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.


Keywords: Program Synthesis, RESTful API, Type Inference

ACM Reference Format:

1 Introduction

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

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

1. First, call conversations_list1 to retrieve the array of all channel objects, and then search for a channel object with a given name and get its ID;
2. Next, call conversations.members on the channel ID to get all user IDs of its members;
3. Finally, for each user ID, call users.info to retrieve a user object u, and then access the user’s email via u.profile.email.

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

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

---

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

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

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

2 APIphany by Example

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

2.1 API Analysis by Example

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

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

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

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

Mining types from witnesses. To this end, we build upon an algorithm first proposed in [1] that infers types by mining them from execution traces, based on the insight that equal values observed at different locations likely have the same type. More specifically, our type mining algorithm starts by ascribing a unique semantic type to each String location and then merges locations that share a value anywhere in the witness set. As an illustration, consider Fig. 4, which lists two witnesses for the API methods from our running example. In this witness set we observe the same value "U5RH6G4S" 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
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`.

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.

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

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

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

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

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

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

2.3 Ranking via Retrospective Execution

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

Fortunately, most of the 1000 candidates are easy to weed out because they produce uninteresting results. Consider:

```plaintext
\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.
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.

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

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

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

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

\[
\begin{align*}
\sigma &::= \text{User} | \text{Channel} | \ldots & \text{object names} \\
f &::= \text{u.info} | \ldots & \text{method names} \\
l &::= \text{in} | \text{out} | 0 | \text{id} | \text{name} | \ldots & \text{field labels} \\
\ell &::= \text{l}|\ell & \text{record fields} \\
loc &::= o^7 | f^7 & \text{locations} \\
\end{align*}
\]

**Terms**

\[
\begin{align*}
e &::= x | e.l & \text{expressions, projection} \\
f &::= f(l_i = e_i) & \text{method call, pure binding} \\
\text{let } x &= e; e & \text{guard, monadic binding} \\
\text{return } e & & \text{pure value lifting} \\
E &::= \lambda x.e & \text{Top Level Programs} \\
\end{align*}
\]

**Values**

\[
\begin{align*}
v &::= \ldots | [v] & \text{strings, arrays, objects} \\
t &::= \text{String} & \text{Syntactic types} \\
o | \{t\} &::= \{l_i : t_i\} & \text{named objects, arrays, records} \\
s &::= t \rightarrow t & \text{function types} \\
i &::= \text{loc} & \text{loc-sets} \\
o | \{t\} &::= \{l_i : t_i\} & \text{named objects, arrays, records} \\
\end{align*}
\]

**Types**

\[
\begin{align*}
\Lambda &::= \sigma ; t ; f : s & \text{object and method definitions} \\
\hat{\Lambda} &::= o ; i ; f : \hat{s} & \text{semantic definitions} \\
\end{align*}
\]

**Figure 6.** Syntax of the language \(\lambda_A\)

In this section, we formalize the core of APIPHANYS’s DSL as \(\lambda_A\), a functional language specialized for manipulating semi-structured data. The syntax of \(\lambda_A\) is summarized in Fig. 6.

**Types.** The types of \(\lambda_A\) include syntactic types \(t\) (those used in the OpenAPI spec) and semantic types \(i\), which we infer. Both categories of types have named objects \(o\), arrays \([t]\), and records \(\{l_i : t_i\}\).

\(\text{Records are mappings from field labels to types; some fields are optional, indicated with a } ? \text{ before its label. For example, the record type } \{\text{id} : \text{String}, ?\text{time}\text{.zone} : \text{String}\}, \text{has a required field } \text{id} \text{ and an optional field } \text{time}\text{.zone}.\)

\(\text{Because in REST some arguments are optional, the same method can be called with different subsets of arguments.}\)

\(\text{We write } \overline{X} \text{ to denote zero or more occurrences of a syntactic element } X.\)
The two categories of types differ in their base types: the sole primitive syntactic type is `String`, 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 → 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 `Λ`, 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 `Λ`.

**Terms.** Values of `λ` 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 ← e` evaluates `e` for each element `x` of the array `e`, and concatenates all resulting arrays. In contrast, the pure binding `let x = e; e2` binds `x` to the entire result of `e` and then evaluates `e2`. 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 `E` is an abstraction with a list of arguments `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 `Λ ⊢ loc → t`, which assigns a semantic type `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 `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:

- `Λ ⊢ c . members . out → [{c . members . out . 0}]` because array types do not themselves get replaced with `locs`; instead, we recursively assign a location-based type to an array’s element.
- `Λ ⊢ u . info . out . id ⊢ {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, `Λ ⊢ u . info . out → user`, and then follow the field `id` of the `user` object.

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

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

`DS` supports two efficient operations: `insert` and `find`. `insert` takes a pair `{loc, v}` and checks whether either of its components already appears in `DS`; if so, it merges the new pair into the corresponding group, and otherwise puts it into a new group. `find` takes a location `loc` and returns a semantic type `t`; internally, `find` locates the group to which the pair `{loc, _}` belongs in `DS` and returns the `loc-set` `{loc, loc1, ...}` that contains all locations in that group.

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

In the first phase, the algorithm iterates over the witnesses, registering the input value `v_in` at the location `f.in` and the output value `v_out` at the location `f .out`. To this end, we call a helper function `AddWitness`, which drills down into composite values (arrays and objects) to get to string literals, and then inserts each string into `DS` with its location-based type. For example, when processing the response from the first witness in Fig. 4, `AddWitness` iterates over all channel objects in the array, and over all fields of each channel object; once it reaches the value “U5RMHG4S”, it computes the type of its location as `Λ ⊢ c . list . out . 0 . creator`.

The second phase is a for loop that iterates over all pairs in `DS`, and `loc-set`s represent the final set of semantic types.

In the second phase, the algorithm calls `AddDefinitions` to iterate over all object and method definitions in `Λ`, and
add corresponding definitions to \( \Lambda \), relying on \text{find} to retrieve the semantic type for each location. For example, when adding the method \( \text{u.info} \), we \text{query} \( \text{DS.u.info.in.user} \), which finds the group mentioned above and returns its loc-set: \{\text{User.id}, \text{Channel.creator}, \ldots\}. If the requested location is not in \( DS \)—because \( W \) 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 \( \Lambda \) and a semantic query type \( \tilde{s} \). For our running example, we use the semantic library from Fig. 7 and the query type \( \text{Channel.name} \rightarrow \text{[Profile.email]} \).\(^7\) A \textit{candidate solution} is any program \( E \) that type-checks against \( \tilde{s} \). To formalize this notion, we introduce the program typing judgment \( \Lambda \vdash E :: \tilde{s} \), 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 \( \text{if } e_1 = e_2; e \text{, } e \) must have an array type, while \( e_1 \) and \( e_2 \) must have (the same) loc-set type, since equality is only supported over string values. Full definition can be found in the technical report [12].

**Type transition nets.** To efficiently enumerate well-typed programs we follow prior work [8, 13] and encode the search space as a special kind of Petri net, called \textit{type-transition net} (TTN). Intuitively, a TTN encodes how each API method transforms values of one semantic type into another; e.g. \( \text{u.info} \) transforms a \text{User.id} into a \text{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 \( \lambda \Lambda \) projections (e.g. \text{proj}_{\text{User.profile}} \) and \text{proj}_{\text{Profile.email}} \) and guards (e.g. \text{filter}_{\text{Channel.name}}).

\textbf{Array-oblivious search.} For our search space encoding to be useful, we need to make sure that every well-typed \( \lambda \Lambda \) program corresponds to a path in the TTN. This is where we encounter a challenge: there is no straightforward way to encode \( \lambda \Lambda \)’s monadic bind operation into the TTN. Although prior work on Hoogle+ [13] supports higher-order functions, the arguments to those functions are syntactically restricted to variables (i.e. inner lambda abstractions are not supported), which is insufficient for our purposes. To address this problem, we introduce a new, \textit{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 \text{c.members} returns \text{User} instead of \text{[User]}, and hence its output can be passed directly to \text{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 \textit{token} in the input type \text{Channel.name} and search for a path (a sequence of transitions) that would get this token to the output type \text{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 \text{c.list} which does not consume any

---

\(^7\)Here and throughout this section, we write loc-set types using an arbitrarily chosen representative; the user can \text{query} APIPhany using any locations of their choosing, and the tool interprets them as the loc-sets they belong to.
tokens) to produce an extra token in Channel. Next, we fire filterChannel.name, which consumes the two tokens in Channel and Channel.name, and produces a single token in Channel. The remaining five transitions on the bold path simply move this one token along until it reaches Profile.email.

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

**Synthesis algorithm.** APHIDNY’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 $\pi$, the algorithm iterates over the corresponding array-oblivious programs $\mathcal{E}$ and lifts them into well-typed $\Lambda$ programs. The reason $\pi$ might yield multiple programs is that the TTN does not distinguish different arguments of the same type, and hence we must try all their combinations.

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

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

**Lifting array-oblivious programs.** The function $\text{Progs}(\pi)$ (line 5 in Fig. 10) converts a TTN path $\pi$ into a set of array-oblivious programs in A-Normal Form (ANF). Fig. 11 (left) shows the full array-oblivious program extracted from the bold path in Fig. 9. As you can see from this example, array-oblivious programs can be ill-typed: for example, the projection $x_1$.name in line 4 does not type-check since $x_1$ actually has an array type [channel]. What we really want this program to do is to project name (and execute the remaining steps in the program) for each channel in $x_1$. This can be accomplished by inserting a monadic binding $x_1' \leftarrow x_1$ and using $x_1'$ instead of $x_1$ in line 4 (and elsewhere in the program where a non-array version of $x_1$ is required, such as line 6). We refer to this process of repairing type errors by inserting monadic bindings and returns as lifting.\(^6\)

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

\(^6\)A reader familiar with monads might think of the array-oblivious program as written in the identity monad instead of the list monad, and lifting as lifting the program back into the list monad.
returns in order to fix the mismatch. This is always possible because the only kind of type mismatch we can encounter is between an actual type $[\.\.f\.\.]$ and the expected type $\tilde{i}$, 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 in lines 4 and 6. An alternative would be to iterate over $x_1$ only once (line 3), and reuse the same “iterator” variable $x_1$.

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 $E$, 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 $E$ based on its result set $res$ and the return type $\tilde{i}$ of the query as follows:

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

\begin{figure}[h]
\centering
\begin{align*}
\text{Retrospective Execution} & \quad \langle W; \Gamma; \Sigma \mid e \rangle \Rightarrow \nu \\
E-If-True-L & \quad \langle W; \Gamma; \Sigma \mid 1 \rangle \Rightarrow \nu_1 \\
E-If-True-R & \quad \langle W; \Gamma; \Sigma \mid if \ x_1 = x_2; e \rangle \Rightarrow \nu_2 \\
E-Method-Val & \quad \langle W; \Gamma; \Sigma \mid f \rangle \Rightarrow \nu_0 \in W \\
E-Method-Name & \quad \langle W; \Gamma; \Sigma \mid f \rangle \Rightarrow \nu_0 \in W
\end{align*}
\caption{Retrospective execution.}
\end{figure}

\textit{Retrospective execution.} We formalize RE as a judgement $\langle W; \Gamma; \Sigma \mid e \rangle \Rightarrow \nu$, stating that $\nu$ is a valid result for executing the expression $e$ in the environment $\Sigma$ (which maps variables to values). The judgment is also parameterized by a type context $\Gamma$ and witness set $W$, used to replay method calls and sample program inputs. To run a candidate solution $E$, we execute its body in an empty $\Sigma$ and with $\Gamma$ storing the types of $E$’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.

\textit{Replaying method calls.} Most of the rules for the RE judgement describe standard big-step operational semantics (they can be found in the technical report [12]), but two groups of rules, shown in Fig. 12, deserve more attention. The first group of interest includes $E$-Method-VAL and $E$-Method-NAME, which replay a method call by looking it up in $W$. The rule $E$-Method-VAL applies when $W$ 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 $f(\tilde{l} = \nu_0)$, there might be multiple approximate matches in $W$, 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 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 $\Sigma$ 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 $W$.

### Evaluation

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

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

(RQ1) Can APIphany find solutions for a wide range of realistic tasks across multiple popular APIs?

(RQ2) Is type mining effective and necessary for enabling type-directed synthesis?

(RQ3) Is retrospective execution effective and necessary for prioritizing relevant synthesis results?

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

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

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

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

### Experiment setup: program synthesis.

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

#### 7.1 RQ1: Overall Effectiveness

The last four columns of Tab. 2 detail APIphany’s performance on the 32 synthesis benchmarks. APIphany finds the
Table 2. Synthesis benchmarks and results. Benchmarks marked with † are effective. 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 \(r_{\text{orig}}\), and the lower and upper bound on its rank with RE \(r_{\text{RE}}\) and \(r_{\text{TO}}\). ‘-’ means no solution is found in 150 seconds.

<table>
<thead>
<tr>
<th>API</th>
<th>ID</th>
<th>Solution Size</th>
<th>Time</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Solution Size</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AST</td>
<td>(n_f)</td>
<td>(n_p)</td>
</tr>
<tr>
<td>Slack</td>
<td>1.1</td>
<td>17</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>1.2†</td>
<td>12</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>16</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>14</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1.5†</td>
<td>10</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1.6†</td>
<td>9</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1.7†</td>
<td>12</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1.8</td>
<td>9</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Stripe</td>
<td>2.1†</td>
<td>9</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2.2†</td>
<td>10</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2.3†</td>
<td>12</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2.4</td>
<td>8</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2.5</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2.6†</td>
<td>9</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2.7</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2.8</td>
<td>16</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2.9</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2.10†</td>
<td>10</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2.11†</td>
<td>7</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2.12†</td>
<td>11</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2.13†</td>
<td>10</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Square</td>
<td>3.1</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3.2</td>
<td>16</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3.3</td>
<td>10</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3.4</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3.5†</td>
<td>14</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3.6</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3.7</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3.8</td>
<td>9</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3.9</td>
<td>8</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3.10†</td>
<td>16</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>3.11†</td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 13. Comparison of synthesis performance between APIphany and its two variants that do not use type mining. Average APIphany takes 17.8 seconds to find the desired solution (median time 1.3 seconds).

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

7.2 RQ2: Type Mining
Recall that type mining involves replacing primitive syntactic types in the spec with unique location-based types, and then merging those based on the witness set to obtain semantic types. The merging process is not perfect: it might fail to merge two location that should have the same type because the witness set lacks evidence to justify the merge; or it might spuriously merge two locations if they share a value by chance. It is hard to measure the accuracy of inferred types directly, since we do not have an oracle for semantic types. Instead, we evaluate type mining indirectly in two ways: 1) we run an ablation study to measure its impact on the overall performance of the synthesizer, and 2) we perform a small-scale qualitative analysis of inferred types.

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

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

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

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

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

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

![Figure 14](image)

**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.

**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: \(r_{\text{org}}\) 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); \(r_{\text{RE}}\) denotes the RE-based rank of the solution at the time it was generated, and \(r_{\text{TO}}\) denotes its RE-based rank by the timeout (which can be lower than \(r_{\text{RE}}\) 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 \(r_{\text{RE}}\) and \(r_{\text{TO}}\).

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.
We ran our experiments using a particular witness set, which APIphany most likely would not be able to solve them, since they would be ill-typed with inferred semantic types. We ran our experiments using a particular witness set, which we collected using one methodology (described in the technical report [12]); our findings might not generalize to using APIphany with witness sets collected by other means.

Effectful methods. We observe that effectful methods in REST APIs have an interesting property: they make the effect explicit in their response. For example, the method for posting a message on SLACK also returns the message object, and the method for deleting a catalog item in SQUARE returns the ID of the deleted item (instead of just returning void). This property makes REST APIs particularly suitable for type-directed synthesis and expressing user intent with types: for example, the query “Send a message to a user with a given email” can be expressed as the type Profile.email ➞ Message instead of a much less informative type Profile.email ➞ void. The downsize, 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, 16, 25] scales poorly with the size of the component library. An alternative, more scalable graph-based approach was introduced in PROSPECTOR [22] for unary components, and generalized to n-ary components in SYPET [8], by replacing graphs with Petri nets. TYGAR [13] further extends SYPET’s search to polymorphic components using the idea of abstract types, which are inspired by succinct types from another component-based synthesizer, INSYNTX [15]. APIphany’s program search phase is using the Petri net encoding from SYPET and TYGAR with minor adaptations (support for optional arguments and ILP encoding). Our array-oblivious encoding is related to abstract and succinct types in that it helps make the Petri net smaller, but it is also substantially different in that, unlike prior work, it can efficiently encode a certain class of higher-order programs (array comprehensions) into the Petri net.

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

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

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

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

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

**Ranking solutions.** Specifications in program synthesis are often ambiguous, so synthesizers have to rank their candidate solutions and return the top result(s). Existing tools most commonly rely on hand-crafted [11] or learned [15, 28, 30] ranking functions based on syntactic features of generated programs. Hoogle++ 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**


