInfo. https://github.com/tzinfo/tzinfo.

Loren Segal. 2020. YARD: Yay! A Ruby Documentation Tool. http://yardoc.org.

Jeremy Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Seventh Workshop on Scheme and Functional Programming*. ACM, Portland, OR, USA, 81–92.

T. Stephen Strickland, Brianna M. Ren, and Jeffrey S. Foster. 2014. Contracts for Domain-Specific Languages in Ruby. In *Proceedings of the 10th ACM Symposium on Dynamic Languages* (Portland, Oregon, USA) *(DLS '14)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/2661088.2661092

Stripe. 2020. Sorbet: A static type checker for Ruby. https://sorbet.org/.

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 395–406. https://doi.org/10.1145/1328438.1328486

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. (2017).

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages* (Portland, Oregon, USA) *(DLS '14)*. Association for Computing Machinery, New York, NY, USA, 45–56. https://doi.org/10.1145/2775052.2661101

Andrew K. Wright and Robert Cartwright. 1997. A Practical Soft Type System for Scheme. *ACM Trans. Program. Lang. Syst.* 19, 1 (Jan. 1997), 87–152. https://doi.org/10.1145/239912.239917