RestPi: Path-Sensitive Type Inference for REST APIs

MARK W. ALDRICH, Tufts University, USA
KYLA H. LEVIN, University of Massachusetts at Amherst, USA
MICHAEL COBLENZ, University of California at San Diego, USA
JEFFREY S. FOSTER, Tufts University, USA

REST APIs form the backbone of modern interconnected systems by providing a language-agnostic communication interface. REST API specifications should clearly describe all response types, but automatically generating specifications is difficult with existing tools.

We present REST$_\pi$, a type inference engine capable of automatically generating REST API specifications. The novel contribution of REST$_\pi$ is our use of *path-sensitive type inference*, which encodes symbolic path-constraints directly into a type system. This allows REST$_\pi$ to enumerate all response types by considering each distinct execution path through an endpoint implementation. We implement path-sensitive type inference for Ruby, a popular language used for REST API servers. We evaluate REST$_\pi$ by using it to infer types for 132 endpoints across 5 open-source REST API implementations without utilizing existing specifications or test suites. We find REST$_\pi$ performs type inference efficiently and produces types that are more precise and complete than those obtained via an HTTP proxy. Our results suggest that path-sensitivity is a key technique to enumerate distinct response types for REST endpoints.

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

Additional Key Words and Phrases: Path-Sensitive Analysis, RESTful APIs, Type Inference

ACM Reference Format:
Mark W. Aldrich, Kyla H. Levin, Michael Coblenz, and Jeffrey S. Foster. 2025. RestPi: Path-Sensitive Type Inference for REST APIs. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 277 (October 2025), 22 pages. https://doi.org/10.1145/3763055

1 Introduction

REST APIs serve as a ubiquitous mechanism to enable communication and interaction between networked components [19]. They commonly use JSON as a shared format and communicate over HTTP, making them extremely flexible. Documentation is key with REST APIs: their behavior can be highly dynamic, and the server and client may not be implemented in the same language. Unlike intra-language APIs there is no type system to aid in the usage of a REST API. It is up to the client to carefully invoke endpoints at the correct URL with the correct parameters. It is also up to the client to handle the response from the REST API correctly. This can be particularly difficult to do, as not all REST APIs follow the same convention for denoting failed or successful responses [12].

There are a few existing methods for automatically producing REST API specifications. One solution involves utilizing an HTTP proxy to inspect real API traffic [39, 40]. However, using an HTTP proxy to analyze production traffic comes with several issues. Using a proxy not only incurs a slowdown but presents a privacy risk as well. Additionally, analyzing production traffic is likely to expose only successful interactions with the API, creating an incomplete picture of

Authors' Contact Information: Mark W. Aldrich, Tufts University, Medford, USA, mark.aldrich@tufts.edu; Kyla H. Levin, University of Massachusetts at Amherst, Amherst, USA, khlevin@umass.edu; Michael Coblenz, University of California at San Diego, La Jolla, USA, mcoblenz@ucsd.edu; Jeffrey S. Foster, Tufts University, Medford, USA, jeffrey.foster@tufts.edu.

ACM 2475-1421/2025/10-ART277
https://doi.org/10.1145/3763055

the API's behavior. This technique is commonly used when executing a test suite for a REST API [14, 25, 36, 37], but then necessitates perfect code coverage to achieve a complete specification. REST API specifications can also be generated by extracting information from some existing form of documentation, such as natural-language documentation [9, 34] or manually annotated types [23, 35, 37], but this necessitates some form of documentation to begin with.

In this paper, we address this problem by introducing REST$_\pi$, a novel path-sensitive type inference system for REST APIs. REST$_\pi$ is able to produce complete and accurate specifications for REST endpoints that output JSON objects with no existing test-suite or specification. By utilizing path-sensitivity, REST$_\pi$ is able to explore each distinct path through an endpoint implementation to infer all response types and the symbolic path-constraints that cause them. We present the theory of this system, along with an implementation for Ruby built on the RDL type inference system [29]. Combined with several other features from RDL, such as flow-sensitivity, computed-types, and singleton types, REST$_\pi$ is able to produce precise and accurate specifications types for REST endpoints.

We evaluated REST$_\pi$ by using it to infer types for several REST API endpoint implementations from popular Ruby on Rails projects. We conducted experiments on the performance of REST$_\pi$ and show how the design of our type system effectively mitigates the exponential blow-up commonly observed with path-sensitive static program analyses. Finally, we compare the output of REST$_\pi$ to specifications obtained via an HTTP proxy on its test suite. We found the types produced by REST$_\pi$ is more complete, more precise, and enumerated more response cases than those observed in their test suites.

In summary, the contributions of this paper are:

(1) We present a path-sensitive type inference system capable of enumerating distinct return types of functions and the path conditions under which they occur.
(2) We describe REST$_\pi$, an open-source implementation [3] of this system for Ruby.
(3) We evaluate REST$_\pi$ on several REST API implementations and show how path-sensitive type inference is performant and effective at enumerating REST API response types.

Our results suggest path-sensitivity is a key feature for enumerating response types for REST APIs, and that path-sensitive type inference is an effective tool for REST API developers.

2 Overview

We motivate REST$_\pi$ by considering type inference for the Discourse REST API [24]. Discourse is open-source forum software that lets organizations self-host their own online forums. Discourse's website claims there are over 20,000 instances running, and it is the software that powers community forums for GitLab, OpenAI, Docker, Unreal Engine, and many more. Discourse uses `rswag` to simultaneously specify and test its API. The `rswag` tool automatically generates a specification for the Discourse API by invoking a test suite and observing the behavior of each API endpoint.

The Discourse REST API. Figure 1 shows how Discourse's API specification describes three endpoints from the Discourse API related to user creation and password changes. Each endpoint represents a distinct API method, and is uniquely identified by a URL and an HTTP *verb* (such as **GET**, **POST**, or **PUT**). The items in the "request body" section indicate arguments to the endpoint given using HTTP headers. The "response" section lists distinct response types. These responses consist of an HTTP status code, as well as a specification of the response body.

To use these three endpoints, in theory, all that is required is to invoke each endpoint with correctly typed arguments and handle the listed responses. However, this specification leaves out important details about each of these endpoints. Firstly, the specification states the **POST** `/users.json` endpoint will respond with HTTP status code 200, as well as a JSON object indicating if it was

Create a user
URL: **POST** `/users.json`
Request body:
- `name: string`
- `email: string`
- `password: string`
- `username: string`

Response: `Status Code 200` with JSON schema:
- `success: boolean`
- `user_id: integer`

Send password reset email
URL: **POST** `/session/forgot_password`
Request body:
- `login: string`

Response: `Status Code 200` with JSON schema:
- `success: string`
- `user_found: boolean`

Get a user by username
URL: **GET** `/users/{username}.json`
Request body: empty
Response: `Status Code 200` with JSON schema: `User`

Fig. 1. Excerpt of Discourse API Spec

successful. However, if the forum is set to invite-only, this endpoint will respond with HTTP status code 403 (forbidden) and an *empty* response body. It is easy to see how this will break client code that depends on a JSON object being returned.

Regarding the second endpoint, **POST** `/session/forgot_password`, the specification states the `user_found` field will always be present in the response, indicating if the user for which a password reset request is being issued exists at all. However, this is not the case. Whether or not this field appears in the output depends on how the site is configured; specifically, the `hide_email_address_taken` option in the site's configuration file. If this option is changed in production, the endpoint will no longer follow its specification.

Lastly, the specification for **GET** `/users/{username}.json` states that it will return a JSON object of the type `User`, with fields like `name: String`, `last_seen_at: String`, `post_count: Integer`. However, this endpoint only satisfies its specification when the requested profile is not private; in that case, many of the fields listed in the specification do not appear. Additionally, when requesting information on your own user, much *more* information is returned, such as your API keys and blocked users. In the case that you are not logged in, this endpoint may respond with an HTTP redirect to the login URL.

Although Discourse already uses an automated specification tool for its REST API, it is clear that the specification is incomplete. Ensuring a complete specification, even when using an automated

```
1  class SessionController < ApplicationController
2    def forgot_password
3      user = User.find_by_username_or_email(params[:login])
4      user_presence = user.present? && user.human? && !user.staged
5      if user_presence
6        # generate new token and send email to user
7      end
8
9      json = success_json
10     unless SiteSetting.hide_email_address_taken
11       json[:user_found] = user_presence
12     end
13
14     render json: json
15   end
16 end
```

Fig. 2. (Partial) Code for `/session/forgot_password`.

tool like `rswag`, still requires developers to manually identify the relationship between input, application state, and output, and then exhaustively test every configuration.

Applying REST$_\pi$ to Discourse. REST$_\pi$ works by performing path-sensitive type inference on the source code that implements an endpoint.

Consider the implementation of **POST** `/session/forgot_password`, a Ruby method shown in Figure 2. This endpoint first retrieves the `login` parameter on line 3 and uses it to find the requested User object. A check is performed on line 4 to confirm if the requested user exists and can have its password reset, stored in the `user_presence` variable. If this check succeeds, the endpoint creates a fresh token and sends an email to the user's email address. On line 9, the `json` variable is defined, which will eventually be rendered (sent back to the client) as output. It is initially set to the result of calling `success_json`, a helper method defined in this class that returns the Ruby hash `{success: "OK"}`. On lines 10-12, the `json` variable takes on another field, `:user_found`, on the condition that the `hide_email_address_taken` setting is configured to false. Finally, on line 14, the resulting `json` variable is serialized and rendered as an HTTP response to the API call.

The key feature of path-sensitive type inference is combining flow-sensitivity with path-sensitivity to determine when the types of local variables change depending on control-flow. For example, here we'd like to infer that the type of the `json` variable actually has two distinct types: (1) `{success: String}`, and (2) `{success: String, user_found: Boolean}`, and that the type observed depends on `SiteSetting.hide_email_address_taken`. To do this, REST$_\pi$ maintains a symbolic path-constraint ϕ for every expression it typechecks, along with a flow-sensitive type environment Γ. When typechecking the conditional statement on line 10, the type environment Γ is duplicated into Γ_1 and Γ_2, and typechecking proceeds on both branches of the condition. Line 11 is typechecked with its duplicated type-environment Γ_1 and a new path-constraint $\phi = \neg$`hide_email_address_taken`. When typechecking the variable assignment on line 11, a new type is produced for `json` containing its new field `user_found`, and is stored in Γ_1. The other side of the conditional contains no code, but produces a path-constraint $\phi =$ `hide_email_address_taken`.

When these type environments are merged together following the if-statement, a *path-type* is created, encoding that the type of the `json` differs depending on which branch of the conditional is taken: (`SiteSetting.hide_email` $\rightarrow$ `{success: String}`, $\neg$`SiteSetting.hide_email` $\rightarrow$ `{success: String, user_found: Boolean}`). This process of environment merging is illustrated in Figure 3. By observing how the types of variables can differ depending on control flow, REST$_\pi$

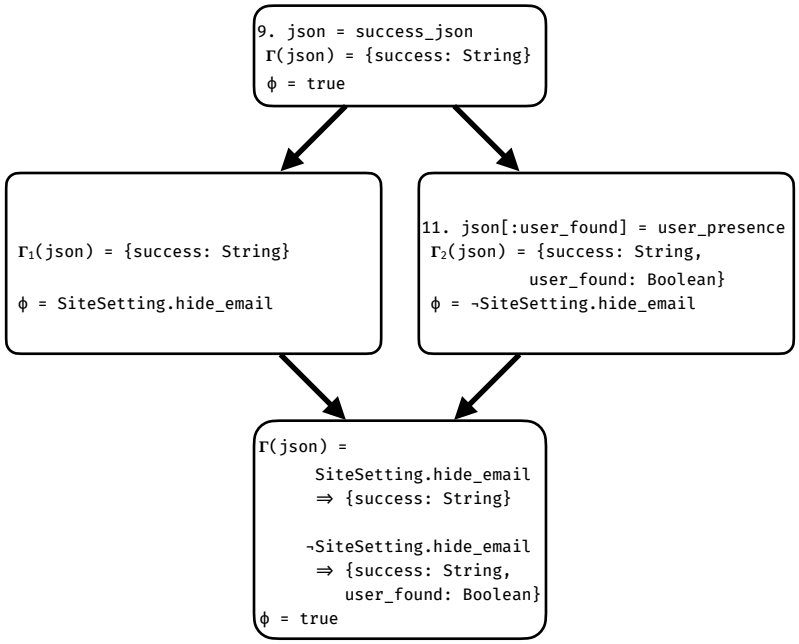


Fig. 3. Path-type creation. When inferring lines 10-12 of `forgot_password`, the type of `json` differs along the two branches of the if-statement. Merging the two type environments results in the creation of a path-type, which encodes its possible types along with the symbolic path-constraints that produced them.

Values	v	::=	$n \mid \text{true} \mid \text{false}$
Expressions	e	::=	$v \mid x \mid x = e \mid e; e$
		$\mid$	$\textbf{if}^{\ell}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3\ \textbf{end}$
Types	τ	::=	$bool \mid int \mid \bot \mid (\phi_1 : \tau_1,\ \phi_2 : \tau_2)$
Path Condition	ϕ	::=	$true \mid false \mid \ell \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi$
Type Environments	Γ	::=	$\cdot \mid \Gamma[x \mapsto \tau]$

Fig. 4. Source language.

is able to effectively enumerate all response types of an endpoint, even when they only render a single value as above.

In Section 3, we present our novel theory of path-sensitive type inference. In Section 4, we show how this can be combined with singleton types and *computed-types* [28] to further increase the precision of our inferred types.

3 Path-Sensitive Type Inference

We formalize REST$_\pi$ on the small imperative language shown in Figure 4. Values v in the language include integers n and booleans true and false. Expressions consist of values, local variables x, assignment $x = e$, and sequencing $e; e$. Expressions also include conditionals $\textbf{if}^{\ell}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3\ \textbf{end}$, which are labeled with their syntactic location ℓ. Our type system does not attempt to interpret conditional guards, so it tracks paths in terms of which syntactic conditional branches were taken.

$$\frac{}{\Gamma, \phi \vdash n : int \rightarrow \Gamma}\ \textsc{TInt} \qquad \frac{}{\Gamma, \phi \vdash \mathsf{true} : bool \rightarrow \Gamma}\ \textsc{TTrue}$$

$$\frac{x \in dom(\Gamma)}{\Gamma, \phi \vdash x : \Gamma(x) \rightarrow \Gamma}\ \textsc{TVar} \qquad \frac{}{\Gamma, \phi \vdash \mathsf{false} : bool \rightarrow \Gamma}\ \textsc{TFalse}$$

$$\frac{\Gamma, \phi \vdash e : \tau \rightarrow \Gamma'}{\Gamma, \phi \vdash x = e : \tau \rightarrow \Gamma'[x \mapsto \tau]}\ \textsc{TAsgn} \qquad \frac{\Gamma, \phi \vdash e_1 : \tau_1 \rightarrow \Gamma_1 \quad \Gamma_1, \phi \vdash e_2 : \tau_2 \rightarrow \Gamma_2}{\Gamma, \phi \vdash e_1; e_2 : \tau_2 \rightarrow \Gamma_2}\ \textsc{TSeq}$$

$$\frac{\Gamma, \phi \vdash e_1 : bool \rightarrow \Gamma_1 \qquad \Gamma_1, \phi \wedge \ell \vdash e_2 : \tau_2 \rightarrow \Gamma_2 \qquad \Gamma_1, \phi \wedge \neg\ell \vdash e_3 : \tau_3 \rightarrow \Gamma_3}{\Gamma, \phi \vdash \mathbf{if}^{\ell}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3\ \mathbf{end} : (\phi \wedge \ell : \tau_2,\ \phi \wedge \neg\ell : \tau_3) \rightarrow (\Gamma_2, \phi \wedge \ell) \uplus (\Gamma_3, \phi \wedge \neg\ell)}\ \textsc{TIf}$$

(a) Type Inference Rules.

$$\frac{\phi_1; \phi_2 \vdash \Gamma_1 \uplus \Gamma_2 : \Gamma'}{\phi_1; \phi_2 \vdash \Gamma_1[x \mapsto \tau_1] \uplus \Gamma_2[x \mapsto \tau_2] : \Gamma'[x \mapsto (\phi_1 : \tau_1,\ \phi_2 : \tau_2)]}$$

$$\frac{\phi_1; \phi_2 \vdash \Gamma_1 \uplus \Gamma_2 : \Gamma' \qquad x \notin dom(\Gamma_2)}{\phi_1; \phi_2 \vdash \Gamma_1[x \mapsto \tau_1] \uplus \Gamma_2 : \Gamma'[x \mapsto (\phi_1 : \tau_1,\ \phi_2 : \bot)]} \qquad \frac{}{\phi_1; \phi_2 \vdash \cdot \uplus \cdot : \cdot}$$

(b) Environment merging.

Fig. 5. Type Inference Rules and environment merging.

Types τ for this language include primitive types *bool* and *int* and the bottom type $\bot$ indicating unreachable code. Types also include *path types* $(\phi_1 : \tau_1,\ \phi_2 : \tau_2)$, indicating the type is τ_i under *path condition* ϕ_i. Here path conditions can either be *true*, *false*, a location ℓ, or a conjunction, disjunction, or negation of a path condition. Finally, type environments are maps from variables to types.

Figure 5a gives path-sensitive type inference rules for our language. These rules prove judgments of the form $\Gamma, \phi \vdash e : \tau \rightarrow \Gamma'$, meaning in type environment Γ and with path condition ϕ, expression e has type τ, and after evaluating e the new type environment is Γ'. Notice that our type system is flow-sensitive, i.e., the type of a variable can change after evaluating an expression.

Rules (TInt), (TTrue), and (TFalse) are straightforward: literal values are assigned their standard types and do not change the type environment. Rule (TVar) is similar, retrieving the type of the bound variable from the type environment and leaving the type environment unchanged.

Rule (TAsgn) types e to obtain a type τ, along with a possibly modified type environment. It then binds x to τ in the output type environment. Rule (TSeq) types its two subexpressions in sequence, threading the type environments through the derivation in the standard way.

Rule (TIf) describes several key features of symbolic execution. Control-flow splits with if-expressions, and the rule TIf types each branch under modified path-conditions. The code executed when the guard evaluates to true is typed under the path-condition $\phi \wedge \ell$. This corresponds to duplicating the symbolic state and updating the path-condition, however as this is a type system, the type environment is duplicated. The same process occurs when typing the code executed when the guard evaluates to false, under a new path-condition $\phi \wedge \neg\ell$. When control-flow is joined following the if-expression, the rule TIf joins both resulting type environments. If-expressions simplify to values as well, and as the resulting value is clearly dependent on a path condition, a

path-type is created for the result of the if-expression. In this case, the path-type created is made of two path-conditions which are guaranteed to be disjoint and satisfiable.

The type environments are merged following this process for each bound variable v: (1) if the type of the variable is defined in both type environments with different types τ_1 and τ_2, the resulting type environment will include the binding $v \mapsto (\phi_1 : \tau_1,\ \phi_2 : \tau_2)$, a path-type; (2) if the type of the variable is defined in only the first type environment, the resulting type environment will include the binding $v \mapsto (\phi_1 : \tau,\ \phi_2 : \bot)$; (3) similarly if the type of the variable is defined in only the second type environment, the resulting type environment will include the binding $v \mapsto (\phi_1 : \bot,\ \phi_2 : \tau)$.

To demonstrate how this type system works, consider the following example program:

```
1       a = 5;
2       if readInput then
3         a = true
4       else
5         b = true
6       end
```

As this example is a sequence of expressions, we will type one expression at a time, with the resulting type environment and path condition flowing from one expression to the next. We start by typing the first expression `a = 5` on line 1 with an empty type environment Γ and the path condition true. Applying rule TAsgn, our resulting state is $\Gamma = \{a : int\}$.

The next expression is an if-expression, and we proceed by typing e_1 on line 2. e_1 simply reads a boolean input from the environment, and does not change our state. We type e_2 on line 3 with the state $\Gamma = \{a : int\}$, $\phi = \text{true} \wedge readInput \equiv readInput$. e_2 is a variable assignment. Applying TAsgn produces the state $\Gamma = \{a : bool\}$. e_3 on line 5 is another variable assignment. We type e_3 under the state $\Gamma = \{a : int\}$, $\phi = \text{true} \wedge \neg readInput \equiv \neg readInput$. Applying TAsgn produces the state $\Gamma = \{a : int, b : bool\}$.

Finally, we must merge the two states using EnvJoin. This results in the creation of path-types for both `a` and `b`, with the final state of $\Gamma = \{a \mapsto (readInput : bool,\ \neg readInput : int), b \mapsto (readInput : \bot,\ \neg readInput : bool)$. This path-type fully describes all possible types of the variables `a` and `b`, and the path-conditions required to observe each type.

4 REST$_\pi$ in Practice

In this section, we describe how we bridge the gap between the theoretical presentation of the path-sensitive type system and our actual implementation. Our implementation, REST$_\pi$, is built on top of RDL [29], an existing type inference engine for Ruby. We extend RDL with path-sensitivity, fully integrating the path-sensitive typing rules described in Section 3. Combined with several of RDL's features, such as singleton types and computed types, REST$_\pi$ is able to extract precise types for JSON responses, encoding the HTTP status codes and JSON keys in the resulting type. To tackle the exponential blow-up commonly seen in path-sensitive analyses, we implement *path-type merging* to avoid considering unnecessary paths.

4.1 Manipulating Symbolic Values

Quality REST API documentation depends on values just as much as types. For example, it is not sufficient for a developer to know that the HTTP status code of the served response will be of type `Integer`. Similarly, it is not useful for a developer to know that the JSON object served as response is of type `Hash<String, Object>`. Yet, these are precisely the types we expect from type inference engines.

```ruby
def success_json
  { success: "OK" }
end

def failed_json
  { failed: "FAILED" }
end
```

(a) Definition of response types.

```ruby
def create_user(username)
  if User.create(username)
    render json: success_json
  else
    render json: failed_json,
      status: 422
  end
end
```

(b) Creating a user.

Fig. 6. Example endpoint implementation that creates a new User. The `create_user` method implements the endpoint and returns a JSON response.

For example, when typing the expression `status_code = 404`, traditional type inference engines will bind `status_code` with a type of `Integer`, to denote that this variable can be used anywhere an `Integer` is expected. To lift this value into the type system, REST$_\pi$ utilizes singleton types. REST$_\pi$ always assigns singleton types to literal values, where typing that same expression binds `status_code` with the singleton type `404`. The singleton type will be bound in the type environment, only promoting it to its supertype `Integer` when being compared against a concrete type, such as during a method call. This applies to all literal types, including hash objects and strings.

These singleton types can then be used to construct a specification for an endpoint. Consider the `success_json` and `failed_json` functions defined in Figure 6 (a). These functions come from our Discourse benchmark and are used pervasively throughout the application to construct API responses indicating success or failure.

REST$_\pi$ infers a type for `success_json` as `() -> {success: "OK"}`, with the return type being a singleton hash type with one key and a singleton string type for its value. Similarly for `failed_json`, REST$_\pi$ infers a type of `() -> {failed: "FAILED"}`.

Tying this together with path-sensitivity, consider the example code in Figure 6 (b).

When asked to infer a type for `create_user`, REST$_\pi$ utilizes path-sensitivity to determine that the output of this function is dependent on the condition `User.create(username)`, and utilizes singleton types to lift the exact hashes directly into the type. Our final inferred type is:

$$\big(\texttt{User.create(username)} : \texttt{HTTPResponse<200, \{success: "OK"\}>},$$
$$\neg\texttt{User.create(username)} : \texttt{HTTPResponse<422, \{failed: "FAILED"\}>}\big)$$

Singleton types allow values to be lifted into the type system, but do not let us manipulate them. To accomplish this, we utilize *computed-types* [28]. Computed-types allow parts of a function type to be computed at type inference time. When a call to a function with a computed-type is typechecked, its actual return type is computed based on the exact argument types it received. Consider the following example:

```ruby
1  def find_users(query)
2    users = Users.find(query)
3    if users == nil then render json: failed_json, status: 404
4    else render json: success_json.merge({users: users}) end
5  end
```

In REST$_\pi$, the function type for `Hash#merge` is defined as a computed-type. If both the receiver type and argument type are singleton types, this computed-type is able to directly merge the two hashes together to return the singleton type `{success: "OK", users: List<User>}`. Additionally, a computed-type is defined for `ActiveRecord::Base#as_json`, which is implicitly called on database model objects when serialized, as is the case on Line 4 of the `find_users` function. This computed-type searches the database schema used by a Ruby on Rails application to determine the types of its columns. Combined with path-sensitivity, REST$_\pi$ infers the return type of the function `find_users` as:

```
(users == nil : HTTPResponse<404, {failed: "FAILED"}>,
¬users == nil : HTTPResponse<200, {success: "OK",
                                   users: List<{id: Integer,
                                                name: String,
                                                username: String
                                                ...}> >)
```

By combining singleton types with computed types, REST$_\pi$ can essentially symbolically execute a function, retaining as much information as possible from the literal values in the source code. When important details of a REST API response such as the status code or keys of the JSON schema appear as literals in the source code, REST$_\pi$ retains the actual values and presents them in the final type signature. Non-singleton types appearing in the final type signature, such as a username being of type String, are obtained through constraint-based path-sensitive type inference.

4.2 Inferring Response Types for Endpoints

In this section, we describe how REST$_\pi$ operates in the context of an API request in a real application. In many REST API implementations, the functionality of an endpoint is split up into several distinct layers, where each layer is responsible for handling some part of the endpoint's behavior. For example, a typical endpoint may be composed of layers that authenticate the user, ensure valid arguments to the endpoint, implement the core logic, and finally a layer to serialize the requested resource. In order for REST$_\pi$ to gather precise information about the flow through the entire API request's lifecycle, each layer needs to be inferred in order.

Consider an example endpoint in Figure 7 for retrieving a post by its `id`. Although the core logic of the endpoint is implemented in the `show` function in the `PostsController` class, every line of code in this figure is reachable. The lifecycle of this request starts with the `before_action authenticate_user`, which ensures the user is authenticated before proceeding with the API request. If the user is not authenticated, an empty response with HTTP status code 401 is rendered to the client. Note that in Ruby on Rails, only one layer may render a response; rendering in an earlier layer halts execution of further layers. If the user is authenticated, execution then moves to the `show` function, which looks up the requested post in the database. This call to `find` may throw an exception of type `ActiveRecord::RecordNotFound`, which would be handled in the `rescue_from` lambda that renders an empty response with HTTP status code 404 to the client. When the post is found, it is then serialized in the `post_json` function which, in the context of this API endpoint, always includes the number of views in the response.

To paint an accurate picture of this endpoint's behavior, REST$_\pi$ runs type inference on each layer in order, passing the context from one layer to the next. REST$_\pi$ starts by creating a type variable α to represent the response type of this endpoint. Whenever a response is rendered,

```ruby
class PostsController
  rescue_from ActiveRecord::RecordNotFound do
    render status: 404
  end

  before_action :authenticate_user

  def show # GET /posts/{id}
    post = Post.find(params[:id])
    render json: post_json(post, true)
  end
end
```

(a) Rendering a post.

```ruby
def authenticate_user
  unless logged_in
    render status: 401
  end
end

def post_json(post, include_views)
  json = {title: post.title,
          body: post.body}
  if include_views
    json[:num_views] = post.view_count
  end
  return json
end
```

(b) Defining additional layers.

Fig. 7. Example endpoint implementation with multiple layers.

a path-sensitive type constraint will be produced of the form $\tau \leq_\phi \alpha$, indicating that τ must be a subtype of α under path-condition ϕ. Inference starts with the `authenticate_user` function, and is done with an initial path-condition of $\phi = \mathit{true}$. The call to `render` produces the path-sensitive type constraint `HTTPResponse<401>` $\leq_{\neg \texttt{logged\_in}} \alpha$. Next, path-sensitive type inference is run on the `show` function. This type inference starts with a path-condition of $\phi =$ `logged_in` as this function is only executed if the earlier layer did not render a response. The call to `Post.find` may throw an exception, so REST_π runs path-sensitive type inference on the `rescue_from` lambda with the path-condition $\phi =$ `logged_in` $\wedge \neg$`Post.find`, which produces a constraint `HTTPResponse<404>` $\leq_{(\texttt{logged\_in} \wedge \neg \texttt{Post.find})} \alpha$. Inference now continues with the path-condition $\phi =$ `logged_in` $\wedge$ `Post.find`. The call to `post_json` invokes another layer, so REST_π runs path-sensitive type inference on `post_json` with the argument types found at its callsite: `Post` and *true*. Since the `include_views` argument was bound to the singleton type *true*, REST_π sees that the if-statement is always executed and does not produce an additional path constraint for the case where `include_views` is *false*. Finally, the call `render json: post_json(post, true)` produces a constraint `HTTPResponse<200, {title: String, body: String, num_views: Integer}>` $\leq_{(\texttt{logged\_in} \wedge \texttt{Post.find})} \alpha$. The three constraints on α now describe the complete behavior of this endpoint under all possible paths, and a path-type is produced for its output:

$$
\begin{aligned}
(&\neg\texttt{logged\_in} : \texttt{HTTPResponse<401>},\\
&\texttt{logged\_in} \wedge \neg\texttt{Post.find} : \texttt{HTTPResponse<404>},\\
&\texttt{logged\_in} \wedge \texttt{Post.find} : \texttt{HTTPResponse<200, \{title: String,}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{body: String,}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{num\_views: Integer\}})
\end{aligned}
$$

4.3 Path Explosion

Path-sensitive program analysis suffers from the *path-explosion problem* [17, 18], wherein analyzing a program's behavior for both branches of a conditional must double the number of paths explored.

This quickly leads to an exponential blow-up, as the number of distinct types a variable can express correlates with the number of feasible execution paths. In REST$_\pi$, the impact of this exponential blow-up is mitigated in two ways: (1) path-type merging, and (2) limiting path-sensitivity to only the API layers.

Path-type merging. In REST$_\pi$'s type system, a variable with a type that depends on a path-condition is represented by a *path-type*: $(\phi_1 : t_1,\ \phi_2 : t_2)$. For example, on line 3 of the program below, when the if-statement joins, the type environments are joined and bind `hash` to the path-type:

$$(x : \{\mathsf{key1} : \alpha\},\ \neg x : \{\})$$

Then, on line 4, when the if-statement joins, the type environments are joined again and bind `hash` to the path-type

$$(x \wedge y : \{\mathsf{key1} : \alpha, \mathsf{key2} : \beta\}, x \wedge \neg y : \{\mathsf{key1} : \alpha\}, \neg x \wedge y : \{\mathsf{key2} : \beta\}, \neg x \wedge \neg y : \{\})$$

which accurately represents the four possible types this variable can take on. Now looking at line 6, it must be typechecked four separate times, once for each feasible execution path through the function.

```
1  def request
2    hash = {}
3    if x then hash[:key1] = a end
4    if y then hash[:key2] = b end
5
6    hash[:c] = c
7    return hash
8  end
```

Here we introduce one additional optimization that greatly reduces this exponential blow-up: *path-type merging*. When two type environments with paths ϕ and ϕ' are joined, and a variable has a hash type in both environments, each key of the hash type will be incrementally merged through the following process:

$$\begin{gathered}
\phi, \{k_1 : t_1, \ldots, k_n : t_n,\ l_{n+1} : t_{n+1}, \ldots, l_{n+m} : t_{n+m}\} \\
\uplus \\
\phi', \{k_1 : t'_1, \ldots, k_n : t'_n,\ m_{n+1} : t'_{n+1}, \ldots, m_{n+p} : t'_{n+p}\} \\
= \\
\{k_1 : (\phi : t_1,\ \phi' : t'_1), \ldots, k_n : (\phi : t_n,\ \phi' : t'_n), \\
l_{n+1} : (\phi : t_{n+1},\ \phi' : \bot), \ldots, l_{n+m} : (\phi : t_{n+m},\ \phi' : \bot), \\
m_{n+1} : (\phi : \bot,\ \phi' : t'_{n+1}), \ldots, m_{n+p} : (\phi : \bot,\ \phi' : t'_{n+p})\}
\end{gathered}$$

This process is similar to the environment merging rules, but applied to each key of a hash: if a key-value pair (k, t) was only found in the type environment with path ϕ, a path-type is created for its type $(\phi : t,\ \phi' : \bot)$; when only found in the type environment with path ϕ', its type becomes $(\phi : \bot,\ \phi' : t)$. When it is found in both type environments, (k, t) with path ϕ and (k, t') with path ϕ', its type becomes $(\phi : t,\ \phi' : t')$. The result of this process is a single hash type where each key is merged. For example, on line 3 when the if-statement joins, merging its path-type would produce:

$$\{\mathsf{key1} : (x : \alpha,\ \neg x : \bot)\}$$

On line 4 when the if-statement joins, merging its path-type would produce:

$$\{\mathsf{key1} : (x : \alpha,\ \neg x : \bot), \mathsf{key2} : (y : \beta,\ \neg y : \bot)\}$$

Finally, on line 6, the operation `hash[:c] = c` only needs to be typechecked a single time, as this operation does not depend on the values of the `key1` or `key2` keys whatsoever. This optimization

```ruby
class UsersController
  def show # GET /user
    opts = {}
    if params[:id]
      opts[:id] = params[:id]
    else
      opts[:full_name] = params[:name]
    end
# opts :
#   (params[:id] : {id: Integer},
#   !params[:id] : {full_name: String})
    user = User.find_by(opts)
    render json: user
  end
end
```

(a) Call with a path-sensitive argument.

```ruby
class PostsController
  def view_count # GET /post/{id}/view_count
    if params[:type] == "video"
      model = VideoPost.find(params[:id])
    else
      model = TextPost.find(params[:id])
    end
# model :
#   (params[:type] == "video" : VideoPost,
#   !params[:type] == "video" : TextPost)
    view_count = model.view_count
    render json: view_count
  end
end
```

(b) Call with a path-sensitive receiver.

Fig. 8. Calls to library functions with path-sensitive arguments and receivers.

is strikingly similar to *value-merging* [38] in symbolic execution but applied in a type-theoretic context. We measure the performance gains of this optimization in our evaluation, and show how the tractability of path-sensitive type inference is enabled by this optimization.

Limiting path-sensitivity to API layers. REST$_\pi$ takes another approach to limiting the effects of the exponential blow-up associated with path-sensitive program analysis by limiting the amount of code typechecked in a path-sensitive manner. API code typically relies on a large number of library functions such as ORM functions to retrieve and manipulate database objects. Additionally, API code may rely on many helper functions defined in the application, such as helpers that determine if resources are available, filter data according to permissions, or determine if a request should be blocked due to rate limiting. While determining accurate return types for these functions is essential for the precision of the analysis, the paths inside these functions are often uninteresting as they do not affect the response type directly. To avoid considering any extraneous paths, REST$_\pi$ typechecks these library function calls in a path-insensitive manner.

This is accomplished by treating any path-type arguments or path-type receivers as union types and typing the call under path-insensitive rules. Consider the code examples in Figure 8. In Figure 8 (a), the ORM function `User.find_by` is invoked with a path-type argument, where the type signature for `User.find_by` is `(Hash) -> User`. Here, REST$_\pi$ checks that the method call is valid for each type the argument can express (both `{id: Integer}` $\leq$ `Hash` and `{full_name: String}` $\leq$ `Hash` must hold). In both cases, the resulting type is `User` which will be bound to the `user` variable.

In Figure 8 (b), the method `view_count` is invoked on a path-type receiver. Here, REST$_\pi$ typechecks the method call once for each type the receiver may express (once for `VideoPost.view_count` and once for `TextPost.view_count`). In both cases, the resulting type is `Integer` which will be bound to the `view_count` variable.

5 Evaluation

This evaluation aims to answer the following research questions:

(RQ1) Is path-sensitive type inference effective at enumerating response types dependent on control-flow?

(RQ2) What is the performance of REST$_\pi$?

Table 1. Information on evaluation applications.

Project	# Endpoints Inferred	Total LoC Inferred
Canvas [27]	11	1,006
Code.org [13]	25	589
Discourse [24]	66	2,095
Journey [8]	8	112
OpenFarm [11]	22	112
Total	132	3,914

(RQ3) How do the response types inferred by REST$_\pi$ compare to specifications obtained via dynamic proxying?

5.1 Experimental Setup

Five benchmarks were selected for evaluation, detailed in Table 1. Three of the benchmarks (Code.org, Discourse, and Journey) were selected as they have previously been typechecked by RDL in prior work [28]. Two additional benchmarks (Canvas and OpenFarm) were selected by searching GitHub for repositories written in Ruby, sorting by stars, and selecting the first two repositories that: implement a REST API, had at least 10 endpoints that called `render` multiple times in one function body, utilize the Ruby on Rails ORM and serialization frameworks, and contained a functioning test suite for its REST API. Individual endpoints were selected if they called `render` multiple times in one function body. This choice was made to focus on the most dynamic and complex endpoints in each application; endpoints that lack control-flow in their final layer often implement simple CRUD functionality that only fail if the user is not authenticated or the requested resource cannot be found. Additionally, as the REST API in Canvas is expansive, endpoints in that benchmark were further limited to those related to course assignments.

We gathered OpenAPI specifications based on HTTP proxying for each benchmark. For Discourse, we used the official documentation generated with `rswag`. For Canvas and OpenFarm, the OpenAPI specification was produced automatically by `rspec-openapi` [1], which supports generating OpenAPI specifications for REST API test suites written in the `rspec` testing framework. Code.org and Journey do not use the `rspec` testing framework; for these, we manually instrumented their test suites to record all HTTP requests and responses issued. To generate a specification for these benchmarks, the instrumentation dumped all HTTP requests and responses in `HAR` format, and the resulting specification was produced automatically by `har-to-openapi` [2].

All experiments were performed on a consumer desktop with an Intel i7-3770 and 16GB of RAM.

5.2 RQ1: Is path-sensitive type inference effective at enumerating response types dependent on control-flow?

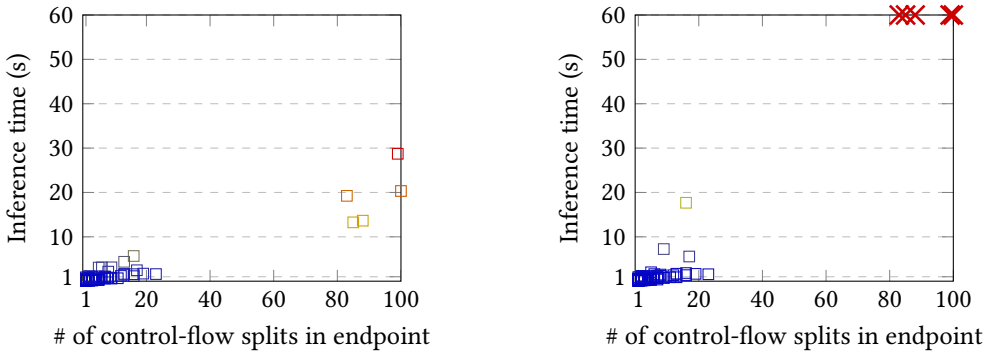
All 132 endpoints selected for evaluation render responses that are dependent on control flow. The goal of this research question is to ensure path-sensitive type inference is effective at exploring all control-flow paths through each endpoint and enumerating these distinct response types for each endpoint. In Table 2, we present the results of our inference, broken down by application and Ruby class in which the endpoint is implemented. In total, REST$_\pi$ inferred 545 unique response types for the 132 total endpoints, with 166 unique HTTP 200 responses identified and 379 unique responses with other status codes identified. These results suggest path-sensitive type inference is effective at identifying the relationship between control-flow and a rendered response.

Table 2. Information on the number of response types inferred by REST$_\pi$ per file. The first line reads: REST$_\pi$ inferred types for 2 endpoints from `AccountCalendarsApiController` in Canvas, inferred a total of 2 unique HTTP 200 responses, and inferred a total of 12 unique responses with other HTTP status codes.

Endpoint Class	# of Endpoints	HTTP 200	HTTP Other Code
Canvas			
AccountCalendarsApiController	2	2	12
AssignmentGroupsApiController	3	3	13
AssignmentsApiController	6	15	33
Code.org			
ApiController	5	7	3
CalloutsController	2	0	4
LevelAssetsController	1	1	1
MakerController	5	5	7
SurveyResultsController	1	1	1
TransfersController	1	0	10
SectionsStudentsController	3	3	9
CensusController	1	0	10
FormsController	1	1	2
WorkshopsController	4	5	7
PublicGalleryController	1	1	1
Discourse			
CategoriesController	3	3	8
DraftController	1	2	0
GroupsController	4	5	10
InvitesController	5	7	9
NotificationsController	1	2	1
PostActionUsersController	1	1	0
PostActionsController	2	2	7
PostsController	4	4	38
ReviewablesController	2	1	8
SessionController	4	12	5
StepsController	1	1	1
TagsController	3	3	6
TopicsController	14	19	56
UploadsController	1	1	3
UserActionsController	1	2	1
UserApiKeysController	1	0	1
UserBadgesController	2	2	3
UsersController	15	33	33
UsersEmailController	1	1	6
Journey			
PagesController	2	3	1
QuestionOptionsController	2	3	1
QuestionnairesController	2	3	1
QuestionsController	2	1	4
OpenFarm			
CropsController	2	2	5
GardenCropsController	5	3	15
GardensController	4	2	11
GuidesController	2	0	7
StageActionsController	2	1	5
StagesController	3	1	9
TokensController	2	0	6
UsersController	2	2	5
Total	132	166	379

5.3 RQ2: What is the performance of REST$_\pi$?

Figure 9 details the performance of REST$_\pi$ when inferring response types for the endpoints selected for RQ1. Times were gathered over 10 runs, and the times presented in Figure 9 are the medians of these runs for each endpoint. We are particularly interested in the performance of REST$_\pi$ on endpoints with complex control-flow: exponential slow-downs as control-flow complexity increases



(a) Path-type merging enabled

(b) Path-type merging disabled

Fig. 9. Time for REST$_\pi$ to infer each endpoint vs. # of control-flow splits in each endpoint. Red X's indicate timeouts after 1 minute.

would indicate REST$_\pi$ being affected by the path-explosion problem. To measure this, for each endpoint, we gathered the time spent inferring response types, as well as the number of points where control-flow split across all layers of the endpoint. In Ruby, this happens for each conditional statement as well as when exceptions are thrown. All endpoints selected for evaluation contain at least 1 control-flow split, with the maximum being 100 in the Canvas benchmark. Additionally, this same experiment was performed with path-type merging disabled.

The results show a modest increase in runtime as control-flow complexity increases, with an average runtime of 1.353 seconds per endpoint with path-type merging enabled. With path-type merging disabled, a much larger increase in runtime is observed for endpoints with complex control-flow. Without path-type merging, five endpoints were unable to be inferred within a timeout of 1 minute.

We find this runtime to be acceptable, and we can imagine utilizing path-sensitive type inference in a continuous integration environment, either for generating specifications or for checking an implementation against a handwritten specification.

5.4 RQ3: How do the response types inferred by REST$_\pi$ compare to specifications obtained via dynamic proxying?

Out of the 132 endpoints inferred for RQ1, 76 of them were invoked during the test suites in our benchmarks. A detailed comparison between the response types inferred by REST$_\pi$ and those found in the specifications obtained by dynamic proxying is shown in Table 3. These specifications contain three key pieces of information for each endpoint: (1) which HTTP status codes the endpoint can respond with; (2) for each status code, which fields will appear in its JSON reponse; and (3) for each of those fields, if they are *required* to appear in the JSON response for that status code. When producing specifications via dynamic proxying, a field is listed as required when it is observed in all responses with that status code; otherwise, that field will be listed as optional. For each endpoint, we compare the number of HTTP status codes and JSON fields in its specification with those found by REST$_\pi$. We report cases where REST$_\pi$ found more JSON fields than were present in its dynamic specification, indicating dynamic testing of the endpoint was insufficient to expose the control-flow necessary to include these fields. We also report cases where the specification lists a field as required to appear, but REST$_\pi$ found an execution path where that field does not appear in

Table 3. Comparison between specifications obtained via dynamic proxying and the response types inferred by REST_π, detailing number of HTTP 200 responses (S), number of HTTP responses with other status codes (O), the number of additional fields found by REST_π, and the number of fields where the specification was inaccurate about its presence.

Endpoint Class	# of Endpoints	Spec S	Spec O	REST_π S	REST_π O	Additional Fields Found	Req. Field Issues
Canvas							
AssignmentGroupsApiController	2	2	2	2	6	9	0
AssignmentsApiController	6	5	14	5	20	416	224
Code.org							
ApiController	3	3	0	3	1	6	3
MakerController	5	5	7	5	7	0	0
SurveyResultsController	1	1	0	1	1	0	0
TransfersController	4	3	9	3	12	1	1
CensusController	1	0	2	0	2	0	0
WorkshopsController	4	4	2	4	6	4	0
PublicGalleryController	1	0	1	1	1	0	0
Discourse							
CategoriesController	1	1	0	1	3	31	0
GroupsController	2	2	0	2	4	1	0
InvitesController	2	2	0	2	2	4	0
NotificationsController	1	1	0	1	1	10	1
PostActionsController	2	2	0	2	6	144	0
PostsController	1	1	0	1	4	29	0
SessionController	1	1	0	1	3	0	0
TopicsController	5	5	0	5	15	8	8
UploadsController	1	1	0	1	1	2	0
UserActionsController	1	1	0	1	1	1	0
UserBadgesController	2	2	0	2	3	4	0
UsersController	4	4	0	4	6	101	18
UsersEmailController	1	1	0	1	3	0	0
Journey							
QuestionOptionsController	2	1	1	2	1	0	0
QuestionnairesController	2	2	0	2	1	1	0
QuestionsController	2	1	1	1	3	0	0
OpenFarm							
CropsController	2	2	1	2	5	32	0
GardenCropsController	5	3	9	3	13	19	0
GardensController	4	2	6	2	10	40	0
GuidesController	2	0	5	0	7	53	0
StagesController	3	1	7	1	9	30	0
TokensController	2	0	4	0	6	2	0
UsersController	1	1	1	1	2	37	0
Total	76	60	72	62	165	985	255

the response. Here, we present these results broken down by Ruby class in which the endpoint is implemented; this data broken down by each individual endpoint is available in the Appendix.

In total, the specifications produced from dynamic proxying yielded a total of 132 response types with unique status codes: 60 responses with status code 200 and 72 responses with other status codes. The specifications produced from REST_π yielded a total of 227 (+71.97%) response types with unique status codes: 62 (+3.33%) responses with status code 200 and 165 (+129.17%) responses with other status codes. We have manually verified that REST_π correctly over-approximates the dynamic proxying: for all status codes and JSON fields observed dynamically, REST_π will point out the path conditions necessary to observe them.

Status Codes. In general, REST_π found many additional status codes across all projects. The majority of additional status codes discovered correspond to unsuccessful interaction with the API.

While some of these additional codes correspond to simple errors like a lack of authentication, there are many interesting cases discovered by REST_π that correspond to interesting code paths, such as complex validation conditions and interactions with other APIs. There were also two cases where a test suite failed to successfully interact with an endpoint, and REST_π discovered a path through the program that yields an HTTP 200 response.

Additional Fields Found. While REST_π found additional fields across all five benchmarks, it found the most additional fields on Canvas, Discourse, and OpenFarm. For Canvas, this is mostly due to a feature in the course assignments API that allows the user to request the resource be serialized as either an assignment, a quiz, or a calendar event. Dynamic testing of its endpoints only requested these endpoints serialize their resources as assignments, but in reality, all three serialization formats are supported, leading to many additional fields discovered. In Discourse, the visibility of many fields in the resulting JSON is dependent on the access control level of the user. Dynamic testing of its endpoints did not test all levels of access control, and therefore missed many fields. In OpenFarm, all resources are serialized in JSONAPI format, where associations and relations between objects are serialized along with the requested resource. Dynamic testing of its endpoints did not always test serializing a resource with and without its associations, leading to missing fields.

Required Field Issues Found. REST_π found issues with required fields in Canvas, Code.org, and Discourse. In Canvas, this was due to the serialization format mentioned above; dynamic testing of its endpoints always yielded the JSON fields for an assignment, and so marked these fields as required to appear. In reality, none of them are guaranteed to appear, as choosing a different serialization format yields an entirely different set of fields. In Code.org and Discourse, required field issues were discovered by REST_π because dynamic testing failed to cover all cases regarding access control levels, existence of associated objects, and input parameters. Additionally, dynamic testing for one endpoint, Code.org's **GET** `/dashboardapi/lockable_state.json`, yielded a specification stating it would return a JSON object `{2378: T}`, but in reality it returns a JSON object `Hash<Integer, T>`. This highlights a common challenge with dynamic specification inference where it is impossible to generalize this type without more dynamic testing.

6 Threats to Validity

The main threat to validity is the assumption of the software stack being used to implement the REST API: Ruby on Rails. Our prototype implementation supports inferring specifications for REST APIs implemented using Ruby on Rails and its associated ORM library. Leveraging this assumption solves several practical challenges. Firstly, the database schema is immediately available when using Ruby on Rails. REST_π relies on this database schema to automatically define type signatures for operations on database models. While this may not be the case for other frameworks, we believe this reliance on the availability of the database schema is reasonable as REST_π is intended to be used by API developers. Secondly, database models are serialized to JSON objects using Rails's standard API. Extending this approach to another framework would require developing additional functionality to model its serialization approach.

The main theoretical concern is the absence of type-inference rules for loops. Language-level loops are rare in Ruby programs, and even more so in Rails applications. Instead, Ruby programmers typically iterate through data structures using higher-order functions, which take a lambda and apply it iteratively. In REST_π, calls to these methods do not introduce path-conditions; their looping behavior is typed path-insensitively, and therefore soundly. Our implementation does include an unsound analysis of language-level loop constructs like `while` and `for` that analyzes the loop body once. However, none of the endpoints analyzed in our evaluation use these language-level loop constructs.

7 Related Work

As this work defines a new path-sensitive type system and applies it to the automated discovery of REST API behavior, related work falls into three major categories: intersection type systems, path-sensitive program analyses, and automated REST API analysis.

Intersection Type Systems. First described in 1978 [15], intersection types describe the notion of a value holding two types simultaneously: $v : \tau_1 \wedge \tau_2$. While initially motivated by the desire to prove normalizing properties of the λ-calculus, they have found their way into many modern programming languages. In class-based object-oriented languages such as Java or C#, they are used to describe interface composition (i.e. v : `Object` $\wedge$ `Serializable` $\wedge$ `Iterable`). In more dynamic languages, they are used to describe ad-hoc polymorphism via occurrence typing [10, 30, 41]. For example, the + operator having a type (`String`, `String`) $\rightarrow$ `String` $\wedge$ (`Integer`, `Integer`) $\rightarrow$ `Integer`. The path-types described in this paper are a form of intersection types with a crucial difference from the systems above: instead of mapping types to types, path-types map *symbolic path-constraints* to types. This lets us encode how output types depend on conditions more complex than input types, such as database state and input validation.

Path Sensitive Program Analysis. Early work on path-sensitive program analysis focused on the ability to perform abstract interpretation [16] and symbolic execution [7, 32, 33]. Path-sensitive static program analysis is, in general, infeasible due to the path explosion problem. In order to simulate a program taking both paths of a conditional, the number of paths must double at each point. This means that quickly, the number of unique paths to be considered becomes unmanageable even for small programs. Much work has been done to confine the exponential explosion by reducing the precision of the analysis [17, 18]. Instead of the full path being duplicated at each conditional branch, paths are categorized into a finite set in order to merge "similar" paths. Sen et. al. [38] reduce the impact of path-explosion for path-sensitive symbolic-execution by utilizing *value summaries*. These value summaries correspond very closely with the path-type merging utilized by REST$_\pi$.

Automated REST API Analysis. Since the inception of REST APIs, many tools have been created to support programmers working as authors or clients.

Automated test generation has been explored using white-box [5, 20] and black-box [4, 22, 42] methods. A survey of these techniques [31] found that none of them are capable of achieving high code coverage due to the lack of precise type information in specifications (beyond primitive types) and the mismatch between API specifications and implementations.

REST documentation can be informal (a natural language description) or formal (an accurate OpenAPI specification [26]). We are concerned with formal documentation as it enables precise program analysis. Swagger [25] is a very popular tool for *manually* producing an OpenAPI specification, but continuous effort is needed to keep an API specification up to date with a changing API implementation. Solutions that automatically generate OpenAPI specifications rely on analyzing endpoint traffic in production [39, 40] or from a test-suite [14, 36], analyzing existing natural-language documentation [9, 34], or by utilizing existing types or annotations [6, 37]. Huang et. al. [23] utilize a novel path-sensitive static analysis to infer OpenAPI specifications for Java, but this relies on existing manual type annotations by the developers.

Program Synthesis for REST APIs is very novel with only one group exploring this space. Guo et. al. [21] implement APIphany, a program synthesis tool for performing complex actions utilizing a REST API. This work introduces many new abstractions for reasoning about REST APIs, including semantic types, witness mining, and retrospective execution.

8 Data Availability

An artifact [3] is provided, containing all of the software and code necessary to reproduce the data presented in this paper.

A Appendix

The following tables correspond to Table 3, but broken down by each individual endpoint rather than each Ruby class. The following tables show a comparison between specifications obtained via dynamic proxying and the response types inferred by REST$_\pi$, detailing number of HTTP 200 responses (S), number of HTTP responses with other status codes (O), the number of additional fields found by REST$_\pi$, and the number of fields where the specification was inaccurate about its presence.

Canvas:

Endpoint URL	Spec S	Spec O	REST$_\pi$ S	REST$_\pi$ O	Additional Fields Found	Req. Field Issues
PUT /api/v1/courses/{id}/assignment_groups/{id}	1	2	1	3	2	0
DEL /api/v1/courses/{id}/assignment_groups/{id}	1	0	1	3	7	0
POST /api/v1/courses/{id}/assignments/{id}/duplicate	1	2	1	3	47	26
POST /api/v1/courses/{id}/assignments/{id}/retry_alignment_clone	1	1	1	1	139	62
GET /api/v1/courses/{id}/assignments/{id}	1	1	1	3	42	29
POST /api/v1/courses/{id}/assignments	0	3	0	5	93	55
PUT /api/v1/courses/{id}/assignments/{id}	1	4	1	4	92	52
PUT /api/v1/courses/{id}/assignments/bulk_update	1	3	1	4	3	0

Code.org:

Endpoint URL	Spec S	Spec O	REST$_\pi$ S	REST$_\pi$ O	Additional Fields Found	Req. Field Issues
GET /dashboardapi/lockable_state.json	1	0	1	0	0	1
GET /dashboardapi/section_progress/1.json	1	0	1	1	0	0
GET /dashboardapi/user_progress_for_stage.json	1	0	1	0	6	2
POST /maker/apply.json	1	1	1	1	0	0
POST /maker/schoolchoice.json	1	2	1	2	0	0
POST /maker/complete.json	1	2	1	2	0	0
GET /maker/application_status.json	1	1	1	1	0	0
POST /maker/override.json	1	1	1	1	0	0
POST /survey_results.json	1	0	1	1	0	0
POST /dashboardapi/sections/transfers.json	0	4	0	4	0	1
PUT /dashboardapi/sections/1/students/1.json	1	2	1	3	0	0
POST /dashboardapi/sections/1/students/bulk_add.json	1	2	1	2	0	0
POST /dashboardapi/sections/1/students/1/remove.json	1	1	1	3	1	0
POST /dashboardapi/v1/census/1.json	0	2	0	2	0	0
GET /api/v1/pd/workshops/filter.json	1	0	1	1	1	0
GET /api/v1/pd/workshops/upcoming_teachercons.json	1	0	1	1	1	0
PUT /api/v1/pd/workshops/1.json	1	1	1	2	1	0
POST /api/v1/pd/workshops.json	1	1	1	2	1	0
GET /api/v1/projects/gallery/public/1/1/1.json	0	1	1	1	0	0

Discourse:

Endpoint URL	Spec S	Spec O	REST$_\pi$ S	REST$_\pi$ O	Additional Fields Found	Req. Field Issues
POST /categories.json	1	0	1	3	31	0
GET /groups/{group_name}/members.json	1	0	1	1	0	0
PUT /groups/{group_id}/members.json	1	0	1	3	1	0
POST /invites	1	0	1	1	2	0
POST /invites/link	1	0	1	1	2	0
GET /notifications.json	1	0	1	1	10	1
POST /post_actions	1	0	1	3	72	0
DEL /post_actions/{id}	1	0	1	3	72	0
PUT /posts/{id}	1	0	1	4	29	0
POST /session/forgot_password	1	0	1	3	0	0
GET /t/{id}.json	1	0	1	4	3	8
PUT /t/-/{id}.json	1	0	1	3	0	0
PUT /t/{id}/bookmark	1	0	1	3	0	0
POST /t/{id}/invite	1	0	1	4	4	0
PUT /t/{id}/change-timestamp	1	0	1	1	1	0
POST /uploads.json	1	0	1	1	2	0
GET /user_actions.json	1	0	1	1	1	0
POST /user_badges.json	1	0	1	2	3	0
DEL /user_badges/{id}	1	0	1	1	1	0
GET /users/{username}.json	1	0	1	1	85	18
POST /users	1	0	1	1	6	0
PUT /users/password-reset/{token}	1	0	1	1	9	0
PUT /users/{username}/preferences/avatar/pick	1	0	1	3	1	0
PUT /users/{username}/preferences/email	1	0	1	3	0	0

Journey:

Endpoint URL	Spec S	Spec O	REST$_\pi$ S	REST$_\pi$ O	Additional Fields Found	Req. Field Issues
POST /questionnaires/{id}/pages/{id}/questions/{id}/question_options	0	1	1	1	0	0
PUT /questionnaires/{id}/pages/{id}/questions/{id}/question_options/{id}	1	0	1	0	0	0
GET /questionnaires/{id}	1	0	1	1	1	0
PUT /questionnaires/{id}	1	0	1	0	0	0
POST /questionnaires/{id}/pages/{id}/questions	0	1	0	2	0	0
PUT /questionnaires/{id}/pages/{id}/questions/{id}	1	0	1	1	0	0

OpenFarm:

Endpoint URL	Spec S	Spec O	REST$_\pi$ S	REST$_\pi$ O	Additional Fields Found	Req. Field Issues
POST /api/v1/crops	1	0	1	2	16	0
PUT /api/v1/crops/{id}	1	1	1	3	16	0
GET /api/v1/gardens/{garden_id}/garden_crops	1	2	1	2	5	0
POST /api/v1/gardens/{garden_id}/garden_crops	0	2	0	3	6	0
GET /api/v1/gardens/{garden_id}/garden_crops/{id}	1	2	1	2	0	0
PUT /api/v1/gardens/{garden_id}/garden_crops/{id}	1	0	1	3	8	0
DEL /api/v1/gardens/{garden_id}/garden_crops/{id}	0	3	0	3	0	0
POST /api/v1/gardens	0	1	0	3	24	0
GET /api/v1/gardens/{id}	1	1	1	2	15	0
PUT /api/v1/gardens/{id}	1	1	1	2	1	0
DEL /api/v1/gardens/{id}	0	3	0	3	0	0
POST /api/v1/guides	0	2	0	4	53	0
DEL /api/v1/guides/{id}	0	3	0	3	0	0
POST /api/v1/stages	0	3	0	3	14	0
PUT /api/v1/stages/{id}	1	1	1	3	16	0
DEL /api/v1/stages/{id}	0	3	0	3	0	0
POST /api/v1/token	0	2	0	3	1	0
DEL /api/v1/token	0	2	0	3	1	0
GET /api/v1/users/{id}	1	1	1	2	37	0

References

[1] 2025. Generate OpenAPI schema from RSpec request specs. https://github.com/exoego/rspec-openapi.

[2] 2025. HAR to OpenAPI spec generator. https://github.com/jonluca/har-to-openapi.

[3] Mark W Aldrich, Kyla H Levin, Michael Coblenz, and Jeffrey S Foster. 2025. Replication Package for REST$_\pi$: Path Sensitive Type Inference for REST APIs. doi:10.5281/zenodo.16916395

[4] Juan C Alonso, Alberto Martin-Lopez, Sergio Segura, Jose Maria Garcia, and Antonio Ruiz-Cortes. 2022. ARTE: Automated generation of realistic test inputs for web APIs. *IEEE Transactions on Software Engineering* 49, 1 (2022), 348–363.

[5] Andrea Arcuri. 2018. Evomaster: Evolutionary multi-context automated system test generation. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 394–397.

[6] Luke Autry. 2024. Build OpenAPI-compliant REST APIs using TypeScript and Node. https://github.com/lukeautry/tsoa.

[7] Robert S Boyer, Bernard Elspas, and Karl N Levitt. 1975. SELECT—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices* 10, 6 (1975), 234–245.

[8] Nat Budin. 2024. Journey: An online questionnaire application. https://github.com/nbudin/journey/.

[9] Hanyang Cao, Jean-Rémy Falleri, and Xavier Blanc. 2017. Automated generation of REST API specification from plain HTML documentation. In *Service-Oriented Computing: 15th International Conference, ICSOC 2017, Malaga, Spain, November 13–16, 2017, Proceedings*. Springer, 453–461.

[10] Giuseppe Castagna, Victor Lanvin, Mickaël Laurent, and Kim Nguyen. 2022. Revisiting occurrence typing. *Science of Computer Programming* 217 (2022), 102781.

[11] OpenFarm CC. 2024. A free and open database for farming and gardening knowledge. https://github.com/openfarmcc/OpenFarm.

[12] Michael Coblenz, Wentao Guo, Kamatchi Voozhian, and Jeffrey S Foster. 2023. A Qualitative Study of REST API Design and Specification Practices. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 148–156.

[13] Code.org. 2024. The code powering code.org and studio.code.org. https://github.com/code-dot-org/code-dot-org.

[14] Ruby Swagger Community. 2024. Seamlessly adds a Swagger to Rails-based API's. https://github.com/rswag/rswag.

[15] Mario Coppo and Mariangiola Dezani-Ciancaglini. 1978. A new type assignment for λ-terms. *Archiv für mathematische Logik und Grundlagenforschung* 19 (1978), 139–156.

[16] Patrick Cousot. 1996. Abstract interpretation. *ACM Computing Surveys (CSUR)* 28, 2 (1996), 324–328.

[17] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 57–68.

[18] Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 270–280.

[19] Roy Thomas Fielding. 2000. *Architectural styles and the design of network-based software architectures*. University of California, Irvine.

[20] Gordon Fraser and Andrea Arcuri. 2012. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2012), 276–291.

[21] Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. 2022. Type-directed program synthesis for RESTful APIs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 122–136.

[22] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving semantics-aware fuzzers from web api schemas. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 345–346.

[23] Ruikai Huang, Manish Motwani, Idel Martinez, and Alessandro Orso. 2024. Generating REST API Specifications through Static Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[24] Civilized Discourse Construction Kit Inc. 2024. Discourse: A platform for community discussion. https://github.com/discourse/discourse.

[25] SmartBear Software Inc. 2024. Swagger. https://swagger.io.

[26] OpenAPI Initiative. 2024. *OpenAPI Specification v3.1.0*. https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.1.0.md

[27] Inc. Instructure. 2024. The open LMS by Instructure, Inc. https://github.com/instructure/canvas-lms.

[28] 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*. 966–979.

[29] 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*. 112–125.

[30] Andrew M Kent, David Kempe, and Sam Tobin-Hochstadt. 2016. Occurrence typing modulo theories. *ACM SIGPLAN Notices* 51, 6 (2016), 296–309.

[31] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated test generation for rest apis: No time to rest yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 289–301.

[32] James C King. 1975. A new approach to program testing. *ACM Sigplan Notices* 10, 6 (1975), 228–233.

[33] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[34] Papa Alioune Ly, Carlos Pedrinaci, and John Domingue. 2012. Automated information extraction from web APIs documentation. In *Web Information Systems Engineering-WISE 2012: 13th International Conference, Paphos, Cyprus, November 28-30, 2012. Proceedings 13*. Springer, 497–511.
[35] Kamil Mysliwiec. 2024. OpenAPI (Swagger) module for Nest framework (node.js). https://github.com/nestjs/swagger.
[36] Badr Nasslahsen. 2024. Library for OpenAPI 3 with spring-boot. https://github.com/springdoc/springdoc-openapi.
[37] Sebastián Ramírez. 2024. FastAPI: a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints. https://github.com/fastapi/fastapi.
[38] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 842–853.
[39] SM Sohan, Craig Anslow, and Frank Maurer. 2017. Automated example oriented REST API documentation at Cisco. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 213–222.
[40] Sheikh Mohammed Sohan, Craig Anslow, and Frank Maurer. 2015. Spyrest: Automated restful API documentation using an HTTP proxy server. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 271–276.
[41] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of Typed Scheme. *ACM SIGPLAN Notices* 43, 1 (2008), 395–406.
[42] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. Resttestgen: automated black-box testing of restful apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 142–152.

Received 2024-10-16; accepted 2025-08-12