Sound, Heuristic Type Annotation Inference for Ruby

Milod Kazerounian
University of Maryland
College Park, Maryland, USA
milod@cs.umd.edu

Brianna M. Ren
University of Maryland
College Park, Maryland, USA
bren@cs.umd.edu

Jeffrey S. Foster
Tufts University
Medford, MA, USA
jfoster@cs.tufts.edu

Abstract

Many researchers have explored retrofitting static type systems to dynamic languages. This raises the question of how to add type annotations to code that was previously untyped. One obvious solution is type inference. However, in complex type systems, in particular those with structural types, type inference typically produces most general types that are large, hard to understand, and unnatural for programmers. To solve this problem, we introduce `InferDL`, a novel Ruby type inference system that infers sound and useful type annotations by incorporating heuristics that guess types. For example, we might heuristically guess that a parameter whose name ends in count is an integer. `InferDL` works by first running standard type inference and then applying heuristics to any positions for which standard type inference produces overly-general types. Heuristic guesses are added as constraints to the type inference problem to ensure they are consistent with the rest of the program and other heuristic guesses; inconsistent guesses are discarded. We formalized `InferDL` in a core type and constraint language. We implemented `InferDL` on top of `RDL`, an existing Ruby type checker. To evaluate `InferDL`, we applied it to four Ruby on Rails apps that had been previously type checked with `RDL`, and hence had type annotations. We found that, when using heuristics, `InferDL` inferred 22% more types that were as or more precise than the previous annotations, compared to standard type inference without heuristics. We also found one new type error. We further evaluated `InferDL` by applying it to six additional apps, finding five additional type errors. Thus, we believe `InferDL` represents a promising approach for inferring type annotations in dynamic languages.

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

Keywords: type inference, dynamic languages, Ruby

ACM Reference Format:
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 (DLS '20), November 17, 2020, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3426422.3426985

DLS '20, November 17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8175-8/20/11…$15.00
https://doi.org/10.1145/3426422.3426985

1 Introduction

Many researchers have explored ways to add static types to dynamic languages [3, 4, 15, 19, 29, 31, 36, 37], to help programmers find and prevent type errors. One key challenge in using such retrofitted type systems is finding type annotations for code that was previously untyped. *Type inference*, which aims to type check programs with few or no type annotations, is an obvious solution, and indeed there are several type inference systems for dynamic languages [2–4, 15]. Beyond type checking, type inference can also be extended to generate type annotations for program values. These annotations provide a useful form of documentation, and can be used in other forms of program analysis such as code completion for IDEs.

However, type inference systems typically aim to find the *most general type* for every position, i.e., the least restrictive possible type. If the type language is rich—particularly if it includes structural types—the most general possible annotations might be large, hard to read, and unnatural for programmers. For example, An et al. [2] describe a type inference system for Ruby that infers that a certain position accepts any object with >, <<, >>, &, and ^ methods. In contrast, a programmer would mostly likely, and much more concisely, say that position takes an Integer. Moreover, even if we are not interested in producing annotations, large, complex types can lead to difficult-to-understand error messages.

In this paper, we present `InferDL`, a novel Ruby type inference system that aims to infer sound and useful type annotations. More specifically, `InferDL` allows the programmer to specify heuristics for guessing type annotations. For example, one simple but effective heuristic is to guess the type Integer for any variables whose names end with id, num, or count. `InferDL` runs such heuristics for positions for which standard type inference is overly-general (meaning, among others, positions with inferred structural types). Heuristic guesses are added as additional type constraints and checked for consistency with the rest of the program. Only consistent

solutions are kept. In this way, InferDL maintains soundness while producing a less general, but potentially more useful, solution than standard type inference. (§ 2 gives an overview of InferDL.)

We describe InferDL more formally on a core type and constraint language. We present standard constraint resolution rules, which rewrite a set of constraints into *solved form* from which most general solutions can be extracted. We describe that solution extraction procedure in detail and then show how to incorporate heuristics. (See § 3 for our formal description.)

We implemented InferDL as an extension to RDL, an existing Ruby type checker [13, 19, 31]. We modified RDL to generate and resolve type constraints, run heuristics, and extract solutions to produce annotations. InferDL currently includes eight heuristics: one that replaces structural types with nominal types that match the structure; six that look at variable names, such as the one mentioned above for names ending in id, num, or count; and one that produces precise hash types. We also extended RDL with *choice types*, an idea inspired by variational typing [7] that helps type inference work in the presence of overloaded methods. (§ 4 describes the implementation of InferDL.)

We evaluated InferDL by applying it to four Ruby on Rails apps for which RDL type annotations already existed [19, 31]. We note that our results are preliminary, and further work is needed to affirm they generalize beyond our benchmarks. For these apps, InferDL inferred 496 type annotations. Of these, 399 exactly matched or were more precise than the programmer-supplied annotations, compared to only 290 such annotations when not using heuristics. InferDL also found one previously unknown type error. We also applied InferDL to six additional Ruby programs for which we did not have annotations, and InferDL found five previously unknown type errors. (§ 5 discusses our evaluation.)

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

2 Overview

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

2.1 Standard Type Inference

Figure 1a shows a code snippet taken from *Discourse*, a Ruby on Rails web app used in our evaluation (§ 5). The code defines two methods, normalize_username and find_by_name, in the class User. Because User is a subclass of ActiveRecord::Base, it is a Rails *model*, meaning instances of the class represent rows of a database table.

```
1  class User < ActiveRecord::Base
2    # α → β
3    def self.normalize_username(name)
4      name.unicode_normalize.downcase if name.present?
5    end
6    # γ → δ
7    def self.find_by_name(name)
8      find_by(name_lower: normalize_username(name))
9    end
10 end
```

(a) Source code from Discourse app.

Constraints Generated
(1) $\alpha \leq [\text{unicode\_normalize} : \bot \to \epsilon]$
(2) $\alpha \leq [\text{present?} : \bot \to \zeta]$
(3) $\epsilon \leq [\text{downcase} : \bot \to \eta]$
(4) $\eta \leq \beta$
(5) $\gamma \leq \alpha$
(6) $\text{User} \leq \delta$
Resolved Constraints
(7) $\gamma \leq [\text{unicode\_normalize} : \bot \to \epsilon]$
(8) $\gamma \leq [\text{present?} : \bot \to \zeta]$

(b) Constraints generated on type variables.

Figure 1. Inferring method types in *Discourse*.

Suppose we wish to infer types for these two methods using the standard, constraint-based approach. We first generate a *type variable* for the method argument and return types, as shown in the comments on lines 2 and 6, e.g., normalize_username takes a value of type α and returns type β. Then we analyze the method body, generating *constraints* of the form $x \leq y$, indicating that x must be a subtype of y. In this case we also say x is a *lower bound* on y and y is an *upper bound* on x.

The top portion of Figure 1b shows the constraints generated from this example. Constraint (1) arises from the call name.unicode_normalize[1]. In this constraint, the *structural type* $[\text{unicode\_normalize} : \bot \to \epsilon]$ represents an object with a unicode_normalize method that takes no argument (here written $\bot$) and returns ϵ, a fresh type variable generated at the call. Hence, by standard subtyping rules, α must be a type that contains at least this method with appropriate argument and return types. Constraint (2) is similar.

Constraint (3) arises from calling downcase on the result of unicode_normalize. Constraint (4) arises because the result of the call to downcase is returned. Note that normalize_username may also return nil (if the conditional guard is false), but nil is a subtype of all other types in InferDL, so we omit this constraint here. Finally, constraint (5) arises from the call to normalize_username on line 8, and constraint

[1] In Ruby, the parentheses in a method call are optional.

(6) arises because User's find_by method returns a User (as indicated by find_by's type annotation, omitted here).

After generating constraints, InferDL performs *constraint resolution*, which applies a series of rewriting rules to the constraints. For example, one resolution rule is transitive closure: If $a \leq b$ and $b \leq c$ then we add constraint $a \leq c$ (see § 3 for a complete list of constraint resolution rules). In our example, constraint resolution generates the two new constraints (7) and (8).

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

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

Fortunately, after constraint resolution, the constraints are in *solved form* [27], which means that to extract a variable's solution we need only look at its lower and upper bounds. More specifically, for a return, we compute the union of its lower bounds, ignoring variables (since any transitive constraints from them have been propagated by resolution), and for an argument, we compute the intersection of its upper bounds. Thus, in our example, the solution for α and γ is [unicode_normalize : $\bot \to \epsilon$, present? : $\bot \to \zeta$], i.e., an object that has those methods, and the solution for δ is User.

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

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

2.2 Type Inference with Heuristics

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

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

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

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

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

Moreover, the additional constraints on α and γ in turn yield better solutions elsewhere because:

- $\Rightarrow$ String $\leq \alpha$ is added as a constraint. Transitively propagating to α's upper bounds yields...
- $\Rightarrow$ String $\leq$ [unicode_normalize : $\bot \to \epsilon$]. To check this constraint, we look up String's unicode_normalize method type and generate the constraint...
- $\Rightarrow \bot \to$ String $\leq \bot \to \epsilon$. Propagating to the methods' return types yields...
- $\Rightarrow$ String $\leq \epsilon$. Transitively propagating through ϵ yields...
- $\Rightarrow$ String $\leq$ [downcase : $\bot \to \eta$]. Looking up String's type for downcase, we get the constraint...
- $\Rightarrow \bot \to$ String $\leq \bot \to \eta$, and propagating to return types yields...
- $\Rightarrow$ String $\leq \eta$. Finally, propagating to η's upper bound yields...
- $\Rightarrow$ String $\leq \beta$

Thus, now β has nominal type String as a solution. Putting this together with the (most general) solution for δ and the

(heuristic) solutions for α and γ, InferDL has now inferred fully nominal type annotations for our example:

normalize_username: String → String
find_by_name: String → User

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

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

We wish to infer the type of find_answer. However, notice that only id is called on each argument. In Rails, the method id is typically defined for all model classes, e.g., in *Journey*, 48 different classes include an id method. This means STRUCT-TO-NOMINAL will fail to infer a precise annotation for the arguments in this case.

Instead, we develop another heuristic that takes advantage of a common practice: Ruby programmers often name a variable after the class of the value it will hold, especially for Rails models. Indeed, the arguments response and question are intended to take instances of the model classes Response and Question, respectively. We define a new heuristic IS_MODEL that guesses types based on this convention:

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

To define this heuristic, we call RDL::Heuristic.add, passing the name of the heuristic, in this case :is_model, and a *code block*. Code blocks are Ruby's version of anonymous functions or lambdas. The RDL::Heuristic.add method expects a code block that takes a single argument, which is the type variable whose solution the heuristic should guess. Note that in InferDL, which is built on RDL, types are actual values we can compute with; we discuss this in greater detail in § 4. The code block returns either nil, if there is no guess, or the guessed type.

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

During type inference, InferDL runs all heuristics for each type variable, accepting the solution from the first heuristic that produces a consistent type. For find_answer, the IS_MODEL heuristic produces appropriate nominal types for the arguments, which then become the final type annotations for those positions.

$$
\begin{array}{lrcl}
\textit{Types} & \tau & ::= & \alpha \mid A \mid [m : \tau_m] \mid \\
 & & & \tau \cup \tau \mid \tau \cap \tau \mid \bot \mid \top \\
\textit{Method Types} & \tau_m & ::= & \tau \to \tau \\
\textit{Constraints} & C & ::= & \tau \leq \tau \mid C \cup C
\end{array}
$$

$A \in$ class IDs, $m \in$ meth IDs

Figure 2. Core types and constraints.

3 Constraints, Solutions, and Heuristics

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

3.1 Types and Constraints

Figure 2 formally defines a core subset of the types and constraints in InferDL. Types τ include type variables α and nominal types A, which is the set of class IDs. Structural types $[m : \tau_m]$ name a method m and its corresponding method type τ_m. For brevity, structural types can only comprise a single method, and method types may only take a single argument. Types also include union types $\tau \cup \tau$, intersection types $\tau \cap \tau$, the bottom type $\bot$, and the top type $\top$. Constraints C consist of subtyping constraints $\tau_1 \leq \tau_2$ and unions of constraints $C_1 \cup C_2$, which allow us to build up sets of constraints.

Generating Constraints. Constraint generation is a straightforward modification of standard type checking in which, instead of type rules checking constraints, we view them as *generating* constraints. For example, the rule for typing a method call is

$$
\frac{\Gamma \vdash e_1 : \tau_{rec} \qquad \Gamma \vdash e_2 : \tau_{arg} \qquad \tau_{rec} \leq [m : \tau_{arg} \to \tau_{ret}] \qquad \tau_{ret} \text{ is fresh}}{\Gamma \vdash e_1.m(e_2) : \tau_{ret}}
$$

This rule types a method call $e_1.m(e_2)$ (where each e is an expression) in type environment Γ (a map from local variables to types), yielding type τ_{ret}. To apply this rule, we recursively type the receiver and the argument, yielding types τ_{rec} and τ_{arg}, respectively. We then generate a constraint $\tau_{rec} \leq [m : \tau_{arg} \to \tau_{ret}]$, where τ_{ret} is a fresh type variable. Then, the return type of the method call has type τ_{ret}. By convention, we assume any constraint in the premise of a rule is automatically added to a global set of constraints C.

Full details of type checking rules for a core Ruby language can be found in Ren and Foster [31] or Kazerounian et al. [19], both of which formalize Ruby in a core language and provide type checking rules. As with the above example, those rules can be turned into inference rules by viewing them as generating constraints and adjusting them as needed to make all constraints explicit, e.g., in the rule above, we

(1) $C \cup \tau_1 \le \alpha \cup \alpha \le \tau_2 \Rightarrow C \cup \tau_1 \le \alpha \cup \alpha \le \tau_2 \cup \tau_1 \le \tau_2$
(2) $C \cup A \le A' \Rightarrow C$ if $A = A'$ or A is a subclass of A'
(3) $C \cup A \le A' \Rightarrow error$ if $A \neq A'$ and A is not a subclass of A'
(4) $C \cup A \le [m : \tau_1 \to \tau_2] \Rightarrow C \cup \tau_1 \le \tau_1' \cup \tau_2' \le \tau_2$ if A has method m with type $\tau_1' \to \tau_2'$
(5) $C \cup A \le [m : \tau_1 \to \tau_2] \Rightarrow error$ if A has no method m
(6) $C \cup (\tau_1 \cup \tau_2 \le \tau_3) \Rightarrow C \cup \tau_1 \le \tau_3 \cup \tau_2 \le \tau_3$
(7) $C \cup (\tau_1 \le \tau_2 \cap \tau_3) \Rightarrow C \cup \tau_1 \le \tau_3 \cup \tau_2 \le \tau_3$
(8) $C \cup \bot \le \tau \Rightarrow C$
(9) $C \cup \tau \le \top \Rightarrow C$

Figure 3. Standard constraint resolution rules.

specify the type of e_1 as τ_{rec} and write an explicit constraint on τ_{rec}, rather than implicitly constraining e_1's type to have a particular shape in the rule.

Resolving Constraints. Figure 3 gives standard constraint resolution rules. Each rule has the form $C \Rightarrow C'$, meaning a set of constraints matching C can be rewritten to C'. The rules are applied exhaustively until they either yield *error* or no additional constraints can be generated.

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

3.2 Solution Extraction

Recall from § 2.1 that after generating and resolving constraints, the next step in standard type inference is to extract solutions for type variables. More precisely, standard type inference produces solutions using the following procedure:

```
procedure STANDARD_SOLUTION(C, α)
    if (α represents arg) then
        sol = ⊤
        for each constraint α ≤ τ ∈ C do
            sol = sol ∩ τ
    else                                  ▷ α represents return
        sol = ⊥
        for each constraint τ ≤ α ∈ C do
            sol = sol ∪ τ
    return sol
```

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

`InferDL` uses the same procedure as a subroutine to its heuristic inference algorithm, described next.

Heuristics. Formally, we can model `InferDL`'s heuristics as a set of additional constraint resolution rules beyond those in Figure 3. For example, we can express STRUCT-TO-NOMINAL as the following constraint rewriting rule:

STRUCT-TO-NOMINAL(C, α) =
$C' \cup \alpha \le [m_1 : ...] \cup \ldots \cup \alpha \le [m_n : ...] \Rightarrow C' \cup (\alpha = (A_1 \cup \ldots \cup A_k))$
if $k \le 10$ and $A_1, ..., A_k$ are all classes with $m_1 ... m_n$.

This rule applies to a type variable α that has one or more structural type upper bounds. If there are at most 10 classes $A_1 \ldots A_k$ matching those structural types, then we replace the structural constraints with a solution $A_1 \cup \ldots \cup A_k$ for α. Recall from § 2 that a *solution constraint* of the form $\tau_1 = \tau_2$ is shorthand for the two constraints $\tau_1 \le \tau_2$ and $\tau_2 \le \tau_1$.

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

```
procedure HEURISTIC_SOLUTION(H, C, α)
    for h ∈ H do
        C', sol = h(C, α)
        C' = RESOLVE(C')
        if C' ≠ error and sol ≠ nil then
            return C', sol
    return C, nil
```

Here we abuse notation slightly and assume that heuristic rules return a pair C', sol, where C' is the new set of constraints after running the rule, and sol is the solution found for α. If h does not match C, then h returns the pair nil, nil. When h does return a set of constraints C' and a solution, we perform constraint resolution on C' to propagate the new solution and detect any inconsistencies in the set of constraints. This is done by calling the RESOLVE procedure, which simply invokes the constraint resolution rules of Figure 3. If the resulting resolved constraint set C' is valid, then we return C' and sol. If all heuristics run without finding a valid solution, we return the original constraints C and nil.

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

Extracting Solutions. The next step is to combine standard and heuristic solution extraction, doing the latter when a standard solution is overly-general. Thus, we need to define what *overly-general* means. Based on our prior experience, we believe that, in most cases, nominal types are far easier for programmers to understand and use than structural types because they are smaller and simpler. The developers of Sorbet [35], another Ruby type checker, feel the same way—they

have found that structural types can be less intuitive and more difficult to read when used in error messages [26].

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

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

procedure EXTRACT_SOLUTION(H, C, α)
  sol = STANDARD_SOLUTION(C, α)
  if *overly_general*(sol) **then**
    C', sol_h = HEURISTIC_SOLUTION(H, C, α)
    if $sol_h \neq nil$ **then**
      $sol = sol_h$
  else
    C' = *RESOLVE*($C \cup (\alpha = sol)$)
  return C', sol

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

Finally, as shown in § 2.2, one extracted solution may lead to the discovery of other solutions. Thus, we continue to extract solutions for type variables until no new constraints are generated. More precisely, the following procedure takes H, C, and a set of all type variables $\mathcal{V}$ as input, and extracts solutions until no new constraints are generated.

procedure EXTRACT_ALL_SOLUTIONS(H, C, $\mathcal{V}$)
  solutions_map = { }
  repeat
    for each $\alpha \in \mathcal{V}$ **do**
      C, sol = EXTRACT_SOLUTION(H, C, α)
      solutions_map[α] = sol
  until no new constraints are added to C
  return *solutions_map*

4 Implementation

InferDL is built on top of RDL [13], an existing Ruby type checker. In this section, we briefly describe some of the implementation challenges in InferDL.

Type Checking with RDL. RDL uses an expressive type language, including nominal, singleton, generic, union, variable, and structural types; tuple types for fixed-size arrays and finite hash types for fixed-size hashes[2] [32]; and type-level computations [19].

One key feature of RDL is that it performs type checking *at runtime*. That is, method bodies are statically type checked, but this checking takes place dynamically at a time specified by the user. For example, say a programmer wanted to type check the body of normalize_username from Figure 1a. To do so, they could write the following RDL type annotation above the method:

```
RDL.type "() → String", typecheck: :later
```

This annotation is actually a call to the method RDL.type, which stores the given type annotation in a global table. The argument typecheck: :later associates this type annotation with the symbol :later. After this method call, the programmer can choose when to call:

```
RDL.do_typecheck :later
```

to actually perform type checking of all methods associated with the symbol :later, including normalize_username.

This approach to type checking allows RDL to handle metaprogramming, which is ubiquitous in Ruby [31]. Moreover, this design streamlines the implementation of heuristics in InferDL, because RDL types are runtime values that can be computed with, as in the IS_MODEL heuristic in § 2.2.

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

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

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

RDL already included type variables to support parametric polymorphism. InferDL extends type variables to store constraints as a list of upper and lower bounds on each type variable. Then, to perform constraint generation, InferDL modifies RDL's type checker so that, whenever two types are checked for subtyping, and at least one of the types is a type variable, we store the subtyping constraint. After constraint generation, InferDL performs constraint resolution and solution extraction, as explained in § 3.

Heuristics. Though heuristics are not baked-in to InferDL and are thus configurable, we have written eight heuristics that we found useful in practice, listed below. Recall from § 3.2 that heuristics are applied in a specified order. We list heuristics in the order in which they are applied.

[2]Hashes are Ruby's implementation of heterogeneous dictionaries.

- IS_MODEL: See § 2.2 for a description. This rule is only used for Rails apps.
- IS_PLURALIZED_MODEL: If a variable name is the pluralized version of the name of model X, then guess solution Array<X> ∪ ActiveRecord_Relation<X>. Note that ActiveRecord_Relation is a data structure provided by Rails that extends common array operations with some database queries. This rule is only used for Rails apps.
- STRUCT-TO-NOMINAL: See § 2.2 and § 3.2.
- INT_NAMES: If a variable name ends with id, count, or num, guess solution Integer.
- INT_ARRAY_NAME: If a variable name ends with ids, counts, or nums, guess solution Array<Integer>.
- PREDICATE_METHOD: If a method name ends with ?, guess solution %bool (RDL's boolean type) for the method return type.
- STRING_NAME: If a variable name ends with name, guess solution String.
- HASH_ACCESS: Like all other values, hashes are objects in Ruby, not built-in constructs. As such, they are accessed using the method [], and written to using the method []=. This rule states that, if all of an argument type variable's upper bounds are structural types consisting of the methods [] and []=, and all of the keys given to these methods are symbols, then guess the solution that is the finite hash type consisting of the keys and the unions of corresponding assigned values for these methods. For example, if an argument type variable α had constraints $\alpha \leq$ [:[]= : :id → Integer], $\alpha \leq$ [:[]= : :id → String], and $\alpha \leq$ [:[]= : :name → String], then the solution for α would be the finite hash type { id: String ∪ Integer, name: String }. This type says that α is a hash mapping the symbol :id to a String or Integer, and :name to a String.

As discussed in § 3.2, InferDL treats type variables, union, intersection, structural, and bottom and top types as overly-general. In our implementation, we also treat Object and nil (which were omitted from the formalism for brevity but are almost the same as the top and bottom types, respectively) as overly-general. In addition to nominal types, RDL also includes several additional kinds of types that we treat as sufficiently precise: generic types (which are parameterized nominal types), finite hash and tuple types (which are more precise versions of Hash and Array types), and singleton types (such a type has only one value as an inhabitant).

Choice Types. Ruby methods often have intersection types, which pose a challenge for type inference. Consider the Array indexing method [], which has the following type in RDL:

(Integer) → t ∩ (Range<Integer>) → Array<t>

Here, t is the type parameter for the Array class. When given an Integer index, [] returns a single element, and when given a Range<Integer> corresponding to multiple indexes, [] returns the subarray of elements at those indexes. Now consider the following contrived code snippet:

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

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

To address this issue, we introduce *choice types*, a type system feature loosely inspired by variational type checking [7]. A choice type, written $\text{Choice}_i(\tau_1, \ldots, \tau_n)$, represents a choice among the types τ_j. Each choice type also has a label i. During inference, if one τ_j of a choice type would result in a type error, then arm j is eliminated from all choice types with the same index i.

In the example above, the call to arr[x] would result in the constraint
(1) $\alpha \leq \text{Choice}_1$(Integer, Range<Integer>)
because InferDL reasons that it has a choice between the two input types of Array's [] method. Additionally, the return type of arr[x] would be
(2) Choice_1(Integer, Array<Integer>)
representing both possible returns. Both choice types have the label 1, indicating that they are decided together. Then, when type checking the call arr[x] + 1, InferDL would recognize that the Array<Integer> arm of type (2) results in a type error, and it would eliminate that arm from both (2) and (1). Effectively, this would retroactively make the return type of arr[x] be the sole type Integer, and it would allow us to infer α's solution as the sole type Integer. If InferDL ever eliminates all arms of a choice type, it raises a type error.

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

InferDL's approach to library types avoids this issue. During constraint generation, if InferDL encounters a call to a method that both lacks a type annotation and is not itself the target of inference, InferDL finds the method definition to determine the method's arity. InferDL then creates a type signature for the method with fresh type variables for the

return and each argument. If no method definition is found, InferDL raises a type error.

This approach is similar to type inference for a method, except we do not generate constraints from the method body. Thus, type inference might be unsound, producing a solution that would be impossible if we knew the library method's implementation. However, in practice, we found this approach was essential for allowing us to apply inference to each new benchmark, and it also helped us discover a previously unknown bug in one of our benchmarks (§ 5.1).

Variable Types. In addition to method types, InferDL can infer type annotations for the three non-local kinds of Ruby variables: global, class, and instance variables. Recall that STANDARD_SOLUTION (§ 3.2), used as a subroutine in InferDL, aims to infer most general types for methods. However, observe that variables are both read from and written to, i.e., for a field @x, there is conceptually a getter of type $\bot \to \alpha$ and a setter of type $\alpha \to \bot$. Notice that the α appears both co- and contravariantly. Hence, unlike method arguments and returns, it is not the case that a least or greatest solution will always be most general.

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

5 Evaluation

We evaluated InferDL on four Ruby on Rails web apps:

- *Journey* [5] is a web app that provides a graphical interface to create surveys and collect responses from participants.
- *Discourse* [18] is an open-source discussion platform built on Rails.
- *Code.org* [8] is a Rails app that powers code.org, a programming education website.
- *Talks* [14] is a Rails app written by one of the authors for sharing talk announcements.

We chose these apps because they have all been used as type checking benchmarks in prior work [19, 31]. Thus, we could use the previously written type annotations for these apps as "gold standards" to compare against. We inferred type annotations for all methods and global, class, and instance variables for which type annotations existed in prior work. This includes both methods (and the variables they used) that were type checked in prior work and those that were not checked, but were annotated to assist in the type checking of other methods.

The only additional type annotations used were for the special Rails params hash, which contains values that come from a user's browser. The params hash always maps symbols to various types of objects. Without annotations, InferDL typically infers that params has type Hash<K, V>, where V was the union of all observed value types for the hash. This effectively treats all values from the hash as belonging to the same type, which causes false positive type errors during inference. Rather than add type casts for these cases, we instead used the type annotations for params from the prior type checking work [19, 31]. In the future, we plan to incorporate special handling of the Rails params hash to avoid this issue.

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

5.1 Results

Table 1 summarizes our type inference results. The first column gives the number of methods we inferred types for, totalling 250 methods across the four apps. The subsequent group of three columns counts the number of types we inferred. The first of these columns, Meth Typs, counts the number of method argument and return types inferred. We count each argument and return type separately so that we can more precisely evaluate InferDL's performance. The second of these columns, Var Typs, shows the number of global, class, and instance variable types inferred. Finally, the Total Typs column counts the total number of types inferred, i.e., Meth Typs + Var Typs.

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

The subsequent column reports InferDL's running time on a 2014 MacBook Pro with a 3GHz i7 processor and 16GB RAM. We give the time as the median and semi-interquartile range (SIQR) of 11 runs. For comparison, we provide InferDL's runtime when using the heuristics presented in § 4 (shown under "*heur*"), and when not using any heuristics (shown under "*std*"). In total, InferDL took 31.91s to run on all benchmarks when using heuristics, with an SIQR of just 0.95s, indicating little variance across runs. By comparison, when not using any heuristics, InferDL took 8.68s to run on all benchmarks. Upon closer examination, we found that approximately 75% of InferDL's runtime when using heuristics was spent on just one rule, STRUCT-TO-NOMINAL. The rule involves searching through the space of all existing classes, and for each one, searching through the names of all its methods. This can be quite expensive for larger programs. We found we could achieve speedups by caching search results, and by building a mapping from method names to the

Table 1. Type inference results.

Program	Num Meths	Meth Typs	Var Typs	Total Typs	Type Casts	Time (s) Median ± SIQR	Correct Meths	Correct Vars	Correct Total	Heuristic Uses
						heur / *std*	*heur*/ *std*	*heur*/*std*	*heur*/ *std*	STN/Name/Hash
Journey	23	33	26	59	1	1.68 ± 0.05 /1.01 ± 0.09	33 / 30	19 / 13	52 / 43	0 / 13 / 0
Discourse	43	77	0	77	0	7.70 ± 0.64 /0.59 ± 0.04	61 / 47	0 / 0	61 / 47	3 / 42 / 1
code.org	74	152	12	164	4	20.1 ± 0.22 /5.01 ± 0.10	111 / 60	10 / 10	121 / 70	0 / 80 / 5
Talks	110	149	47	196	8	2.43 ± 0.04 /2.08 ± 0.15	127 /102	38 / 28	165 /130	7 / 58 / 3
Total	250	411	85	496	13	31.91 ± 0.95/8.68 ± 0.39	332 /239	67 / 51	399 /290	10 / 193 / 9

classes that implement them in advanced of running the rule. Nevertheless, this remains an expensive operation.

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

The first of these columns gives the number of method argument and return types correctly inferred for heuristic and standard inference. For example, InferDL correctly inferred 332 out of 411 total argument and return types for all apps when using heuristics, compared to just 239 correct types when performing standard inference. The next column gives the number of variable types correctly inferred, and finally, the Correct Total column gives the number of total types inferred correctly. As shown, the use of heuristics enables InferDL to infer about 22% more correct type annotations, a significant improvement. We found this percentage was fairly consistent across the benchmarks, indicating that the heuristics we used were not specific to one app, but rather captured some more common, general properties. We also found this improvement was approximately the same for types of global, class, and instance variables, and types of method inputs/outputs, indicating our approach to variables (discussed in § 4) is effective.

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

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

Finally, the last column shows the number of times a heuristic successfully found a type for each app, that is, the heuristic's guess was actually used as a solution. For brevity, we present the counts for all of the six name-based heuristics (IS_MODEL, IS_PLURALIZED_MODEL, INT_NAMES, INT_ARRAY_NAME, PREDICATE_METHOD, and STRING_NAME) under a single column Name, while the STN column gives the count for STRUCT-TO-NOMINAL, and the Hash column for HASH_ACCESS. It is clear that the name-based rules were by far the most useful heuristics for inferring types, comprising a total of 193 of the successful heuristic applications. Of those 193 applications, 103 were of the PREDICATE_METHOD rule. Overall, this suggests that variable and method names are a strong indicator of intended types.

Error Caught. In the process of inferring types, we discovered a previously unknown bug in the *Journey* app. This was particularly surprising because the method it was found in was *already type checked* in prior work [19]. The bug existed in the following code, which creates and saves a new person:

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

The bug arises because there is no save method for the IllyanClient::Person class, so the call invitee.save always raises an error. Moreover, because this error exists within a begin...rescue clause, the bug will never be directly seen at runtime since control will always pass to the rescue clause. The bug could potentially have been detected via manual programmer inspection of the error log, though it never was. We confirmed this bug with the *Journey* developer. This

bug was not caught by type checking in prior work because the programmer who wrote type annotations in that work wrongly assumed that the IllyanClient::Person#save method did exist. Thanks to InferDL's handling of library types (§ 4), the same mistake was not made here.

5.2 Case Studies

To further evaluate InferDL, we applied it to five additional Ruby libraries and one additional Ruby app:

- *Active Merchant* [34], a payment abstraction library.
- *Diff-LCS* [40] a library for generating difference sets between Ruby sequences.
- *MiniMagick* [24], an image processing library.
- *Optcarrot* [11], a Nintendo Entertainment System (NES) emulator implemented in Ruby and intended as a benchmark for runtime performance evaluation.
- *Sidekiq* [25], a background job handler library.
- *TZInfo* [33], a time management library.

Because we do not have gold standard type annotations for these programs, we refer to these experiments as case studies. With the exception of Optcarrot, we picked these programs because they are all highly popular, well-maintained, and well-tested. We chose Optcarrot because of its intended use as a Ruby benchmark and because, as an emulator, it relies heavily on binary arithmetic, which distinguishes it from other Ruby programs we looked at.

We ran inference for all methods defined in these programs, excluding methods that use features not supported by RDL; the most common unsupported feature was mixin methods. We also excluded methods defined in the *Active Merchant* payment gateways, a set of 215 distinct payment gateways comprising over 60,000 lines of code. Running InferDL for this many lines of code would have required a significant manual effort to add type casts to circumvent false positive errors, so we decided to leave them out of our case study. We discuss the issue of type casts further below.

***Using* InferDL.** It would be tedious and time-consuming to call InferDL's infer method (§ 4) on every method in our subject programs. Instead, we used InferDL's infer_file and infer_path methods, which take a file or path, respectively, as an argument and then call infer on every method statically defined in that file or path. We called these methods for all code in a program's lib/ directory, which by convention holds the program's implementation (and excludes testing code, code for handling dependencies, etc.).

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

- True errors resulting from bugs in the program. We discuss these below.
- False positives due to InferDL's conservatism. We inserted appropriate type casts to suppress these type errors. We discuss type casts below.
- Errors resulting from features unsupported by InferDL. As mentioned earlier, we exclude such methods from future rounds of inference.

As an aside, we note that currently, it can sometimes be difficult to find the underlying cause of a type error reported by InferDL. If an invalid constraint is generated during resolution, InferDL reports the invalid constraint and the line number origins of the left- and right-hand sides of the constraint. But often these constraints were generated through a series of propagations resulting from many different places in the code, so their origins do not always reveal the underlying cause. In the future, we hope to incorporate ideas from prior work on diagnosing type inference errors [20, 21, 39].

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

We wrote 104 total type casts to run inference for these programs, or approximately one type cast for every 30 types we inferred. In addition to the need for type casts when accessing values from heterogeneous data structures (discussed in § 5.1), we encountered many cases where type casts were necessary for path-sensitive typing. For example, consider the code snippet below, simplified from the *TZInfo* library:

```
1 index = @transitions.length
2 index.downto(0) do |i|
3   start_transition = i > 0 ? @transitions[i - 1] : nil
4   end_transition = @transitions[i]
5   offset = start_transition ? start_transition.offset
        : end_transition.previous_offset
6   ...
7 end
```

This snippet refers to the instance variable @transitions, which has type Array<TimezoneTransition>. On line 2, we enter a loop for values of i=index down to i=0. On line 3, we use Ruby's *ternary operator* to conditionally assign the variable start_transition to either a TimezoneTransition or to nil. Then, on line 5, we use the ternary operator again, this time with the variable start_transition as our condition. In Ruby, the value nil is falsey. Thus, on line 5, we only evaluate the expression start_transition.offset if start_transition is non-nil. This call is safe, because TimezoneTransition has a method offset defined.

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

Table 2. Case Study Inference Results.

Program	Num Meths	Meth Typs	Var Typs	Total Typs	Type Casts	Time (s) Median ± SIQR	Heuristic Uses
						heur / *std*	STN/Name/Hash
Active Merchant	148	275	62	337	5	1.71 ± 0.06 /0.80 ± 0.07	29 / 103 / 0
Diff-LCS	80	187	40	227	23	2.65 ± 0.02 /2.38 ± 0.06	20 / 14 / 0
MiniMagick	79	166	13	179	7	0.57 ± 0.15 /0.26 ± 0.01	4 / 10 / 0
Optcarrot	430	763	367	1130	48	38.9 ± 2.64 /78.6 ± 17.4	204 / 22 / 1
Sidekiq	344	623	96	719	12	3.28 ± 0.56 /2.12 ± 0.13	37 / 63 / 3
TZInfo	251	511	57	568	9	3.15 ± 0.25 /5.06 ± 0.07	118 / 42 / 0
Total	1332	2525	635	3160	104	50.22 ± 3.67/89.2 ± 17.8	412 / 254 / 4

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

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

The next column of Table 2 reports the median time and SIQR taken across 11 runs of InferDL, when using vs. not using heuristics. Interestingly, on these apps InferDL actually took *less* total time when using heuristics compared to not using them. This was attributable to two apps in particular, *Optcarrot* and *TZInfo*, that took 2× and 1.6× as long, respectively, when not using heuristics. This can occur due to the way that InferDL performs type inference. As shown in § 3.2, InferDL will repeatedly perform constraint resolution and solution extraction until no new constraints are generated. In some cases, heuristics may lead to solutions for type variables earlier on in this process, thereby helping to reach a constraint set fixpoint sooner. For instance, *Optcarrot* performed just 6 rounds of solution extraction when using heuristics, compared with 15 rounds of solution extraction when not using heuristics; for *TZInfo*, the numbers were 3 and 9, respectively.

Finally, we report the number of successful applications of heuristics for inferring types. Notably, the STRUCT-TO-NOMINAL heuristic is far more useful for our case study programs than for the Rails apps in Table 1. It was used 412 times when running inference for 3,160 total types in our case studies, versus just 10 times for 496 total types for the Rails apps. This disparity is at least partly attributable to the order in which heuristics are run (§ 4). STRUCT-TO-NOMINAL is applied after the rules IS_MODEL and IS_PLURALIZED_MODEL. But the latter two rules are only used for Rails apps, meaning STRUCT-TO-NOMINAL is applied third for Rails apps, and first for non-Rails apps. We tried re-running InferDL on the Rails apps with STRUCT-TO-NOMINAL ordered first, and found it was applied 28 times for the Rails apps, which at least partly closes the gap with non-Rails apps.

We also more closely examined the uses of STRUCT-TO-NOMINAL across all 10 programs in § 5.1 and § 5.2, and we found that approximately 14% of the time the heuristic produced a union type, while the remainder of the time it produced just a single, nominal type. The usefulness of the produced union types varied. Sometimes, the unions were quite sensible. For example, in the *TZInfo* program, STRUCT-TO-NOMINAL produced the type TZInfo::Timestamp ∪ Time as the solution for a number of variables. Both the TZInfo::Timestamp and Time classes represent time values, and many of *TZInfo*'s methods are implemented to handle objects from both of these classes, so this is a sensible solution. In other cases, we found that STRUCT-TO-NOMINAL produced unions of unrelated classes that happened to have some same-named methods, thereby producing a solution that is less useful.

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

Finally, note that the HASH_ACCESS rule was used only four times across the programs in Table 2, and only nine total times across the programs in Table 1. This is partly attributable to the invariance of hashes (as discussed in § 5.1). Though the rule was applied few times in practice, we still believe it was useful in the cases it was used for converting structural type solutions to a more readable finite hash type. For example, for one type variable in the *code.org* app, InferDL used the HASH_ACCESS rule to infer the finite hash type solution { id: Integer, email: String, gender: α } (some key/-value pairs omitted for brevity), rather than a much larger and more difficult to read intersection of structural types.

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

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

Table 3. Testing Implementation Choices.

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

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

```
rescue Exception ⇒ ex
  ...
  raise e
end
```

  The notation rescue Exception => ex catches exceptions of type Exception and binds the specific Ruby exception object to ex. This rescue clause was meant to perform some error handling (elided with the ... above) and then raise the original error, but it erroneously referred to an undefined e rather than ex.
- In *Sidekiq*, `InferDL` caught a reference to an undefined constant UNKNOWN.
- In *Diff-LCS*, `InferDL` caught two different calls to an undefined method Diff::LCS.YieldingCallbacks.
- In *Diff-LCS*, `InferDL` caught a reference to an undefined constant Text::Format.

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

5.3 Testing Implementation Choices

Finally, in § 4 we discussed two novel design features of `InferDL`: the use of choice types for resolving calls to overloaded methods and `InferDL`'s handling of calls to library methods for which we do not have types. To evaluate these choices, we provide some relevant data in Table 3 collected from all 10 of the programs discussed in § 5.1 and § 5.2.

For each program, the first column gives the number of choice types used while running `InferDL` on the program. As discussed in § 4, a choice type is used when type checking a call to an overloaded method, when `InferDL` is not able to determine which type of the method to use. They can help avoid false positive type errors (and thus reduce the need for type casts), and to infer more precise types. In total, we used 307 choice types across all benchmarks. The vast majority of these uses were in just two programs, *Diff-LCS* and *Optcarrot*. This is likely because these programs rely heavily on array manipulation, and therefore make frequent use of the Array accessing method [], which requires choice types to resolve its overloaded method type (see § 4 for an example). Thus, we found that the need for choice types commonly arises in programs, but they are especially useful for programs that make frequent use of overloaded methods.

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

6 Related Work

Researchers have been studying type inference for many decades. Traditionally, the problem is posed as follows: given a program without type annotations, can we determine the most general type for each expression in the program, and rule out any type errors? The problem was first formulated and solved by Curry and Feys [9] for the simply typed lambda calculus. Perhaps most famously, Hindley [17], Milner [23], and Damas and Milner [10] developed an approach known today as Hindley-Milner-Damas type inference. Its central algorithm, Algorithm W, works by generating constraints on type variables, then resolving those constraints through a process known as unification.

Later efforts by Cartwright and Fagan [6], Aiken et al. [1], and Flanagan and Felleisen [12], among others, sought to extend static type inference to programs written in dynamic languages. The goal is to develop a system with the flexibility of dynamic languages, but some of the correctness guarantees of static languages. Pottier [27] more specifically focused on type inferencce in the face of subtyping, making use of type constraint graphs and constraint resolution rules to put the constraints in solved form. Many of the aforementioned ideas have been incorporated into type inference systems for popular dynamic languages like JavaScript [3] and Python [4]. To the best of our knowledge, unlike `InferDL`, these systems focus exclusively on uncovering type errors and do not generate type annotations.

Furr et al. [15] present DRuby, a static type inference system for Ruby, which features an expressive type language including intersection, union, optional, and structural types. While DRuby also focuses exclusively on finding type errors

in programs, many of the type system features it includes are part of RDL [13], on which InferDL is built. Beyond inference alone, numerous static type systems have been explored for Ruby and other dynamic languages. Ren and Foster [31] explore type checking Ruby programs that use metaprogramming. Their central idea, to defer static type checking until runtime, has been incorporated into RDL. Kazerounian et al. [19] make use of type-level computations to more precisely type check Ruby programs, another idea that is now included in RDL. More broadly, static type systems have been explored for many dynamic languages including Racket [36], JavaScript [29], and Python [37].

Beyond formal static analyses, a number of probabilistic approaches to type inference have been proposed in recent years. JSNice [30] uses probabilistic graph modeling methods, such as conditional random fields, to predict JavaScript program properties including type annotations. It is restricted to predicting a limited set of types seen in training data. In a similar vein, Xu et al. [38] infer types for Python programs by building probabilistic graph models that incorporate multiple sources of information such as variable names, attribute accesses, and dataflow information. Unlike JSNice, they train on individual programs, allowing them to predict program-specific types. DeepTyper [16] uses bidirectional RNNs trained on JavaScript source code to infer types from over 11,000 types in its training dataset, while NL2Type [22] trains an RNN exclusively on JavaScript programs' natural language information, such as comments and identifier names, to predict from a set of 1,000 types. TypeWriter [28] uses a neural model to predict Python types based on natural language and code context information, and uses a gradual type checker to rule out incorrect types. They are limited to predicting from a finite, configurable type vocabulary.

Similar to the above approaches, InferDL also incorporates identifier names when performing type inference. Of course, unlike the probabilistic approaches, InferDL relies on formal rules and thus may miss out on the complexity and expressiveness found in natural language. However, InferDL's rule-based approach also has its advantages. For one, this approach avoids the need for a large dataset of type annotations, which is not readily available in Ruby. Moreover, Ruby on Rails programs emphasize convention over configuration, which includes rigid variable/method naming conventions—it is extremely straightforward to write heuristics in InferDL that take direct advantage of such naming conventions (e.g., the IS_MODEL heuristic). Beyond being easy to express, InferDL's heuristic rules are highly configurable and not baked-in to the inference system, allowing programmers to remove rules that do not apply, or add new rules that capture their own conventions. Additionally, unlike most of the existing probabilistic approaches, InferDL is able to predict rare and user-defined types since it is not limited to the data in a training set. Finally, InferDL can also fall back on inferring types based on constraint solving, which the above approaches are unable to do.

7 Conclusion

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

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

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

Acknowledgments

We thank Sankha Narayan Guria and the anonymous reviewers for their helpful comments. This research was supported in part by NSF CCF-1918233 and DGE-1840340.

References

[1] 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 (POPL '94)*. ACM, New York, NY, USA, 163–173. https://doi.org/10.1145/174675.177847

[2] Jong-hoon An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic Inference of Static Types for Ruby. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Austin, TX, USA, 459–472. https://doi.org/10.1145/1926385.1926437

[3] 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

[4] John Aycock. 2000. Aggressive Type Inference.

[5] Nat Budin. 2020. Journey: An online questionnaire application. https://github.com/nbudin/journey/.

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

[7] Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2014. Extending Type Inference to Variational Programs. *ACM Trans. Program. Lang. Syst.* 36, 1 (2014). https://doi.org/10.1145/2518190

[8] Code.org. 2020. The code powering code.org and studio.code.org. https://github.com/code-dot-org/code-dot-org.

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

[10] 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 (POPL '82)*. Association for Computing Machinery, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176

[11] Yusuke Endoh. 2020. Optcarrot. https://github.com/mame/optcarrot.

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

[13] 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.

[14] Jeffrey S. Foster. 2020. Talks. https://github.com/jeffrey-s-foster/talks.

[15] 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

[16] 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 (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 152âĂŞ162. https://doi.org/10.1145/3236024.3236051

[17] 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

[18] Civilized Discourse Construction Kit Inc. 2020. Discourse: A platform for community discussion. https://github.com/discourse/discourse.

[19] 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 (PLDI 2019)*. ACM, New York, NY, USA, 966–979. https://doi.org/10.1145/3314221.3314630

[20] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for Type-Error Messages. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 425–434. https://doi.org/10.1145/1250734.1250783

[21] Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. 2016. A Practical Framework for Type Inference Error Explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '16)*. Association for Computing Machinery, New York, NY, USA, 781–799. https://doi.org/10.1145/2983990.2983994

[22] 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 âĂŹ19)*. IEEE Press, 304âĂŞ315. https://doi.org/10.1109/ICSE.2019.00045

[23] 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

[24] MiniMagick. 2020. MiniMagick. https://github.com/minimagick/minimagick.

[25] Mike Perham. 2020. Sidekiq. https://github.com/mperham/sidekiq.

[26] Dmitry Petrashko. 2020. Personal communication.

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

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

[29] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe and Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 167–180. https://doi.org/10.1145/2775051.2676971

[30] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from âĂIJBig CodeâĂİ. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL âĂŹ15)*. Association for Computing Machinery, New York, NY, USA, 111âĂŞ124. https://doi.org/10.1145/2775051.2677009

[31] 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. https://doi.org/10.1145/2908080.2908127

[32] Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. 2013. The Ruby Type Checker. In *Object-Oriented Program Languages and Systems (OOPS) Track at ACM Symposium on Applied Computing*. ACM, Coimbra, Portugal, 1565–1572. https://doi.org/10.1145/2480362.2480655

[33] Phil Ross. 2020. TZInfo. https://github.com/tzinfo/tzinfo.

[34] Shopify and Spreedly. 2020. Active Merchant. https://github.com/activemerchant/active_merchant.

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

[36] 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

[37] 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 (DLS '14)*. Association for Computing Machinery, New York, NY, USA, 45–56. https://doi.org/10.1145/2775052.2661101

[38] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python Probabilistic Type Inference with Natural Language Support. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 607âĂŞ618. https://doi.org/10.1145/2950290.2950343

[39] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2015. Diagnosing Type Errors with Class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 12–21. https://doi.org/10.1145/2813885.2738009

[40] Austin Ziegler. 2020. Diff-LCS. https://github.com/halostatue/diff-lcs/.