Type-Directed Program Synthesis for RESTful APIs

Zheng Guo
UC San Diego
USA
zhg069@ucsd.edu

David Cao
UC San Diego
USA
dmcao@ucsd.edu

Davin Tjong
UC San Diego
USA
dtjong@ucsd.edu

Jean Yang
Akita Software
USA
jean@akitasoftware.com

Cole Schlesinger
Akita Software
USA
cole@akitasoftware.com

Nadia Polikarpova
UC San Diego
USA
npolikarpova@ucsd.edu

Abstract

With the rise of software-as-a-service and microservice architectures, RESTful APIs are now ubiquitous in mobile and web applications. A service can have tens or hundreds of API methods, making it a challenge for programmers to find the right combination of methods to solve their task.

We present APIphany, a component-based synthesizer for programs that compose calls to RESTful APIs. The main innovation behind APIphany is the use of precise *semantic types*, both to specify user intent and to direct the search. APIphany contributes three novel mechanisms to overcome challenges in adapting component-based synthesis to the REST domain: (1) a type inference algorithm for augmenting REST specifications with semantic types; (2) an efficient synthesis technique for "wrangling" semi-structured data, which is commonly required in working with RESTful APIs; and (3) a new form of simulated execution to avoid executing APIs calls during synthesis. We evaluate APIphany on three real-world APIs and 32 tasks extracted from GitHub repositories and StackOverflow. In our experiments, APIphany found correct solutions to 29 tasks, with 23 of them reported among top ten synthesis results.

CCS Concepts: • **Software and its engineering** → **Automatic programming**.

Keywords: Program Synthesis, RESTful API, Type Inference

ACM Reference Format:
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 (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3519939.3523450

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523450

1 Introduction

Software-as-a-service has emerged as a widely-used means for developers to leverage third-party software. Developers might send requests to Stripe to handle payments or integrate with Slack to publish notifications, all while making use of cloud providers to provision various form of storage and compute. According to recent industry surveys, more than 80% of respondents' services offer RESTful APIs [27, 31], and these APIs are extensive. Slack, for example, has 174 API methods as of version 1.5.0. Amazon Web Services offers over two hundred products and services, each with tens or hundreds of API methods. Even with comprehensive documentation—which is by no means guaranteed—using a new service can be a daunting proposition.

As an example, consider a question posed on StackOverflow about the Slack API: *How do I retrieve all member emails from a Slack channel with a given name?* The answer is surprisingly complicated:

1. First, call `conversations_list`[1] to retrieve the array of all channel objects, and then search for a channel object with a given name and get its ID;
2. Next, call `conversations_members` on the channel ID to get all user IDs of its members;
3. Finally, for each user ID, call `users_info` to retrieve a user object `u`, and then access the user's email via `u.profile.email`.

To come up with this solution, one must be familiar with `channel` objects, `user` objects, and three different API methods.

Component-based program synthesis [8, 15, 19, 22] has been previously used to help programmers navigate APIs in Java, Scala, and Haskell. Component-based synthesizers take as input a type signature and (in most cases) a set of input-output examples, and return a list of program snippets that compose API calls and have the desired type and input-output behavior. This is a powerful approach for navigating APIs, because it allows developers to start with information

[1] We shorten method names for brevity and elide the distinction between REST *methods* and *endpoints*, irrelevant in this context.

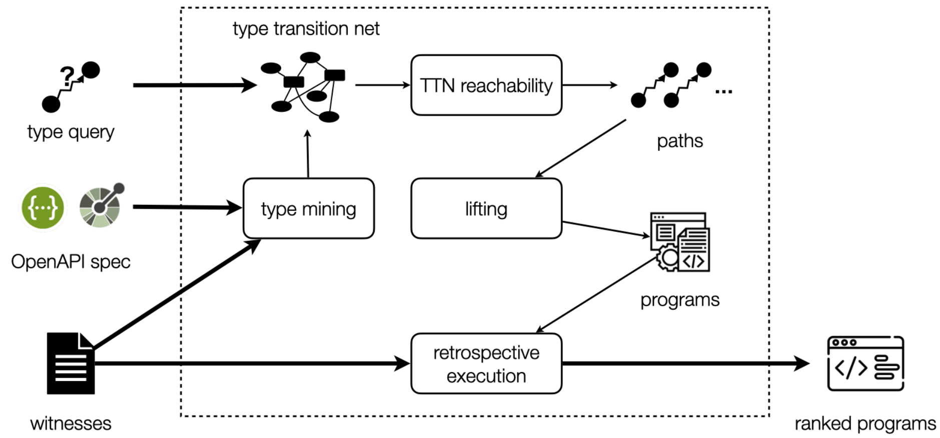

Figure 1. Overview of APIPHANY

easily at hand—the types of inputs they have and the outputs they desire—and requires no knowledge of which API methods to apply.

Challenges. Unfortunately, there are three significant challenges in applying component-based synthesis to RESTful APIs. First, component-based synthesis relies on *types* both for expressing user intent and for efficient search, but types in REST APIs are quite shallow. For example, in the SLACK API specification, both channel names and emails have type `String`, so our example, which transforms a channel name into an array of emails, would have a very imprecise type signature `String` ⟶ `[String]`.

Second, RESTful APIs commonly transmit *semi-structured data*, *i.e.* arrays of objects, which may themselves contain nested objects and arrays. As a result, using an API is often not as simple as sequencing together a handful of method calls; instead, the calls must be interleaved with "data wrangling" operations such as projections, maps, and filters. These data wrangling operations are challenging for component-based synthesis: they are extremely generic, and hence significantly expand the search space.

Finally, to compensate for the inherent ambiguity of types, component-based synthesis typically relies on *executing* candidate program snippets and matching them against user-provided input-output examples. In a software-as-a-service environment, this is a complete non-starter: not only is the user generally unaware of the internal state of the service and hence unable to provide accurate examples, but executing API calls during synthesis can also be prohibitively expensive due to rate limits imposed by the services and, even more importantly, can have unrecoverable side effects, such as deleting accounts or publishing messages.

APIPHANY: synthesis with semantic types. Our core insight is that type-based specifications are actually a good fit for REST APIs, as long as the types are more fine-grained. In our example, if the SLACK API had dedicated types for `Channel.name` and `Profile.email`, the programmer could specify their intent as the type `Channel.name` ⟶ `[Profile.email]`. Although this specification is still somewhat ambiguous, intuitively it has enough information to narrow down the synthesis results to a manageable number such that the programmer can manually inspect the remaining solutions. We refer to such fine-grained types as *semantic types*.

In this paper, we present APIPHANY, a component-based synthesizer for REST APIs guided by semantic types. Fig. 1 shows a high-level overview of our approach, which is structured into two phases: (1) the *analysis* phase infers semantic type annotations for a given API; (2) the *synthesis* phase uses these type annotations to perform component-based synthesis. For the SLACK API, APIPHANY is able to infer, for example, that the method `conversations_members` has the semantic type `Channel.id` ⟶ `[User.id]`. At synthesis time, given the *type query* `Channel.name` ⟶ `[Profile.email]`, APIPHANY returns a ranked list of programs of this type, where the desired solution (shown in Fig. 2) appears among the top ten. APIPHANY's output is expressed in a compact DSL inspired by Haskell's monadic `do`-notation and Scala's `for`-comprehensions, which, however, can be easily translated into the user's language of choice for communicating with the API.

Contributions. We present the design, implementation, and evaluation of APIPHANY, including:

1. *Type mining* (Sec. 4), a technique that infers semantic types from a set of *witnesses* (observed invocations of API methods). Witnesses can be generated in a sandbox or by tapping live production traffic; in either case, they are collected ahead of time, once per API, which avoids inducing side effects during synthesis.
2. Efficient synthesis of wrangling operations for semi-structured data via *array-oblivious search* (Sec. 5), which omits challenging array operations during search, and recovers them later via type-directed lifting.

```
\channel_name → {
  c   ← conversations_list()
  if c.name = channel_name
  uid ← conversations_members(channel=c.id)
  let u = users_info(user=uid)
  return u.profile.email
}
```

Figure 2. Solution for retrieving all member emails from a Slack channel in APIphany DSL.

3. Ranking synthesis results with the help of *retrospective execution* (Sec. 6), a type of simulated execution using previously collected witnesses. Retrospective execution helps APIphany weed out uninteresting programs (*e.g.* programs that always return an empty array), reducing the number of synthesis results the user has to inspect to find their expected solution.

We evaluate APIphany on three real-world APIs, and 32 tasks extracted from GitHub repositories and StackOverflow (Sec. 7). Our evaluation shows that APIphany can find solutions to the majority of tasks (29/32) within 150 seconds. Moreover, semantic types are crucial to its effectiveness: without type mining, APIphany can only solve four tasks. Finally, ranking significantly improves the quality of reported solutions, increasing the number of correct solutions appearing in top ten results from 12/29 to 23/29.

2 APIphany by Example

In this section we use the task of retrieving all member emails in a Slack channel as a running example to illustrate the APIphany workflow depicted in Fig. 1.

2.1 API Analysis by Example

API analysis is performed once per API. It takes as input a *spec* in the popular OpenAPI format[2] and a set of *witnesses* (successful API method calls); it produces a spec annotated with semantic types. OpenAPI specs are publicly available for most popular APIs.[3] Witnesses can be generated in a number of ways, for example, by running an integration test suite in a sandbox or by passively listening to production API traffic. We envision witness collection and API analysis being performed by the API maintainer (or another interested party), not by regular users of the APIphany synthesizer.

OpenAPI specs. Fig. 3 shows a fragment of the OpenAPI spec provided by Slack. An OpenAPI spec consists of object definitions and method definitions. We show definitions of three objects, `user`, `profile` and `channel`, and two methods, `users_info` and `conversations_list`, relevant to our example.

[2]https://swagger.io/. APIphany supports both OpenAPI v2 and v3.
[3]Slack OpenAPI spec is available at: https://raw.githubusercontent.com/slackapi/slack-api-specs/master/web-api/slack_web_openapi_v2.json

As you can see, the spec does provide precise type information for some of the locations: for example, the response of `users_info` clearly has type `User` (it is annotated with a reference to the corresponding object definition). The bulk of the locations, however, such as the field `user.id` or the parameter of `users_info`, are simply annotated with `String`, which is not very helpful for the purposes of type-directed synthesis. Our goal is to replace these `String` annotations with more fine-grained types.

Mining types from witnesses. To this end, we build upon an algorithm first proposed in [1] that infers types by *mining* them from execution traces, based on the insight that *equal values observed at different locations likely have the same type*. More specifically, our type mining algorithm starts by ascribing a unique semantic type to each `String` location and then merges locations that share a value anywhere in the witness set. As an illustration, consider Fig. 4, which lists two witnesses for the API methods from our running example. In this witness set we observe the same value `"UJ5RHEG4S"` in three locations: (1) the *parameter* of `users_info`, (2) the `id` *field* of a `User` object (we know from the spec that `users_info` returns a `User`), and (3) the `creator` *field* of a `Channel` object (we know from the spec that `conversations_list` returns an array of `Channels`). Hence we merge all three locations into the same semantic type. For presentation purposes, we assign the name `User.id` to this type, which is derived from location (2) above. The choice of name is not important, however: the user is free to refer to this semantic type via any of its representative locations; for example, `Channel.creator` also denotes the same type.

2.2 Program Synthesis by Example

The program synthesis phase of APIphany is meant to be used by regular programmers, any time they need help accomplishing a task with one of the supported APIs. The programmer queries APIphany with a type signature built from semantic types. Although the UI for constructing queries is beyond the scope of this paper, we envision the programmer browsing object definitions and selecting relevant fields as semantic types. For our running example, the programmer knows that they need to go from a channel name to an array of user emails; they might first look through the `channel` object definition and find the `name` field; they might then search globally for a field called `email` and find it inside the `profile` object; hence they settle on the type query `Channel.name` → [`Profile.email`].

The program synthesis phase itself comprises two steps, beginning with a program *search* step to generate a list of candidate programs with a given type, followed by a *ranking* step to identify promising candidates (described in Sec. 2.3).

Challenge: components meet control flow. Given the type query `Channel.name` → [`Profile.email`], how would APIphany go about enumerating all programs of this type? This task

```
{"user": {"type": "object",
          "properties": {"id":{"type": "string"},
                         "name": {"type": "string"},
                         "profile":{"$ref": "#/definitions/profile"}}},
 "profile": {"type": "object",
             "properties": {"display_name": {"type": "string"},
                            "email": {"type": "string"}}},
 "channel": {"type": "object",
             "properties": {"creator": {"type": "string"},
                            "name": {"type": "string"},
                            "id": {"type": "string"}}}
}
```

```
{"users_info":
  {"parameter": [
    {"in": "query", "name": "user", "type": "string"}],
   "responses": {"200": {"schema": {"properties":
       {"user": {"$ref": "#/definitions/user"}}
  }}}},
 "conversations_list":
  {"parameter": [],
   "responses": {"200": {"schema": {"properties":
     {"channels":
       {"type": "array",
        "items": {"$ref": "#/definitions/channel"}
       }}
  }}}}
}
```

Figure 3. Fragment of the Slack API's OpenAPI specification. (left) Definitions of `user`, `profile` and `channel` objects. (right) Parameters and responses of the methods `users_info` and `conversations_list`.

```
GET /conversations_list
{
  "ok": true,
  "channels": [
    {
      "id": "C12345678",
      "name": "general",
      "creator": "UJ5RHEG4S",
      ...
```

```
GET /users_info?user=UJ5RHEG4S
{
  "ok": true,
  "user": {
    "id": "UJ5RHEG4S",
    "name": "demo_user",
    "profile": {
      "email": "xyz@gmail.com",
      ...
```

Figure 4. Witnesses for two Slack API methods. Arrows connect equal values observed at different locations. Type mining ascribes the type `User.id` to all the boxed locations.

```
\channel_name → {
  c ← conversations_list()
  if c.name = channel_name
  let uid = c.creator
  let u = users_info(user=uid)
  return u.profile.email
}
```

```
\channel_name → {
  c ← conversations_open()
  if c.name = channel_name
  let uid = c.creator
  let u = users_info(user=uid)
  return u.profile.email
}
```

Figure 5. A sample of incorrect candidate solutions.

presents a challenge to existing synthesis techniques because our candidate programs have *both* a large component library to choose from—from dozens to hundreds of methods—*and* non-trivial control flow—*e.g.* the solution to our running example has to *loop* over the members of a channel. One line of prior work that scales to large component libraries is graph-based search using *type-transition nets* (TTNs) [8, 13]; unfortunately, this approach can only generate sequences of method calls, and does not support loops.

The APIphany DSL. We observe that the loops we need for manipulating semi-structured data are restricted to iterating over (possibly nested) arrays of objects. To capture this restricted class of programs we have designed a DSL inspired by Scala's `for`-comprehensions, Haskell's monadic `do`-notation, and `LINQ` [23]. The solution to our running example in this DSL is given in Fig. 2. In this language, iteration over an array is expressed using the monadic *bind* operation (written ←). For example, the second bind in Fig. 2 has the effect of performing the subsequent computation for every element `uid` of the array returned in line 4:

```
4  uid  ← conversations_members(channel=c.id);
5  let u = users_info(user=uid);
6  return u.profile.email
```

Array-oblivious search. The main idea behind APIphany's search is that although we cannot directly synthesize the program above using existing TTN-based techniques, we can synthesize an *array-oblivious* version of this program, where we pretend that `conversations_members` returns a single `User.id` instead of an array, and hence we can simply sequence the two method calls, without monadic binding:

```
4  let uid = conversations_members(channel=c.id);
5  let u   = users_info(user=uid);
6  u.profile.email
```

To transform an array-oblivious program into the final solution, APIphany *lifts* it into a comprehension by replacing each **let** binding that causes a type mismatch with a monadic bind. In our example, the **let** in line 4 causes a type error (because `conversations_members` returns `[User.id]`, while `users_info` expects a single `User.id`), while the **let** in line 5 does not (since `users_info` returns a single `User`); hence lifting replaces the first **let**-binding with ← but not the second.

2.3 Ranking via Retrospective Execution

Although semantic types are less ambiguous than primitive types for expressing user intent, they are still not precise enough to exactly identify the desired program. For example, our synthesizer generates more than 1000 candidates for the type signature `Channel.name` → `[Profile.email]`; clearly, it is infeasible for the user to manually go through all of them. Hence, APIphany must be able to rank the candidates in order to show the user a small number of likely solutions.

Fortunately, most of the 1000 candidates are easy to weed out because they produce uninteresting results. Consider two of the candidates depicted in Fig. 5, which differ from our desired solution (Fig. 2) in the highlighted fragments: the first program returns the email of the channel's *creator* (as opposed to all of its members), and the second one gets the list of channels from `conversations_open`, which is intended for opening a direct message channel. It turns out that the second program *always fails* at run time, because a successful call to `conversations_open` requires providing exactly one of its two optional arguments (a channel ID or a list of users). The first program executes successfully, but it always returns a single email, while the user asked for an array of emails. For these reasons, both of these programs are less likely to be the intended solution than the program in Fig. 2, which successfully returns multiple emails at least sometimes.

A natural idea is to test all candidate programs on random inputs and rank them based on the results they produce. Unfortunately, as we have hinted above, there are several barriers to systematically executing many candidate programs that make calls to REST APIs. First, most REST APIs set a rate limit on how frequently a user can make method calls or how many calls a user can make in a day. Second, many REST API methods are side-effecting. Unlike a self-contained binary, a remotely-hosted service cannot be restarted from a clean state for each execution.

Retrospective execution. We propose *retrospective execution* (RE) as an efficient, non-side effecting alternative to program execution. The main idea is to simulate execution by "replaying" witnesses collected for the API analysis phase. When evaluating a candidate program, rather than executing an API call, RE instead searches for a matching witness and substitutes its response at the call site. If done naively, however, this process almost always yields failure or an empty array; so making RE useful for ranking purposes requires explicitly *biasing* execution towards meaningful results.

As an illustration, consider executing the program in Fig. 2 using the witnesses in Fig. 4. As the first step, we simulate the call to `conversations_list` using the first witness; the response is an array of channels with names `"general"`, `"private-test"`, and `"team"`. The second step is to filter this array, retaining only those channels whose name is equal to the input parameter `channel_name`. If we had sampled the value for `channel_name` eagerly, before running the program, we could scarcely have chosen one of the three names actually present in the array, so the filtering step (and hence the whole program) would almost always return an empty array. Instead we sample the value for `channel_name` *lazily*, once we encounter the filter, picking one of the names present in the array.

Assume that we picked `channel_name = "general"`, and hence the filter returns the first channel. Next, we simulate the call to `conversations_members` on this channel's ID. Because our witness set is sparse, we may or may not find an exact match for this call; in the latter case, we sample the response from the set of *approximate matches*, *i.e.* witnesses with the same

$$
\begin{array}{rll}
o ::= & \texttt{User} \mid \texttt{Channel} \mid \ldots & \text{object names} \\
f ::= & \texttt{u\_info} \mid \ldots & \text{method names} \\
l ::= & \texttt{in} \mid \texttt{out} \mid \texttt{0} \mid \texttt{id} \mid \texttt{name} \mid \ldots & \text{field labels} \\
\ell ::= & l \mid ?l & \text{record fields} \\
loc ::= & o.\overline{l} \mid f.\overline{l} & \text{locations}
\end{array}
$$

Terms

$$
\begin{array}{rll}
e ::= & & \textit{Expressions} \\
 & \mid x \mid e.l & \text{variable, projection} \\
 & \mid f(\overline{l_i = e_i}) \mid \textbf{let } x = e; e & \text{method call, pure binding} \\
 & \mid \textbf{if } e = e; e \mid x \leftarrow e; e & \text{guard, monadic binding} \\
 & \mid \textbf{return } e & \text{pure value lifting} \\
\mathcal{E} ::= & \lambda \overline{x}.e & \textit{Top Level Programs}
\end{array}
$$

Values

$$
v ::= \texttt{"..."} \mid [\overline{v}] \mid \{\overline{l_i = v_i}\} \quad \text{strings, arrays, objects}
$$

Types

$$
\begin{array}{rll}
t ::= & & \textit{Syntactic types} \\
 & \mid \texttt{String} & \text{strings} \\
 & \mid o \mid [t] \mid \{\overline{\ell_i : t_i}\} & \text{named objects, arrays, records} \\
s ::= & t \to t & \text{function types} \\
\hat{t} ::= & & \textit{Semantic types} \\
 & \mid \{\overline{loc}\} & \text{loc-sets} \\
 & \mid o \mid [\hat{t}] \mid \{\overline{\ell_i : \hat{t}_i}\} & \text{named objects, arrays, records} \\
\hat{s} ::= & \hat{t} \to \hat{t} & \text{function types}
\end{array}
$$

Libraries

$$
\begin{array}{rll}
\Lambda ::= & \overline{o : t}; \overline{f : s} & \text{object and method definitions} \\
\hat{\Lambda} ::= & \overline{o : \hat{t}}; \overline{f : \hat{s}} & \text{semantic definitions}
\end{array}
$$

Figure 6. Syntax of the language λ_A

method names and argument names,[4] but not necessarily the same argument values. Due to approximate matching, RE results do not always equal the results of a real execution, but they are still useful for estimating whether a program candidate is able to produce meaningful outputs. For each candidate, we run RE multiple times (with different random seeds) and use the outputs to assign a rank to each candidate.

3 The Core Language

In this section, we formalize the core of APIphany's DSL as λ_A, a functional language specialized for manipulating semi-structured data. The syntax of λ_A is summarized in Fig. 6.

Types. The types of λ_A include *syntactic types* t (those used in the OpenAPI spec) and *semantic types* $\hat{t}$, which we infer. Both categories of types have named objects o, arrays $[t]$, and records $\{\overline{\ell_i : t_i}\}$.[5] Records are mappings from field labels to types; some fields are optional, indicated with a ? before its label. For example, the record type $\{\texttt{id} : \texttt{String}, ?\texttt{time\_zone} : \texttt{String}\}$, has a required field `id` and an optional field `time_zone`.

[4]Because in REST some arguments are optional, the same method can be called with different subsets of arguments.

[5]We write $\overline{X}$ to denote zero or more occurrences of a syntactic element X.

The two categories of types differ in their base types: the sole primitive syntactic type is `String`,[6] while the sole primitive semantic type is a *loc-set*, *i.e.* a set of locations.

A *location* is an object or method name followed by a sequence of labels, such as `User.id`. Apart from field labels that correspond to object fields in the OpenAPI spec, we introduce three reserved labels—in, out, and 0—for addressing method parameters and responses, and array elements, respectively. For example, `c_list`.out.0 refers to an element type of the response array of the method `c_list`.

Function types are written $t \to t$, and multiple arguments are represented as a record whose fields encode argument names (with optional fields encoding optional arguments).

A library Λ models an OpenAPI spec. It contains object definitions, which bind object identifiers to (record) types, and method definitions, which bind method names to function types. A semantic library $\hat{\Lambda}$, which is the output of type mining, binds object identifiers and method names to semantic types. As an example, Fig. 7 shows Λ definitions that correspond to a portion of the Slack OpenAPI spec (with method names shortened for brevity), and their corresponding definitions in the semantic library $\hat{\Lambda}$.

Terms. Values of λ_A include string literals, arrays, and objects; objects are mappings from field labels to values. Similarly to Haskell's `do`-notation, **`return`** e returns an array with a single element e, and the monadic binding $x \leftarrow e_1; e_2$ evaluates e_2 for each element x of the array e_1, and concatenates all resulting arrays. In contrast, the pure binding **`let`** $x = e_1; e_2$ binds x to the entire result of e_1 and then evaluates e_2. The guard expression **`if`** $e_1 = e_2; e$ evaluates e if the guard holds and returns an empty array otherwise; guards are restricted to equalities, since these are the only guards generated by APIphany. At the top level, a program $\mathcal{E}$ is an abstraction with a list of arguments $\overline{x}$ and body e.

4 Type Mining

In this section we detail APIphany's type mining algorithm, using the library Λ in Fig. 7 and the witnesses in Fig. 4 as a running example. Informally, the idea is to first assign every `String` location *loc* in Λ a unique type $\{loc\}$, and then merge the types of some locations based on the witnesses.

Assigning location-based types. We formalize the first step as a judgement $\Lambda \vdash loc \Longrightarrow \hat{t}$, which assigns a semantic type $\hat{t}$ to location *loc* based only on the information present in the syntactic library Λ. The reader might be wondering why isn't the assigned type $\hat{t}$ always simply $\{loc\}$. This is indeed the case for `String`-annotated locations explicitly present in Λ, such as `User.id` or `u_info`.in.`user`. But in other cases, location-based type assignment is more involved; for example:

- $\Lambda \vdash$ `u_info`.out $\Longrightarrow$ `User` because this location is annotated with a named object type.
- $\Lambda \vdash$ `c_members`.out $\Longrightarrow$ [{`c_members`.out.0}] because array types do not themselves get replaced with loc-sets; instead, we recursively assign a location-based type to an array's element.
- $\Lambda \vdash$ `u_info`.out.`id` $\Longrightarrow$ {`User.id`} because type assignment *canonicalizes* locations inside types to make sure they explicitly appear in Λ; to this end, we recursively assign a type to location's prefix, $\Lambda \vdash$ `u_info`.out $\Longrightarrow$ `User`, and then follow the field `id` of the `User` object.

The formalization of location-based type assignment is mostly straightforward and relegated to the technical report [12].

Merging types via a disjoint-set. Type mining relies on a variant of the *disjoint-set* data structure (also known as *union-find* [32]). Our disjoint-set *DS* stores disjoint groups of pairs (loc, v), where *loc* is a location and v is a string value. When two pairs are in the same group, their corresponding locations have the same semantic type.

DS supports two efficient operations: insert and find. insert takes a pair (loc, v) and checks whether either of its components already appears in *DS*; if so, it merges the new pair into the corresponding group, and otherwise puts it into a new group. find takes a location *loc* and returns a semantic type $\hat{t}$; internally, find locates the group to which the pair $(loc, \_)$ belongs in *DS* and returns the loc-set $\{loc, loc_1, \ldots\}$ that contains all locations in that group.

Type mining algorithm. Fig. 8 presents the top-level algorithm MineTypes, which takes as input a syntactic library Λ and a set of witnesses $\mathcal{W}$, and returns a semantic library $\hat{\Lambda}$. A *witness* W is a triple $\langle f, v_{in}, v_{out} \rangle$, where f is a method name and v_{in}, v_{out} are its argument and response value (multiple arguments are represented as an object). MineTypes operates in two phases: in lines 2–5 it builds the disjoint-set *DS* from $\mathcal{W}$ and in line 6 it build $\hat{\Lambda}$ from *DS*.

In the first phase, the algorithm iterates over the witnesses, registering the input value v_{in} at the location f.in and the output value v_{out} at the location f.out. To this end, we call a helper function AddWitness, which drills down into composite values (arrays and objects) to get to string literals, and then inserts each string into *DS* with its location-based type. For example, when processing the response from the first witness in Fig. 4, AddWitness iterates over all channel objects in the array, and over all fields of each channel object; once it reaches the value `"UJ5RHEG4S"`, it computes the type of its location as $\Lambda \vdash$ `c_list`.out.0.`creator` $\Longrightarrow$ {`Channel.creator`}, and inserts (`Channel.creator`, `"UJ..."`) into *DS*. Processing the second witness results in inserting the pairs (`u_info`.in.`user`, `"UJ..."`) and (`User.id`, `"UJ..."`), which share the same string value, and hence all three pairs get merged into the same group. Once all the witnesses are added to *DS*, its groups represent the final set of semantic types.

In the second phase, the algorithm calls AddDefinitions to iterate over all object and method definitions in Λ, and

[6] In practice, REST APIs also include integers and booleans; these types are handled slightly differently in APIphany, as discussed in Sec. 7.4.

	Syntactic library Λ	Semantic library $\hat{\Lambda}$

Objects	`Channel: { id: String,`
	`name: String,`
	`creator: String }`
	`User: { id: String,`
	`name: String,`
	`profile: Profile }`
		`Channel: { id: Channel.id ,`
		`name: Channel.name ,`
		`creator: User.id }`
		`User: { id: User.id ,`
		`name: User.name ,`
		`profile: Profile }`

Methods	`c_list:`
	`{} → [Channel]`
	`u_info:`
	`{user: String} → User`
	`c_members:`
	`{channel: String} → [String]`
		`c_list:`
		`{} → [Channel]`
		`u_info:`
		`{user: User.id } → User`
		`c_members:`
		`{channel: Channel.id } → [ User.id ]`

Figure 7. Library Λ that models a portion of the Slack OpenAPI spec and the corresponding semantic library $\hat{\Lambda}$. Each gray box is a loc-set type inferred by type mining, depicted for brevity using a single representative location from the set.

Input: A library Λ and witnesses $\mathcal{W}$
Output: A semantic library $\hat{\Lambda}$

```
function MineTypes(Λ, 𝒲)
  DS ← empty disjoint-set
  for ⟨f, v_in, v_out⟩ ∈ 𝒲 do
    AddWitness(DS, f, in, v_in)
    AddWitness(DS, f, out, v_out)
  Λ̂ ← AddDefinitions(Λ, DS)
  return Λ̂

function AddWitness(DS, loc, v)
  match v
    case "...":
      Λ ⊢ loc ⟹ {loc′}
      DS ← insert(DS, loc′, v)
    case [v̄_i]:
      forall i : AddWitness(DS, loc.0, v_i)
    case {l_i = v_i}:
      forall i : AddWitness(DS, loc.l_i, v_i)
```

Figure 8. Type mining algorithm.

add corresponding definitions to $\hat{\Lambda}$, relying on find to retrieve the semantic type for each location. For example, when adding the method `u_info`, we query find(DS, `u_info`.in.`user`), which finds the group mentioned above and returns its loc-set: {`User.id`, `Channel.creator`, ...}. If the requested location is not in DS—because $\mathcal{W}$ has no witnesses for the enclosing method or object—it is annotated with the unmerged location-based type.

5 Type-Directed Synthesis

In this section, we discuss how APIphany generates a set of well-typed programs given a query type, using the same running example as in previous sections.

Synthesis problem. Formally, our synthesis problem is defined by a semantic library $\hat{\Lambda}$ and a semantic query type $\hat{s}$. For our running example, we use the semantic library from Fig. 7 and the query type `Channel.name` $\to$ [`Profile.email`].[7] A *candidate solution* is any program $\mathcal{E}$ that type-checks against $\hat{s}$. To formalize this notion, we introduce the program typing judgment $\hat{\Lambda} \vdash \mathcal{E} :: \hat{s}$, which is mostly straightforward. We note only that in a monadic binding $x \leftarrow e_1; e_2$, both e_1 and e_2 must have array types; in a guard **if** $e_1 = e_2; e$, e must have an array type, while e_1 and e_2 must have (the same) loc-set type, since equality is only supported over string values. Full definition can be found in the technical report [12].

Type transition nets. To efficiently enumerate well-typed programs we follow prior work [8, 13] and encode the search space as a special kind of Petri net, called *type-transition net* (TTN). Intuitively, a TTN encodes how each API method transforms values of one semantic type into another; *e.g.* `u_info` transforms a `User.id` into a `User`. Fig. 9 shows a TTN for our running example. Places (circles) correspond to semantic types, transitions (rectangles) correspond to methods, and edges connect methods with their input and output types. In addition to API methods, the TTN contains transitions that correspond to λ_A projections (*e.g.* $\mathsf{proj}_{\texttt{User.profile}}$ and $\mathsf{proj}_{\texttt{Profile.email}}$) and guards (*e.g.* $\mathsf{filter}_{\texttt{Channel.name}}$).

Array-oblivious search. For our search space encoding to be useful, we need to make sure that every well-typed λ_A program corresponds to a path in the TTN. This is where we encounter a challenge: there is no straightforward way to encode λ_A's monadic bind operation into the TTN. Although prior work on Hoogle+ [13] supports higher-order functions, the arguments to those functions are syntactically restricted to variables (*i.e.* inner lambda abstractions are not supported), which is insufficient for our purposes. To address this problem, we introduce a new, *array-oblivious* TTN encoding, which does not distinguish between array types and types of their elements, and hence does not require monadic binds. For example, in Fig. 9 `c_members` returns `User` instead of `[User]`, and hence its output can be passed directly to `u_info`, without iterating over it.

Search in the TTN. Once the TTN is built, we enumerate paths from the input to the output type (or rather, array-oblivious versions thereof). In our example, we place a *token* in the input type `Channel.name` and search for a path (a sequence of transitions) that would get this token to the output type `Profile.email`, possibly generating and consuming extra tokens along the way. The bold path in Fig. 9 corresponds to our desired solution from Fig. 2. On this path, we first fire the transition `c_list` (which does not consume any

[7]Here and throughout this section, we write loc-set types using an arbitrarily chosen representative; the user can query APIphany using any locations of their choosing, and the tool interprets them as the loc-sets they belong to.

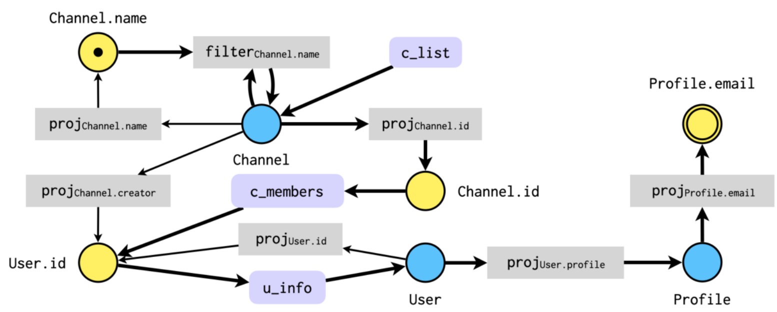

Figure 9. A fragment of the type-transition net (TTN) for Slack. Places (circles) are semantic types; transitions (rectangles) are API methods and data transformations. The bold path represents the solution to our running example.

Input: Semantic library $\hat{\Lambda}$ query type $\hat{s}$
Output: Set of candidate solutions $\overline{\mathcal{E}}$

1: **function** Synthesize($\hat{\Lambda}$, $\hat{s}$)
2: $\quad\mathcal{N} \leftarrow$ BuildTTN($\hat{\Lambda}$)
3: $\quad I, F \leftarrow$ PlaceTokens($\hat{s}$)
4: $\quad$**for** $\pi \in$ Paths($\mathcal{N}, I, F$) **do**
5: $\quad\quad$**for** $\mathcal{E} \in$ Progs(π) **do**
6: $\quad\quad\quad$**yield** Lift($\hat{\Lambda}, \hat{s}, \mathcal{E}$)

Figure 10. Synthesis algorithm

tokens) to produce an extra token in `Channel`. Next, we fire filter$_{\texttt{Channel.name}}$, which consumes the two tokens in `Channel` and `Channel.name`, and produces a single token in `Channel`. The remaining five transitions on the bold path simply move this one token along until it reaches `Profile.email`.

Like in prior work [8, 13], a path is only considered valid if the final state contains exactly one token in the output type (and no tokens in any other types); this condition ensures that the generated programs use all their inputs.

Synthesis algorithm. APIphany's top-level synthesis algorithm is depicted in Fig. 10. The algorithm first constructs a TTN $\mathcal{N}$ and encodes the query type $\hat{s}$ as an initial and final token placement, I and F; it then enumerates all paths from I to F in $\mathcal{N}$ in the order of length (until timeout). For each path π, the algorithm iterates over the corresponding array-oblivious programs $\mathcal{E}$ and *lifts* them into well-typed λ_A programs. The reason π might yield multiple programs is that the TTN does not distinguish different arguments of the same type, and hence we must try all their combinations.

Because TTN construction and search for valid paths is similar to prior work, we omit their detailed description and refer an interested reader to our technical report [12].

One difference worth mentioning, however, is that we use an *integer linear programming* (ILP) solver to find paths in the TTN, unlike prior approaches, which relied on SAT/SMT solvers. We found that although both solvers are equally quick at finding *one valid path*, when it comes to computing *all valid paths* of a given length, the ILP solver is much more efficient, as it has native support for enumerating multiple solutions.

Lifting array-oblivious programs. The function Progs(π) (line 5 in Fig. 10) converts a TTN path π into a set of array-oblivious programs in A-Normal Form (ANF). Fig. 11 (left) shows the full array-oblivious program extracted from the bold path in Fig. 9. As you can see from this example, array-oblivious programs can be ill-typed: for example, the projection x_1`.name` in line 4 does not type-check since x_1 actually

```
1  \channel_name →                    \channel_name →
2    let x1 = c_list({});               let x1 = c_list({});
3                                       x1' ← x1;
4    let x2 = x1.name;                  let x2 = x1'.name;
5    if x2 = channel_name;              if x2 = channel_name;
6    let x3 = x1.id;                    let x3 = x1'.id;
7    let x4 = c_members(channel=x3);    let x4 = c_members(channel=x3);
8                                       x4' ← x4;
9    let x5 = u_info(user=x4);          let x5 = u_info(user=x4');
10   let x6 = x5.profile;               let x6 = x5.profile;
11   let x7 = x6.email;                 let x7 = x6.email;
12                                      let x7' = return x7
13   x7                                 x7'
```

Figure 11. Array-oblivious program built from the bold path in Fig. 9 (left) and its lifted version (right).

has an array type `[Channel]`. What we really want this program to do is to project `name` (and execute the remaining steps in the program) *for each* channel in x_1. This can be accomplished by inserting a monadic binding $x_1' \leftarrow x_1$ and using x_1' instead of x_1 in line 4 (and elsewhere in the program where a non-array version of x_1 is required, such as line 6). We refer to this process of repairing type errors by inserting monadic bindings and **return**s as *lifting*.[8]

The function Lift (line 6 in Fig. 10) takes as input a semantic library $\hat{\Lambda}$, a query type $\hat{s}$, and an array-oblivious program $\mathcal{E}$, and produces a program $\mathcal{E}'$ that is well-typed at $\hat{s}$. Fig. 11 (right) depicts the result of lifting the program in Fig. 11 (left) to the query type `Channel.name` $\rightarrow$ `[Profile.email]` with $\hat{\Lambda}$ from Fig. 7. The full definition of lifting can be found in the technical report [12]. Informally, lifting type-checks the program "line by line", and whenever it encounters a type mismatch (in a projection, guard, or a method argument), it inserts the appropriate number of monadic bindings or

[8]A reader familiar with monads might think of the array-oblivious program as written in the identity monad instead of the list monad, and lifting as lifting the program back into the list monad.

`returns` in order to fix the mismatch. This is always possible because the only kind of type mismatch we can encounter is between an actual type $[..[\hat{t}]..]$ and the expected type $\hat{t}$, or vice versa. One thing worth noting is that we assume that the top-level return type of the program is an array type: since the lifted programs have top-level monadic bindings, they can only return arrays. If the user requests a scalar return type, we take this into account at the ranking stage by prioritizing programs that always return singleton arrays.

Completeness. Strictly speaking, array-oblivious search is incomplete: there are multiple programs that map to the same array-oblivious program, but lifting only returns a single, canonical representative. For example, consider the program in Fig. 11 (right), where we iterate over the array `x1` only once (line 3), and reuse the same "iterator" variable `x1'` in lines 4 and 6. An alternative would be to iterate over `x1` the second time before line 6, effectively retrieving names and IDs from all *pairs* of channels (instead of the name and the ID belonging to the same channel). We consider this a benign incompleteness because it is much less likely that the user intended to loop twice over the same array. If they did, we believe they would be able to repair the program by hand, as we discuss in Sec. 7.4.

6 Ranking

As we mentioned in Sec. 2, the algorithm Synthesize may generate hundreds or even thousands of well-typed candidate solutions, most of which, however, are uninteresting. We now formalize how APIphany ranks these candidates with the help of *retrospective execution* (RE).

Cost computation. To rank the programs, we assign them a positive cost, and then order them from lowest to highest cost. To compute the cost of a program $\mathcal{E}$, we retrospectively execute it multiple times, accumulating execution results in a set *res*; retrospective execution is non-deterministic, and executing a program more times lead to more precise cost estimates. We then compute the cost of $\mathcal{E}$ based on its result set *res* and the return type $\hat{t}$ of the query as follows:

1. The base cost is the size of $\mathcal{E}$ in AST nodes.
2. If $res = \emptyset$ (all executions have failed), the candidate receives a large penalty.
3. If $res = \{[]\}$ (all executions return an empty array), the candidate receives a medium penalty.
4. Finally, we compare the values $v \in res$ with the desired result type $\hat{t}$; recall that λ_A programs always return an array, while $\hat{t}$ might or might not be an array type. We assign a small penalty for a *multiplicity mismatch*, *i.e.* if either $\hat{t}$ is a scalar type and *any* value v has more than one element, or $\hat{t}$ is an array type and *all* values v have a single element.

Retrospective Execution $\boxed{\langle \mathcal{W}; \Gamma; \Sigma \mid e \rangle \Rightarrow v}$

$$\textsc{E-If-True-L}\ \frac{x_1 \in \Sigma \quad x_2 \notin \Sigma \quad \Sigma(x_1) = v_1 \qquad \langle \mathcal{W}; \Gamma; x_2 \mapsto v_1, \Sigma \mid e \rangle \Rightarrow v}{\langle \mathcal{W}; \Gamma; \Sigma \mid \texttt{if}\ x_1 = x_2; e \rangle \Rightarrow v}$$

$$\textsc{E-If-True-R}\ \frac{x_1 \notin \Sigma \quad \langle \mathcal{W}; \Gamma; \Sigma \mid x_2 \rangle \Rightarrow v_2 \qquad \langle \mathcal{W}; \Gamma; x_1 \mapsto v_2, x_2 \mapsto v_2, \Sigma \mid e \rangle \Rightarrow v}{\langle \mathcal{W}; \Gamma; \Sigma \mid \texttt{if}\ x_1 = x_2; e \rangle \Rightarrow v}$$

$$\textsc{E-Method-val}\ \frac{(f, \overline{l_i = v_i}, v_{out}) \in \mathcal{W}}{\langle \mathcal{W}; \Gamma; \Sigma \mid f(\overline{l_i = v_i}) \rangle \Rightarrow v_{out}}$$

$$\textsc{E-Method-name}\ \frac{\forall (f, \overline{l_i = v'_i}, v_{out}) \in \mathcal{W}.\ \exists i : v'_i \neq v_i \qquad (f, \overline{l_i = v'_i}, v_{out}) \in \mathcal{W}}{\langle \mathcal{W}; \Gamma; \Sigma \mid f(\overline{l_i = v_i}) \rangle \Rightarrow v_{out}}$$

Figure 12. Retrospective execution.

Retrospective execution. We formalize RE as a judgement $\langle \mathcal{W}; \Gamma; \Sigma \mid e \rangle \Rightarrow v$, stating that v is a valid result for executing the expression e in the environment Σ (which maps variables to values). The judgment is also parameterized by a type context Γ and witness set $\mathcal{W}$, used to replay method calls and sample program inputs. To run a candidate solution $\mathcal{E}$, we execute its body in an *empty environment* $\Sigma = \cdot$ and with Γ storing the types of $\mathcal{E}$'s arguments. As we explain in more detail below, program inputs are selected lazily, during execution, in order to maximize its chances of producing meaningful results.

Replaying method calls. Most of the rules for the RE judgement describe standard big-step operational semantics (they can be found in the technical report [12]), but two groups of rules, shown in Fig. 12, deserve more attention. The first group of interest includes E-Method-Val and E-Method-Name, which replay a method call by looking it up in $\mathcal{W}$. The rule E-Method-Val applies when $\mathcal{W}$ contains an exact match for the current call, *i.e.* we have previously observed a call to the same method, with the same parameter names and parameter values. The rule E-Method-Name applies when an exact match cannot be found (see first premise); in this case we pick an approximate match, where only the method name and parameter names match. Matching parameter names is important because many REST API methods admit optional parameters, and behave very differently based on which pattern of optional parameters is provided. If an approximate match cannot be found either, RE fails. Note that for a given call $f(\overline{l_i = v_i})$, there might be multiple approximate matches in $\mathcal{W}$, which makes RE non-deterministic (in fact, there can even be multiple precise matches because services are stateful). Due to hidden state and approximate matches, the results of RE are not guaranteed to match actual execution, but our experiments show that they are precise enough for the purposes of ranking.

Table 1. APIs used in our experiments. For each API we report the number of methods $|\Lambda.f|$, min/max number of arguments per method n_{arg}, the number of objects $|\Lambda.o|$, and min/max size of the objects s_{obj}. We also report the number of witnesses $|\mathcal{W}|$ we collected for type mining and the number of methods covered by those witnesses n_{cov}.

	API size				API Analysis	
API	$\lvert\Lambda.f\rvert$	n_{arg}	$\lvert\Lambda.o\rvert$	s_{obj}	$\lvert\mathcal{W}\rvert$	n_{cov}
Slack	174	0 - 15	79	1 - 70	3834	60
Stripe	300	0 - 145	399	1 - 66	25402	124
Square	175	0 - 20	716	1 - 34	1749	67

Lazy sampling of program inputs. The remaining two rules in Fig. 12 are responsible for choosing program inputs so as to bias guard expressions to evaluate to true. We observe that when inputs are sampled eagerly ahead of time, guard expressions almost always evaluate to false, causing RE to return an empty array; as a result, our ranking heuristic cannot distinguish meaningful candidates from those that return an empty array regardless of the input. To address this issue, we postpone adding program inputs to the environment Σ until they are used. If the first usage of a program input is in a guard, the rules E-If-True-L and E-If-True-R pick its value to make the guard true: E-If-True-L applies when only the right-hand side of a guard is undefined, and E-If-True-R applies when the left-hand side or both are undefined. If the first usage of an input is in a method call or a projection, we instead randomly sample from all values of the same type observed in $\mathcal{W}$.

7 Evaluation

We implemented APIphany in Python, except for retrospective execution, where we used Rust for performance reasons. We used the Gurobi ILP solver [14] v9.1 as the back-end for TTN search. We ran all the experiments on a machine with an Intel Core i9-10850K CPU and 32GB of memory.

We designed our empirical evaluation to answer the following research questions:

(RQ1) Can APIphany find solutions for a wide range of realistic tasks across multiple popular APIs?
(RQ2) Is type mining effective and necessary for enabling type-directed synthesis?
(RQ3) Is retrospective execution effective and necessary for prioritizing relevant synthesis results?

API selection. For our evaluation, we selected three popular REST APIs: the Slack communication platform and two online payment platforms, Stripe and Square. We selected these APIs because they are widely used and have both an OpenAPI specification and a web interface, which allowed us to set up the test environment and collect witnesses easily. As shown in Tab. 1, these APIs are quite complex: each has over a hundred methods with up to 145 arguments; all three feature optional arguments. The three APIs also contain a large number of object definitions, with up to 70 fields.

Experiment setup: type mining. Recall that type mining relies on a witness set $\mathcal{W}$. Witnesses are straightforward to collect for API owners, or when an integration test suite is publicly available; neither was the case in our setting. Instead, we collected witnesses by observing traffic from the services's web interface, and then enhancing this initial (very sparse) witness set via random testing; this process is described in more detail in our technical report [12]. As shown in Tab. 1, we collected between 1.7K and 25K witnesses per API, which covered 30–40% of all methods. It is hard to obtain full coverage for these closed source APIs as an outsider, for instance, because many methods are only available to paid accounts; our experiments show, however, that APIphany performs well with this witness set.

Benchmark selection. For each API, we extracted programming tasks from StackOverflow questions that mention this API as well as GitHub repositories that use the API. After excluding the tasks that were out of scope of our DSL, we manually translated each of the remaining tasks from a natural-language description or a code snippet into a type query, resulting in 32 benchmarks (see Tab. 2). Apart from our running example (benchmark 1.1), these include, for instance: "Send a message to a user given their email" in Slack (1.2), "Create a product and invoice a customer" in Stripe (2.3), and "Delete catalog items with given names" in Square (3.10). As noted in Tab. 2, many of these tasks are *effectful*: they require creating, modifying, or deleting objects.

Each benchmark comes with a "gold standard" solution: the accepted solution on StackOverflow or the snippet we found on GitHub. We manually translated these solutions into APIphany's DSL. As shown in the "Solution Size" portion of Tab. 2, these solutions range in complexity from 7 to 22 AST nodes, containing up to three method calls and guards and up to seven projections, which makes them nontrivial for programmers to solve manually. A complete list of tasks, type queries, and solutions can be found in [12].

Experiment setup: program synthesis. For each of the 32 benchmarks, we ran the synthesizer with a timeout of 150 seconds. For each new candidate generated, we estimated its cost using 15 rounds of RE and recorded the synthesis time (including both TTN search and RE time). After the timeout, we checked whether the gold standard solution appears among the generated candidates and compared its RE-based rank vs the original rank at which it was generated (based on path length). Below we report average time and median rank over three runs to reduce the impact of randomness.

7.1 RQ1: Overall Effectiveness

The last four columns of Tab. 2 detail APIphany's performance on the 32 synthesis benchmarks. APIphany finds the

Table 2. Synthesis benchmarks and results. Benchmarks marked with † are effectful. For each benchmark we report the size of the desired solution: AST, n_f, n_p and n_g correspond to number of AST nodes, method calls, projections and guards, respectively. We also report the time to find the correct solution (in seconds), its rank without RE (r_{orig}), and the lower and upper bound on its rank with RE (r_{RE} and r_{RE}^{TO}). '-' means no solution is found in 150 seconds.

API	ID	Solution Size				Time	Rank		
		AST	n_f	n_p	n_g	(sec)	r_{orig}	r_{RE}	r_{RE}^{TO}
Slack	1.1	17	3	6	1	83.5	25230	5	5
	1.2†	12	3	5	0	5.6	2224	10	10
	1.3	16	3	7	0	-	-	-	-
	1.4	14	2	4	1	1.3	489	24	31
	1.5†	10	2	3	0	3.4	788	5	5
	1.6†	9	2	2	0	1.7	573	8	19
	1.7†	12	2	4	1	1.3	757	8	9
	1.8	9	2	3	0	42.0	16438	29	30
Stripe	2.1†	9	2	2	0	95.4	4952	3	3
	2.2†	10	2	2	0	92.4	4854	4	4
	2.3†	12	3	2	0	121.2	6363	1	1
	2.4	8	1	2	1	0.5	3	1	1
	2.5	8	2	2	0	1.0	10	4	4
	2.6†	9	3	2	0	12.2	270	3	3
	2.7	5	1	2	0	0.6	4	2	2
	2.8	16	2	7	1	20.2	679	17	17
	2.9	6	1	2	0	0.5	2	1	1
	2.10†	10	2	3	0	7.8	187	6	6
	2.11†	7	2	1	0	17.2	490	6	6
	2.12†	11	3	2	0	-	-	-	-
	2.13†	10	3	2	0	-	-	-	-
Square	3.1	4	1	1	0	0.2	2	1	1
	3.2	16	1	4	3	0.5	10	4	4
	3.3	10	1	3	1	0.4	6	1	1
	3.4	5	1	2	0	0.7	2	1	1
	3.5†	14	2	3	0	2.2	99	2	2
	3.6	5	1	2	0	0.2	1	1	1
	3.7	6	1	2	0	0.3	7	4	4
	3.8	9	1	3	0	0.7	1	1	1
	3.9	8	1	2	1	0.2	3	2	2
	3.10†	16	2	5	1	1.9	174	10	12
	3.11†	8	2	3	0	1.0	68	16	16

correct solution for 29 benchmarks. The remaining three benchmarks fail with a timeout because their type queries are too ambiguous; for example, in benchmark 1.3 ("Get unread messages of a user") the type query has no means to specify that we are only interested in *unread* messages; as a result, the solution is drowned among thousands of other programs that map a user ID to messages.

We plot the number of benchmarks solved as a function of time (including RE) in Fig. 13. As the plot shows, majority of benchmarks (19/32) can be solved within five seconds. On average APIPHANY takes 17.8 seconds to find the desired solution (median time 1.3 seconds).

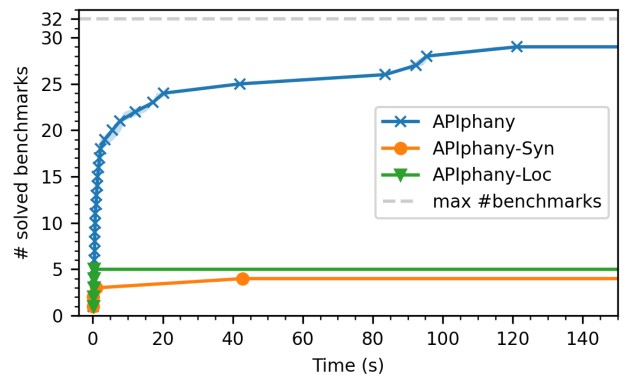

Figure 13. Comparison of synthesis performance between APIPHANY and its two variants that do not use type mining.

Takeaway: APIPHANY is able to solve 91% of tasks from three real-world APIs.

7.2 RQ2: Type Mining

Recall that type mining involves replacing primitive *syntactic types* in the spec with unique *location-based types*, and then merging those based on the witness set to obtain *semantic types*. The merging process is not perfect: it might *fail to merge* two location that should have the same type because the witness set lacks evidence to justify the merge; or it might *spuriously merge* two locations if they share a value by chance. It is hard to measure the accuracy of inferred types directly, since we do not have an oracle for semantic types. Instead, we evaluate type mining indirectly in two ways: 1) we run an *ablation study* to measure its impact on the overall performance of the synthesizer, and 2) we perform a small-scale *qualitative analysis* of inferred types.

Ablation study. For this experiment, we compare the performance of APIPHANY and its two variants: (a) APIPHANY-SYN, which builds the TTN directly from syntactic types, and (b) APIPHANY-LOC, which builds the TTN from (unmerged) location-based types. We plot the number of benchmarks solved by each variant as the function of time in Fig. 13.

As expected, both variants perform poorly: APIPHANY-SYN only solves 4/32 benchmarks and APIPHANY-LOC solves 5. All these benchmarks are "easy" (solved by APIPHANY in under a second). Intuitively, the two variants represent two extremes in terms of type *granularity*. Syntactic types are *too coarse-grained* (all `String` locations have the same type), which leads TTN search to return too many well-typed candidates. As a result, APIPHANY-SYN struggles to solve all but the simplest tasks, with many benchmarks running out of memory. Location-based types, on the other hand, are *too fine-grained* (each `String` location has a unique type), which leads to most desired solutions simply being ill-typed, because there is no way for one method to use values returned by another. The solutions to all of the five benchmarks solved by APIPHANY-LOC have only one method call with no parameters, followed by several projections or filters.

As you can see from Fig. 13, APIPHANY drastically outperforms both variants. This result indicates that type mining strikes a good balance between coarse- and fine-grained types: all 32 benchmarks have a well-typed solution in terms of the mined types, and APIPHANY is able to find most of them within a reasonable time.

Qualitative analysis. To give a more direct account of the quality of inferred semantic types, we randomly sampled five methods from each API (among the methods covered by the collected witnesses), and manually inspected the inferred types to check if they match our expectations. More specifically, for each `String` location in a method spec, we pick a location type loc^*, which we deem most natural for a programmer to use in a type query (for example, for the parameter to `users_info`, loc^* = `User.id`); we consider the inferred loc-set type sufficient if it contains loc^*. The detailed results appear in the technical report [12].

In the methods we examined, type mining was able to infer a sufficient semantic type for all responses, required parameters, and about half of optional parameters. The remaining optional parameters were assigned unmerged location types, because they were never used in our witness set. This is almost unavoidable, because of the sheer number of obscure optional parameters in real-world APIs (which, fortunately, are rarely needed to solve programmer's tasks).

Recall that the other failure mode of type mining is spuriously merging unrelated locations. We did not observe any spurious merges among the randomly sampled methods, but anecdotally we did encounter one such merge elsewhere in the Slack API: between `Channel.name` and `Message.name`. Note that spurious merges might slow down the search and produce some "semantically ill-typed" solutions, but they do not prevent APIPHANY from finding the desired solution.

Takeaway: Type mining increases the percentage of solved benchmarks from 12% to 91%.

7.3 RQ3: Ranking

To measure the effectiveness of RE-based ranking, we compare the last three columns of Tab. 2: r_{orig} denotes the rank of the desired solution in the order it was generated by TTN search (which is based on path length, and hence correlated with solution size); r_{RE} denotes the RE-based rank of the solution at the time it was generated, and r_{RE}^{TO} denotes its RE-based rank by the timeout (which can be lower than r_{RE} as other candidates generated later might end up being ranked higher). We report both of these RE-based ranks because we envision an APIPHANY user inspecting the candidate solutions some time between they are generated and the timeout,

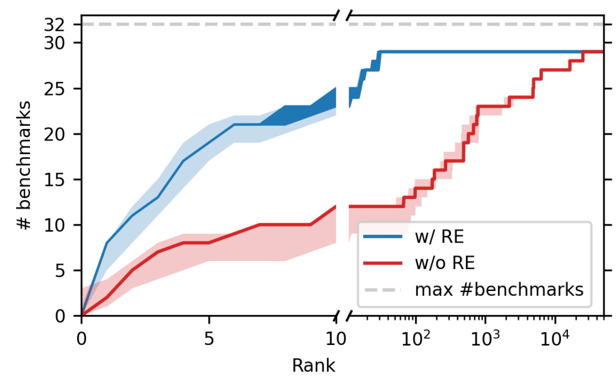

Figure 14. Number of benchmarks whose solution is reported within a given rank. The filled blue area is the range of ranks one might get depending on when they inspect the candidates. The shaded area is the 95% confidence interval.

and hence the relevant rank value is between r_{RE} and r_{RE}^{TO}. We plot the number of benchmarks whose solutions lie at or below each rank in Fig. 14, with the range between r_{RE} and r_{RE}^{TO} represented as a filled area.

As you can see from Fig. 14, RE-based ranking significantly increases the chances that the desired solution makes the short-list of candidates. In particular, *without RE-based ranking* only 8 benchmarks (28% of solved) return the correct solution in top five, and only 12 (41%) return it in top ten; in contrast, *with RE-based ranking*, 19 (65%) benchmarks return the correct solution in top five (after timeout), and 23 (79%) in top ten. Moreover, as we can see from Tab. 2, the solution's rank never gets worse after RE, in all but two cases it strictly improves, and for all long-running benchmarks it improves drastically (the average rank improves from 2230.5 to 7.0).

A closer look at the six benchmarks that do not land in top ten after RE reveals two main reasons for these suboptimal rankings. In most cases the solution is simply large, and there are many smaller candidates that are still meaningful. For example, the query "Delete all catalog items" (3.11) takes no arguments and returns an array of all deleted items; there are many valid and simple ways to construct an array of catalog items without deleting them. In a few cases, APIphany fails to throw out meaningless programs due to the imprecision of retrospective execution. For example, in 1.6 it reports a solution that posts an update to a given channel with a given timestamp, even though this timestamp might be invalid for this channel; APIphany instead thinks that this call always succeeds by relying on approximate matches during retrospective execution.

We also recorded the time APIPHANY takes to compute the cost for all generated candidates (which involves executing each candidate 15 times). Although APIPHANY generates thousands of well-typed candidates for most benchmarks, cost computation only takes about 1% of total synthesis time.

Takeaway: RE-based ranking takes a negligible amount of time and increases the percentage of correct solutions reported in top ten from 41% to 79%.

7.4 Discussion and Limitations

Witness generation. One threat to validity of our evaluation is that the results of type minings (and therefore synthesis) depend heavily on the witness set. In particular, if our benchmarks required methods that are not covered by the witness set, APIPHANY most likely would not be able to solve them, since they would be ill-typed with inferred semantic types. We ran our experiments using a particular witness set, which we collected using one methodology (described in the technical report [12]); our findings might not generalize to using APIPHANY with witness sets collected by other means.

Effectful methods. We observe that effectful methods in REST APIs have an interesting property: they make the effect explicit in their response. For example, the method for posting a message on SLACK also returns the message object, and the method for deleting a catalog item in SQUARE returns the ID of the deleted item (instead of just returning `void`). This property makes REST APIs particularly suitable for type-directed synthesis and expressing user intent with types: for example, the query "Send a message to a user with a given email" can be expressed as the type `Profile.email` ⟶ `Message` instead of a much less informative type `Profile.email` ⟶ `void`. The downside, of course, is that the return type of an effectful method might not be obvious to the user (for example, does deleting a catalog item return an object or its ID?) One way to overcome this limitation is to let the user specify the name of the last method they want to call (*e.g.* `catalog_object_delete`) instead of the output type; this kind of specification is straightforward to integrate into TTN search.

DSL restrictions. In our search for benchmarks, we encountered (very few) snippets that were inexpressible in our DSL because they required functional transformations on primitive values, as opposed to just structural transformations on objects and arrays, for example: "Get all members of a channel and *concatenate* them together". We consider such functional transformations beyond the scope of APIPHANY because its type-based specifications are too coarse to distinguish between different functional transformations. This is also the reasoning behind our design decision to only support equality inside guards, as opposed to more general predicates: if the specification cannot distinguish between, say, $=$ and $\leq$, there is little use in generating programs with both. More generally, we view programs synthesized by APIPHANY as a starting point, which helps the programmer figure out how to plumb data through a set of API calls; we envision the user building on top of those programs to add functional modifications and more expressive predicates. This interaction model motivates both our DSL restrictions and our type-based specifications.

Value-based location merging. Value-based merging works well for strings, since their large domain makes it unlikely that two `String` locations share a value by chance. It works less well for other primitive types, such as integers and booleans. To reduce the risk of spurious merges, our implementation performs value-based merging only for strings and large integers (> 1000), but not for booleans or small integers. In the future, we plan to investigate more sophisticated approaches to location merging. One idea is to use probabilistic reasoning to estimate the likelihood of two locations having the same type based on (1) how common a value is across locations and (2) what proportion of values is shared between the two locations. Another approach is to cluster locations using NLP techniques, such as sentiment analysis of object and field names, as well as documentation.

User interface. Another important direction for future work is to investigate usable ways of specifying semantic type queries and comprehending synthesis results. In particular, existing work from the HCI community [9, 10] might help users quickly explore a large space of related candidate solutions, thereby mitigating the limitations of ranking.

8 Related Work

APIPHANY is a component-based synthesizer and primarily compares with related work in this space. It also draws on techniques from specification mining and type inference.

Type-directed component-based synthesis. The goal of component-based synthesis is to find a *composition* of components (library functions) that implements a given task. In *type-directed* component-based synthesis both the task and the components are specified using types. The traditional approach to this problem based on proof search [3, 16, 25] scales poorly with the size of the component library. An alternative, more scalable *graph-based approach* was introduced in PROSPECTOR [22] for unary components, and generalized to *n*-ary components in SYPET [8], by replacing graphs with Petri nets. TYGAR [13] further extends SYPET's search to polymorphic components using the idea of *abstract types*, which are inspired by *succinct types* from another component-based synthesizer, INSYNTH [15]. APIPHANY's program search phase is using the Petri net encoding from SYPET and TYGAR with minor adaptations (support for optional arguments and ILP encoding). Our array-oblivious encoding is related to abstract and succinct types in that it helps make the Petri net smaller, but it is also substantially different in that, unlike prior work, it can efficiently encode a certain class of higher-order programs (array comprehensions) into the Petri net.

API navigation. Beyond type-directed synthesis, other work focuses on smart auto-completion [21, 26, 28] but relies on static analysis and mining client code, which APIphany does not require. Among tools that leverage dynamic analysis, EdSynth [35] uses test executions to generate snippets that involve both API calls and control structures. MatchMaker [37] and DemoMatch [36] are similar to APIphany in that they rely on observed program traces to suggest code that uses complex APIs (the former from types and the latter from demonstrations). All these techniques work in the context of Java, and hence assume that sufficiently precise types are already present.

SQL synthesis. The problem of generating projections and filters is related to synthesis of SQL queries [33, 34]. Existing SQL synthesis techniques are not directly applicable to our problem domain, because (1) our programs also contain arbitrary API method invocations, and (2) we manipulate semi-structured data instead of relational data.

API discovery and specification mining. A complimentary approach to API navigation using program synthesis is to infer specifications [1, 24, 29] or example usages [4, 6, 18] to help the user understand the API better. APIphany's type mining is inspired by Ammons et al. [1], where they build probabilistic finite state automata representing data and temporal dependencies between API methods. APIphany implements a simpler form of their algorithm, which discovers data flows (but not temporal dependencies), but the novelty lies in using this information to drive program synthesis.

Type mining is also related to prior work on inferring type annotations for dynamically typed languages from executions [2, 5, 7]. However, this work is for structural types, whereas we infer domain-specific nominal types.

Simulated execution. An alternative to our retrospective execution is to synthesize a *model* of the API, and evaluate program candidates against that model. Previous work [17, 20] synthesizes models for complex frameworks and opaque code; our retrospective execution is simpler: it skips the extra step of model synthesis.

Ranking solutions. Specifications in program synthesis are often ambiguous, so synthesizers have to rank their candidate solutions and return the top result(s). Existing tools most commonly rely on hand-crafted [11] or learned [15, 28, 30] ranking functions based on syntactic features of generated programs. Hoogle+ [19] is most similar to APIphany in that it ranks programs based on the results of their *execution*, using heuristics like whether the program always fails, and how similar it is to other candidates.

Acknowledgments

The authors would like to thank the anonymous reviewers, our shepherd Yuepeng Wang, as well as Hila Peleg and Ilya Sergey for their valuable feedback on earlier drafts of this paper. This work was supported by the National Science Foundation under Grants No. 1943623, 1911149, and 2107397.

References

[1] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 4–16. https://doi.org/10.1145/503272.503275

[2] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic inference of static types for ruby. In *POPL. Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 459–472.

[3] Lennart Augusstson. 2005. Djinn. https://github.com/augustss/djinn.

[4] Celeste Barnaby, Koushik Sen, Tianyi Zhang, Elena Glassman, and Satish Chandra. 2020. Exempla Gratis (E.G.): Code Examples for Free. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1353–1364. https://doi.org/10.1145/3368089.3417052

[5] Ambrose Bonnaire-Sergeant. 2019. *Typed Clojure in Theory and Practice*. Ph.D. Dissertation. Indiana University, Bloomington.

[6] Raymond P. L. Buse and Westley Weimer. 2012. Synthesizing API Usage Examples. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 782–792.

[7] Ravi Chugh, Sorin Lerner, and Ranjit Jhala. 2011. Type Inference with Run-time Logs. In *Workshop on Scripts to Programs (STOP)*.

[8] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. In *POPL*.

[9] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Trans. Comput.-Hum. Interact.* 22, 2, Article 7 (mar 2015), 35 pages. https://doi.org/10.1145/2699751

[10] Elena L. Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. 2018. *Visualizing API Usage Examples at Scale*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3174154

[11] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 317–330. https://doi.org/10.1145/1926385.1926423

[12] Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. 2022. Type-Directed Program Synthesis for RESTful APIs (Technical Report). *arXiv preprint arXiv:2203.16697* (2022). https://arxiv.org/abs/2203.16697

[13] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2020. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.* 4, POPL (2020), 12:1–12:28.

[14] Gurobi Optimization, LLC. 2021. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[15] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *PLDI*.

[16] George T. Heineman, Jan Bessai, Boris Düdder, and Jakob Rehof. 2016. A Long and Winding Road Towards Modular Synthesis. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*. 303–317. https://doi.org/10.1007/978-3-319-47166-2_21

[17] Stefan Heule, Manu Sridharan, and Satish Chandra. 2015. Mimic: Computing Models for Opaque Code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy)

(ESEC/FSE 2015). Association for Computing Machinery, New York, NY, USA, 710–720. https://doi.org/10.1145/2786805.2786875

[18] Abbas Heydarnoori, Krzysztof Czarnecki, and Thiago Tonelli Bartolomei. 2009. Supporting Framework Use via Automatically Extracted Concept-Implementation Templates. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming* (Italy) *(Genoa)*. Springer-Verlag, Berlin, Heidelberg, 344–368. https://doi.org/10.1007/978-3-642-03013-0_16

[19] Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. 2020. Digging for fold: synthesis-aided API discovery for Haskell. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 205:1–205:27.

[20] Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. 2016. Synthesizing Framework Models for Symbolic Execution. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 156–167. https://doi.org/10.1145/2884781.2884856

[21] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code Recommendation via Structural Code Search. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 152 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360578

[22] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *PLDI*.

[23] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 706. https://doi.org/10.1145/1142473.1142552

[24] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-Based Semantic Code Search over Partial Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 997–1016. https://doi.org/10.1145/2384616.2384689

[25] Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. 230–266. https://doi.org/10.1007/978-3-642-04652-0_5

[26] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. 2012. Type-directed completion of partial expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 275–286. https://doi.org/10.1145/2254064.2254098

[27] Inc. Postman. 2020. 2020 State of The API Report. https://www.postman.com/state-of-api/api-technologies/.

[28] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. *SIGPLAN Not.* 49, 6 (June 2014), 419–428. https://doi.org/10.1145/2666356.2594321

[29] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. 2007. Static specification mining using automata-based abstractions. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*. 174–184.

[30] Rishabh Singh and Sumit Gulwani. 2015. Predicting a Correct Program in Programming by Example. In *CAV - 27th International Conference, 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 398–414.

[31] Inc. SmartBear. 2020. The State of API Report 2020. https://static1.smartbear.co/smartbearbrand/media/pdf/smartbear_state_of_api_2020.pdf.

[32] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (April 1975), 215–225. https://doi.org/10.1145/321879.321884

[33] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

[34] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 63 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133887

[35] Z. Yang, J. Hua, K. Wang, and S. Khurshid. 2018. EdSynth: Synthesizing API Sequences with Conditionals and Loops. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 161–171. https://doi.org/10.1109/ICST.2018.00025

[36] Kuat Yessenov, Ivan Kuraj, and Armando Solar-Lezama. 2017. DemoMatch: API Discovery from Demonstrations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 64–78. https://doi.org/10.1145/3062341.3062386

[37] Kuat Yessenov, Zhilei Xu, and Armando Solar-Lezama. 2011. Data-Driven Synthesis for Object-Oriented Frameworks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 65–82. https://doi.org/10.1145/2048066.2048075