RbSyn: Type- and Effect-Guided Program Synthesis

Sankha Narayan Guria
University of Maryland
College Park, Maryland, USA
sankha@cs.umd.edu

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

David Van Horn
University of Maryland
College Park, Maryland, USA
dvanhorn@cs.umd.edu

Abstract

In recent years, researchers have explored component-based synthesis, which aims to automatically construct programs that operate by composing calls to existing APIs. However, prior work has not considered efficient synthesis of methods with side effects, e.g., web app methods that update a database. In this paper, we introduce RbSyn, a novel type- and effect-guided synthesis tool for Ruby. An RbSyn synthesis goal is specified as the type for the target method and a series of test cases it must pass. RbSyn works by recursively generating well-typed candidate method bodies whose write effects match the read effects of the test case assertions. After finding a set of candidates that separately satisfy each test, RbSyn synthesizes a solution that branches to execute the correct candidate code under the appropriate conditions. We formalize RbSyn on a core, object-oriented language λ_{syn} and describe how the key ideas of the model are scaled-up in our implementation for Ruby. We evaluated RbSyn on 19 benchmarks, 12 of which come from popular, open-source Ruby apps. We found that RbSyn synthesizes correct solutions for all benchmarks, with 15 benchmarks synthesizing in under 9 seconds, while the slowest benchmark takes 83 seconds. Using observed reads to guide synthesize is effective: using type-guidance alone times out on 10 of 12 app benchmarks. We also found that using less precise effect annotations leads to worse synthesis performance. In summary, we believe type- and effect-guided synthesis is an important step forward in synthesis of effectful methods from test cases.

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

Keywords: program synthesis, type and effect systems, Ruby

ACM Reference Format:
Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2021. RbSyn: Type- and Effect-Guided Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454048

1 Introduction

A key task in modern software development is writing code that composes calls to existing APIs, such as from a library or framework. *Component-based synthesis* aims to carry out this task automatically, and researchers have shown how to perform component-based synthesis using SMT solvers [26]; how to synthesize branch conditions [31]; and how to perform synthesis given a very large number of components [12].

This prior work guides the synthesis process using types or special properties of the synthesis domain, which is critical to achieving good performance. However, prior work does not explicitly consider *side effects*, which are pervasive in many domains. For example, consider synthesizing a method that updates a database. Without reasoning about effects—in this case, that the method body needs to change the database—synthesis of such a method reduces to brute-force search, limiting its performance.

In this paper, we address this issue by introducing RbSyn, a new tool for synthesizing Ruby methods. In RbSyn, the user specifies the desired method by its type signature and a series of test cases it must pass. RbSyn then searches for a solution by enumerating candidates and checking them against the tests. The key novelty of RbSyn is that the search is both *type- and effect-guided*. Specifically, the search begins with a *typed hole* tagged with the method's return type. Each step either replaces a typed hole with an expression of that type, possibly introducing more typed holes; inserts an *effect hole*, annotated with a write effect that may be needed to satisfy a test assertion; or replaces an effect hole with an expression with the given write effect, possibly inserting another effect hole. Once this process finds a set of method bodies that cumulatively pass all tests, RbSyn uses a novel merging strategy to construct a complete solution: It creates a method whose body branches among the conditions, executing the corresponding (passing) code, thus yielding a single method that passes all tests. (§ 2 gives a complete example of RbSyn's synthesis process.)

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454048

We formalize RbSyn for λ_{syn}, a core object-oriented language. The synthesis algorithm is comprised of three parts. The first part, type-guided synthesis, is similar to prior work [16, 30, 32], but is geared towards imperative, object-oriented programs. The second part is *effect-guided synthesis*, which tries to fill an effect hole $\diamond : \epsilon$ with an expression with effect ϵ. In λ_{syn}, an effect accesses a *region* $A.r$, where A is a class and r is an uninterpreted identifier. For example, `Post.author` might indicate reading instance field `author` of class `Post`. This notion of effects balances precision and tractability: effects are precise enough to guide synthesis effectively, yet coarse enough that reasoning about them is simple. The last part of the synthesis algorithm synthesizes branch conditions to create a *merged* program that combines solutions for individual tests into an overall solution for the complete problem. (§ 3 discusses our formalism.)

Our implementation of RbSyn is built on top of RDL, a Ruby type system [15]. Our implementation extends RDL to include effect annotations, including a `self` region to give more precise effect information in the presence of inheritance. Our implementation also makes use of RDL's *type-level computations* [27] to provide precise typing during synthesis. Finally, when searching for solutions, our implementation heuristically prioritizes further exploration of candidates that are small and have passed more assertions. (§ 4 describes our implementation.)

We evaluated RbSyn on a suite of 19 benchmarks, including seven benchmarks we wrote and 12 benchmarks extracted from three widely used, open-source Ruby apps: Discourse, Gitlab, and Disaspora. For the former, we wrote our own specifications. For the latter, we used unit tests that came with the benchmarks. We found that RbSyn synthesizes correct solutions for all benchmarks and does so quickly, taking less than 9 seconds each for 15 of the benchmarks, and 83 seconds for the slowest benchmark. Moreover, type- and effect-guidance is critical. Without it, a majority of the benchmarks time out after five minutes. Finally, we examine the tradeoff of effect precision versus performance. We found that restricting effects to class names only causes 3 benchmarks to time out, and restricting effects to only purity/impurity causes 10 benchmarks to time out. (§ 5 discusses the evaluation in detail.)

We believe that RbSyn is an important step forward in synthesis of effectful methods from test cases.

2 Overview

In this section, we illustrate RbSyn by using it to synthesize a method from a hypothetical web blogging app. This app makes heavy use of ActiveRecord, a popular database access library for Ruby on Rails. It is the ActiveRecord methods whose side effects RbSyn uses to guide synthesis.

```
1  # User schema {name: Str, username: Str}
2  # Post schema {author: Str, title: Str, slug: Str}
3
4  define :update_post, "(Str, Str, {author: ?Str, title:
5      ?Str, slug: ?Str}) → Post", [User, Post] do
6    spec "author can only change titles" do
7      setup {
8        seed_db # add some users and their posts to db
9        @post = Post.create(author: 'author', slug:
10         'hello-world', title: 'Hello World')
11       update_post('author', 'hello-world', author:
12         'dummy', title: 'Foo Bar', slug: 'foobar')
13     }
14     postcond { |updated|
15       assert { updated.id == @post.id }
16       assert { updated.author == "author" }
17       assert { updated.title == "Foo Bar" }
18       assert { updated.slug == 'hello-world' }
19     }
20   end
21   spec "other users cannot change anything" do
22     setup { ... # same setup as above except next line
23       update_post('dummy', ...) # other args same
24     }
25     postcond { |updated| ... # same other three asserts
26       assert { updated.title == "Hello World" }
27     }
28 end end
```

Figure 1. Specification for `update_post` method

Figure 1 shows the synthesis problem. This particular app includes database tables for users and posts. In ActiveRecord, rows of these tables are represented as instances of classes `User` and `Post`, respectively. For reference, the table schemas are shown in lines 1 and 2. Each user has a `name` and `username`. Each post has the `author`'s username, the post's `title`, and a `slug`, used to compute a permalink.

The goal of this particular synthesis problem, given by the call to `define`, is to create a method `update_post` that allows users to change the information about a post. Lines 4 and 5 specify the method's type signature in the format of RDL [15], a Ruby type system that RbSyn uses for types and type checking. Here, the first two arguments are strings, and the last is a *finite hash type* that describes an instance of `Hash` with optional (indicated by ?) keys `author`, `title`, and `slug` (all *symbols*, which are just interned strings) that map to strings. The method itself returns a `Post`.

In addition to the type signature, the synthesis problem also includes a list of constants that can be used in the target method. In this case, those constants are the classes `User` and `Post`, as given by the last argument to `define` on line 5. These classes can then be used to invoke singleton (class) methods in the synthesized method. For simplicity, we assume that RbSyn can use only these constants for this example. In practice, RbSyn can synthesize predefined numeric or string constants like 0, 1 or the empty string.

Finally, the synthesis problem includes a number of *specs*, which are just test cases. Each spec has a title, for human convenience; a `setup` block to establish any necessary preconditions and call the synthesized method; and a `postcond` block with assertions that must hold after the synthesized method runs. As we will see below, separating the pre- and postconditions allows RbSyn to more easily use effects to guide synthesis. In this example, both specs add a few users and a post created by each of them to the database (call to `seed_db`, details not shown) and then create a post titled "Hello World" by the user `author`. The first spec asserts that `update_post` allows `author` to update a post's title. The second spec asserts that a `dummy` user cannot update the post. The check for `id` ensures that only existing posts are updated (any new posts will have a new unique id).

The final, synthesized solution is shown on the right of Figure 2. Notice the synthesized code calls several ActiveRecord methods (`exists?`, `where`, and `first`) as well as the hash access method `[]`. Applying solver-aided synthesis to this problem would require developing accurate models of these methods, which is a difficult challenge [29]. To address this limitation, RbSyn instead enumerates candidates, which can then be run to check them against the specs. As the search space is vast, RbSyn uses `update_post`'s type signature and the effects from the specs' `postcond`s the guide the search. Finally, RbSyn uses a novel merging algorithm to synthesize the necessary branch condition to yield a solution that satisfies both specs.

2.1 Synthesizing Spec Solutions

The left portion of Figure 2 shows the search process RbSyn uses to solve this synthesis problem. To begin, RbSyn observes that the return type of `update_post` is `Post`. Thus, the search begins (upper left) by creating a candidate method body □:`Post`, which is a *typed hole* that must be filled by an expression of type `Post`. RbSyn then iteratively expands holes in candidates, running the specs whenever it produces fully concretized candidates with no holes.

In general, RbSyn can fill a typed hole with a local variable, a constant, or a method call. As there are no local variables (which so far are just parameters) or constants of the appropriate type, RbSyn chooses a method call. To do so, it searches through the available method type annotations to find those that could return `Post`. In this case, RbSyn takes advantage of RDL's type annotations for ActiveRecord [27] to synthesize candidates C1 and C2, among others (not shown). It is straightforward for the user to add type annotations for any other library methods that might be needed by the synthesized method. For illustration purposes, we also show a candidate C3 that returns the wrong type. Such candidates are discarded by RbSyn, vastly reducing the search space. Note that C2 contains two method calls, and thus would take two steps to produce, but we show it here as a single step for conciseness.

Next, RbSyn tries to fill holes in candidate expressions, starting with smaller candidates. In this case, it first considers C1, which has a hole of type `Class<Post>`, which is the singleton type for the constant `Post`. Thus, there is only one choice for the hole, yielding candidate C4. Since C4 has no holes, RbSyn runs it against the specs. More specifically, it runs it against the first spec—as we will discuss shortly, RbSyn synthesizes solutions for each spec independently, and then combines them. In this case, C4 fails the spec (because the first post in the database is not the one to be updated, due to the initial database seeding) and hence is rejected.

Continuing with C5, RbSyn fills in the (finite hash-typed) hole, yielding choices that include C6 and C7. RbSyn rejects C6 since there is no way to construct an expression of type `Int`. However, for C7, there are two local variables of type `Str` from the method arguments. Substituting these yields C8 and C9. C8 uses `arg0`, the username, to query the `Post` table's slug, so it fails. C9 queries the `Post` table with the correct slug value `arg1`. This passes the first two assertions (line 15 onwards) but fails the third, which expects the post title to be updated from "Hello World" to "Foo Bar."

RbSyn extends RDL's type annotations to include read and write effects. When the expression inside an `assert` evaluates to `false`, RbSyn infers the `assert`'s read and write effects based on those of the methods it calls. For example, we can give the `Post#title`[1] method, used by the third assertion, the following signature:

```
type Post, :title, '() → Str', read: ['Post.title']
```

Thus, RbSyn sees that the failing assertion reads `Post.title`, an abstract effect label. To make the assertion succeed, RbSyn inserts an *effect hole* ◇ : `Post.title` in the candidate program (C10). It also saves the value of the previous candidate expression in a temporary variable, and inserts a hole with the candidate's type at the end. RbSyn then continues the search, trying to fill the effect hole with a call to a method whose *write* effect matches the hole—such a call could potentially satisfy the failed assertion. Here, RbSyn replaces the effect hole (C11) with a call to `Post#title=`, which is such a method. (We should note that all previous candidates that failed a spec due to a side effect will also have effect holes added in a similar fashion. We omit these candidates from the discussion as they do not lead to a solution.)

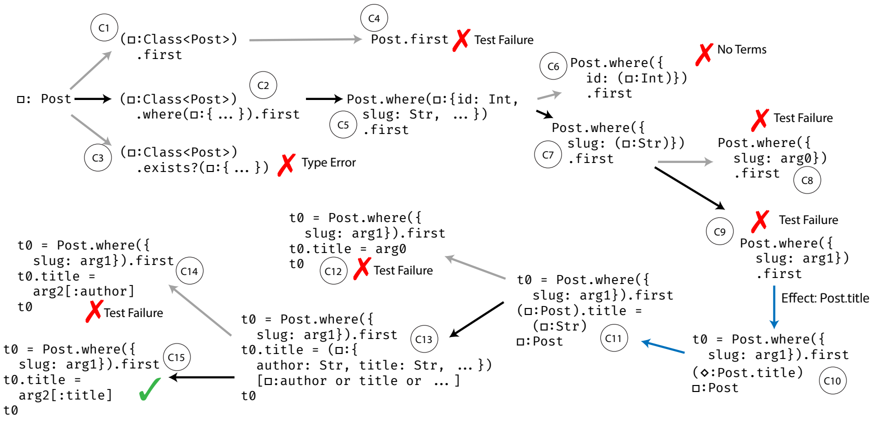
RbSyn continues by using type-guided synthesis for the typed holes of C11—yielding C12, rejected due to assertion failures—and then C13. After several steps (not shown), RbSyn arrives at C14, which fails the spec, and C15, which fully satisfies the first spec. Indeed, we see this exact expression in lines 4–6 of the solution in Figure 2.

2.2 Merging Solutions

RbSyn next uses the same technique to synthesize an expression that satisfies the second spec, yielding the expression

---

[1] `A#m` indicates instance method `m` of class `A`.



```
1  def update_post(arg0, arg1, arg2)
2    if Post.exists?(author: arg0,
3        slug: arg1)
4      t0 = Post.where(slug:arg1).first
5      t0.title=arg2[:title]
6      t0
7    else
8      Post.where(slug: arg1).first
9    end
10 end
```

Figure 2. *Left:* Steps in the synthesis of solution to the first specification. Note C2 takes two steps to synthesize but is shown as a single composite step. Some choices available to the synthesis algorithm have been omitted for simplicity. *Right:* Synthesized `update_post` method.

shown on line 8. Now RbSyn needs to merge these individual solutions into a single solution that passes all specs. At a high-level, it does so by constructing a program `if` b_1 `then` e_1 `else if` b_2 `then` e_2 `end`, where the e_i are the solutions for the specs and the b_i are *branch conditions* capturing the conditions under which those expressions pass the specs.

To create the b_i, RbSyn uses the same technique again, this time synthesizing a boolean-valued expression that evaluates to `true` under the `setup` of spec i. In this case, this process results in the same branch condition `true` for both specs. However, since this trivially holds for both specs, this branch condition does not work—we need to find a branch condition that distinguishes the two cases.

Next RbSyn tries to synthesize a branch condition b_1' that evaluates to `true` for the `setup` of the first spec and `false` for the `setup` of the second. This yields the more precise branch condition b_1' = `Post.exists?(author: arg0, slug: arg1)`. This is a sufficient condition, as the `update_post` method is supposed to update a post only if a post with slug `arg1` is authored by `arg0`. It solves an analogous synthesis problem for the second spec, yielding b_2' = `!Post.exists?(author: arg0, slug: arg1)`. As these are the negation of each other, RbSyn then merges these two together as `if-then-else` (rather than an `if-then-else if-then-else`), yielding the final synthesized program in Figure 2.

$$
\begin{array}{llll}
\textit{Values} & v & ::= & \texttt{nil} \mid \texttt{true} \mid \texttt{false} \mid [A] \\
\textit{Expressions} & e & ::= & v \mid x \mid e;e \mid e.m(e) \\
 & & \mid & \textbf{if}\ b\ \textbf{then}\ e\ \textbf{else}\ e \\
 & & \mid & \textbf{let}\ x = e\ \textbf{in}\ e \mid \square : \tau \mid \diamond : \epsilon \\
\textit{Conditionals} & b & ::= & e \mid !b \mid b \vee b \\
\textit{Types} & \tau & ::= & A \mid \tau \cup \tau \\
\textit{Programs} & P & ::= & \textbf{def}\ m(x) = e \\
\\
\textit{Specs} & s & ::= & \langle S, Q \rangle \\
\textit{Setup} & S & ::= & e;\ x_r = P(e) \\
\textit{Postconditions} & Q & ::= & \textbf{assert}\ e \mid Q;Q \\
\textit{Spec Set} & \Psi & := & \{s_i\} \\
\textit{Synthesis Goal} & G & ::= & \langle \tau \to \tau, \Psi \rangle \\
\\
\textit{Class Table} & CT & ::= & \emptyset \mid A.m : \sigma, CT \\
\textit{Method Types} & \sigma & ::= & \tau \xrightarrow{\langle \epsilon_r, \epsilon_w \rangle} \tau \\
\textit{Type Env.} & \Gamma & ::= & \emptyset \mid x : \tau, \Gamma \\
\textit{Dynamic Env.} & E & ::= & x \to v \\
\textit{Constants} & \Sigma & ::= & \emptyset \mid v : \tau, \Sigma \\
\textit{Effect} & \epsilon & ::= & \bullet \mid * \mid A.* \mid A.r \mid \epsilon \cup \epsilon
\end{array}
$$

$$
\begin{gathered}
r \in \text{effect regions} \quad \bullet \subseteq \epsilon \quad \epsilon \subseteq * \\
A_1.* \subseteq A_2.* \text{ and } A_1.r \subseteq A_2.r \text{ and } A_1.r \subseteq A_2.* \text{ if } A_1 \leq A_2 \\
\epsilon^1 \subseteq \epsilon^1 \cup \epsilon^2 \quad \epsilon^2 \subseteq \epsilon^1 \cup \epsilon^2 \\
\langle \epsilon_r^1, \epsilon_w^1 \rangle \cup \langle \epsilon_r^2, \epsilon_w^2 \rangle = \langle \epsilon_r^1 \cup \epsilon_r^2, \epsilon_w^1 \cup \epsilon_w^2 \rangle
\end{gathered}
$$

$$
\begin{gathered}
x \in \text{variables},\ m \in \text{methods},\ A \in \text{classes}, \\
\mathit{Nil} \leq \tau \quad \tau \leq \mathit{Obj} \quad \tau_1 \leq \tau_1 \cup \tau_2 \quad \tau_2 \leq \tau_1 \cup \tau_2
\end{gathered}
$$

Figure 3. Syntax and Relations of λ_{syn}.

3 Formalism

In this section, we formalize RbSyn on λ_{syn}, a core object-oriented calculus shown in Figure 3. Values v include `nil`, `true`, `false`, and objects $[A]$ of class A. Note that we omit fields to keep the presentation simpler. Expressions e include values, variables x, sequences $e;e$, method calls $e.m(e)$, conditionals **if** b **then** e **else** e, and variable bindings **let** $x = e$ **in** e. A conditional guard b can be an expression e, a negation $!b$, or a disjunction $b \vee b$. The grammar for guards is limited to match what RbSyn can actually synthesize.

Expressions also include *typed holes* $\square : \tau$ and *effect holes* $\diamond : \epsilon$, which are placeholders that are eventually filled with an expression of the given type, or expression with the given write effect, respectively. We note our synthesis algorithm only inserts effect holes at positions that can have any type. Types are either classes or unions of types, and we assume classes form a lattice with *Nil* (the class of `nil`) as the bottom element and *Obj* as the top element. We write $A \leq B$ when class A is a subclass of B according to the lattice. We defer the definition of effects for the moment. Finally, a synthesized program P is a single method definition $\textbf{def}\ m(x) = e$. We restrict the method to one argument for convenience.

A spec s in λ_{syn} is a pair of setup code S and a postcondition Q. A setup $e_1; x_r = P(e_2)$ includes some initialization e_1 followed by a special form indicating calling the synthesized method in P with argument e_2 and binding the result to x_r. The postcondition is a sequence of assertions that can test x_r and inspect the global state using library methods. We write Ψ for a set of specs, and a *synthesis goal* G is a pair $\langle \tau_1 \rightarrow \tau_2, \Psi \rangle$, where τ_1 and τ_2 are the method's domain and range types, respectively, and Ψ are the specs the synthesized method should satisfy.

The next part of Figure 3 defines additional notation used in the formalism. Synthesized methods can use classes and methods from a *class table CT*, which maps class and method names to the methods' types. For example, the class table has type information for other methods of a target app and library methods such as those from ActiveRecord. A method type σ has the form $\tau \xrightarrow{\langle \epsilon_r, \epsilon_w \rangle} \tau'$, where τ and τ' are the domain and range types, respectively, and $\langle \epsilon_r, \epsilon_w \rangle$ specifies the method's read effect ϵ_r and write effect ϵ_w (discussed shortly). During type-guided synthesis, RbSyn maintains a type environment Γ mapping variables to their types. When executing a synthesized program, the operational semantics (omitted) uses a dynamic environment E mapping variables to their values. During synthesis, Σ is a list of user-supplied constants that can fill holes.

Effects. The last part of Figure 3 defines effects ϵ. In RbSyn, effects are hierarchical names that abstractly label the program state. The empty effect $\bullet$ denotes no side effect, used for pure computations. The effect $*$ is the top effect, indicating a computation that might touch any state in the program. Lastly, effect $A.*$ denotes code that touches any state within class A, and $A.r$ denotes code that touches the region labeled r in A, where region names are completely abstract. Effects can also be unioned together.

We define subsumption $\epsilon_1 \subseteq \epsilon_2$ on effects to hold when ϵ_2 includes ϵ_1. Effects $\bullet$ and $*$ are the bottom and top, respectively, of the $\subseteq$ relation, and if $A_1 \leq A_2$ then $A_1.r \subseteq A_2.r$ and $A_1.r \subseteq A_2.*$ and $A_1.* \subseteq A_2.*$. We also have standard rules for subsumption with effect unions.

In RbSyn, all effects arise from calling methods from the class table CT, which have effect annotations of the form $\langle \epsilon_r, \epsilon_w \rangle$, where ϵ_r and ϵ_w are the method's read and write effects, respectively. We extend subsumption to such paired effects in the natural way. During synthesis, if RbSyn observes the failure of an assertion with some read effect ϵ_r, it tries to fix the failure by inserting a call to some method with write effect ϵ_w such that $\epsilon_r \subseteq \epsilon_w$, i.e., it tries writing to the state that is read. For example, in Section 2, this technique generated a call to `Post#title`.

Our effect language is inspired by the region path lists approach of Bocchino Jr et al. [4], but is much simpler. We opted for coarse-grained, abstract effects to make it easier to write annotations for library methods. Although class names are included in the effect language, such names are for human convenience only—nothing precludes a method in class A being annotated with an effect to $B.r$ for some other class B. We found that this approach works well for our problem setting of synthesizing code for Ruby apps, where trying to precisely model heap and database state would be difficult. However, we believe the core of this approach—pairing effects (in our case, reads and writes) and then creating candidates using the opposing element of such a pair—can be generalized to more complex effect systems.

Synthesis Problem. We can now formally specify the synthesis problem. Given a synthesis goal $\langle \tau_1 \rightarrow \tau_2, \{\langle S_i, Q_i \rangle\} \rangle$, RbSyn searches for a program P such that, for all i, assuming that S_i calls P with an argument of type τ_1, evaluating to x_r of type τ_2, it is the case that $P \vdash S_i; Q_i \Downarrow v$. In other words, evaluating the setup followed by the postcondition yields some value rather than aborting with a failed assertion. We omit the evaluation rules as they are standard.

3.1 Type-Guided Synthesis

The first component of RbSyn is type-guided synthesis, which creates candidate expressions of a given type by trying to fill a hole $\square : \tau_2$ where τ_2 is the method return type. Figure 4 shows a subset of the type-guided synthesis rules; the full set can be found in the companion technical report [22]. These rules have the form $\Sigma, \Gamma \vdash_{CT} e_1 \rightsquigarrow e_2 : \tau$, meaning with constants Σ, in type environment Γ, under class table CT, the holes in e_1 can be rewritten to yield e_2, which has type τ.

The rules in Figure 4 have two forms. The T- rules apply to expressions whose outermost form is not rewritten. Thus these rules perform standard type checking. For example, T-Var type checks a variable x by checking its type against the type environment Γ, leaving the term unchanged. T-Let type-checks and recursively rewrites (or not) the subexpressions and then rewrites those new expressions into a let-binding, ensuring the resulting term is type-correct. Finally, T-Hole applies to a typed hole that is not being rewritten, in which case it remains the same and has the given type.

The S- rules rewrite typed holes. S-Const replaces a hole by a constant of the correct type from Σ. S-Var is similar,

$$\boxed{\Sigma, \Gamma \vdash_{CT} e \rightsquigarrow e : \tau}$$

$$\frac{\Gamma(x) = \tau}{\Sigma, \Gamma \vdash_{CT} x \rightsquigarrow x : \tau}\ \textsc{T-Var}$$

$$\frac{\Sigma, \Gamma \vdash_{CT} e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Sigma, \Gamma[x \mapsto \tau_1] \vdash_{CT} e_2 \rightsquigarrow e_2' : \tau_2}{\Sigma, \Gamma \vdash_{CT} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rightsquigarrow \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2' : \tau_2}\ \textsc{T-Let}$$

$$\frac{}{\Sigma, \Gamma \vdash_{CT} \square : \tau \rightsquigarrow (\square : \tau) : \tau}\ \textsc{T-Hole}$$

$$\frac{v : \tau_1 \in \Sigma \qquad \tau_1 \leq \tau_2}{\Sigma, \Gamma \vdash_{CT} \square : \tau_2 \rightsquigarrow v : \tau_1}\ \textsc{S-Const}$$

$$\frac{\Gamma(x) = \tau_1 \qquad \tau_1 \leq \tau_2}{\Sigma, \Gamma \vdash_{CT} \square : \tau_2 \rightsquigarrow x : \tau_1}\ \textsc{S-Var}$$

$$\frac{m : \tau_1 \rightarrow \tau_2 \in CT(A) \qquad \tau_2 \leq \tau_3}{\Sigma, \Gamma \vdash_{CT} \square : \tau_3 \rightsquigarrow (\square : A).m(\square : \tau_1) : \tau_2}\ \textsc{S-App}$$

Figure 4. Type-guided synthesis rules (selected).

replacing a hole by a variable from Γ. Finally, S-App replaces a hole with a call to a method with the right return type, inserting typed holes for the method receiver and argument.

Type Narrowing. Notice that in these three S-rules, the term replacing the hole may actually have a subtype of the original hole's type. Thus, type-guided synthesis could *narrow* types in a synthesized program, potentially also narrowing the search space. For example, consider an expression $(\square_1 : \texttt{Str})$.append$(\square_2 : \texttt{Str})$ that joins two strings, and assume the set of constants Σ includes `nil`. Notice that `nil` is a valid substitution for $\square_1$, which will then cause the type of the receiver to narrow to *Nil*. But then the typing derivation fails because the *Nil* type has no append method, stopping further exploration along this path. In contrast, if we had typed the replacement term at `Str`, then RbSyn would have fruitlessly continued the search, trying various replacements for $\square_2$ only to reject them due to a runtime failure for invoking a method on `nil`.

3.2 Effect-Guided Synthesis

The second component of RbSyn is effect-guided synthesis, used when type-guided synthesis creates a candidate that does not satisfy the postcondition of the tests. If this happens, RbSyn computes the effect $\langle \epsilon_r, \epsilon_w \rangle$ of the failed assertion in the postcondition. (We defer the formal rules for computing this effect to the technical report [22], as they simply union the effects of method calls in the assertion.) Then, we hypothesize that the assertion may have failed because the region denoted by ϵ_r is in the wrong state.

To potentially fix the state, RbSyn applies a new rule S-Eff, shown in Figure 5. The hypothesis computes the type τ of e, the candidate expression that failed the postcondition. In the conclusion, e is rewritten to $\mathbf{let}\ x = e\ \mathbf{in}\ (\Diamond : \epsilon_r; \square : \tau)$, i.e., e is computed, bound to x, and two holes are sequenced. The first must be filled with an expression of the desired effect ϵ_r. The second must have e's type τ, to preserve type-correctness. For example, it could be filled by x, as happened in Figure 2 when `t0` is returned.

$$\boxed{\Sigma, \Gamma, \epsilon_r \vdash_{CT} e \twoheadrightarrow e}$$

$$\frac{\Sigma, \Gamma \vdash_{CT} e \rightsquigarrow e : \tau}{\Sigma, \Gamma, \epsilon_r \vdash_{CT} e \twoheadrightarrow \mathbf{let}\ x = e\ \mathbf{in}\ (\Diamond : \epsilon_r; \square : \tau)}\ \textsc{S-Eff}$$

$$\boxed{\Sigma, \Gamma \vdash_{CT} e \rightsquigarrow e : \tau}$$

$$\frac{}{\Sigma, \Gamma \vdash_{CT} \Diamond : \epsilon \rightsquigarrow (\Diamond : \epsilon) : \mathit{Obj}}\ \textsc{T-EffObj}$$

$$\frac{\epsilon_r \subseteq \epsilon_w' \qquad m : \tau_1 \xrightarrow{\langle \epsilon_r', \epsilon_w' \rangle} \tau_2 \in CT(A)}{\Sigma, \Gamma \vdash_{CT} \Diamond : \epsilon_r \rightsquigarrow \square : \epsilon_r'; (\square : A).m(\square : \tau_1) : \tau_2}\ \textsc{S-EffApp}$$

$$\frac{}{\Sigma, \Gamma \vdash_{CT} \Diamond : \epsilon \rightsquigarrow \texttt{nil} : \mathit{Nil}}\ \textsc{S-EffNil}$$

Figure 5. Effect guided synthesis rule

The rules for working with effect holes are shown in the bottom of Figure 5, which extends Figure 4. T-EffObj gives an effect hole, that is not rewritten, type *Obj*. Since this is the top of the type hierarchy, this ensures an effect hole can safely be replaced by a term with any type. In other words, effect holes are filled for their effects, not their types. S-EffApp does the heavy lifting, filling an effect hole with a call to a method m with a write effect ϵ_w' that subsumes the desired effect ϵ_r. Of course, this call may itself read state ϵ_r', so the rule precedes the method call with a hole with that effect, in case said state needs to change. Finally, S-EffNil replaces an effect hole with `nil`, which removes it from the program. This is used in case some extra effect holes are added that are not actually needed.

3.3 Merging Solutions

The last component of RbSyn combines expressions that pass individual specs into a final program that passes all specs. More specifically, given a synthesis goal $\langle \tau_1 \rightarrow \tau_2, \{s_i\} \rangle$, RbSyn first uses type- and effect-guided synthesis to create expressions e_i such that e_i is the solution for spec s_i. Then, RbSyn combines the e_i into a branching program roughly of the form $\mathbf{if}\ b_1\ \mathbf{then}\ e_1\ \mathbf{else}\ \mathbf{if}\ b_2\ \mathbf{then}\ e_2 \ldots$ for some b_i.

For each i, RbSyn uses the type-guided synthesis rules in § 3.1 to synthesize a b_i such that under the setup S_i of spec s_i, conditional b_i evaluates to `true`, i.e., $\mathbf{def}\ m(x) = b_i \vdash S_i; \mathbf{assert}\ x_r \Downarrow v$. Note effect-guided synthesis is not used here as the asserted expression x_r is pure.

$$\langle e_1, b_1, \Psi_1\rangle \oplus \langle e_2, b_2, \Psi_2\rangle = \langle e_1, b_1, \Psi_1 \cup \Psi_2\rangle \quad \text{if } e_1 \equiv e_2 \text{ and } b_1 \implies b_2 \tag{1}$$

$$\langle e_1, b_1, \Psi_1\rangle \oplus \langle e_2, b_2, \Psi_2\rangle = \langle e_1, b_1 \vee b_2, \Psi_1 \cup \Psi_2\rangle \quad \text{if } e_1 \equiv e_2 \text{ and } b_1 \not\Longrightarrow b_2 \tag{2}$$

$$\begin{aligned} \langle e_1, b_1, \Psi_1\rangle \oplus \langle e_2, b_2, \Psi_2\rangle &= \langle e_1, b_1^{syn}, \Psi_1\rangle \oplus \langle e_2, b_2^{syn}, \Psi_2\rangle \\ &\qquad \text{if } e_1 \not\equiv e_2 \text{ and } b_1 \implies b_2 \\ \text{where } \forall\langle S_i, Q_i\rangle \in \Psi_1.&\mathbf{def}\ m(x) = b_1^{syn} \vdash S_i; \mathbf{assert}\ x_r \Downarrow v \\ \wedge \quad \forall\langle S_j, Q_j\rangle \in \Psi_2.&\mathbf{def}\ m(x) = b_1^{syn} \vdash S_j; \mathbf{assert}\ !x_r \Downarrow v \\ \text{and } \forall\langle S_i, Q_i\rangle \in \Psi_1.&\mathbf{def}\ m(x) = b_2^{syn} \vdash S_i; \mathbf{assert}\ !x_r \Downarrow v \\ \wedge \quad \forall\langle S_j, Q_j\rangle \in \Psi_2.&\mathbf{def}\ m(x) = b_2^{syn} \vdash S_j; \mathbf{assert}\ x_r \Downarrow v \end{aligned} \tag{3}$$

Figure 6. Rewriting rules.

Notice that while each initial b_i evaluates to `true` under the precondition, there is no guarantee it is a sufficient condition for s_i to satisfy the postcondition—especially because RbSyn aims to synthesize small expressions, as discussed further in § 4. Moreover, there may be multiple e_i that are actually the same expression, and therefore could be combined to yield a smaller solution.

Thus, RbSyn next performs a *merging* step to create the final solution. This process operates on tuples of the form $\langle e, b, \Psi\rangle$, which is a hypothesis that the program fragment **if** b **then** e satisfies the specs Ψ. RbSyn repeatedly merges such tuples using an operation $\langle e_1, b_1, \Psi_1\rangle \oplus \langle e_2, b_2, \Psi_2\rangle$ to represent that **if** b_1 **then** e_1 **else if** b_2 **then** e_2 satisfies the specs $\Psi_1 \cup \Psi_2$. We define $\textsc{Specs}(\langle e_1, b_1, \Psi_1\rangle \oplus ...) = \bigcup \Psi_i$, i.e., the specs from merged tuples, and $\textsc{Prog}(\langle e_1, b_1, \Psi_1\rangle \oplus ...)$ = **def** $m(x)$ = **if** b_1 **then** e_1 **else** ..., a definition with the expression represented by the merged tuples.

Figure 6 defines rewriting rules that are applied to create the final solution. Rule 1 simplifies the case where e_1 and e_2 are the same and b_1 implies b_2, yielding a single expression and branch that satisfy $\Psi_1 \cup \Psi_2$. Note we omit the symmetric case for all rules due to space limitations. Rule 2 applies when b_1 does not imply b_2 but e_1 and e_2 are the same. In this case, e_1 satisfies the union of the specs under the disjunction of the branch conditions. (Note this rule could also applied if $b_1 \Rightarrow b_2$, but the resulting solution would be longer than Rule 1 generates.) Finally, Rule 3 applies when e_1 and e_2 differ but b_1 implies b_2. In such a scenario, b_2 holds for both e_1 and e_2 and thus it must be that b_1 and b_2 are insufficient to branch among e_1 and e_2. Thus, RbSyn synthesizes a stronger conditional b_1^{syn} that hold for all specs in Ψ_1 and does not hold for the specs in Ψ_2, and the reverse for b_2^{syn}. For example, recall the application of this rule in the example of § 2, to synthesize a more precise branch condition because the initial condition `true` was the same for both branches.

Algorithm 1 Merge programs

```
 1: procedure MergeProgram(candidates = $\{\langle e_i, b_i, \Psi_i\rangle\}$)
 2:     merged ← $\{\bigoplus \langle e_i, b_i, \Psi_i\rangle\}$
 3:     final ← {}
 4:     for all $m \in$ merged do
 5:         $m$ ← apply (1)-(3) to $m$ until no rewrites possible
 6:         final ← final ∪ $\{m\}$ if $\forall\langle S_i, Q_i\rangle \in \textsc{Specs}(m)$.
 7:                          $\bigwedge_i \textsc{Prog}(m) \vdash S_i; Q_i \Downarrow v$
 8:     end for
 9:     return Prog($m$) s.t. $m \in$ final
10: end procedure
```

RbSyn also includes a number of other merging rules, deferred to the technical report [22], for further simplifying expressions. Like, **if** b_1 **then** e_1 **else if** $!b_1$ **then** e_2 **else** `nil` can be rewritten as **if** b_1 **then** e_1 **else** e_2, which was used to generate the solution in Figure 2.

Checking Implication. Checking the implications in Figure 6 is challenging since branch conditions may include method calls whose semantics is hard to reason about. To solve this problem, RbSyn checks implications using a heuristic approach that is effective in practice. Each unique branch condition b is mapped to a fresh boolean variable z. Similarly, $!b$ is encoded as $\neg z$, and $b_1 \vee b_2$ is encoded as $z_1 \vee z_2$. Then to check an implication $b_1 \Rightarrow b_2$, RbSyn uses a SAT solver to check the implication of the encoding. While this check could err in either direction (due to not modeling the semantics of the b_i precisely), we found it works surprisingly well in practice. In case the implication check fails due to lack of precision, we fall back on the original $\oplus$ form which represents the complete program **if** b_1 **then** e_1 **else if** ... without loss of precision. Should the implication check incorrectly succeed, it will be caught by running the merged program against the assertions.

Constructing the Final Program. Finally, notice that the merge operation $\oplus$ is not associative, and it may yield different results depending on the order in which it is applied. Thus, to get the best solution, RbSyn uses Algorithm 1. It builds the set of all possible merged fragments (line 2). Then it simplifies each candidate solution using the rewrite rules and only considers a candidate valid if it passes all tests. It returns any such program as the solution. This branch merging strategy tries all combinations, so it is less sensitive to spec order than other component based synthesis approaches [31]. In practice, we found that reordering the specs does not have much effect.

3.4 Discussion

Before discussing our implementation in the next section, we briefly discuss some design choices in our algorithm.

Our effect system uses pairs of read and write effects in regions. As mentioned, this core idea could be extended to any effects in a test assertion that can be paired with an effect in the synthesized method body. For example, throwing and catching exceptions, I/O to disk or network, or enabling/disabling features in a UI could all be expressed this way. We leave exploring such effect pairs to future work.

One convenient feature of our algorithm is that correctness is determined by passing specs, which are directly executed. Thus, the synthesizer can generate as many candidates as it likes—i.e., be as over approximate as it likes—as long as its set of candidates includes the solution. This feature enables RbSyn to use a fairly simple effect annotation system compared to effect analysis tools [4].

We could potentially adapt our algorithm to work in a capability-based setting, using the observation that capabilities and effects are related [6, 8, 19]. In this setting, assertion failures in tests would indicate specific capabilities needed by the synthesized code. We leave exploring this idea further to future work.

Finally, we distinguish typed holes from effect holes, rather than have a single type-and-effect hole, to control where to use type-guidance and where to use effect-guidance. When initially trying to synthesize a method body, we omit effects because it is unclear which effects are needed. For example, in Figure 1, the second spec has read effects on all fields of the post, and yet the target method does not write any fields, as the spec is checking the case when the post is not modified. Thus, we cannot simply compute the union of all read effects in all assertions and use those for effect guidance. Moreover, type-guided synthesis often will synthesize effectful expressions, e.g., the call to `Post.where` in Figure 2. Conversely, our algorithm only places effect holes in positions where the type does not matter—hence type information for such a hole would not add anything. Nonetheless, type-and-effect holes would be a simple extension of our approach, and we leave exploration of them to future work in other synthesis domains.

4 Implementation

RbSyn is implemented in approximately 3,600 lines of Ruby, excluding its dependencies.

Synthesis specifications, as discussed in § 2, are written in a custom domain-specific language. Each has the form:

```
define :name, "method-sig", [consts,...] do
  spec "spec1" do setup { ... } postcond { ... } end ...
end
```

where `:name` names the method to be synthesized; `method-sig` is its type signature; and `consts` lists constants that can be used in the synthesized method. Each `spec` is a test case the method must pass: `setup` describes the test case setup, and `postcond` makes assertions about the results.

In Ruby, `do...end` and `{...}` are equivalent syntax for creating *code blocks*, i.e., closures. Having the setup and postcondition in separate code blocks allows RbSyn to run the setup code and check the postcondition independently.

RbSyn also has optional hooks for resetting the global state before any `setup` block is run. This ensures candidate programs are tested in a clean slate without being affected by side-effects from previous runs. In our experiments, RbSyn resets the global state by clearing the database.

Program Exploration Order. While our synthesis rules are non-deterministic, our implementation is completely deterministic. This makes it sensitive to the order in which expressions are explored. RbSyn uses two metrics to prioritize search. First, programs are explored in order of their size; smaller programs are preferred over larger ones. Program size is calculated as the number of AST nodes in the program.

Second, RbSyn prefers trying effect-guided synthesis for expressions that have passed more assertions rather than fewer. (The technical report [22] formally describes counting passed assertions.) Untested candidates are assumed to have passed zero assertions. In general, expressions are explored in decreasing order of number of passed assertions, then in increasing order of program size.

These metrics combined also help when RbSyn synthesizes a candidate that does not make any progress towards a solution: after running tests and effect-guided synthesis on such candidates, their size increases, but if they do not pass more assertions, they are pushed further down the search queue. We leave experimenting with other search strategies to future work.

Effect Annotations. We extended RDL to support effect annotations along with type annotations for library methods. Programmers specify read and write effects following the grammar in § 3. For example a method annotated with a write effect `Post.author` writes to some region `author` in some object of class `Post`. Here `author` is an uninterpreted string, selected by the programmer. Similarly the labels "." and "$*$" stand for pure and any region (or simply "impure"), respectively. A region `Post.*` is written as `Post` for convenience. One important extension is a `self` effect region, which indicates a read or write to the class of the receiver. This is essential for supporting ActiveRecord, whose query methods are inherited by the actual Rails model classes. For example, we use the `self` effect on the `exists?` query method of `ActiveRecord::Base`. Then at a call `Post.exists?`, where `Post` inherits from `ActiveRecord::Base`, we know the query reads the `Post` table and not any other table.

Effect annotations are similar to frame conditions [5, 14, 28] used in verification literature. More precise effect annotations help RbSyn find a solution faster because it will have fewer methods with subsumed effects than an imprecise one, shrinking the search space. But effect precision does not affect the correctness of the synthesized program, since correctness is ensured by the specs. For example, if the effect annotation for the method Post#title= shown in § 2.1 had just `Post` as its write annotation, synthesis would still work, but would try more candidate programs. In some cases, coarse effects are required, e.g. the Post.where method queries records from the Post table. It has the coarser Post annotation because which columns such a query will access cannot be statically specified: it depends on the arguments. We evaluate some of the tradeoffs in effect precision in § 5.4.

Type Level Computations. RbSyn uses RDL [15, 35] to reason about types, e.g., checking if one type is a subtype of another, and using the type environment and class table to find terms that can fill holes. RDL includes *type-level computations* [27], or *comp types*, in which certain methods' types include computations that run during type checking. For example, a comp type for the `ActiveRecord#joins` method can compute that `A.joins(B)` returns a model that includes all columns of tables `A` and `B` combined. Using a comp type for `joins` encodes a quadratic number of type signatures, for different combinations of receivers and arguments, into a single type, and more for joins of more than two tables [27].

RbSyn uses RDL's comp types, but with new type signatures designed for synthesis. In particular, the previous version of RDL's comp types gave precise types when the receiver and arguments were known, e.g., in `A.joins(B)`, RDL knows exactly which two classes are being joined. But this may not hold during synthesis, e.g., if `B` is replaced by a hole in the example, then the exact return type of the `joins` call cannot be computed.

To address this issue, we modified RDL's existing comp type signatures for `ActiveRecord` methods like `joins` so that they compute all possible types. For example, if a hole is an argument to `joins`, then the type finds all models `B1`, `B2`, ... that could be joined (i.e., those with associations); gives the hole type $\mathtt{B1} \cup \mathtt{B2} \cup \ldots$; and sets the return type of joins to a table containing the columns of $\mathtt{A}, \mathtt{B1}, \mathtt{B2}, \ldots$. This over-approximation is narrowed as the argument terms are synthesized, leading to cascading narrowing of types throughout the program as discussed in § 3.1.

Optimizations. Synthesis of terms that pass a spec is an expensive procedure. In practice, we found solutions to a single spec often satisfy others. Thus, when confronted with a new spec, RbSyn first tries existing solutions and conditionals to see if they hold for the spec, before falling back on synthesis from scratch if needed. This makes the bottleneck for synthesis not the number of tests, but the number of unique paths through the program. Moreover, this reduces the number of tuples for merging, as a single expression and conditional tuple can represent multiple specs Ψ.

Finally, we found that in practice, the condition in one spec often turns out to be the negation of the condition in another. Thus during synthesis of conditionals, RbSyn tries the negation of already synthesized conditionals before falling back on synthesis from scratch.

Limitations. While RbSyn works on a wide range of programs, as we will demonstrate next, it does have several key limitations. First, RbSyn currently only synthesizes code that does not need type casts to be well-typed. This ensures programs do not have type errors at run time, but eliminates some valid programs from consideration. Second, the set of constants RbSyn can use during synthesis is fixed ahead of time. This places programs that use unlikely constants out of reach, e.g., we have encountered Rails model methods that include raw SQL query strings (instead of only using ActiveRecord). Finally, because RbSyn uses enumerative search, it can face a combinatorial explosion when searching for nested method calls, e.g., if there are n possible method calls, available, synthesizing $A.m(A.m(A.m(x)))$ may require an $O(n^3)$ search. In practice, we did not face this problem as deeply nested method calls are rarely used in Rails apps.

5 Evaluation

We evaluated RbSyn by using it to synthesize a range of benchmarks extracted from widely used open source applications that use a variety of libraries. We pose the following questions in our evaluation:

- How does RbSyn perform using code based on existing unit tests in widely deployed applications? (§ 5.2)
- How much improvement is type-and-effect guidance compared to alternatives such as only type-guidance or only effect-guidance? (§ 5.3)
- How does the precision of effect annotations affect synthesis performance? (§ 5.4)

5.1 Benchmarks

To answer the questions above, we collected a benchmark suite comprised of programs from the following sources:

- *Synthetic benchmarks* is a set of minimal examples that demonstrate features of RbSyn.
- *Discourse* [24] is a Rails-based discussion platform used by over 1,500 companies and online communities.
- *Gitlab* [18] is a web-based Git repository manager with wiki, issue tracking, and CI/CD tools built on Rails.
- *Diaspora* [9] is a distributed social network, with groups of independent nodes (called Pods), also built on Rails.

We selected these apps because they are popular, well-maintained, widely used, and representative of programs that are written with supporting unit tests. We selected a subset of the app's methods for synthesis, choosing ones that fall into the Ruby grammar we can synthesize: method calls, hashes, sequences of statements and branches. We currently do not synthesize blocks (lambdas), for/while loops, case statements, or meta-programming in the synthesized code.

All benchmarks from apps have side effects due to either database accesses or reading and writing globals.

Table 1 lists the benchmarks. The first column group lists the app name (or *Synthetic* for the synthetic benchmarks); the benchmark id; the benchmark name; and the number of specs. The synthetic benchmarks exercise features of RbSyn by synthesizing pure methods, methods with side effects, methods in which multiple branches are folded into a single line program, etc. The Discourse benchmarks include a number of effectful methods in the User model, such as methods to activate an user account, unstage a placeholder account created for email integration, etc. The Gitlab benchmarks include methods that disable two factor authentication for a user, methods to close and reopen issues, etc. Finally, the Disaspora benchmarks include methods to confirm a user's email, accept a user invitation, etc.

We derived the specs for the non-synthetic benchmarks directly from the unit tests included in the app. We split each test into *setup* and *postcondition* blocks in the obvious way, and we added an appropriate type annotation to the synthesis goal. Across all benchmarks, we started with a base set of constants (Σ in § 3) to be `true`, `false`, 0, 1 and the empty string. Then we added `nil` and singleton classes (for calling class methods) on a per benchmark basis as needed. (As with many enumerative search based methods, we rely on the user to provide the right set of constants.)

A few apps have several different unit tests with exactly the same setup but different assertions in the postcondition. We merged any such group of tests into a single spec with that setup and the union of the assertions as the postcondition, to ensure that every spec setup can be distinguished with a unique branch condition, if necessary. We indicate this in the *# Specs* column of Table 1 by listing the final number of specs followed by the original number of tests in parentheses if they differ. We report the minimum and maximum number of assertions over all specs per benchmark in the *Asserts* columns and the number of paths through the method in the true canonical solution (from the app) in the *# Orig Paths* column.

Annotations for Benchmarks. Finally, the *# Lib Meth* column lists the number of library methods available during synthesis. These are methods for which we provided type-and-effect annotations. In total, 164 such methods are shared across all benchmarks, including, e.g., ActiveRecord and core Ruby libraries. Since our benchmarks are sourced from full apps, they often also depend on some other methods in the app. We wrote type-and-effect annotations for such methods and included those annotations only when synthesizing that app. Since RbSyn needs to run the synthesized code, when running specs we include the code for both general-purpose methods, such as those from ActiveRecord, and required app-specific methods. We slightly modify the set of library methods for A9, as discussed further below.

To find effect labels for app-specific methods, we found examining the method name and quickly scanning its code was typically quite helpful. Often it was clear if a method was pure or impure. For impure methods, there were a few cases. Sometimes, methods access the same object fields irrespective of how the method is called, so we give such methods the most precise labels, e.g., the effect `InvitationCode.count` was used for benchmark A10. Other times, it is apparent the method accesses different fields of a class depending on the method's arguments or the global state, so we give these class effect labels, e.g., `User` (equivalent to `User.*`). Overall, the simplicity of the effect system helped here, as we could use human-readable region identifiers even without any object references, e.g., the effect `InvitationCode.count` abstracts over all possible instances of `InvitationCode` class.

The other main category of effect labels was for Rails libraries such as ActiveRecord. We constructed these labels by following the documentation. For metaprogramming-generated column accessor methods, we extended RDL's existing type generating annotations [35] to also generate effects. For example, when RDL creates the type signature for an accessor method `Post#title` for the title column of the Post table, it now also creates a read effect annotation `Post.title` for it.

Overall, we found writing effect annotations to be easier than our previous efforts writing type annotations for Ruby [27, 35], though of course we relied on that previous experience. We leave a systematic evaluation of the effort of writing effect annotations to future work.

5.2 Synthesis Correctness and Performance

RbSyn successfully synthesized methods that pass the specs for every benchmark. We manually examined the output and found that the synthesized code is equivalent to the original, human-written code, modulo minor differences that do not change the code's behavior in practice. For example, one such difference occurs with original code that updates multiple database columns with a single ActiveRecord call, and then has a sequence of asserts to check that each updated column is correct. Because RbSyn considers the effects of assertions in the postcondition one by one, it instead synthesizes a sequence of database updates, one per column. Another difference occurs in Gitlab, which uses the state_machine gem (an external package) to maintain an issue's state (closed, reopened, etc). RbSyn synthesizes correct implementations that work without the gem.

The middle group of columns in Table 1 summarizes RbSyn's running time. We set a timeout of 300 seconds on all experiments. The first column reports performance numbers for the full system as the median and semi-interquartile range (SIQR) of 11 runs on a 2016 Macbook Pro with a 2.7GHz Intel Core i7 processor and 16GB RAM. The next three columns show the median performance when RbSyn uses only type-guidance, only effect-guidance, and naive

Table 1. Synthesis benchmarks and results. *# Specs* is the number of specs used to synthesize the method; *Asserts* reports the minimum and maximum number of assertions over all specs for every benchmark; *# Orig Paths* is the number of paths through the method as written in the app; *# Lib Meth* is the number of library methods used for every benchmark; *Time* shows the median and semi-interquartile range over 11 runs, followed by the median time for synthesis using only types, only effects and naive term enumeration (*Neither*). *Meth Size* is the number of AST nodes in the synthesized method; *# Syn Paths* shows the number of paths through the synthesized method.

Group	ID	Name	#	Asserts		# Orig	# Lib	Time (sec)				Meth	# Syn
			Specs	Min	Max	Paths	Meth	Median ± SIQR	Types	Effects	Neither	Size	Paths
Synthetic	S1	lvar	1	1	1	1	164	0.34 ± 0.01	1.36	11.97	-	4	1
	S2	false	1	1	1	1	164	0.35 ± 0.01	1.37	12.19	-	4	1
	S3	method chains	2	1	1	1	164	0.98 ± 0.01	9.56	-	-	10	1
	S4	user exists	2	1	1	1	164	0.98 ± 0.02	9.52	-	-	9	1
	S5	branching	3	1	1	2	165	2.49 ± 0.07	38.37	-	-	17	2
	S6	overview (ext)	3	4	4	3	164	12.78 ± 0.09	-	-	-	72	3
	S7	fold branches	3	1	1	1	164	82.44 ± 0.95	218.51	-	-	13	1
Discourse	A1	User#clear_glob…	3	2	2	3	169	2.11 ± 0.04	-	-	-	24	3
	A2	User#activate	2 (3)	1	4	2	170	8.95 ± 0.23	-	-	-	28	2
	A3	User#unstage	3 (4)	1	5	2	164	50.02 ± 0.55	-	-	-	31	2
	A4	User#check_site…	5	1	1	2	168	51.6 ± 0.23	-	-	-	28	3
Gitlab	A5	Discussion#build	1	4	4	1	167	0.24 ± 0.01	-	-	-	18	1
	A6	User#disable_two…	1	10	10	1	164	0.25 ± 0.01	-	0.44	-	22	1
	A7	Issue#close	1 (2)	3	3	1	166	0.77 ± 0.03	25.99	0.13	0.37	15	1
	A8	Issue#reopen	1 (3)	5	5	1	166	3.68 ± 0.1	-	0.55	45.66	17	1
Diaspora	A9	Pod#schedule_…	3 (4)	1	1	2	161	2.44 ± 0.04	-	-	-	19	2
	A10	User#process_inv…	1	2	2	2	165	2.64 ± 0.05	0.81	-	0.85	12	1
	A11	InvitationCode#use!	1	1	1	1	165	4.23 ± 0.06	-	-	-	12	1
	A12	User#confirm_email	7	4	4	2	166	7.28 ± 0.11	-	-	-	31	3

enumeration, respectively. The SIQRs (omitted due to space constraints) for these runs are very small compared to the median runtime, similar to the performance numbers with all features enabled. We discuss the runs with certain guidance disabled in detail in § 5.3. The right-most group of columns shows the synthesized method size (in terms of number of AST nodes) and the number of paths through the method (1 for straight-line code).

Overall, RbSyn runs quickly, with around 80% of benchmarks solving in less than 9s. Benchmarks like A3 take longer because it requires synthesis of `nil` terms—recall `nil` is the bottom element of our type lattice, causing RbSyn to synthesize `nil` at every typed hole for method arguments. Consequently, this requires testing all completed candidates—even though they eventually fail—consuming significant time.

For one benchmark, A9, we changed the set of default library methods slightly due to some pathological behavior. This benchmark includes an assertion that invokes ActiveRecord's `reload` method, which touches all fields of that record. But then when RbSyn tries to find matching write effects, it explores a combinatorial explosion of writes to different subsets of the fields. This effort is almost entirely wasted, because the remainder of the assertion looks at only one particular field—but that one read is subsumed by the effect of the `reload`, making it invisible to RbSyn's search. As a result, synthesis for A9 slows down by two orders of magnitude. We addressed this by removing four ActiveRecord methods that manipulate specific fields and adding ActiveRecord's `update!` method as the only way to write a field back to the database. An alternative approach would have been to move the `reload` call to be outside the assertion.

As this example shows, and as is common with many synthesis problems, performance is very hard to predict. Indeed, we can see from Table 1 that performance is generally not well correlated with either the size of the output program or with the number of branches. The number of assertions (which direct the side effect guided synthesis) does not correlate with the synthesis time. We do observe that RbSyn's branch merging strategy is effective, often producing fewer conditionals than there are specs, e.g., in A12 there are seven specs but only three conditionals. Though, sometimes the results are not always optimal if the branch merging strategy finds a program that passes all tests, but a program with fewer branches exists, e.g., for A4 and A12, RbSyn produces a program with one more branch than the hand-written.

5.3 Performance of Type- and Effect-Guidance

Next, we explore the performance benefits of type- and effect-guidance. Figure 7 plots the running times from Table 1 when all features of RbSyn are enabled (*TE Enabled*), with

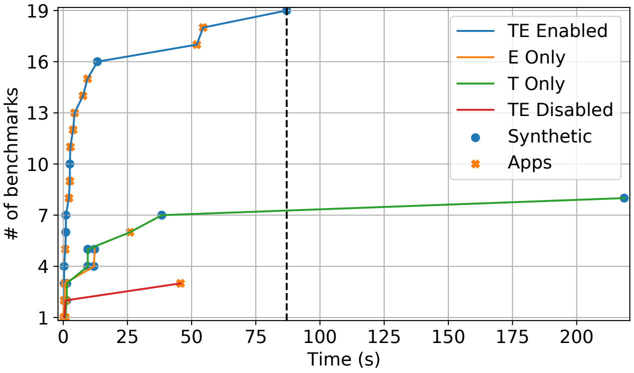


Figure 7. Number of benchmarks synthesized using type-and-effect (*TE Enabled*) guided synthesis relative to using only type (*T Only*) or effect (*E Only*) guidance separately and naive enumeration (*TE Disabled*). Higher is better.

only type-guidance (*T Only*), with only effect-guidance (*E Only*) and with neither (*TE Disabled*). The plot shows the number of benchmarks that complete (y-axis) in a given amount of time (x-axis), based on the median running times. This experiment serves as a proxy to show how a synthesis procedure that uses type-guidance but not effect-guidance, such as SyPet [12] or Myth [16, 30], may have performed if adapted for Ruby.

We can clearly see that type- and effect-guided synthesis performs best, successfully synthesizing all benchmarks; the slowest takes 83s. In contrast, with both strategies disabled, all but three small benchmarks time out. Performance with only type- or only effect-guidance lies in between. With only type-guidance, synthesis completes on eight benchmarks, of which the majority are pure methods from the synthetic benchmarks. From apps, it only synthesize A7 and A10. In these benchmarks, the needed effectful expressions are small and hence can be found with essentially brute-force search. With only effect-guidance, synthesis performance significantly worse, completing only five benchmarks, of which only three are from apps. These benchmarks succeeded because effect-guided synthesis quickly generates the template for the effectful method calls and then correctly fills them since they are small and can be found quickly by naive enumeration.

5.4 Effect Annotation Precision vs. Performance

Finally, we explore the tradeoff between effect annotation precision and synthesis performance. Recall that we found writing effect annotations easier for our benchmarks than writing type annotations. However, the effort can be further minimized by writing less precise annotations. This will not affect correctness, since RbSyn only accepts synthesis candidates that pass all specs, but it does affect performance.

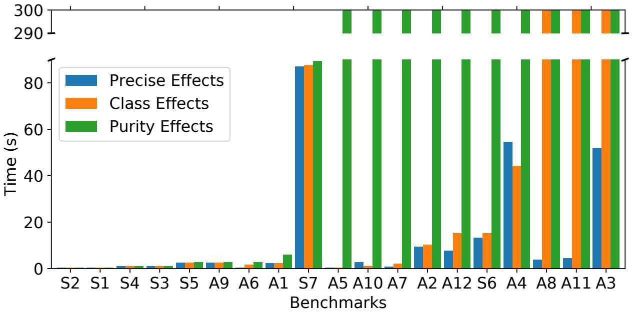


Figure 8. Performance of RbSyn with varying effect annotation precision: full, class effects only, and purity annotations on library methods. Lower is better. Full height indicates timeout.

Figure 8 plots the median of synthesis times for benchmarks over 11 runs under three conditions: *Precise Effects*, which are the effects used above; *Class Effects*, in which annotations include only class names and eliminate region labels (e.g., `Post.title` becomes `Post`); and *Purity Effects*, in which the only effect annotations are pure or impure (the $\bullet$ and $*$ effects, respectively, in our formalism). The benchmarks (x-axis) are ordered in increasing order of time for *Purity Effects*, then *Class Effects*, and finally *Precise Effects*.

From these experiments, we see that synthesis time increases as effect annotation precision decreases, often leading to a timeout. Class labels were sufficient to synthesize 16 of 19 benchmarks. Overall, class labels take time similar to precise labels, except for the three cases (A8, A11, and A3) where side-effecting method calls require precise labels to quickly find the candidate. As all precise effects are reduced to class effects, RbSyn must try many candidates with class effect before finding the correct one, leading to timeouts.

We note that A1 and A4 are slightly faster when using class effects. The reason is an implementation detail. The effect holes in these benchmarks can only be correctly filled by methods whose regular annotations are class annotations (more precise annotations are not possible). However, when trying to fill holes, RbSyn first tries all methods with precise annotations, only afterward trying methods with class annotations. Since the precise annotations never match, this yields worse performance under the precise effect condition than under the class effect condition, when the search could by chance find the matching methods sooner.

Purity labels only enabled synthesis of 9 benchmarks, including just 3 of 12 app benchmarks. The purity annotations are slow in general and only effective in the cases where the number of impure library methods is small.

6 Related Work

Component-Based Synthesis. Several researchers have proposed component-based synthesis, which creates code by composing calls to existing APIs, as RbSyn does. For example, Jha et al. [26] propose synthesis of loop-free programs for bit-vector manipulation. Their approach uses formal specifications for synthesis, in contrast to RbSyn, which uses unit tests. Hoogle+ [25] uses Haskell tests and types to synthesize potential solutions, primarily geared towards API discovery. CodeHint [17] synthesizes Java programs, using a probabilistic model to guide the search towards expressions more often used in practice. SyPet [12] also synthesizes programs that use Java APIs, by modeling them as a petri net and using SAT-based techniques to find a solution. These approaches do not support synthesis of programs with branches, which are common in the domain of web apps. While SyPet supports synthesis with side-effecting methods and CodeHint detects undesirable side effects during the search and avoids them, RbSyn uses side effect information from test cases to guide the search.

Programming by Example. Myth [16, 30] uses bidirectional type checking to synthesize programs, using input/output examples as the specification. However, Myth expects examples to be *trace complete*, meaning the user has to provide input/output examples for any recursive calls on the function arguments. RbSyn does not synthesize recursive functions, as they are rarely needed in our target domain of Ruby web apps. Escher [1] and spreadsheet manipulation tools [20, 21, 23] all accept input/output examples as a partial specification for synthesis. These tools primarily target users who cannot program, whereas RbSyn is targeted towards programmers. In addition, RbSyn's specs are full unit tests, so they can check both return values and side effects. λ^2 [13] synthesizes data structure transformations using higher-order functions, a feature not handled by RbSyn because of our target domain of Rails web apps, which rarely use such functions. STUN [2] uses a program merging strategy that is similar to ours, but it depends on defining domain-specific unification operators to safely combine programs under branches. In contrast, our approach may be more domain-independent, using preconditions and tests to find correct branch conditions. There have been multiple approaches to synthesizing database programs [7, 11]. Perhaps the closest in purpose to RbSyn is Scythe [39], which synthesizes SQL queries based on input/output examples. Scythe uses a two-phased synthesis process to synthesize an abstract query, after which enumeration is used to concretize the abstract query. In contrast, the use of comp types [27] allows RbSyn to quickly construct a template for a database query. With precise types for the method argument holes, this essentially builds abstract queries for free, whose holes are then filled later during synthesis.

Solver-Aided Synthesis. In solver-aided synthesis, synthesis specifications are transformed to a set of constraints for a SAT or SMT solver. Synquid [32] uses polymorphic refinement types as the specification for synthesis. Lifty [34] is a similar type system that verifies information flow control policies and synthesizes program repairs as needed to satisfy the policies. Both Synquid and Lifty synthesize conditionals using logical abduction. In contrast, RbSyn uses branch merging to synthesize conditionals, since translating Rails code and libraries into logical formulas is impractical.

Sketch [36] allows users to write partial programs, called sketches, where the omitted parts are then synthesized by the tool. Migrator [40] uses *conflict-driven learning* [10] to synthesize raw SQL queries, for use in database programs for schema refactoring. In contrast, programs synthesized by RbSyn use ActiveRecord to access the database. Rosette [37, 38] is a solver-aided language that provides access to verification and synthesis. It relies on symbolic execution, and thus requires significant modeling of external libraries for synthesizing programs that use such libraries.

eusolver [3] synthesizes programs with branches, using an information-gain heuristic via decision tree learning. While, the decision tree learning procedure can produce branches in an enumerative search setting (provided the input/output example set is complete), we leave an exploration of how it compares to our rule-based merging to future work. However, eusolver requires a SMT solver to produce counterexamples to build the input/output example set which has the additional cost of requiring formal specifications of library method semantics, an impractical task in the Rails setting. SuSLik [33] synthesizes heap-manipulating programs using separation logic to precisely model the the heap. RbSyn, in contrast, uses very coarse effects to track accesses that can go beyond the heap, such as database reads and writes.

7 Conclusion

We presented RbSyn, a system for type- and effect-guided program synthesis for Ruby. In RbSyn, the synthesis goal is described by the target method type and a series of specs comprising preconditions followed by postconditions that use assertions. The user also supplies the set of constants the synthesized method can use, and type-and-effect annotations for any library methods it can call. RbSyn then searches for a solution starting from a hole $\square : \tau$ typed with the method's return type, inserting (write) effect holes $\diamond : \epsilon$ derived from the read effects of failing assertions. Finally, RbSyn merges together solutions for individual specs by synthesizing branch conditions to select among the different solutions as needed. We evaluated RbSyn by running it on a suite of 19 benchmarks, 12 of which are representative programs from popular open-source Ruby on Rails apps. RbSyn synthesized correct solutions to all benchmarks, completing synthesis of 15 of the 19 benchmarks in under 9s, with the slowest benchmark solving in 83s. We believe RbSyn demonstrates a promising new approach to synthesizing effectful programs.

Acknowledgments

Thanks to the anonymous reviewers for their helpful comments. This research was supported in part by National Science Foundation awards #1900563 and #1846350.

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *International conference on computer aided verification*. Springer, 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

[2] Rajeev Alur, Pavol Černỳ, and Arjun Radhakrishna. 2015. Synthesis through unification. In *International Conference on Computer Aided Verification*. Springer, 163–179. https://doi.org/10.1007/978-3-319-21668-3_10

[3] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 10205. 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

[4] Robert L Bocchino Jr, Vikram S Adve, Danny Dig, Sarita V Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 97–116. https://doi.org/10.1145/1640089.1640097

[5] Alexander Borgida, John Mylopoulos, and Raymond Reiter. 1995. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering* 21, 10 (1995), 785–798. https://doi.org/10.1109/32.469460

[6] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 126:1–126:30. https://doi.org/10.1145/3428194

[7] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Vol. 48. ACM New York, NY, USA, 3–14. https://doi.org/10.1145/2499370.2462180

[8] Aaron Craig, Alex Potanin, Lindsay Groves, and Jonathan Aldrich. 2018. Capabilities: Effects for Free. In *Formal Methods and Software Engineering - 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast, QLD, Australia, November 12-16, 2018, Proceedings (Lecture Notes in Computer Science)*, Jing Sun and Meng Sun (Eds.), Vol. 11232. Springer, 231–247. https://doi.org/10.1007/978-3-030-02450-5_14

[9] Diaspora Inc. 2020. Diaspora: A privacy-aware, distributed, open source social network. https://github.com/diaspora/diaspora.

[10] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Vol. 53. ACM New York, NY, USA, 420–435. https://doi.org/10.1145/3192366.3192382

[11] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Vol. 52. ACM New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351

[12] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. 2017. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 599–612. https://doi.org/10.1145/3093333.3009851

[13] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* 50, 6 (2015), 229–239. https://doi.org/10.1145/2737924.2737977

[14] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—where programs meet provers. In *European symposium on programming*. Springer, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

[15] Jeffrey Foster, Brianna Ren, Stephen Strickland, Alexander Yu, Milod Kazerounian, and Sankha Narayan Guria. 2020. RDL: Types, type checking, and contracts for Ruby. https://github.com/tupl-tufts/rdl.

[16] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed synthesis: a type-theoretic interpretation. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* 51, 1 (2016), 802–815. https://doi.org/10.1145/2837614.2837629

[17] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*. 653–663. https://doi.org/10.1145/2568225.2568250

[18] GitLab B.V. 2020. GitLab is an open source end-to-end software development platform with built-in version control, issue tracking, code review, CI/CD, and more. https://gitlab.com/gitlab-org/gitlab.

[19] Colin S. Gordon. 2020. Designing with Static Capabilities and Effects: Use, Mention, and Invariants (Pearl). In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs)*, Robert Hirschfeld and Tobias Pape (Eds.), Vol. 166. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:25. https://doi.org/10.4230/LIPIcs.ECOOP.2020.10

[20] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* 46, 1 (2011), 317–330. https://doi.org/10.1145/1925844.1926423

[21] Sumit Gulwani, William R Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105. https://doi.org/10.1145/2240236.2240260

[22] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2021. RbSyn: Type- and Effect-Guided Program Synthesis. arXiv:cs.PL/2102.13183

[23] William R Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* 46, 6 (2011), 317–328. https://doi.org/10.1145/1993316.1993536

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

[25] 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. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27. https://doi.org/10.1145/3428273

[26] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 215–224. https://doi.org/10.1145/1806799.1806833

[27] Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S Foster, and David Van Horn. 2019. Type-level computations for Ruby libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 966–979. https://doi.org/10.1145/3314221.3314630

[28] Bertrand Meyer. 2015. Framing the Frame Problem. In *Dependable Software Systems Engineering*. Vol. 40. IOS Press, 193–203.

[29] Jaideep Nijjar and Tevfik Bultan. 2011. Bounded verification of Ruby on Rails data models. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 67–77. https://doi.org/10.1145/2001420.2001429

[30] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* 50, 6 (2015), 619–630. https://doi.org/10.1145/2737924.2738007

[31] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven synthesis. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* 49, 6, 408–418. https://doi.org/10.1145/2666356.2594297

[32] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* 51, 6, 522–538. https://doi.org/10.1145/2908080.2908093

[33] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the synthesis of heap-manipulating programs. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30. https://doi.org/10.1145/3290385

[34] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2020. Liquid information flow control. *Proc. ACM Program. Lang.* 4, ICFP (2020), 105:1–105:30. https://doi.org/10.1145/3408987

[35] Brianna M Ren and Jeffrey S Foster. 2016. Just-in-time static type checking for dynamic languages. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 462–476. https://doi.org/10.1145/2908080.2908127

[36] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 404–415. https://doi.org/10.1145/1168918.1168907

[37] Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 135–152. https://doi.org/10.1145/2509578.2509586

[38] Emina Torlak and Rastislav Bodik. 2014. A lightweight symbolic virtual machine for solver-aided host languages. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* 49, 6 (2014), 530–541. https://doi.org/10.1145/2594291.2594340

[39] 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*. 452–466. https://doi.org/10.1145/3062341.3062365

[40] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 286–300. https://doi.org/10.1145/3314221.3314588