

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*

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

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

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9265-5/22/06.

<https://doi.org/10.1145/3519939.3523450>

(PLDI '22), June 13–17, 2022, San Diego, CA, USA. ACM, New York, NY, USA, 27 pages. <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 [26, 30], 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, 14, 18, 21] 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

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

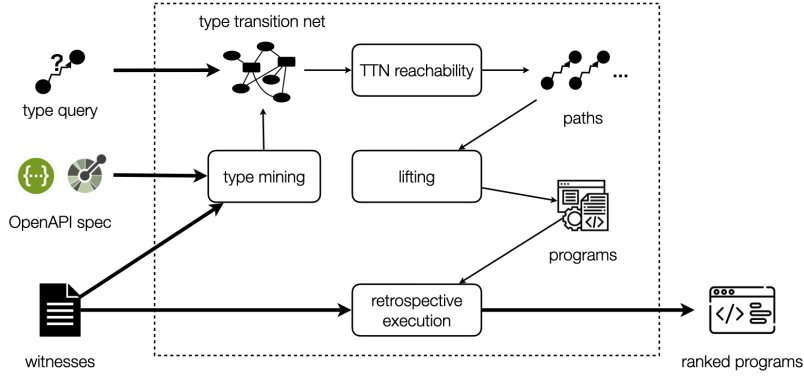


Figure 1. Overview of APIPHANY

APIs, because it allows developers to start with information 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.
3. Ranking synthesis results with the help of *retrospective execution* (Sec. 6), a type of simulated execution using

```

1 \channel_name → {
2   c ← conversations_list()
3   if c.name = channel_name
4     uid ← conversations_members(channel=c.id)
5     let u = users_info(user=uid)
6     return u.profile.email
7 }

```

Figure 2. Solution for retrieving all member emails from a SLACK channel in APIPHANY DSL.

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 format² 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. 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 presents a challenge to existing synthesis techniques because our candidate programs have *both* a large component library

²<https://swagger.io/>. APIPHANY supports both OpenAPI v2 and v3.

³SLACK OpenAPI spec is available at: https://raw.githubusercontent.com/slackapi/slack-api-specs/master/web-api/slack_web_openapi_v2.json

```

{
  "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.

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, 12]; 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` [22]. 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 \leftarrow). 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 \leftarrow 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

$o ::=$	User Channel ...	object names
$f ::=$	u_info ...	method names
$l ::=$	in out 0 id name ...	field labels
$\ell ::=$	$l \mid ?l$	record fields
$loc ::=$	$o.\bar{l} \mid f.\bar{l}$	locations
Terms		
$e ::=$	$x \mid e.l$	Expressions variable, projection
	$f(\overline{l_i = e_i}) \mid \mathbf{let} \ x = e; e$	method call, pure binding
	$\mathbf{if} \ e = e; e \mid x \leftarrow e; e$	guard, monadic binding
	$\mathbf{return} \ e$	pure value lifting
$\mathcal{E} ::=$	$\lambda \bar{x}. e$	Top Level Programs
Values		
$v ::=$	"..." $[\bar{v}] \mid \{\overline{l_i = v_i}\}$	strings, arrays, objects
Types		
$t ::=$	String	Syntactic types strings
	$o \mid [t] \mid \{\overline{\ell_i : t_i}\}$	named objects, arrays, records
$s ::=$	$t \rightarrow t$	function types
$\hat{t} ::=$	$\{\overline{loc}\}$	Semantic types loc-sets
	$o \mid [\hat{t}] \mid \{\overline{\ell_i : \hat{t}_i}\}$	named objects, arrays, records
$\hat{s} ::=$	$\hat{t} \rightarrow \hat{t}$	function types
Libraries		
$\Lambda ::=$	$\overline{o : t}; \overline{f : s}$	object and method definitions
$\hat{\Lambda} ::=$	$\overline{o : \hat{t}}; \overline{f : \hat{s}}$	semantic definitions

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 $\{\ell_i : t_i\}$.⁵ Records are mappings from field labels to types; some fields are optional, indicated with a ? before its label. For example, the record type $\{\text{id} : \text{String}, ?\text{time_zone} : \text{String}\}$, has a required field `id` and an optional field `time_zone`.

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

⁵We write \bar{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 $t \rightarrow t$, 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 = e1; e2` binds x to the entire result of e_1 and then evaluates e_2 . The guard expression `if e1 = e2; 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 \bar{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 $\Lambda \vdash loc \Rightarrow \hat{t}$, 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:

- $\Lambda \vdash u_info.out \Rightarrow User$ because this location is annotated with a named object type.

- $\Lambda \vdash c_members.out \Rightarrow [\{c_members.out.0\}]$ because array types do not themselves get replaced with loc-sets; instead, we recursively assign a location-based type to an array’s element.
- $\Lambda \vdash u_info.out.id \Rightarrow \{User.id\}$ 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, $\Lambda \vdash u_info.out \Rightarrow User$, and then follow the field `id` of the `User` object.

The formal definition of location-based type assignment is mostly straightforward and relegated to Appendix A (Fig. 15).

Merging types via a disjoint-set. Type mining relies on a variant of the *disjoint-set* data structure (also known as *union-find* [31]). 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 $\{loc, loc_1, \dots\}$ that contains all locations in that group.

Type mining algorithm. Fig. 8 presents the top-level algorithm `MINE TYPES`, 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). `MINE TYPES` 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 `ADD WITNESS`, 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, `ADD WITNESS` iterates over all channel objects in the array, and over all fields of each channel object; once it reaches the value `"UJ5RHEG45"`, it computes the type of its location as $\Lambda \vdash c_list.out.0.creator \Rightarrow \{Channel.creator\}$, 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 `ADD DEFINITIONS` to iterate over all object and method definitions in Λ , and

⁶In 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	<pre>Channel: { id: String, name: String, creator: String } User: { id: String, name: String, profile: Profile }</pre>	<pre>Channel: { id: Channel.id, name: Channel.name, creator: User.id } User: { id: User.id, name: User.name, profile: Profile }</pre>
Methods	<pre>c_list: {} → [Channel] u_info: {user: String} → User c_members: {channel: String} → [String]</pre>	<pre>c_list: {} → [Channel] u_info: {user: User.id} → User c_members: {channel: Channel.id} → [User.id]</pre>

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.

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 → [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 Appendix B (Fig. 16).

Type transition nets. To efficiently enumerate well-typed programs we follow prior work [8, 12] 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

⁷Here 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.

Input: A library Λ and witnesses \mathcal{W}

Output: A semantic library $\hat{\Lambda}$

```

1: function MINETYPES( $\Lambda, \mathcal{W}$ )
2:    $DS \leftarrow$  empty disjoint-set
3:   for  $\langle f, v_{in}, v_{out} \rangle \in \mathcal{W}$  do
4:     ADDWITNESS( $DS, f, in, v_{in}$ )
5:     ADDWITNESS( $DS, f, out, v_{out}$ )
6:    $\hat{\Lambda} \leftarrow$  ADDDEFINITIONS( $\Lambda, DS$ )
7:   return  $\hat{\Lambda}$ 

8: function ADDWITNESS( $DS, loc, v$ )
9:   match  $v$ 
10:  case "...":
11:     $\Lambda \vdash loc \implies \{loc'\}$ 
12:     $DS \leftarrow$  insert( $DS, loc', v$ )
13:  case  $\overline{v_i}$ :
14:    forall  $i : \text{ADDWITNESS}(DS, loc.0, v_i)$ 
15:  case  $\{l_i = v_i\}$ :
16:    forall  $i : \text{ADDWITNESS}(DS, loc.l_i, v_i)$ 
```

Figure 8. Type mining algorithm.

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. `projUser.profile` and `projProfile.email`) and guards (e.g. `filterChannel.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 HOOGL+ [12] 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

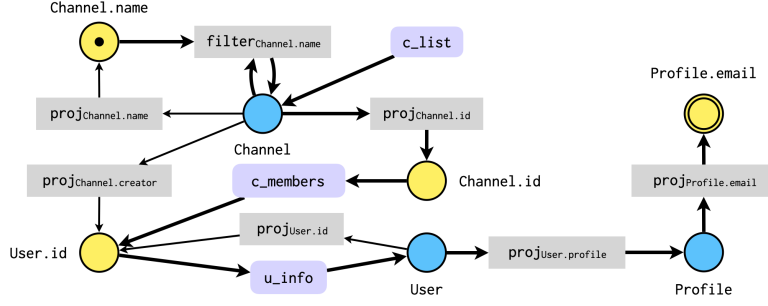


Figure 9. A fragment of the type-transition net (TTN) for SLACK. Places (circles) are semantic types; transitions (rectangles) are API methods and data transformations. The bold path represents the solution to our running example.

tokens) to produce an extra token in Channel. Next, we fire $\text{filter}_{\text{Channel.name}}$, 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, 12], 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 Appendix B. 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 $\text{PROGS}(\pi)$ (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 $x_1.\text{name}$ in line 4 does not type-check since x_1 actually has an array type $[\text{Channel}]$. What we really want this program to do is to project *name* (and execute the remaining

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

```

1: function SYNTHESIZE( $\hat{\Lambda}, \hat{s}$ )
2:    $\mathcal{N} \leftarrow \text{BUILDTTN}(\hat{\Lambda})$ 
3:    $I, F \leftarrow \text{PLACETOKENS}(\hat{s})$ 
4:   for  $\pi \in \text{PATHS}(\mathcal{N}, I, F)$  do
5:     for  $\mathcal{E} \in \text{PROGS}(\pi)$  do
6:       yield LIFT( $\hat{\Lambda}, \hat{s}, \mathcal{E}$ )

```

Figure 10. Synthesis algorithm

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

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

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 **returns** 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 $\text{Channel.name} \rightarrow [\text{Profile.email}]$ with $\hat{\Lambda}$ from Fig. 7. The full definition of lifting can be found in Appendix B (Fig. 18). 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 **returns** in order to fix the mismatch. This is always possible because the only kind of type mismatch we can encounter is

⁸A 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.

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 x_1 only once (line 3), and reuse the same “iterator” variable x_1 in lines 4 and 6. An alternative would be to iterate over x_1 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 $res = \{\emptyset\}$ (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. 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

$$\begin{array}{c}
 \text{Retrospective Execution} \quad \boxed{\langle \mathcal{W}; \Gamma; \Sigma \mid e \rangle \Rightarrow v} \\
 \\
 \text{E-IF-TRUE-L} \frac{x_1 \in \Sigma \quad x_2 \notin \Sigma \quad \Sigma(x_1) = v_1 \quad \langle \mathcal{W}; \Gamma; x_2 \mapsto v_1, \Sigma \mid e \rangle \Rightarrow v}{\langle \mathcal{W}; \Gamma; \Sigma \mid \text{if } x_1 = x_2; e \rangle \Rightarrow v} \\
 \\
 \text{E-IF-TRUE-R} \frac{x_1 \notin \Sigma \quad \langle \mathcal{W}; \Gamma; \Sigma \mid x_2 \rangle \Rightarrow v_2 \quad \langle \mathcal{W}; \Gamma; x_1 \mapsto v_2, x_2 \mapsto v_2, \Sigma \mid e \rangle \Rightarrow v}{\langle \mathcal{W}; \Gamma; \Sigma \mid \text{if } x_1 = x_2; e \rangle \Rightarrow v} \\
 \\
 \text{E-METHOD-VAL} \frac{(f, \bar{l}_i = v_i, v_{out}) \in \mathcal{W}}{\langle \mathcal{W}; \Gamma; \Sigma \mid f(\bar{l}_i = v_i) \rangle \Rightarrow v_{out}} \\
 \\
 \text{E-METHOD-NAME} \frac{\forall (f, \bar{l}_i = v'_i, v_{out}) \in \mathcal{W}. \exists i : v'_i \neq v_i \quad (f, \bar{l}_i = v'_i, v_{out}) \in \mathcal{W}}{\langle \mathcal{W}; \Gamma; \Sigma \mid f(\bar{l}_i = v_i) \rangle \Rightarrow v_{out}}
 \end{array}$$

Figure 12. Retrospective execution.

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 Fig. 19 in Appendix C), 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 $f(\bar{l}_i = v_i)$, 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.

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

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	API size				API Analysis	
	$ \Lambda.f $	n_{arg}	$ \Lambda.o $	s_{obj}	$ \mathcal{W} $	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

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 [13] 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 Appendix D. 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 *effective*: 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 non-trivial for programmers to solve manually. A complete list of tasks, type queries, and solutions can be found in Appendix E.

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 correct solution for 29 benchmarks. The remaining three benchmarks fail with a timeout because their type queries

Table 2. Synthesis benchmarks and results. Benchmarks marked with \dagger 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 (sec)	Rank		
		AST	n_f	n_p	n_g		r_{orig}	r_{RE}	r_{RE}^{TO}
SLACK	1.1	17	3	6	1	83.5	25230	5	5
	1.2 \dagger	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 \dagger	10	2	3	0	3.4	788	5	5
	1.6 \dagger	9	2	2	0	1.7	573	8	19
	1.7 \dagger	12	2	4	1	1.3	757	8	9
	1.8	9	2	3	0	42.0	16438	29	30
STRIPE	2.1 \dagger	9	2	2	0	95.4	4952	3	3
	2.2 \dagger	10	2	2	0	92.4	4854	4	4
	2.3 \dagger	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 \dagger	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 \dagger	10	2	3	0	7.8	187	6	6
	2.11 \dagger	7	2	1	0	17.2	490	6	6
	2.12 \dagger	11	3	2	0	-	-	-	-
	2.13 \dagger	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 \dagger	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 \dagger	16	2	5	1	1.9	174	10	12
	3.11 \dagger	8	2	3	0	1.0	68	16	16

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).

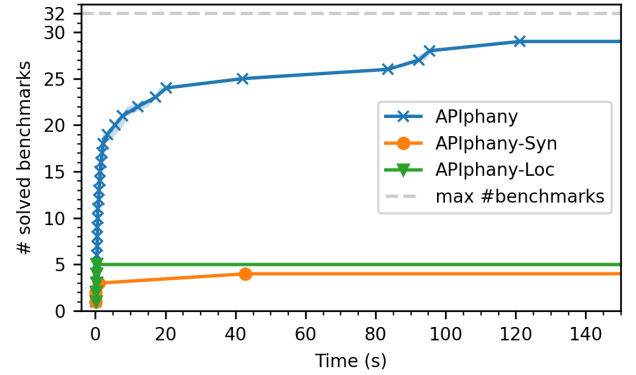


Figure 13. Comparison of synthesis performance between APIPHANY and its two variants that do not use type mining.

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`, $loc^* = \text{User.id}$); we consider the inferred loc -set type sufficient if it contains loc^* . The detailed results appear in Appendix E (Tab. 4).

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, 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

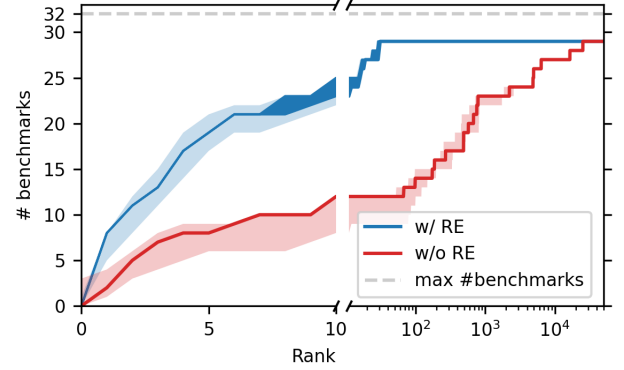


Figure 14. Number of benchmarks whose solution is reported within a given rank. The filled blue area is the range of ranks one might get depending on when they inspect the candidates. The shaded area is the 95% confidence interval.

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 [Appendix D](#)); 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 `≤`, 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, 15, 24] scales poorly with the size of the component library. An alternative, more scalable *graph-based approach* was introduced in PROSPECTOR [21] for unary components, and generalized to *n*-ary components in SyPET [8], by replacing graphs with Petri nets. TYGAR [12] 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 [14]. 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 [20, 25, 27] but relies on

static analysis and mining client code, which APIPHANY does not require. Among tools that leverage dynamic analysis, EDSYNTH [34] uses test executions to generate snippets that involve both API calls and control structures. MATCH-MAKER [36] and DEMOMATCH [35] 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 [32, 33]. 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, 23, 28] or example usages [4, 6, 17] 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 [16, 19] 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 [14, 27, 29] ranking functions based on syntactic features of generated programs. HOOGL+ [18] 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 Bodik, 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 Augusssstson. 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, 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.
- [13] Gurobi Optimization, LLC. 2021. Gurobi Optimizer Reference Manual. <https://www.gurobi.com>
- [14] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *PLDI*.
- [15] George T. Heineman, Jan Bessai, Boris Döder, 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
- [16] 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>
- [17] 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
- [18] 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.
- [19] 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>
- [20] 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>
- [21] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *PLDI*.
- [22] 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>
- [23] 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>
- [24] 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
- [25] 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>
- [26] Inc. Postman. 2020. 2020 State of The API Report. <https://www.postman.com/state-of-api/api-technologies/>.
- [27] 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>
- [28] 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.
- [29] 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.
- [30] Inc. SmartBear. 2020. The State of API Report 2020. https://static1.smartbear.co/smartbearbrand/media/pdf/smartbear_state_of_api_2020.pdf.
- [31] 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>
- [32] 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>
- [33] 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>
- [34] 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>
- [35] 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>
- [36] 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>

$$\begin{array}{c}
\textbf{Expression Typing} \quad \boxed{\hat{\Lambda}; \Gamma \vdash e :: \hat{t}} \quad \boxed{\hat{\Lambda} \vdash \mathcal{E} :: \hat{s}} \\
\text{T-VAR} \frac{x : \hat{t} \in \Gamma}{\hat{\Lambda}; \Gamma \vdash x :: \hat{t}} \quad \text{T-PROJ} \frac{\hat{\Lambda}; \Gamma \vdash e :: \{l : \hat{t}, \dots\}}{\hat{\Lambda}; \Gamma \vdash e.l :: \hat{t}} \\
\text{T-RET} \frac{\hat{\Lambda}; \Gamma \vdash e :: \hat{t}}{\hat{\Lambda}; \Gamma \vdash \textbf{return } e :: [\hat{t}]} \\
\text{T-CALL} \frac{f : \{\ell_j : \hat{t}_j\} \rightarrow \hat{t}_o \in \hat{\Lambda} \quad \hat{\Lambda}; \Gamma \vdash e_i :: \hat{t}_i \quad \forall j. \ell_j = l_j \Rightarrow \exists i. l_j = l_i \wedge \hat{t}_j = \hat{t}_i \quad \forall j, i. \ell_j = ?l_j \wedge l_j = l_i \Rightarrow \hat{t}_j = \hat{t}_i}{\hat{\Lambda}; \Gamma \vdash f(l_i = e_i) :: \hat{t}_o} \\
\text{T-IF} \frac{\hat{\Lambda}; \Gamma \vdash e_x :: \{\overline{loc}\} \quad \hat{\Lambda}; \Gamma \vdash e_y :: \{\overline{loc}\} \quad \hat{\Lambda}; \Gamma \vdash e :: [\hat{t}]}{\hat{\Lambda}; \Gamma \vdash \textbf{if } e_x = e_y; e :: [\hat{t}]} \\
\text{T-LET} \frac{\hat{\Lambda}; \Gamma \vdash e_1 :: \hat{t}_1 \quad \hat{\Lambda}; \Gamma, x : \hat{t}_1 \vdash e_2 :: \hat{t}_2}{\hat{\Lambda}; \Gamma \vdash \textbf{let } x = e_1; e_2 :: \hat{t}_2} \\
\text{T-BIND} \frac{\hat{\Lambda}; \Gamma \vdash e_1 :: [\hat{t}_1] \quad \hat{\Lambda}; \Gamma, x : \hat{t}_1 \vdash e_2 :: [\hat{t}_2]}{\hat{\Lambda}; \Gamma \vdash x \leftarrow e_1; e_2 :: [\hat{t}_2]} \\
\text{T-OBJ} \frac{\hat{\Lambda}; \Gamma \vdash e :: o \quad o : \hat{t} \in \hat{\Lambda}}{\hat{\Lambda}; \Gamma \vdash e :: \hat{t}} \\
\text{T-TOP} \frac{\hat{\Lambda}; x_i : \hat{t}_i \vdash e :: \hat{t}}{\hat{\Lambda} \vdash \lambda \overline{x_i}. e :: \{\overline{x_i} : \hat{t}_i\} \rightarrow \hat{t}}
\end{array}$$

Figure 16. λ_A : typing judgment.

$$\begin{array}{c}
\textbf{Location-Based Type Inference} \quad \boxed{\Lambda \vdash loc \Rightarrow \hat{t}} \\
\text{ObjStart} \frac{\Lambda, o \vdash \bar{l} \Rightarrow \hat{t}}{\Lambda \vdash o.\bar{l} \Rightarrow \hat{t}} \quad \text{FunStart} \frac{\Lambda, f \vdash \bar{l} \Rightarrow \hat{t}}{\Lambda \vdash f.\bar{l} \Rightarrow \hat{t}} \\
\text{ObjBase} \frac{}{\Lambda, o \vdash [] \Rightarrow o} \quad \text{PathBase} \frac{loc \neq o}{\Lambda, loc \vdash [] \Rightarrow \{loc\}} \\
\text{ObjFollow} \frac{\Lambda(loc.l_1) = o \quad \Lambda, o \vdash \bar{l} \Rightarrow \hat{t}}{\Lambda, loc \vdash l_1.\bar{l} \Rightarrow \hat{t}} \\
\text{Arr} \frac{\Lambda(loc.l) = [t] \quad \Lambda, loc.l \vdash o \Rightarrow \hat{t}}{\Lambda, loc \vdash l \Rightarrow [\hat{t}]} \\
\text{AdHoc} \frac{\Lambda(loc.l) = \{l_i : \hat{t}_i\} \quad \Lambda, loc.l \vdash l_i \Rightarrow \hat{t}_i}{\Lambda, loc \vdash l \Rightarrow \{\overline{l_i : \hat{t}_i}\}} \\
\text{PathFollow} \frac{\Lambda(loc.l_1) = \text{String} \vee (\Lambda(loc.l_1) \neq o \wedge \bar{l} \neq []) \quad \Lambda, loc.l_1 \vdash \bar{l} \Rightarrow \hat{t}}{\Lambda, loc \vdash l_1.\bar{l} \Rightarrow \hat{t}}
\end{array}$$

Figure 15. Rules for location-based type inference.

A Type Mining

Location-based type inference. The location-based type inference judgement $\Lambda \vdash loc \Rightarrow \hat{t}$ is defined in Fig. 15. In this

figure, the notation $\Lambda(loc) = t$ denotes looking up the syntactic type of location loc in the library Λ , e.g. $\Lambda(\text{User.profile}) = \text{Profile}$ and $\Lambda(\text{c_members.out.0}) = \text{String}$. At the same time, $\Lambda(\text{User.profile.email})$ is undefined because it does not directly appear in Λ (instead we need to ask for Profile.email). The full definition of syntactic lookup is straightforward and therefore omitted.

The main complication during location-based type inference is that we need to “fold” locations that denote named objects, replacing them with object names; e.g.

$$\Lambda \vdash \text{users_info.out.id} \Rightarrow \{\text{User.id}\}$$

To this end we introduce an auxiliary judgement $\Lambda, loc \vdash \bar{l} \Rightarrow \hat{t}$, where intuitively loc and \bar{l} correspond to a prefix and a suffix of the location of interest, except that loc is sufficiently “folded”. For example, to derive the judgement above, we will start with

$$\Lambda, \text{users_info} \vdash \text{out.id} \Rightarrow \dots$$

but then use **ObjFollow** to fold the path and rewrite the judgement into

$$\Lambda, \text{User} \vdash \text{id} \Rightarrow \dots$$

and then by **PathBase** we can establish

$$\Lambda, \text{User} \vdash \text{id} \Rightarrow \{\text{User.id}\}$$

Note that because loc in this judgement is always sufficiently folded, all applications of $\Lambda(loc.l)$ in our rules are actually well-defined.

The rules **Arr** and **AdHoc** deal with inference of array types and record types. The rule **Arr** applies when the syntactic type of the top-level location of interest is an array; in this case we infer the semantic type \hat{t} of the array element and return $[\hat{t}]$ for the location. Note that this rule is not used when an array-typed location occurs in the middle of a path and not at the top level: in this case the default rule **PathFollow** applies. The rule **AdHoc** takes care of locations that have *ad-hoc* record types in Λ (as opposed to names object types); for example, we infer the type $\{\text{user} : \text{User.id}\}$ for the location users_info.in . Just like **Arr**, **AdHoc** only applies when the top-level location has a record type, and otherwise **PathFollow** suffices.

B Program Synthesis

Expression typing. The (semantic) typing rules for programs of λ_A are presented in Fig. 16. The typing judgement $\hat{\Lambda}; \Gamma \vdash e :: \hat{t}$ states that a term e has the semantic type \hat{t} under context Γ and the semantic library $\hat{\Lambda}$. The rule **T-CALL** check that all required arguments are provided and all provided arguments have correct types. In a monadic binding $x \leftarrow e_1; e_2$, both e_1 and e_2 must have array types (**T-BIND**). In a guard **if** $e_1 = e_2; e$, e must have an array type, while e_1 and e_2 cannot have an array or record type, since equality is only supported over string values (**T-IF**). Finally, typing

$$\begin{array}{c}
\textbf{TTN Construction} \quad \boxed{\hat{\Lambda} \vdash \mathcal{N} \xrightarrow{c} \mathcal{N}'} \quad \boxed{\hat{\Lambda} \vdash \mathcal{N} \xrightarrow{c:\hat{t}} \mathcal{N}'} \\
\text{C-METHOD} \quad \frac{
\begin{array}{l}
f : \{\ell_i : \hat{t}_i\} \rightarrow \hat{t}_o \in \hat{\Lambda} \quad \hat{t}_{in} = \{\ell_i : [\hat{t}_i]\} \\
O' = O[(p, f) \mapsto |\{l_i \mid ?l_i : p \in \hat{t}_{in}\}|] \\
E' = E[(p, f) \mapsto |\{l_i \mid l_i : p \in \hat{t}_{in}\}|][(f, [\hat{t}_o]) \mapsto 1]
\end{array}
}{
\hat{\Lambda} \vdash (P, T, E, O) \xrightarrow{f} (P \cup \{[\hat{t}_i], [\hat{t}_o]\}, T \cup \{f\}, E', O')
} \\
\text{C-OBJECT} \quad \frac{
\begin{array}{l}
o : \{\ell_i : \hat{t}_i\} \in \hat{\Lambda} \quad \hat{\Lambda} \vdash \mathcal{N} \xrightarrow{\text{proj}_{o.l_i} : [\hat{t}_i]} \mathcal{N}'_i \\
\hat{\Lambda} \vdash \bigcup \mathcal{N}'_i \xrightarrow{\text{filter}_{o.l_i} : [\hat{t}_i]} \mathcal{N}''_i
\end{array}
}{
\hat{\Lambda} \vdash \mathcal{N} \xrightarrow{o} \bigcup \mathcal{N}''_i
} \\
\text{C-PROJ} \quad \frac{
E' = E[(o, \text{proj}_{o.l}) \mapsto 1, (\text{proj}_{o.l}, \hat{t}) \mapsto 1]
}{
\hat{\Lambda} \vdash (P, T, E, O) \xrightarrow{\text{proj}_{o.l} : \hat{t}} (P' \cup \{o, \hat{t}\}, T' \cup \{\text{proj}_{o.l}\}, E', O)
} \\
\text{C-FILTER} \quad \frac{
\begin{array}{l}
\hat{t} \text{ not obj. id} \\
E' = E[(o, \text{filter}_{o.l} \neg) \mapsto 1, (\hat{t}, \text{filter}_{o.l} \neg) \mapsto 1, (\text{filter}_{o.l}, o) \mapsto 1]
\end{array}
}{
\hat{\Lambda} \vdash (P, T, E, O) \xrightarrow{\text{filter}_{o.l} : \hat{t}} (P \cup \{o, \hat{t}\}, T \cup \{\text{filter}_{o.l} \neg\}, E', O)
} \\
\text{C-FILTER-OBJ} \quad \frac{
\begin{array}{l}
o' : \{\ell_i : \hat{t}_i\} \in \hat{\Lambda} \quad \hat{\Lambda} \vdash \mathcal{N} \xrightarrow{\text{filter}_{o.l_i} : [\hat{t}_i]} \mathcal{N}'_i \\
\hat{\Lambda} \vdash \mathcal{N} \xrightarrow{\text{filter}_{o.l} : o'} \bigcup \mathcal{N}'_i
\end{array}
}{
\hat{\Lambda} \vdash \mathcal{N} \xrightarrow{\text{filter}_{o.l} : o'} \bigcup \mathcal{N}'_i
}
\end{array}$$

Figure 17. TTN construction from a semantic library.

follows object definitions in $\hat{\Lambda}$: if e has type o and $o : \hat{t} \in \Lambda$, then e also has the type \hat{t} (T-OBJ).

B.1 TTN Construction

A TTN \mathcal{N} is a 4-tuple (P, T, E, O) , where P is a set of *places*, T is a set of *transitions*, and $E : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$ is a matrix of *edge multiplicities*. $E(p, \tau)$ denotes how many required arguments of type p component τ consumes, and $E(\tau, p)$ denotes how many responses of type p it produces. $E(p, \tau) = 0$ means that there is no edge from p to τ (and symmetrically for $E(\tau, p) = 0$). To model optional arguments, in this work we augment the TTN with the matrix of *optional multiplicities* $O : P \times T \rightarrow \mathbb{N}$, which denotes the number of optional arguments of a given type.

A *marking* is a mapping $M : P \rightarrow \mathbb{N}$ that assigns a non-negative number of tokens to every place. A *transition firing* is a triple $M_1 \xrightarrow{\tau} M_2$, such that for all places p : (a) M_1 contains at least as many tokens as τ 's incoming edges require: $M_1(p) \geq E(p, \tau)$, and (b) M_2 loses tokens consumed by the incoming edge, but gains tokens produced by the outgoing edge: $\exists c. E(p, \tau) \leq c \leq E(p, \tau) + O(p, \tau) \wedge M_2(p) = M_1(p) - c + E(\tau, p)$. A *path* between M and M' is a sequence of transitions τ_1, \dots, τ_n such that $M \xrightarrow{\tau_1} M_1 \xrightarrow{\tau_2} \dots \xrightarrow{\tau_{n-1}} M_{n-1} \xrightarrow{\tau_n} M'$ is a sequence of transition firings.

Array-oblivious encoding. Procedure BUILDTTN($\hat{\Lambda}$) (line 2 in Fig. 10) constructs a TTN \mathcal{N} given a semantic library. Intuitively, its goal is to add transitions for all methods in $\hat{\Lambda}$,

as well as other operations of λ_A , such as projections and filtering, such that any well-typed λ_A program can be encoded as a path in \mathcal{N} . There is one major issue, however: λ_A programs also contain the higher-order monadic bind operations (aka “flat maps”), which cannot be easily encoded in a TTN; moreover, the presence of array types in the TTN increases the number of places and transitions, and slows down the search. To combat this issue we propose the *array-oblivious encoding* of λ_A programs into the TTN, which replaces array types with types of their elements and monadic bindings with regular **let**-bindings (which in the TTN corresponds to simple sequencing of transitions). Formally, we define the *downgrading* operation on semantic types $[\hat{t}]$ as follows:

$$[\hat{t}] = \begin{cases} [\hat{t}'] & \text{if } \hat{t} = [\hat{t}'] \\ \hat{t} & \text{otherwise} \end{cases}$$

Construction rules. We formalize TTN construction using the step relation $\hat{\Lambda} \vdash \mathcal{N} \xrightarrow{c} \mathcal{N}'$ defined in Fig. 17. A construction step adds an encoding of a library component c (method or object) to \mathcal{N} and produces a new TTN \mathcal{N}' ; to encode the entire library $\hat{\Lambda}$, we compose steps for all methods and objects in the library. For example, Fig. 9 depicts (a fragment of) the TTN built from the semantic library in Fig. 7; in this figure, all edges have multiplicity 1 (since no method in $\hat{\Lambda}$ takes multiple arguments of the same type), and there are no optional edges. In general, there are three kinds of transitions in a TTN: method transitions, such as `conversations_members`, projection transitions, such as `projUser.id`, and filter transitions, such as `filterChannel.name`. We now describe the construction rules for the three kinds of transitions in more detail.

The rule C-METHOD adds a transition for a method f and connects it to the f 's (downgraded) input and output types. For example, let $f = \text{conversations_members}$, whose downgraded type is $\{\text{channel} : \text{Channel.id}\} \rightarrow \text{User.id}$. In this case, the rule C-METHOD extends the TTN with a transition f and places $\{\text{Channel.id}, \text{User.id}\}$; it also sets $E[(\text{Channel.id}, f)] = 1$, $E[(f, \text{User.id})] = 1$ (and all other incoming and outgoing edge multiplicities to 0). Fig. 17 uses the notation $E[k \mapsto v]$, to denote a map that is equal to E except for mapping k to v . Note that C-METHOD is the only rule that modifies optional multiplicities O , since only methods can have optional arguments.

The rule C-OBJECT adds projection and filter transitions for every field l of an object identifier o , merging the resulting TTNs component-wise. The rule C-PROJ adds a projection transition `projo.l` between an on object identifier o and the type of l .

Filter transitions are a little more involved, as they do not mirror the structure of λ_A guards one-to-one. Guards in their general form are higher-order operations, and hence cannot be directly encoded in the TTN. Fortunately, APIPHANY only uses guards for one purpose: filtering objects from an array xs if their constituent is equal to some y (in code: $x \leftarrow$

xs ; **if** $x.\bar{l} = y$; **return** x). We encode this filtering operation in the TTN as a filter transition filter $_{o,\bar{l}}$, which consumes o (the downgraded type of xs) and the type of y , and produces o . For example, the filter transition filter $_{\text{Channel.name}}$ in Fig. 9 consumes a `Channel` and a `Channel.name` and produces a `Channel`. When $o.\bar{l}$ is an object, the rule C-FILTER-OBJ recursively creates filter transitions for all its fields (recall that λ_A only supports guards on primitive values). For example, for the object ID `User`, we will add a transition filter $_{\text{User.profile.email}}$, but not filter $_{\text{User.profile}}$.

B.2 TTN Search

Once the TTN \mathcal{N} has been constructed, the algorithm SYNTHESIZE proceeds to enumerate paths from the initial marking I to the final marking F in \mathcal{N} . I and F are constructed from the query type $\hat{t}_{in} \rightarrow \hat{t}_o$ as follows:

$$I(p) = |\{l_i \mid l_i : p \in [\hat{t}_{in}]\}|$$

$$F(p) = \text{if } p = [\hat{t}_o] \text{ then } 1 \text{ else } 0$$

For example, for the query `Channel.name` \rightarrow `[Profile.email]`, the initial marking contains a single token in `Channel.name` and the final marking contains a single token in `Profile.email`. Hence, any valid path from I to F must consume all inputs and produce the output of the query type. Note that the TTN as defined in Sec. B.1 encodes a *linear type system*, i.e. can only generate programs where each input is used *exactly once*. Following prior work [8, 12], our implementation adds copy transitions to the TTN, which results in a *relevant type system*, i.e. one where every input has to be used *at least once*. Prior work shows that this relevancy requirement is crucial to filtering out meaningless solutions during search.

ILP encoding. To find paths in the TTN, prior work has relied on a SAT/SMT encoding of TTN reachability. We have found that although the SMT encoding from [12] works well to find a handful of paths, in our domain we often need to enumerate thousands of paths, which becomes very inefficient. To address this problem, we instead use an *integer linear programming* (ILP) solver, which provides native functionality for computing all solutions to a constraint.

Given a TTN $\mathcal{N} = (P, T, E, O)$ with the initial marking I and the final marking F , we show how to build an ILP formula that encodes all valid paths of a given length L . The overall search proceeds by iteratively increasing the path length L . We encode the number of tokens in each place $p \in P$ at each time step $k \in [0, L]$ as a variable tok_k^p . We encode firing of transition $\tau \in T$ at time step $k \in [0, L-1]$ as a variable $\text{fire}_k^\tau \in \{0, 1\}$, such that $\text{fire}_k^\tau = 1$ indicates that τ is fired at time step k . For any $\tau \in T$, we define the pre-image of τ as $\text{pre}(\tau) = \{p \in P \mid E(p, \tau) > 0 \vee O(p, \tau) > 0\}$ and the post-image of τ as $\text{post}(\tau) = \{p \in P \mid E(\tau, p) > 0\}$.

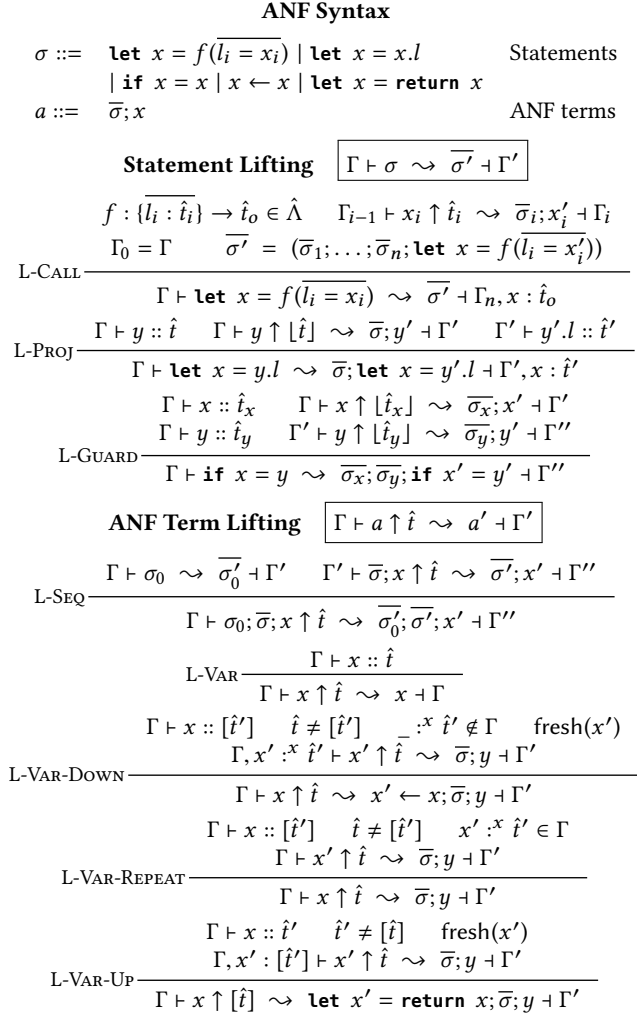
The formula for TTN reachability is a conjunction of the following constraints:

- (1) If a transition τ is fired at time step k , then all required places $p \in \text{pre}(\tau)$ have sufficiently many tokens: $\bigwedge_{k=0}^{L-1} \bigwedge_{\tau \in T} \bigwedge_{p \in \text{pre}(\tau)} \text{tok}_k^p \geq E(p, \tau) \times \text{fire}_k^\tau$
- (2) If a transition τ is fired at time step k , then all places $p \in \text{pre}(\tau) \cup \text{post}(\tau)$ will have their marking updated at time step $k+1$: $\bigwedge_{k=0}^{L-1} \bigwedge_{\tau \in T} \bigwedge_{p \in \text{pre}(\tau) \cup \text{post}(\tau)} \text{tok}_k^p - (E(p, \tau) + O(p, \tau) - E(\tau, p)) \times \text{fire}_k^\tau \leq \text{tok}_{k+1}^p \leq \text{tok}_k^p - (E(p, \tau) - E(\tau, p)) \times \text{fire}_k^\tau$
- (3) At each time step, only one transition is fired: $\bigwedge_{k=0}^{L-1} \sum_{\tau \in T} \text{fire}_k^\tau = 1$
- (4) Variable domains are respected: $\bigwedge_{k=0}^L \bigwedge_{p \in P} 0 \leq \text{tok}_k^p$ and $\bigwedge_{k=0}^{L-1} \bigwedge_{\tau \in T} 0 \leq \text{fire}_k^\tau \leq 1$
- (5) The initial marking I is valid: $\bigwedge_{p \in P} \text{tok}_0^p = I(p)$
- (6) The final marking is valid: $\bigwedge_{p \in P} \text{tok}_L^p = F(p)$

Note that constraint (2) *approximates* consumption of optional arguments as a range between consuming all of them ($E(p, \tau) + O(p, \tau)$) and consuming none of them ($E(p, \tau)$). This encoding is unsound when an optional argument has the same type as the output. Consider firing a transition τ at step k such that $\text{tok}_k^p = 0, E(p, \tau) = 0, O(p, \tau) = 1, E(\tau, p) = 1$; our encoding $0 \leq \text{tok}_{k+1}^p \leq 1$, whereas the TTN definition in Sec. B.1 requires $\text{tok}_{k+1}^p = 1$ (the optional argument could not be consumed since there was no token to consume). We use the approximate encoding because in our experience it is significantly more efficient than the exact alternatives, which require additional variables and/or constraints. In our evaluation, the unsoundness arises very rarely, and when it does, the path is simply rejected by the type checker when converted into a program.

B.3 Program Lifting

From paths to 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). An ANF program is sequence of *statements* σ followed by a variable; the syntax of ANF terms is given in Fig. 18. `PROGS` converts each transition in π into a sequence of statements with a dedicated output variable x_o : a method transition becomes **let** $x_o = f(l_i = x_i)$, a projection transition becomes **let** $x_o = x.l$, and a filter transition becomes **let** $x_1 = x_o.l_1; \dots$ **let** $x_n = x_{n-1}.l_n$; **if** $x_n = y$. All emitted statements are then concatenated into an ANF term $\sigma_1; \dots; \sigma_n; x_n$, where x_n is the output variable of the last transition. Fig. 11 (left) shows the full array-oblivious program extracted from the bold path in Fig. 9. The reason we use ANF as the intermediate representation instead of generating λ_A terms directly is that ANF has a more direct correspondence with TTN paths and also is more convenient to work with during lifting; and ANF term can be translated into a regular λ_A term by recursing through the sequence of statements and replacing sequential composition with λ_A **let** bindings, monadic bindings, or guard expressions.

**Figure 18.** Lifting rules.

Lifting rules. We formalize lifting of ANF terms as a *term lifting* judgement $\Gamma \vdash a \uparrow \hat{t} \rightsquigarrow a' \vdash \Gamma'$, defined in Fig. 18. Here a is the array-oblivious term to be lifted, whose free variables are defined in Γ , \hat{t} is the target type, a' is the lifted term, and Γ' is Γ extended with all the variables bound in a' . The definition of term lifting relies on the auxiliary *statement lifting* judgement $\Gamma \vdash \sigma \rightsquigarrow \overline{\sigma'} \vdash \Gamma'$, which lifts a single statement σ that appears in the environment Γ into a sequence of statements $\overline{\sigma'}$; again Γ' is Γ extended with variables bound in $\overline{\sigma'}$. Both judgments implicitly rely on the semantic library $\hat{\Lambda}$, which we omit for brevity since it is not modified. For example, the statement in line 4 of Fig. 11 is lifted as follows:

$$\begin{aligned} x_1 : [\text{Channel}] \vdash \text{let } x_2 = x_1.\text{name} &\rightsquigarrow \\ x'_1 \leftarrow x_1; \text{let } x_2 = x'_1.\text{name} &\vdash \\ x_1 : [\text{Channel}], x'_1 :^{x_1} \text{Channel}, x_2 : \text{Channel.name} &\end{aligned}$$

Note that the binding for x'_1 in Γ' is annotated with x_1 ; we introduce these annotations to keep track of *mapping variables*: here x'_1 is the mapping variable for the array x_1 , and it

will be reused later in the program whenever an element of x_1 is required again (see line 6).

Let us now describe the rules in Fig. 18 in more detail. The statement lifting rules enforce well-typing for the three kinds of statements that appear in array-oblivious programs: L-CALL enforces that the arguments of a method call agree with its definition in $\hat{\Lambda}$, while L-PROJ and L-GUARD make sure their operands are scalars. The heavy lifting is done by the rules L-VAR-DOWN and L-VAR-UP, both of which handle the case of lifting a variable whose type \hat{t}' in Γ does not match the target type \hat{t} . L-VAR-DOWN applies when $\hat{t}' = [\dots[\hat{t}]\dots]$: in this case we generate a monadic binding $x' \leftarrow x$ (and then lift x' again in case the array type was nested); this rule is responsible for the monadic bindings in lines 3 and 8 of Fig. 11. Note that in the last premise we add the binding $x' :^x \hat{t}'$ to Γ to record that x' is the mapping variable for x . If Γ already has a mapping variable for x , we want to reuse that variable instead of generating a new binding; this is accomplished by the rule L-VAR-REPEAT. Finally, L-VAR-UP applies when the opposite is true, i.e. $\hat{t} = [\dots[\hat{t}']\dots]$; in this case we simply need to wrap x in a **return**. This rule generates line 12 of Fig. 11: here the target return type of the program is $\hat{t} = [\text{Profile.email}]$ and the type of x_7 is $\hat{t}' = \text{Profile.email}$.

C Retrospective Execution

The full definition of the RE judgement is presented in Fig. 19.

D Witness Collection

For each API, we manually created a test environment via the corresponding web interface and filled in some arbitrary test data. We then ran several operations in the test environment, using Google Chrome to record the web traffic in an HTTP Archive (HAR) file: a JSON-formatted file that logs browser interactions with a server. We then extracted initial witnesses \mathcal{W}_0 from the HAR file and ran the algorithm MINETYPES (Fig. 8) on these witnesses, resulting in an initial semantic library $\hat{\Lambda}_0$.

Unfortunately, the set \mathcal{W}_0 obtained this way is sparse, which may prevent the mining algorithm from merging equivalent locations or even inferring types for some methods altogether. For example, if we only have initial witnesses as shown in Fig. 4, APIPHANY is unable to infer the semantic type of `users.lookupByEmail` that is required by one of our benchmarks, and hence fail to solve this task. At the same time, generating useful tests for this method is challenging, because it only succeeds on inputs that correspond to existing users' emails. To improve the quality and coverage of inferred types, APIPHANY generates additional witnesses, using a combination of type-directed random testing and a small amount of manual annotations. Specifically, APIPHANY draws test inputs from the bank of values it has observed in the existing witnesses; in our example,

Retrospective Execution $\langle \mathcal{W}; \Gamma; \Sigma \mid e \rangle \Rightarrow v$	
E-VAR	$\frac{\Sigma(x) = v}{\langle \mathcal{W}; \Gamma; \Sigma \mid x \rangle \Rightarrow v}$
E-VAR-LAZY	$\frac{x \notin \Sigma \quad \Gamma \vdash x :: \hat{t} \quad v \in \mathcal{W}(\hat{t})}{\langle \mathcal{W}; \Gamma; \Sigma \mid x \rangle \Rightarrow v}$
E-BIND-PURE	$\frac{\langle \mathcal{W}; \Gamma; \Sigma \mid e_1 \rangle \Rightarrow v \quad \langle \mathcal{W}; \Gamma; x \mapsto v, \Sigma \mid e_2 \rangle \Rightarrow v'}{\langle \mathcal{W}; \Gamma; \Sigma \mid \text{let } x = e_1; e_2 \rangle \Rightarrow v'}$
E-BIND-MONAD	$\frac{\langle \mathcal{W}; \Gamma; \Sigma \mid e_1 \rangle \Rightarrow [\bar{v}_i] \quad \langle \mathcal{W}; \Gamma; x \mapsto v_i, \Sigma \mid e_2 \rangle \Rightarrow v'_i \quad v' = \bigcup v'_i}{\langle \mathcal{W}; \Gamma; \Sigma \mid x \leftarrow e_1; e_2 \rangle \Rightarrow v'}$
E-RETURN	$\frac{\langle \mathcal{W}; \Gamma; \Sigma \mid \text{return } x \rangle \Rightarrow [v]}{\langle \mathcal{W}; \Gamma; \Sigma \mid x \rangle \Rightarrow v}$
E-PROJECTION	$\frac{\langle \mathcal{W}; \Gamma; \Sigma \mid x \rangle \Rightarrow v \quad \text{hasField}(v, l)}{\langle \mathcal{W}; \Gamma; \Sigma \mid x.l \rangle \Rightarrow v.l}$
E-IF-TRUE-L	$\frac{x_1 \in \Sigma \quad x_2 \notin \Sigma \quad \Sigma(x_1) = v_1 \quad \langle \mathcal{W}; \Gamma; x_2 \mapsto v_1, \Sigma \mid e \rangle \Rightarrow v}{\langle \mathcal{W}; \Gamma; \Sigma \mid \text{if } x_1 = x_2; e \rangle \Rightarrow v}$
E-IF-TRUE-R	$\frac{x_1 \notin \Sigma \quad \langle \mathcal{W}; \Gamma; \Sigma \mid x_2 \rangle \Rightarrow v_2 \quad \langle \mathcal{W}; \Gamma; x_1 \mapsto v_2, x_2 \mapsto v_2, \Sigma \mid e \rangle \Rightarrow v}{\langle \mathcal{W}; \Gamma; \Sigma \mid \text{if } x_1 = x_2; e \rangle \Rightarrow v}$
E-IF-TRUE-LR	$\frac{x_1 \in \Sigma \quad x_2 \in \Sigma \quad \Sigma(x_1) = \Sigma(x_2) \quad \langle \mathcal{W}; \Gamma; \Sigma \mid e \rangle \Rightarrow v}{\langle \mathcal{W}; \Gamma; \Sigma \mid \text{if } x_1 = x_2; e \rangle \Rightarrow v}$
E-IF-FALSE	$\frac{\Sigma(x_1) = v_1 \quad \Sigma(x_2) = v_2 \quad v_1 \neq v_2 \quad \langle \mathcal{W}; \Gamma; \Sigma \mid \text{if } x_1 = x_2; e \rangle \Rightarrow []}{\langle \mathcal{W}; \Gamma; \Sigma \mid \text{if } x_1 = x_2; e \rangle \Rightarrow []}$
E-METHOD	$\frac{\langle \mathcal{W}; \Gamma; \Sigma \mid x_i \rangle \Rightarrow v_i \quad \langle \mathcal{W}; \Gamma; \Sigma \mid f(\bar{l}_i = v_i) \rangle \Rightarrow v_{out}}{\langle \mathcal{W}; \Gamma; \Sigma \mid f(\bar{l}_i = x_i) \rangle \Rightarrow v_{out}}$
E-METHOD-VAL	$\frac{(f, \bar{l}_i = v_i, v_{out}) \in \mathcal{W}}{\langle \mathcal{W}; \Gamma; \Sigma \mid f(\bar{l}_i = v_i) \rangle \Rightarrow v_{out}}$
E-METHOD-NAME	$\frac{\forall (f, \bar{l}_i = v'_i, v_{out}) \in \mathcal{W}. \exists i : v'_i \neq v_i \quad (f, \bar{l}_i = v'_i, v_{out}) \in \mathcal{W}}{\langle \mathcal{W}; \Gamma; \Sigma \mid f(\bar{l}_i = v_i) \rangle \Rightarrow v_{out}}$

Figure 19. Retrospective execution.

one of the strings that appear in Fig. 4 is "xyz@gmail.com". Calling `users_lookupByEmail` with this string succeeds and returns "UJ5RHEG4S". Based on the previously mined types of these two strings, APIPHANY infers the type `Profile.email` \rightarrow `User.id` for the new method.

The top-level API analysis algorithm of APIPHANY is depicted in Fig. 20 (top). The algorithm alternates between calls to `MINE TYPES` (to compute the best semantic library $\hat{\Lambda}$ it can mine from the current witnesses \mathcal{W}) and `GenerateTests` (to

augment \mathcal{W} using the current $\hat{\Lambda}$). Analysis terminates either if $\hat{\Lambda}$ and \mathcal{W} reach a fixpoint, or when a timeout is reached. The algorithm returns both $\hat{\Lambda}$, which is used to build the TTN during the synthesis step, and \mathcal{W} , which is used for retrospective execution during the ranking step; hence augmenting \mathcal{W} also improves the quality of ranking.

Before we introduce the algorithm `GENERATE TESTS`, we augment the semantic library $\hat{\Lambda}$ with a *value bank* \mathcal{V} , which is a mapping from semantic types to sets of values. The value bank gets populated during type mining and contains all values that appear in \mathcal{W} ; for arrays and objects, it also contains their constituents. With the value bank at hand, let us explain the algorithm `GENERATE TESTS` presented in Fig. 20 (bottom). `GENERATE TESTS` iterates through the method signatures in $\hat{\Lambda}$, and for every method $f: \hat{t}_i \rightarrow \hat{t}_o$, it samples a random input of type \hat{t}_i , makes an API call to f with that input, and if the call succeeds, yields the corresponding witness. It uses the value bank (denoted $\hat{\Lambda}.\mathcal{V}$) at line 7 to randomly sample an input from all values stored at semantic type \hat{t} . We also observe that many API methods have optional arguments and behave differently depending on which subset of optional arguments is provided. To cover a wide range of method behaviors, `GENERATE TESTS` partitions the record of all arguments \hat{t}_i into two records, each containing either only required or only optional arguments (line 3). The algorithm then iterates over all subsets of optional arguments (line 4), attempting to make a call for each subset (our implementation only iterates over subsets up to a pre-defined size).

Fully automatic test generation helps us bootstrap semantic type inference for methods that already appear in \mathcal{W}_0 , but it cannot add witnesses for methods that are missing entirely. To address this problem, we manually add consumer-producer annotations to those methods missing from \mathcal{W}_0 .

In the evaluation, we collect witnesses by running the algorithm `ANALYZE API` (Fig. 20) until it converges, alternating test generation and type mining, which resulted in the final set of witnesses \mathcal{W} and library $\hat{\Lambda}$. The total running time for API analysis depends on the number of methods and their arguments, and ranged from several minutes to several hours in our experiments.

E Benchmarks and Solutions

Tab. 3 contains benchmark descriptions and detailed results. Tab. 4 contains the results of the qualitative analysis of mined semantic types.

The rest of this section includes type queries and “gold standard” solutions for all benchmarks. Note that the type queries used here correspond directly to the OpenAPI spec; earlier in the paper, type and method names were simplified for readability.

Input: Library Λ , initial witnesses \mathcal{W}_0
Output: Semantic library $\hat{\Lambda}$
Output: Augmented witnesses \mathcal{W}
1: **function** ANALYZEAPI(Λ, \mathcal{W}_0)
2: $\mathcal{W} \leftarrow \mathcal{W}_0$
3: **repeat**
4: $\hat{\Lambda} \leftarrow \text{MINE_TYPES}(\Lambda, \mathcal{W})$
5: $\mathcal{W} \leftarrow \mathcal{W} \cup \text{GENERATE_TESTS}(\hat{\Lambda})$
6: **until** fixpoint or timeout
7: **return** $\hat{\Lambda}, \mathcal{W}$

Input: Semantic library $\hat{\Lambda}$
Output: Generated witnesses \mathcal{W}
1: **function** GENERATE_TESTS($\hat{\Lambda}$)
2: **for** $f: \{l_i: \hat{t}_i, ?l_j: \hat{t}_j\} \rightarrow \hat{t}_o \in \hat{\Lambda}$ **do**
3: $\hat{t}_{req}, \hat{t}_{opt} \leftarrow \{l_i: \hat{t}_i\}, \{?l_j: \hat{t}_j\}$
4: **for** $\hat{t}_{sub} \subset \hat{t}_{opt}$ **do**
5: $v_{in} \leftarrow \{\}$
6: **for** $l: \hat{t} \in \hat{t}_{req} \cup \hat{t}_{sub}$ **do**
7: $v_{in}.l \leftarrow \text{random}(\hat{\Lambda}.\mathcal{V}[\hat{t}])$
8: $v_{out} \leftarrow \text{call}(f, v_{in})$
9: **if** $v_{out} \neq \perp$ **then yield** $\langle f, v_{in}, v_{out} \rangle$

Figure 20. Top-level API analysis algorithm and test generation.

Table 3. Synthesis benchmarks and results. Benchmarks marked with \dagger 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 spent on RE-based ranking t_{RE} and the total synthesis time t_{Total} in seconds. In the last four columns, we report the rank before RE r_{orig} , the rank after RE among candidates that we find before the desired solution r_{RE} , the total number of candidates we get within timeout, and the rank after RE among all candidates within timeout r_{RE}^{TO} . ‘-’ means no solution is found in 150 seconds.

API	Benchmark		Solution Size				Timing		Rank			
	ID	Description	AST	n_f	n_p	n_g	t_{Total}	t_{RE}	r_{orig}	r_{RE}	# cands	r_{RE}^{TO}
SLACK	1.1	Retrieve emails of all members in a channel	17	3	6	1	83.5	0.5	25230	5	38212	5
	1.2 †	Send a message to a user given their email	12	3	5	0	5.6	0.1	2224	10	30437	10
	1.3	Get the unread messages of a user	16	3	7	0	-	-	-	-	-	-
	1.4	Get all messages associated with a user	14	2	4	1	1.3	0.2	489	24	28012	31
	1.5 †	Create a channel and invite a list of users	10	2	3	0	3.4	0.1	788	5	22426	5
	1.6 †	Reply to a message and update it	9	2	2	0	1.7	0.1	573	8	39276	19
	1.7 †	Send a message to a channel with the given name	12	2	4	1	1.3	<0.1	757	8	39078	9
	1.8	Get the unread messages of a channel	9	2	3	0	42.0	0.8	16438	29	50757	30
STRIPE	2.1 †	Subscribe to a product for a customer	9	2	2	0	95.4	0.6	4952	3	6312	3
	2.2 †	Subscribe to multiple items	10	2	2	0	92.4	0.6	4854	4	6167	4
	2.3 †	Create a product and invoice a customer	12	3	2	0	121.2	2.6	6363	1	6644	1
	2.4	Retrieve a customer by email	8	1	2	1	0.5	<0.1	3	1	1751	1
	2.5	Get a list of receipts for a customer	8	2	2	0	1.0	<0.1	10	4	4548	4
	2.6 †	Get a refund for a subscription	9	3	2	0	12.2	0.1	270	3	4584	3
	2.7	Get the emails of all customers	5	1	2	0	0.6	<0.1	4	2	1382	2
	2.8	Get the emails of the subscribers of a product	16	2	7	1	20.2	0.2	679	17	3407	17
	2.9	Get the last 4 digits of a customer’s card	6	1	2	0	0.5	<0.1	2	1	1812	1
	2.10 †	Update payment methods for a user’s subscriptions	10	2	3	0	7.8	0.1	187	6	3068	6
	2.11 †	Delete the default payment source for a customer	7	2	1	0	17.2	0.1	490	6	1373	6
	2.12 †	Save a card during payment	11	3	2	0	-	-	-	-	-	-
	2.13 †	Send an invoice to a customer	10	3	2	0	-	-	-	-	-	-
SQUARE	3.1	List invoices that match a location id	4	1	1	0	0.2	<0.1	2	1	9544	1
	3.2	List subscriptions by location, customer, and plan	16	1	4	3	0.5	<0.1	10	4	2526	4
	3.3	Get all items a tax applies to	10	1	3	1	0.4	<0.1	6	1	11039	1
	3.4	Get a list of discounts in the catalog	5	1	2	0	0.7	<0.1	2	1	11704	1
	3.5 †	Add order details to order	14	2	3	0	2.2	<0.1	99	2	7222	2
	3.6	Get payment notes of a payment	5	1	2	0	0.2	<0.1	1	1	9590	1
	3.7	Get order ids of current user’s transactions	6	1	2	0	0.3	<0.1	7	4	8669	4
	3.8	Get order names from a transaction id	9	1	3	0	0.7	<0.1	1	1	12323	1
	3.9	Find customers by name	8	1	2	1	0.2	<0.1	3	2	3177	2
	3.10 †	Delete catalog items with names	16	2	5	1	1.9	<0.1	174	10	11336	12
	3.11 †	Delete all catalog items	8	2	3	0	1.0	<0.1	68	16	7429	16

Table 4. A sample of API methods, with their expected and inferred semantic types. Non-string parameters and responses are omitted since we do not perform inference for those.

API	Method	Field Type	Required	Name	Expected Type	Inferred Type
SLACK	/stars.add_POST	Parameter	No	channel	defs_channel	defs_group_id,defs_dm_id
			No	file	defs_file_id	/stars.add.in.file
			No	file_comment	defs_comment_id	/stars.add.in.file.comment
			No	timestamp	defs_ts	/stars.add.in.timestamp
	/conversations.list_GET	Parameter	No	types	objs_conversation.types	/conversations.list.in.types
	/users.profile.get_GET	Parameter	No	user	defs_user_id	objs_file.user, defs_bot_id, defs.topic.purpose.creator, defs_user_id
		Response	-	profile	objs_user_profile	objs_user_profile
STRIPE	/reminders.list_GET	Response	-	reminders	objs_reminder	objs_reminder
	/users.conversations_GET	Parameter	No	user	defs_user_id	objs_file.user, defs_bot_id, defs.topic.purpose.creator, defs_user_id
			No	types	objs_conversation.types	/users.conversations.in.types
	/v1/invoiceitems/{invoiceitem}_POST	Parameter	Yes	invoiceitem	invoiceitem.id	invoiceitem.id,line_item.invoice_item
			No	description	invoiceitem.description	product.name,invoiceitem.description, credit_note_line_item.description, line_item.description
			No	discounts[0][coupon]	discounts.coupon	/v1/invoiceitems/{invoiceitem}.in.discounts.0.coupon
			No	discounts[0][discount]	discounts.discount	/v1/invoiceitems/{invoiceitem}.in.discounts.0.discount
			No	price	price.id	price.id,plan.id
			No	price_data[currency]	invoice.lines.data.price.currency	/v1/invoiceitems/{invoiceitem}.in.price_data.currency
			No	price_data[product]	invoice.lines.data.price.product	/v1/invoiceitems/{invoiceitem}.in.price_data.product
			No	tax_rates[0]	tax_rate.id	/v1/invoiceitems/{invoiceitem}.in.tax_rates.0
	/v1/webhook_endpoints_GET	Response	-	object	N/A (Method returns constant string).	radar.value_list.list_items.object, credit_note.lines.object, customer.sources.object, subscription.items.object, payment.intent.charges.object, charge.refunds.object, file.links.object, customer.subscriptions.object, invoice.lines.object
	/v1/transfers_GET	Parameter	No	destination	account.id	/v1/transfers.in.destination
			No	transfer_group	transfer.transfer_group	/v1/transfers.in.transfer_group
		Response	-	object	N/A (Method returns constant string).	radar.value_list.list_items.object, credit_note.lines.object, customer.sources.object, subscription.items.object, payment.intent.charges.object, charge.refunds.object, file.links.object, customer.subscriptions.object, invoice.lines.object
	/v1/subscription_items_GET	Parameter	Yes	subscription	subscription.id	invoiceitem.subscription, invoice.subscription, discount.subscription, subscription.item.subscription, line_item.subscription,subscription.id
		Response	-	object	subscription.items.object	radar.value_list.list_items.object, credit_note.lines.object, customer.sources.object, subscription.items.object, payment.intent.charges.object, charge.refunds.object, file.links.object, customer.subscriptions.object, invoice.lines.object
	/v1/tax_rates/{tax_rate}_POST	Parameter	Yes	tax_rate	tax_rate.id	tax_rate.id, invoice_tax_amount.tax_rate
			No	country	tax_rate.country	/v1/tax_rates/{tax_rate}.in.country
			No	description	tax_rate.description	tax_rate.description
			No	display_name	tax_rate.display_name	tax_rate.display_name
			No	jurisdiction	tax_rate.jurisdiction	address.state, invoice.account.country, card.country,tax_rate.jurisdiction, tax_rate.country, source_type_card.country, account.country, payment_method_card.country, payment_method_details.card.country, country_spec.supported_transfer_countries.[?], card.address.state, country_spec.id, address.country
			No	state	tax_rate.state	/v1/tax_rates/{tax_rate}.in.state
			No	given_name	Customer.given_name	Customer.given_name, InvoiceRecipient.given_name
SQUARE	/v2/customers_POST	Parameter	No	family_name	Customer.family_name	Customer.family_name, InvoiceRecipient.family_name
			No	company_name	Customer.company_name	Customer.company_name
			No	nickname	Customer.nickname	Customer.nickname
			No	email_address	Customer.email_address	/v2/customers.in.email_address
			No	address	Address	Address
			No	phone_number	Customer.phone_number	Customer.phone_number
			No	reference_id	Customer.reference_id	Customer.reference_id
			No	note	Customer.note	Customer.note
			No	birthday	Customer.birthday	/v2/customers.in.birthday
	/v2/orders/{order_id}_GET	Parameter	Yes	order_id	Order.id	Transaction.id, Payment.order_id,Order.id, Transaction.order_id, Invoice.order_id, Tender.transaction_id, OrderEntry.order_id
	/v2/catalog/list_GET	Parameter	No	types	CatalogObject.type	CatalogObject.type
			No	catalog_version	CatalogObject.version	/v2/catalog/list.in.catalog_version
	/v2/labor/break-types_GET	Parameter	No	location_id	Location.id	/v2/labor/break-types.in.location_id
			No	catalog_object_ids[0]	CatalogObject.id	/v2/inventory/batch-retrieve-counts.in.catalog_object_ids.0
	/v2/inventory/batch-retrieve-counts_POST	Parameter	No	location_ids[0]	Location.id	/v2/inventory/batch-retrieve-counts.in.location_ids.0
			No	updated_after	Counts.updated_at	/v2/inventory/batch-retrieve-counts.in.updated_after
			No	states[0]	Counts.states	/v2/inventory/batch-retrieve-counts.in.states.0

E.1 SLACK

1.1. Retrieve emails of all members in a channel

Type query:

```
{ channel_name: objs_conversation.name } → [objs_user_profile.  
  ↪ email]
```

Solution:

```
\channel_name → {  
  let x0 = /conversations_list_GET()  
  x1 ← x0.channels  
  if x1.name = channel_name  
  let x2 = /conversations_members_GET(channel=x1.id)  
  x3 ← x2.members  
  let x4 = /users_profile_get_GET(user=x3)  
  return x4.profile.email  
}
```

Source: <https://stackoverflow.com/questions/41564027/slack-api-retrieve-all-member-emails-from-a-slack-channel>

1.2. Send a message to a user given their email

Type query:

```
{ email: objs_user_profile.email } → objs_message
```

Solution:

```
\email → {  
  let x0 = /users_lookupByEmail_GET(email=email)  
  let x1 = /conversations_open_POST(users=x0.user.id)  
  let x2 = /chat_postMessage_POST(channel=x1.channel.id)  
  return x2.message  
}
```

Source: <https://stackoverflow.com/questions/43733375/slack-api-post-message-via-user-email>

1.3. Get the unread messages of a user

Type query:

```
{ user_id: defs_user_id } → [[objs_message]]
```

Solution:

```
\user_id → {  
  let x0 = /users_conversations_GET(user=user_id)  
  x1 ← x0.channels  
  let x2 = /conversations_info_GET(channel=x1.id)  
  let x3 = /conversations_history_GET(channel=x2.channel.id,  
    ↪ oldest=x2.channel.last_read)  
  return x3.messages  
}
```

Source: <https://stackoverflow.com/questions/64561594/is-it-possible-to-know-the-number-of-unread-slack-messages-a-user-has-with-the-s>

1.4. Get all messages associated with a user

Type query:

```
{ user_id: defs_user_id,  
  ts: defs_ts  
} → [objs_message]
```

Solution:

```
\user_id ts → {  
  let x0 = /conversations_list_GET()  
  x1 ← x0.channels
```

```
  let x2 = /conversations_history_GET(channel=x1.id, oldest=ts)  
  x3 ← x2.messages  
  if x3.user = user_id  
  return x3  
}
```

Source: <https://github.com/hisabimbola/slack-history-export/blob/e53868d8820ba65e5e726bd5968c80d5eb54c0db/src/utils.js>

1.5. Create a channel and invite a list of users

Type query:

```
{ user_ids: [defs_user_id],  
  channel_name: objs_conversation.name  
} → [objs_conversation]
```

Solution:

```
\user_ids channel_name → {  
  let x0 = /conversations_create_POST(name=channel_name)  
  x1 ← user_ids  
  let x2 = /conversations_invite_POST(channel=x0.channel.id,  
    ↪ users=x1)  
  return x2.channel  
}
```

Source: <https://stackoverflow.com/questions/48328380/slack-api-channels-create-followed-by-channels-invite-info-returns-channel-not>

1.6. Reply to a message and update it

Type query:

```
{ channel: defs_channel,  
  ts: defs_ts  
} → objs_message
```

Solution:

```
\channel ts → {  
  let x1 = /chat_postMessage_POST(channel=channel, thread_ts=ts  
    ↪ )  
  let x2 = /chat_update_POST(channel=channel, ts=x1.ts)  
  return x2.message  
}
```

1.7. Send a message to a channel with the given name

Type query:

```
{ channel: objs_conversation.name } → objs_message
```

Solution:

```
\channel → {  
  let x0 = /conversations_list_GET()  
  x1 ← x0.channels  
  if x1.name = channel  
  let x2 = /chat_postMessage_POST(channel=x1.id)  
  return x2.message  
}
```

Source: <https://github.com/backspace/slack-statsbot/blob/primary/src/atsbot.js>

1.8. Get the unread messages of a channel

Type query:

```
{ channel_id: defs_channel } → [[objs_message]]
```

Solution:

```
\channel_id → {
  let x2 = /conversations_info.GET(channel=channel_id)
  let x3 = /conversations_history.GET(channel=channel_id,
    ↳ oldest=x2.channel.last_read)
  return x3.messages
}
```

Source: <https://stackoverflow.com/questions/64561594/is-it-possible-to-know-the-number-of-unread-slack-messages-a-user-has-with-the-s>

E.2 STRIPE

2.1. Subscribe to a product for a customer

Type query:

```
{ customer_id: customer.id,
  product_id: product.id
} → [subscription]
```

Solution:

```
\customer_id product_id → {
  let x1 = /v1/prices.GET(product=product_id)
  x2 ← x1.data
  let x3 = /v1/subscriptions.POST(customer=customer_id, items
    ↳ [0][price]=x2.id)
  return x3
}
```

Source: <https://github.com/stripe-samples/charging-for-multiple-plan-subscriptions/blob/master/server/node/server.js>

2.2. Subscribe to multiple items

Type query:

```
{ customer_id: customer.id,
  product_ids: [product.id]
} → [subscription]
```

Solution:

```
\customer_id product_ids → {
  x0 ← product_ids
  let x1 = /v1/prices.GET(product=x0)
  x2 ← x1.data
  let x3 = /v1/subscriptions.POST(customer=customer_id, items
    ↳ [0][price]=x2.id)
  return x3
}
```

Source: <https://github.com/stripe-samples/charging-for-multiple-plan-subscriptions/blob/master/server/node/server.js>

2.3. Create a product and invoice a customer

Type query:

```
{ product_name: product.name,
  customer_id: customer.id,
  currency: fee.currency,
  unit_amount: plan.amount
} → invoiceitem
```

Solution:

```
\product_name customer_id currency unit_amount → {
  let x0 = /v1/products.POST(name=product_name)
  let x1 = /v1/prices.POST(currency=currency, product=x0.id,
    ↳ unit_amount=unit_amount)
  let x2 = /v1/invoiceitems.POST(customer=customer_id, price=x1
    ↳ .id)
  return x2
}
```

```
}
```

Source: <https://stripe.com/docs/invoicing/prices-guide>

2.4. Retrieve a customer by email

Type query:

```
{ email: customer.email } → customer
```

Solution:

```
\email → {
  let x0 = /v1/customers.GET()
  x1 ← x0.data
  if x1.email = email
    return x1
}
```

Source: <https://stackoverflow.com/questions/26767150/stripe-is-it-possible-to-search-a-customer-by-their-email>

2.5. Get a list of receipts for a customer

Type query:

```
{ customer_id: customer.id } → [charge]
```

Solution:

```
\customer_id → {
  let x1 = /v1/invoices.GET(customer=customer_id)
  x2 ← x1.data
  let x3 = /v1/charges/{charge}.GET(charge=x2.charge)
  return x3
}
```

Source: <https://stackoverflow.com/questions/24335268/stripe-api-receipts-listing>

2.6. Get a refund for a subscription

Type query:

```
{ subscription: subscription.id } → refund
```

Solution:

```
\subscription → {
  let x0 = /v1/subscriptions/{subscription_exposed_id}.GET(
    ↳ subscription_exposed_id=subscription)
  let x1 = /v1/invoices/{invoice}.GET(invoice=x0.latest_invoice
    ↳ )
  let x2 = /v1/refunds.POST(charge=x1.charge)
  return x2
}
```

Source: <https://stackoverflow.com/questions/62403075/stripe-api-get-upcoming-invoice-for-cancelled-subscription>

2.7. Get the emails of all customers

Type query:

```
{ } → [customer.email]
```

Solution:

```
\ → {
  let x0 = /v1/customers.GET()
  x1 ← x0.data
  return x1.email
}
```

Source: <https://stackoverflow.com/questions/65545997/python3-stripe-api-to-get-all-customer-email>

2.8. Get the emails of the subscribers of a product

Type query:


```
{ product_id: product.id } → [customer.email]
```

Solution:

```
\product_id → {
  let x1 = /v1/subscriptions.GET()
  x2 ← x1.data
  x3 ← x2.items.data
  if x3.price.product = product_id
  let x4 = /v1/customers/{customer}.GET(customer=x2.customer)
  return x4.email
}
```

Source: <https://stackoverflow.com/questions/35882771/use-stripe-api-to-return-a-list-of-valid-subscribers>

2.9. Get the last 4 digits of a customer's card

Type query:

```
{ customer_id: customer.id } → bank_account.last4
```

Solution:

```
\customer_id → {
  let x0 = /v1/customers/{customer}/sources.GET(customer=
    ↪ customer_id)
  x1 ← x0.data
  return x1.last4
}
```

Source: <https://stackoverflow.com/questions/30447026/getting-last4-digits-of-card-using-customer-object-stripe-api-with-php>

2.10. Update payment methods for a user's subscriptions

Type query:

```
{ payment_method: payment_method,
  customer_id: customer.id
} → [subscription]
```

Solution:

```
\payment_method customer_id → {
  let x0 = /v1/subscriptions.GET(customer=customer_id)
  x1 ← x0.data
  let x2 = /v1/subscriptions/{subscription_exposed_id}.POST(
    ↪ subscription_exposed_id=x1.id,
    ↪ default_payment_method=payment_method.id)
  return x2
}
```

Source: <https://stackoverflow.com/questions/58270828/update-credit-card-details-of-user-for-all-subscriptions-in-stripe-using-api>

2.11. Delete the default payment source for a customer

Type query:

```
{ customer_id: customer.id } → payment_source
```

Solution:

```
\customer_id → {
  let x0 = /v1/customers/{customer}.GET(customer=customer_id)
  let x1 = /v1/customers/{customer}/sources/{id}.DELETE(
    ↪ customer=customer_id, id=x0.default_source)
  return x1
}
```

Source: <https://stackoverflow.com/questions/17807881/stripe-api-throwing-error-when-trying-to-delete-a-card>

2.12. Save a card during payment

Type query:

```
{ cur: fee.currency,
  amt: plan.amount,
  pm: payment_method.id
} → payment_intent
```

Solution:

```
\cur amt pm → {
  let x1 = /v1/customers.POST()
  let x2 = /v1/payment_intents.POST(customer=x1.id,
    ↪ payment_method=pm, currency=cur, amount=amt)
  let x3 = /v1/payment_intents/{intent}/confirm.POST(intent=x2.
    ↪ id)
  return x3
}
```

Source: <https://github.com/stripe-samples/saving-card-after-payment/blob/master/without-webhooks/server/node/server.js>

2.13. Send an invoice to a customer

Type query:

```
{ customer_id: customer.id,
  price_id: plan.id
} → invoice
```

Solution:

```
\customer_id price_id → {
  let x1 = /v1/invoiceitems.POST(customer=customer_id, price=
    ↪ price_id)
  let x2 = /v1/invoices.POST(customer=x1.customer)
  let x3 = /v1/invoices/{invoice}/send.POST(invoice=x2.id)
  return x3
}
```

Source: <https://stripe.com/docs/invoicing/integration#send-invoice>

E.3 SQUARE

3.1. List invoices that match a location id

Type query:

```
{ location_id: Location.id } → [Invoice]
```

Solution:

```
\location_id → {
  let x0 = /v2/invoices.GET(location_id=location_id)
  return x0.invoices
}
```

Source: https://github.com/square/connect-api-examples/blob/4283ac967c31b75dc17acebd84f649093477e9a/connect-examples/v2/node_invoices/routes/management.js

3.2. List subscriptions by location, customer, and plan

Type query:

```
{ customer_id: Customer.id,
  location_id: Location.id,
  plan_id: CatalogObject.id
} → [Subscription]
```

Solution:

```
\customer_id location_id plan_id → {
  let x0 = /v2/subscriptions/search_POST()
  x1 ← x0.subscriptions
  if x1.customer_id = customer_id
  if x1.location_id = location_id
  if x1.plan_id = plan_id
  return x1
}
```

Source: https://github.com/square/connect-api-examples/blob/4283ac967c31b75dc17aceebd84f649093477e9a/connect-examples/v2/node_subscription/routes/subscription.js

3.3. Get all items a tax applies to

Type query:

```
{ tax_id: CatalogObject.id } → [CatalogObject]
```

Solution:

```
\tax_id → {
  let x0 = /v2/catalog/search_POST()
  x1 ← x0.objects
  x2 ← x1.item_data.tax_ids
  if x2 = tax_id
  return x1
}
```

Source: <https://github.com/square/catalog-api-demo/blob/85b6754c90fa7b66fc5e605ee7a344314537eade/src/main/java/com/squareup/catalog/demo/example/ApplyTaxToAllItemsExample.java>

3.4. Get a list of discounts in the catalog

Type query:

```
{ } → [CatalogDiscount]
```

Solution:

```
\ → {
  let x0 = /v2/catalog/list_GET()
  x1 ← x0.objects
  return x1.discount_data
}
```

Source: <https://github.com/square/catalog-api-demo/blob/master/src/main/java/com/squareup/catalog/demo/example/ListDiscountsExample.java>

3.5. Add order details to order

Type query:

```
{ location_id: Location.id,
  order_ids: [Order.id],
  updates: [OrderFulfillment]
} → [Order]
```

Solution:

```
\location_id order_ids updates → {
  x0 ← order_ids
  let x1 = /v2/orders/batch-retrieve_POST(location_id=
    ↪ location_id, order_ids[0]=x0)
  x2 ← x1.orders
  let x3 = {fulfillments=updates}
  let x4 = /v2/orders/{order_id}_PUT(order_id=x2.id, order=x3)
  return x4.order
}
```

Source: https://github.com/square/connect-api-examples/blob/4283ac967c31b75dc17aceebd84f649093477e9a/connect-examples/v2/node_orders-payments/routes/checkout.js

3.6. Get payment notes of a payment

Type query:

```
{ } → [Payment.note]
```

Solution:

```
\ → {
  let x0 = /v2/payments.GET()
  x1 ← x0.payments
  return x1.note
}
```

Source: <https://stackoverflow.com/questions/23252751/square-connect-api-list-payments-endpoint-not-showing-description>

3.7. Get order ids of current user's transactions

Type query:

```
{ location_id: Location.id } → [Order.id]
```

Solution:

```
\location_id → {
  let x0 = /v2/locations/{location_id}/transactions_GET(
    ↪ location_id=location_id)
  x1 ← x0.transactions
  return x1.order_id
}
```

Source: <https://stackoverflow.com/questions/46910044/getting-compact-information-from-square-connect-api>

3.8. Get order names from a transaction id

Type query:

```
{ location_id: Location.id,
  transaction_id: Order.id
} → [Invoice.title]
```

Solution:

```
\location_id transaction_id → {
  let x0 = /v2/orders/batch-retrieve_POST(location_id=
    ↪ location_id, order_ids[0]=transaction_id)
  x1 ← x0.orders
  x2 ← x1.line_items
  return x2.name
}
```

Source: <https://stackoverflow.com/questions/58047894/square-connect-how-to-retrieve-product-information-from-transaction-id>

3.9. Find customers by name

Type query:

```
{ name: Customer.given_name } → Customer
```

Solution:

```
\name → {
  let x0 = /v2/customers_GET()
  x1 ← x0.customers
  if x1.given_name = name
  return x1
}
```

Source: <https://developer.squareup.com/forums/t/search-customers-by-name/1567>

3.10. Delete catalog items with names

Type query:

```
{ item_type: CatalogObject.type,
  names: [CatalogItem.name]
} → [CatalogObject.id]
```

Solution:

```
\item_type names → {
  let x0 = /v2/catalog/search_POST(object_types[0]=item_type)
  x1 ← x0.objects
  x2 ← names
  if x1.item_data.name = x2
  let x3 = /v2/catalog/object/{object_id}_DELETE(object_id=x1.
    ↪ id)
  x4 ← x3.deleted_object_ids
  return x4
}
```

Source: <https://github.com/square/catalog-api-demo/blob/85b6754c90fa7b66fc5e605ee7a344314537eade/src/main/java/com/squareup/catalog/demo/example/DeleteCategoryExample.java>

3.11. Delete all catalog items

Type query:

```
{ } → [CatalogObject.id]
```

Solution:

```
\ → {
  let x0 = /v2/catalog/list_GET()
  x1 ← x0.objects
  let x2 = /v2/catalog/object/{object_id}_DELETE(object_id=x1.
    ↪ id)
  return x2.deleted_object_ids
}
```

Source: <https://github.com/square/catalog-api-demo/blob/85b6754c90fa7b66fc5e605ee7a344314537eade/src/main/java/com/squareup/catalog/demo/example/DeleteAllItemsExample.java>