Programming with a Read-Eval-Synth Loop

HILA PELEG, UC San Diego, USA
ROI GABAY, Technion, Israel
SHACHAR ITZHAKY, Technion, Israel
ERAN YAHAV, Technion, Israel

A frequent programming pattern for small tasks, especially expressions, is to repeatedly evaluate the program on an input as its editing progresses. The Read-Eval-Print Loop (REPL) interaction model has been a successful model for this programming pattern. We present the new notion of Read-Eval-Synth Loop (RESL) that extends REPL by providing in-place synthesis on parts of the expression marked by the user. RESL eases programming by synthesizing parts of a required solution. The underlying synthesizer relies on a partial solution from the programmer and a few examples.

RESL hinges on bottom-up synthesis with general predicates and sketching, generalizing programming by example. To make RESL practical, we present a formal framework that extends observational equivalence to non-example specifications.

We evaluate RESL by conducting a controlled within-subjects user-study on 19 programmers from 8 companies, where programmers are asked to solve a small but challenging set of competitive programming problems. We find that programmers using RESL solve these problems with far less need to edit the code themselves and by browsing documentation far less. In addition, they are less likely to leave a task unfinished and more likely to be correct.

CCS Concepts: • **Software and its engineering** → **Source code generation**; **Automatic programming**.

Additional Key Words and Phrases: program synthesis, read-eval-print loops, specification predicates

ACM Reference Format:
Hila Peleg, Roi Gabay, Shachar Itzhaky, and Eran Yahav. 2020. Programming with a Read-Eval-Synth Loop. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 159 (November 2020), 30 pages. https://doi.org/10.1145/3428227

1 INTRODUCTION

A frequent programming pattern for small tasks, especially one-liners, is to repeatedly evaluate the program on an input as its editing progresses. For convenience, users often take such an expression out of its context into a programming environment where a quick cycle of editing and executing is supported. The Read-Eval-Print Loop (REPL) provides such an environment and is a staple of many programming languages, including Python, JavaScript, and Scala. We present the new notion of Read-Eval-Synth Loop (RESL), which extends REPL by providing in-place synthesis on parts of the expression provided by the user. Our experiments show that RESL reduces programmers' effort, increases their task completion rate, and improves program correctness.

Read-Eval-Print Loop (REPL) The REPL model transforms the original development cycle of edit-compile-run-debug, tightening the development loop for small pieces of functionality. In the

Authors' addresses: Hila Peleg, UC San Diego, USA, hpeleg@ucsd.edu; Roi Gabay, Technion, Israel, roigby@gmail.com; Shachar Itzhaky, Technion, Israel, shachari@cs.technion.ac.il; Eran Yahav, Technion, Israel, yahave@cs.technion.ac.il.

2475-1421/2020/11-ART159
https://doi.org/10.1145/3428227

REPL model, the user provides an environment and an expression, an *educated guess* of the desired program, and the machine evaluates the expression and prints the result. REPLs provide exploratory programming and debugging where the programmer iteratively inspects the printed result and adjusts the initial guess accordingly.

Programming with Read-Eval-Synth Loop We wish to apply a similar transformation of the development cycle to the way programmers use synthesizers. RESL offers a synthesis-focused interaction model, supporting an iterative workflow with both the interpreter and the synthesizer, preserving all the benefits of the REPL while also providing synthesis capabilities. Where a REPL operates on values assigned to environment variables, RESL operates on a set of input values, allowing the user to specify the expected output for each input. This functionality meshes well with the paradigm of *Programming by Example* (PBE) [Feser et al. 2015; Gulwani 2011, 2012, 2016; Osera and Zdancewic 2015; Polozov and Gulwani 2015; Wang et al. 2017a; Yaghmazadeh et al. 2018], where the tool produces a program that satisfies the input-output pairs. To augment the examples, the user can pick parts of the program to preserve, and others that should be replaced by the synthesizer, controlling where to focus the synthesis effort.

Existing Techniques Many works in recent years attempted to bring synthesis into the hands of end-users and programmers. A common interaction model relies on sketching [Bornholt et al. 2016; Hua and Khurshid 2017; Smith and Albarghouthi 2016; Wang et al. 2018], where the user provides a program with holes, and some form of specification and the synthesizer fills the holes in a way that satisfies the specification. PBE is a prominent avenue for synthesis specifications, as examples are usually part of task definitions, and can be provided and understood even by non-programmers (though being a partial specification, they do not restrict the behavior for unseen inputs, and often require additional refining). RESL combines the power of the two interaction models and provides an iterative and interactive way to program with a synthesizer in the loop.

RESL can also be viewed as an iterative and interactive repair problem but differs from the many works of repair-by-synthesis [D Le et al. 2017; Hua et al. 2018; Long and Rinard 2015; Xiong et al. 2017] because it is intended for interactive use.

Our Approach We present programming with a Read-Eval-Synth Loop (RESL), a new interaction model that extends the widely used REPL model with an effective synthesis step.

RESL is just like REPL, with the additional help of a synthesizer. This raises two high-level challenges: (1) what is the right interaction model in which the user can express intent to the synthesizer, and (2) how can the synthesizer leverage information from the interactive session to make synthesis efficient and practical in an interactive setting.

(1) Interacting with the synthesizer We conjecture that a good interaction model for small tasks combines: REPL, test cases, synthesis, and specification mechanisms that do not rely solely on input and output values. In addition to test outputs, RESL allows the user to specify their intent using syntactic restrictions on the expected program. We formalize these specifications as predicates on programs.

(2) Efficient synthesis The new specification mechanisms interfere with existing search space pruning techniques, e.g., observational equivalence [Albarghouthi et al. 2013; Udupa et al. 2013], which are required to make the search tractable. Our main technical contribution is to extend observational equivalence, from execution values on inputs to *observers* that differentiate between programs on a per-predicate basis.

Main Contributions The contributions of this paper are:

(1) A new interaction model for small programming tasks in a REPL, which leverages "synthesis in the loop" as part of the iterative Read-Eval-Synth interaction. This model is a strict generalization of synthesis based on input-output examples.
(2) A formal framework for bottom-up synthesis with sketching and specification predicates, which generalizes the original, example-based notion of observational equivalence.
(3) An empirical evaluation that shows our modification of the synthesizer is necessary and yields a tractable procedure.
(4) A user study involving 19 industry developers who are JavaScript novices, which showed RESL helps them correctly solve challenging competitive programming tasks, reducing the editing load, frustration, and the need for documentation.

2 OVERVIEW

2.1 Motivating Example

Our aspiring programming ninja is solving a competitive programming "kata" `numbers-to-digit-tiers`:

> Create a function that takes a number and returns an array of strings containing the number cut off at each digit.
> For example: `420` should return `["4", "42", "420"]`
> `2020` should return `["2", "20", "202", "2020"]`

The user, an experienced programmer but a JavaScript novice, has in mind an approach: iterating over a range of numbers up to the number of digits, they will take the first *i* characters from the string representation of the number each time. They could set out to solve it in a REPL, iteratively testing and honing their proposed solution. However, they are uncertain about how to implement parts of their solution and turn to RESL, where some not readily known parts can be synthesized. The following describes their RESL session as seen in Figure 1:

Step 0 (initial state): a new RESL session starts with a default program `input`, returning the value of the variable `input`, and no examples.

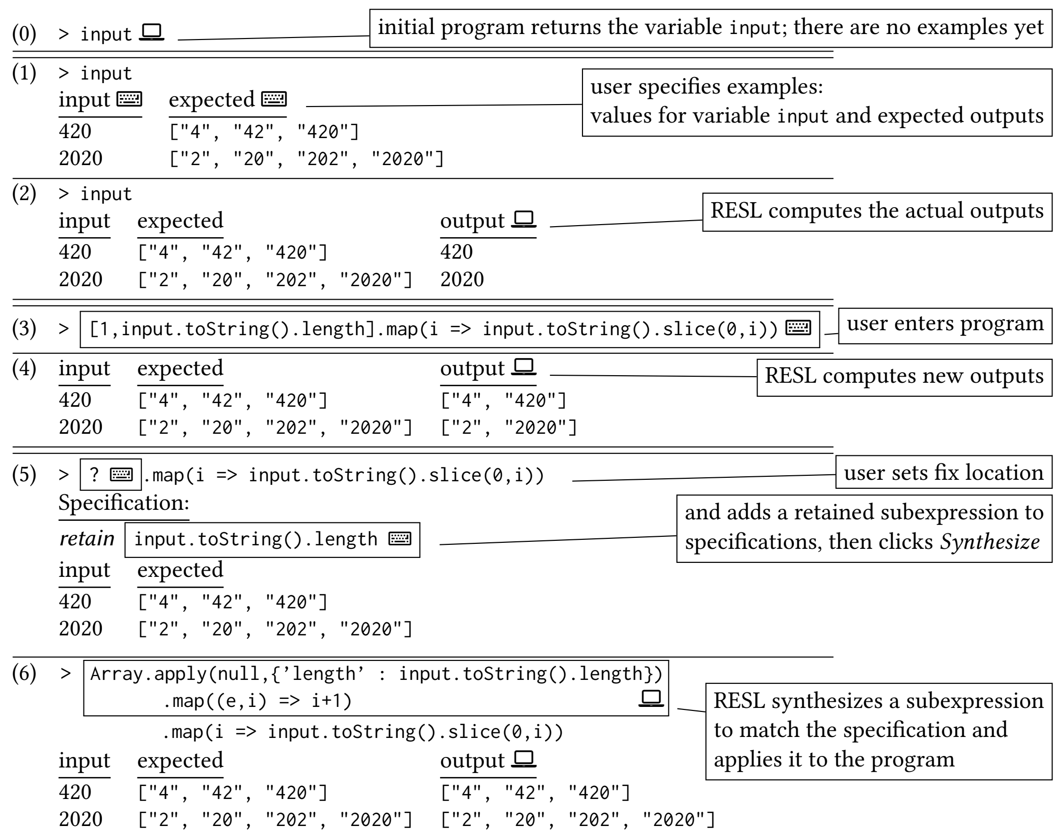
Step 1: the user enters the task's input-output examples into RESL (user edits are denoted by ⌨). Inputs are assigned to the variable `input`.

Step 2: RESL evaluates the inputs on the current program, `input`, and displays the outputs to the user (values computed by the system are denoted by 💻).

Step 3: The user arrives at a partial solution. Arriving at this partial solution could itself be an iterative process, with the user trying a few programs, each of which is executed on both values of `input`. They come up with a `map` that they believe will perform as expected, and test it on the first and last elements of the desired range, 1 and `input.toString().length`.

Step 4: RESL computes the outputs on this program, and the user sees that the first and last elements are as expected.

Step 5: The only thing left to do is to create the range, which should be `1..input.toString().length`. The user does not know how to construct a range, so instead, they *mark the current array expression as a portion of the program that should be replaced.* This is called a *sketch*: the user expresses their intent to keep everything outside this sub-expression as is, while turning it into a *hole* to be replaced with a new expression. The user also adds their intent that `input.toString().length`, the length of the range, be used in the solution by selecting it as a sub-expression to be *retained.* They click the *Synthesize* button to activate the synthesizer.



```
(0)  > input 💻
```
initial program returns the variable `input`; there are no examples yet

```
(1)  > input
     input ⌨    expected ⌨
     420        ["4", "42", "420"]
     2020       ["2", "20", "202", "2020"]
```
user specifies examples:
values for variable `input` and expected outputs

```
(2)  > input
     input   expected                        output 💻
     420     ["4", "42", "420"]              420
     2020    ["2", "20", "202", "2020"]      2020
```
RESL computes the actual outputs

```
(3)  > [1,input.toString().length].map(i => input.toString().slice(0,i)) ⌨
```
user enters program

```
(4)  input   expected                        output 💻
     420     ["4", "42", "420"]              ["4", "420"]
     2020    ["2", "20", "202", "2020"]      ["2", "2020"]
```
RESL computes new outputs

```
(5)  > ? ⌨ .map(i => input.toString().slice(0,i))
     Specification:
     retain input.toString().length ⌨
     input   expected
     420     ["4", "42", "420"]
     2020    ["2", "20", "202", "2020"]
```
user sets fix location

and adds a retained subexpression to specifications, then clicks *Synthesize*

```
(6)  > Array.apply(null,{'length' : input.toString().length})
               .map((e,i) => i+1)                              💻
               .map(i => input.toString().slice(0,i))
     input   expected                        output 💻
     420     ["4", "42", "420"]              ["4", "42", "420"]
     2020    ["2", "20", "202", "2020"]      ["2", "20", "202", "2020"]
```
RESL synthesizes a subexpression to match the specification and applies it to the program

Fig. 1. The steps in a RESL session trying to solve the `numbers-to-digit-tiers` kata. Actions taken by the user indicated by ⌨, values computed by RESL indicated by 💻.
See a video of this example: https://www.youtube.com/watch?v=QF9KtSwtiQQ

Step 6: The synthesizer finds an assignment for the hole which satisfies all provided user intent: input-output examples and the retained expression. It shows the assignment to the user, along with the outputs, which are the expected outputs.

An alternate approach: the user could start off by immediately synthesizing the expression for the range. To that end, they would modify the input-output examples to be `420` ⟶ `[0,1,2]` and `2020` ⟶ `[0,1,2,3]`, specifying the sub-task first, then revert the examples to those appearing in the task and iterate on the content of the `map` to reach the function parameter that appears in step 3.

2.2 The Anatomy of a RESL-ing Session

A RESL-ing session is an iterative loop comprised of two dialogues, as shown in Figure 2. The first loop is between the human user and the Arena, the RESL's user interface, which allows the user to view the program and manipulate it by editing, viewing its output and intermediate states on examples, and preparing a *synthesis query* for a subexpression they would like fixed. The second is between the user and the synthesizer, mediated by the Arena, which is conducted in a sequence of *queries* and *answers* constructed and processed for display for the user by the Arena.

Interaction with the Arena The user can advance the session with the interface in one of two ways: (i) editing the program or the hole to be synthesized, or (ii) instantiating predicates on programs that capture the specification, which will be sent to the synthesizer when it is called.

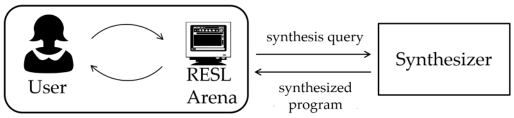


Fig. 2. The dual iterativeness of RESL: the user writes programs iteratively in the RESL Arena as in a REPL, while also entering examples and other specifications. When the user makes a synthesis call, the Arena invokes the synthesizer, which provides the Arena with a new program in place of the user.

The iterations for steps 1&2 and 3&4 are iterations between the user and the interface. The user edits the examples and the program and receives the outputs of the program on those inputs. The user can also use "debug information", intermediate execution values displayed by the interface, to view the progression of loops (e.g., `map`), or track down the source of errors.

Synthesis queries In step 5, the user performs two actions: marking a subexpression C, which becomes a hole and will be replaced by the synthesizer; and marking an expresion to retain in the result. These are precursors to constructing a query to the synthesizer. When the user clicks *Synthesize*, the interface converts the current state of the session into the components of a query q_1.

We define a synthesis query q_i, where $q_i = (\mathcal{V}_i, S_i)$. Each $\mathcal{V}_i$ is a *vocabulary*, or a set of terms, functions and operators that can be used in the solution. S_i is the set of specifications that should hold for the synthesis result, a new *completion* C'. Since the user provides their specification on the complete program (obtained by replacing C with C') the synthesis query contains a modified specification that can be tested directly on any candidate completion C'. Additionally, RESL's initial vocabulary $\mathcal{V}$ is enriched to fit the specific context of C: any additional internal variables available to C are added, and code written by the user is used as an additional expression of intent by adding both new terms from C that are not in $\mathcal{V}$, and subexpressions of C (including C itself), to the vocabulary.

In the example, step 5 generates a query $(\mathcal{V}_1, S_1)$ with:

$$S_1 = \Big\{ retain_{\texttt{input.toString().length}},\quad 420 \rightarrow_{\texttt{?.map(i => input.toString().slice(0,i))}} \texttt{["4","42","420"]},$$
$$2020 \rightarrow_{\texttt{?.map(i => input.toString().slice(0,i))}} \texttt{["2","20","202","2020"]} \Big\}$$

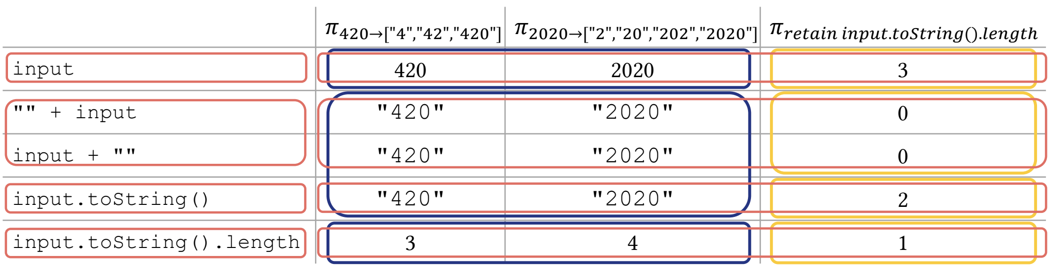
and `input`, `toString()`, `length`, `input.toString()` and `input.toString().length` added to $\mathcal{V}_1$.

Note that input-output examples are a type of predicate, and that they are bound to the sketch the result will be assigned into. The user need not be aware of this binding, as the Arena performs it automatically.

Synthesis loop At any point in the iterative interaction with the Arena, the user can send a synthesis query q_i, to be answered with a response C', which the arena assigns into the sketch remaining after removing C and displays to the user. The user continues the iteration with the Arena until they are happy with the program, either modifying the program and refining the specification, or sending additional synthesis queries. At any point where all predicates entered into the Arena hold for the current program, the user can accept it.

2.3 A Synthesizer Fit to RESL

Previous work has observed that many possible forms of synthesis specification, e.g., examples and syntactic specifications, can be expressed as boolean predicates on programs. Specification with *general predicates* was initially introduced as part of the Granular Interaction Model [Peleg et al. 2018b], where predicates similar to *retain* and *exclude* were found to be useful to users. However, they were tested with a mock synthesizer but never implemented. Our goal is to create a synthesizer for RESL, but this is not trivial: non-example predicates can interfere with the pruning mechanism of enumerative synthesis, either making the search for a program explode or causing the specification to become unsatisfiable.



	$\pi_{420\to[\text{"4","42","420"}]}$	$\pi_{2020\to[\text{"2","20","202","2020"}]}$	$\pi_{retain\ input.toString().length}$
`input`	420	2020	3
`"" + input`	`"420"`	`"2020"`	0
`input + ""`	`"420"`	`"2020"`	0
`input.toString()`	`"420"`	`"2020"`	2
`input.toString().length`	3	4	1

Fig. 3. Equivalence classes induced by specification S_1. Observers (denoted π) compute the values used to determine observational equivalence. The blue boxes denote equivalence classes when using only examples. The red boxes denote the desired equivalence classes that separate `input + ""` and `input.toString()` based on their contribution to the *retain* predicate and ensure `input.toString()` will not be discarded.

To support RESL, we implement an enumerating synthesizer with *observational equivalence* (e.g., Albarghouthi et al. [2013]; Alur et al. [2017]; Wang et al. [2017a]). The goal of Observational equivalence (OE) is to discard equivalent programs during enumeration, however true equivalence is hard to check. OE redefines functional equivalence by limiting it to a few "important" inputs on which programs need to be discernible, and the observational equivalence reduction (OE-reduction) prunes the space by discarding programs equivalent to ones previously encountered. In PBE, the inputs from input-output examples provided to the synthesizer are designated as "important". For example, when tested against the two inputs for examples in S_1, $\langle 420, 2020\rangle$, the program `input.toString()` evaluates to ⟨`'420'`, `'2020'`⟩. Likewise, the JavaScript programs `input + ""` and `"" + input` also evaluate to ⟨`'420'`, `'2020'`⟩. Of these, one (usually the first encountered) will be kept as its representative in the enumeration, and the other discarded. Observational equivalence is the state of the art in pruning a bottom-up enumeration, but we must modify it to suit the needs of RESL.

Specification predicates and observational equivalence To support a varied range of predicates in the user specification, we generalize observational equivalence beyond execution values on inputs. S_1 contains both examples and a *retain* predicate that requires the result to contain `input.toString().length`. Notice that if we discard `input.toString()` and keep `input + ""` in the earlier stages of enumeration, the synthesizer will no longer be able to satisfy S_1 even though it is satisfiable. This is because OE currently takes into account only the examples rather than the entire specification. This can be seen in Figure 3 where the blue boxes represent the equivalence classes created by examples alone.

We introduce the notion of *observers* in order to separate programs such as `input.toString()` and `input + ""` according to non-example parts of the specification. Observers are functions defined per-predicate that provide observational equivalence with values on which to operate—we replace the outputs vector used for equivalence above with the results of observers, one for each predicate in the specification. While for example-predicates the observer will still yield execution values over the input, observers for other predicates such as *retain* will return a value designed to separate programs based on the predicate they observe.

The observer for the non-example predicate, $retain_{\texttt{input.toString().length}}$, must indicate two things: whether the retained expression is already part of the current program, and whether the current program is a subexpression of the retained expression. The first ensures we regard two programs that contain the retained expression as equivalent *in regards to the retain predicate*, and the second ensures we don't discard a subprogram needed to construct the retained expression. Since `input.toString()` is a subexpression of `input.toString().length`, the observer value will encode *which* subexpression it is. Figure 3 shows the observer results for *retain* as well.

The observers for examples preserve the behavior of original OE, and the red boxes in Figure 3 show the refined equivalence classes representing the entire S_1.

Synthesizing a completion to a sketch Examples provided by users in the RESL Arena indicate the desired behavior of the entire program. However, since the user can create a hole anywhere in the current program using the Arena, as in step 5 of Figure 1, the synthesizer is now tasked with synthesizing a completion that, when assigned to the hole, behaves as specified. While the synthesizer generates just the completion, and thereby tests the specification predicates only on the completion, in order to yield a correct one it must consider the completed program, or in other words, test a candidate completion when assigned to the sketch; the sketch is therefore bound to the example predicates in S_1.

In the alternate approach to solving the task in Section 2.1, the user first synthesizes the expression for the range using modified examples: `420` $\rightarrow$ `[1,2,3]` and `2020` $\rightarrow$ `[1,2,3,4]`, and then returns the examples to their original state and attempts to map over the range expression. The user can then append a dummy map, e.g., `.map(i => input)`, which of course does not solve the problem, and create a hole inside the `map`, asking the synthesizer to replace `input` with a completion that will satisfy the examples.

Using observational equivalence when synthesizing an expression with which to replace `input` means values are needed for the variable `i`. In the case of this `map`, we want to perform OE with `i` mapped to each element of the arrays `[1,2,3]` and `[1,2,3,4]`. Leaving out any of these values causes OE to misjudge the real runtime behvior of the expression, which could cause us to miss non-equivalent expressions in the enumeration.

We therefore define observers for examples that can *extend* a given input valuation, yielding all possible valuations for the inner context and returning a set of execution values instead of a single value when comparing two possible completions. For this sketch inside `map`, the extended input valuations will be:

$$\big\{\{input \mapsto 420, i \mapsto 1\}, \{input \mapsto 420, i \mapsto 2\}, \{input \mapsto 420, i \mapsto 3\},$$
$$\{input \mapsto 2020, i \mapsto 1\}, \{input \mapsto 2020, i \mapsto 2\}, \{input \mapsto 2020, i \mapsto 3\}, \{input \mapsto 2020, i \mapsto 4\}\big\}$$

Accepting an extended vocabulary Finally, the synthesizer for RESL must accept a parametric $\mathcal{V}$ with each synthesis query, as each synthesis iteration may have a hole in a different position in the program, adding new inner context variables, and be attempting to replace a different expression that contributes new constants and functions to $\mathcal{V}$ beyond the base vocabulary.

2.4 Key Aspects

The key technical aspects presented in this paper are:

(1) Extended specifications: an interaction model allowing users to easily provide the synthesizer with examples as well as other predicates, define a sketch, and edit the resulting program.
(2) Enriched synthesis vocabulary: the vocabulary given to the synthesizer is enriched with program elements written by the user, not limiting synthesis to a predefined vocabulary.
(3) Generalizing equivalence: a new synthesis algorithm which generalizes Observational Equivalence to allow syntactic predicates and predicates on intermediate states of the program.
(4) Input extension: a bottom-up solution for enumerating programs in an inner scope with new variables, including inside higher-order functions.

3 PRELIMINARIES

Syntax-guided synthesis Syntax-guided synthesis (SyGuS) [Alur et al. 2015] accepts as input a vocabulary $\mathcal{V}$ of constants, functions, and operations to define the possible space of programs. For an element $f \in \mathcal{V}$, $arity(f) \in \mathbb{N}$ denotes the arity of f. We use the elements in $\mathcal{V}$ to construct ASTs bottom-up as follows: given an element f with $arity(f) = k$ and k ASTs $c_1, \dots c_k$, $f(c_1, \dots, c_k)$ is a new AST in the program space of $\mathcal{V}$.

For an AST T, $height(T) \in \mathbb{N}$ denotes the height of the tree, where the height of constants and variables is 0. Another measure for the size of T is $terms(T) \in \mathbb{N}$, which denotes the number of vocabulary terms (or AST nodes) in T.

The subtree relation If expression E_1 is a subexpression of E_2, then the AST for E_1 will be fully embedded in the AST of E_2. (Note that this is a property of expressions, but is not trivially a property of full programs. Though the definition can be modified to fit statements and statement lists, this paper will focus on expressions.) We denote this as $E_1 \subseteq E_2$.

Sketching and sketch trees Sketching (popularized by Sketch [Solar-Lezama 2008] and later extended to include other partial programs, as in [Bornholt et al. 2016; Feldman et al. 2019; Smith and Albarghouthi 2016]) is a variant of synthesis where the outermost part of the program is already known, and only a portion of it (a hole) needs to be filled by the synthesizer. In this paper we handle the specific case of a sketch with a single hole. A *sketch tree* is a an AST in which one node is a *hole node*, denoting a missing subtree, and marked as ?.

Assigning to a sketch In order to attain a full program from a sketch, we need to *assign* a (non-sketch) AST to the hole. We define the sketch assignment operation $T^0 \triangle C$, which accepts a sketch tree T^0 and a full *completion* C, and replaces the hole node in T^0 with C. No renaming or modification is performed on C during the assignment—C is expected to match the specific context of the hole.

4 PREDICATES

This section describes using predicates on programs for communicating user intent to the synthesizer. The way this is integrated into the RESL Arena will be described in Section 5.

Previous work [Peleg et al. 2018b] defines partial specification, including examples, as a set of predicates on programs. The input-output examples the user enters in step 1 of Figure 1 are each an instance of an example-predicate, and the retain operation in step 5 is also a predicate.

Formally, every interaction with the synthesizer is comprised of a set of predicates of type *Tree* $\rightarrow$ *Boolean*. Theoretically [Peleg et al. 2018a], any decidable predicate on programs (preferably quickly decidable) can be used. Peleg et al. [2018b] offer an interaction model where predicates specify syntactic features of the program, which they tested with no implemented synthesizer. In this work, we offer predicates that are both useful and easily instantiated via a point-and-click interface on the program. Their implementation is detailed in Section 7.

RESL supports predicates from several families of predicate schemas:

Input-output predicates: as in PBE, assert that running the program tree on a specific input valuation ι will yield a concrete output o. While the user enters a general $\iota \rightarrow o$ that specifies the entire progam, they may also create a sketch T^0, which means that the synthesis query must specify the behavior of the synthesized completion C. To this end, we define $\iota \rightarrow_{T^0} o$ as follows:

$$(\iota \rightarrow_{T^0} o)(C) \triangleq \left(\llbracket T^0 \triangle C \rrbracket(\iota) = o\right)$$

In a session, T^0 will be the sketch tree created when the user defines a hole in the current program. E.g., in step 5 of Section 2.1, the user creates a hole resulting in the sketch tree t^0 = `?.map(i => input.toString().slice(0,1)`. The first user-provided example in step 1 will be instantiated for the synthesizer as $\{input \mapsto 420\} \rightarrow_{t^0}$ `["4","42","420"]`. If no hole is created, then the entire program is to be synthesized, and $\{input \mapsto 420\} \rightarrow_{?}$ `["4","42","420"]` will be added to S.

Retaining a subexpression: asserts that a certain subexpression must appear in the program tree. Given a full subtree E, we define $retain_E$ as follows:

$$retain_E(C) \triangleq \left(E \subseteq C\right)$$

When used in a session, E is some subexpression of the current program viewed by the user. Any *retain* predicates will only be tested against the completion C, not the full program $T^0 \triangle C$.

For example, in step 5 of Section 2.1, the retain action instantiates a $retain_{\texttt{input.toString().length}}$ predicate in the synthesis query, ensuring that the result in step 6 contains the desired expression.

Excluding a subexpression: the complement of *retain*, *exclude* asserts that a certain subexpression does not appear in the program tree. Given a full subtree E, we define $exclude_E$ as follows:

$$exclude_E(C) \triangleq \left(E \nsubseteq C\right)$$

In a session, E is some subexpression of the current program viewed by the user. Like *retain*, *exclude* is only tested on the completion, not the entire program.

In step 5 of Figure 1, the user could also specify $exclude_1$, meaning this numeric literal should not appear in the result (this would rule out the result returned in step 6).

Requiring and prohibiting subexpression types: asserts that for a given input, evaluation of the completion C will either have (*require*) or not have (*prohibit*) a subexpression of type τ. This is useful both to suggest a desired algorithm and to control type coercion in JavaScript. We define:

$$require_{(\tau,\iota)}(C) \triangleq \exists E \subseteq C.type(\llbracket E \rrbracket(\iota)) = \tau \qquad prohibit_{(\tau,\iota)}(C) \triangleq \forall E \subseteq C.type(\llbracket E \rrbracket(\iota)) \neq \tau$$

Each *require* and *prohibit* is defined for a single input ι. However, in order to make them easier to enter, they are presented to the user as a general specifications for all inputs, and when composing S the Arena, described in the next section, adds the *require* or *prohibit* for each $\iota \rightarrow_{T^0} o$ in S.

5 THE RESL ARENA

Communication with the synthesizer requires both the editing of predicates, and delicate tree operations. While neither is impossible for a human user to do unaided, they are not convenient. As such, we introduce the RESL *Arena*, with which the user interacts. The interaction with the Arena is itself an iterative process with the dual purpose of providing the user with information about the current state of the program and constructing the synthesis queries $q_i = (\mathcal{V}_i, S_i)$, which are the form of communication between the Arena and the synthesizer, and are defined in Section 6.

The Arena is meant to give users the familiar look-and-feel of a REPL, preserving the important feature of REPLs, the speed of iteration [Burckhardt et al. 2013] that allows the programmer to determine whether part of the program behaves as desired in a piecewise manner. The RESL Arena extends this by executing the program on multiple inputs at once, displaying results and intermediate values, and providing a REPL-like interface interface to the synthesizer.

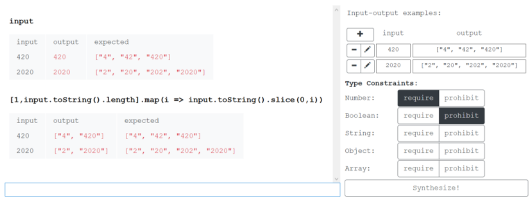
Figure 4 shows the components of the RESL Arena, used in the interaction described in Figure 1. We now describe the actions supported by the Arena.

Entering a program The user can enter a new program (or edit a program from the history of the session) using the prompt at the bottom left. This becomes the *active program.*

Adding and modifying examples The user can add new input-output pairs in the Arena, to evaluate the program with and use as $\iota \rightarrow o$ for the synthesizer, as seen in Figure 4(c). Existing examples can be removed or modified.

Evaluating and testing A program entered into the Arena is evaluated on every input in the provided examples, the input assigned to the `input` variable. Outputs for every input are printed for the user (seen in Figure 4(a)). If the output matches the expected output, a green checkmark is shown, otherwise, the expected output is shown. Changes to the examples are automatically re-evaluated without re-entering the active program.

Viewing debug information While a REPL is not a suitable environment for Live Programming [Lerner 2020; Omar et al. 2019], we still wish to show the user a breakdown of the internals of an evaluated expression. Therefore, for every subexspression of the full program, the Arena exposes

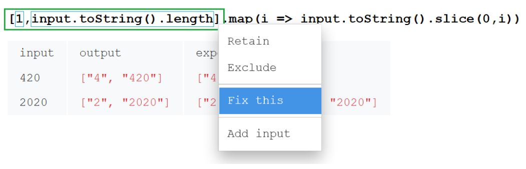


```
input
input   output   expected
420     420      ["4", "42", "420"]
2020    2020     ["2", "20", "202", "2020"]

[1,input.toString().length].map(i => input.toString().slice(0,i))
input   output           expected
420     ["4", "420"]     ["4", "42", "420"]
2020    ["2", "2020"]    ["2", "20", "202", "2020"]
```

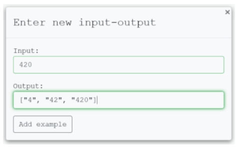
```
Input-output examples:
+     input    output
- ✎   420      ["4", "42", "420"]
- ✎   2020     ["2", "20", "202", "2020"]
Type Constraints:
Number:   [✓ require] prohibit
Boolean:  require [✓ prohibit]
String:   require prohibit
Object:   require prohibit
Array:    require prohibit
Synthesize!
```

(a)



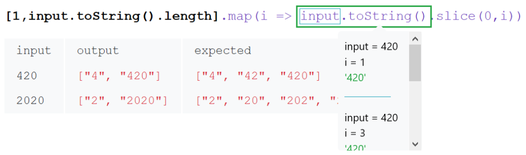
```
[1,input.toString().length].map(i => input.toString().slice(0,i))
input   output          exp
420     ["4", "420"]    ["4
2020    ["2", "2020"]   ["2            "2020"]
```

(b)



```
Enter new input-output
Input:
420
Output:
["4", "42", "420"]
Add example
```

(c)



```
[1,input.toString().length].map(i => input.toString().slice(0,i))
input   output          expected
420     ["4", "420"]    ["4", "42", "420"]
2020    ["2", "2020"]   ["2", "20", "202", "
```

```
input = 420
i = 1
'420'

input = 420
i = 3
'420'
```

(d)

Fig. 4. The RESL Arena. (a) the full Arena window after step 4 in Figure 1, (b) designating a subprogram as the hole to be fixed by the synthesizer, (c) adding an input-output example, and (d) the user viewing debug information on a subexpression.

every intermediate value in the program evaluation for every input. In the case of internal loops, e.g., the lambda inside the `filter` function, every execution context is shown. Figure 4(d) shows the debug information for the subexpression `input.toString()` in the context of $\{input \mapsto 420, i \mapsto 1\}$. Scrolling down will show other valuations, e.g., $\{input \mapsto 420, i \mapsto 3\}$, $\{input \mapsto 2020, i \mapsto 1\}$.

Creating a hole The user can mark a subexpression as the part of the program to be replaced in a synthesis operation. In Figure 4(b), the user is asking the Arena to decompose the current program into a sketch tree (shown in purple in Figure 4(d)) and a completion (shown in black).

Exclude and retain The user can mark a subexpression to exclude or to retain when calling the synthesizer. These will be instantiated as a predicates and appended to the synthesis query.

Restricting subexpression types The user can require or prohibit a specific type to be present in the intermediate state of the synthesized completion. I.e., one of its subexpressions must—or must not, respectively—be of this type. They can be seen in the bottom right of Figure 4(a). In the arena, these constraints are represented as general constraints, as opposed to the way they are formally defined in Section 4. Their conversion into the formal predicate is detailed in Section 6.

Invoking the synthesizer Once the user has provided specifications, they can invoke a call to the synthesizer using the *Synthesize* button, in the lower right corner of Figure 4(a). The following section details how the Arena transforms the user-facing state into a synthesis query.

This synthesis query is then sent to the synthesizer. The completion C' returned by the synthesizer is assigned into the hole and $T^0 \triangle C'$ is displayed as the new active program in the programs display. Subexpression and type constraints (retain, exclude, require type, and prohibit type) are discarded upon receiving a new program. Examples are preserved until changed by the user.

6 SYNTHESIS QUERIES

In this section we describe the interface between the Arena and the synthesizer, which is demonstrated in step 5 in Section 2.1. In each synthesis call, a separate *query* is sent to the synthesis back-end. Each synthesis query is stateless, providing all needed information to the synthesizer.

Synthesis query To initiate a synthesis step, the Arena constructs a synthesis query $q_i = (\mathcal{V}_i, S_i)$ from the Arena's current state. $\mathcal{V}_i$ is the synthesis vocabulary, and its construction will be detailed later in this section. S_i is the modified specification.

Creating the query requires the sketch T^0, the program outside the hole created in the Arena. If the user created no hole, T^0 will ? (a hole node). Then examples, which are specified in the Arena for the full program, are bound to the hole in T^0 by turning each $\iota \to o$ to $\iota \to_{T^0} o$ before adding it to S_i. Subexpression type constraints *require* and *prohibit* are bound to the inputs from any $\iota \to_{T^0} o$. Remaining predicates are simply added to S_i.

Synthesis result A successful result of a synthesis step is an AST C_i such that $C_i \models S_i$. The next active program would consequently be constructed by the Arena by composing $T_i^0 \triangle C_i$.

The synthesizer returns a result as soon as it finds a satisfying expression. If the timeout expires before a result could be found, the synthesizer fails with an empty result denoted $\perp$.

Extending the synthesis vocabulary SyGuS is a variant of program synthesis where the synthesizer is provided with both a specification and a grammar defining the search space as inputs. As $\mathcal{V}_i$ changes at every iteration, RESL is a SyGuS synthesizer.

The size of a synthesizer's vocabulary greatly effects the size of the search space. In a bottom-up enumeration, limiting the vocabulary also limits the available constants. As a result, $\mathcal{V}$ may be too limited for queries in a real-world development session. Additionally, $\mathcal{V}$ is defined for the outer-most scope, but sketching may have introduced new inner scopes.

RESL tackles all of these limitations by *extending* the vocabulary: the Arena incorporates elements of C, the subexpression removed from the hole, and the hole's context into $\mathcal{V}_i$. This preserves any special constants or functions not in the base vocabulary that are part of the user's intent.

We define a *vocabulary extension* $\mathcal{V}(C) \cup ext(T^0)$ that is added to $\mathcal{V}$ as follows:

(1) $\mathcal{V}(C)$ is a vocabulary set comprising elements of C:
  - All node labels in C: constants, variables, operators, and functions with their arity in C.
  - All subtrees $T \subseteq C$, with $arity(T) = 0$. This is not strictly necessary, as they can be recomposed from the node labels of C, but this prioritizes them in a bottom-up enumeration.

(2) $ext(T^0)$ is the context of the hole, comprising the variables whose declaration is on the path from the root of T^0 to the hole node in the sketch tree.

The RESL synthesizer prioritizes the elements of $\mathcal{V}(C) \cup ext(T^0)$ over elements of $\mathcal{V}$ in the synthesis process, making it faster to enumerate programs that are similar to C when replacing C.

In the following sections, we describe the implementation of a synthesizer that supports a query of the format $(\mathcal{V}, S)$ for S comprising predicates described in Section 4.

7 SYNTHESIZER OVERVIEW

In this section, we describe the implementation of RESL synthesizer. This synthesizer supports synthesis queries of the format described in Section 6. While few existing state-of-the-art synthesizers offer sketching as part of their specifications, no synthesizers handle the remaining RESL predicates[1]. For the synthesizer to be correct, it needs to accept general predicates (Section 4) and input-output examples that refer to the entire program, while synthesizing the completion to the sketch. To do this, we tackle several conceptual obstacles addressed in this section.

We begin by motivating our choice of a bottom-up enumeration and introduce pruning with observational equivalence. Next, we extend OE to *observers*, predicate-specific value generators, rather than execution values. We also present an optimization that allows for more pruning using some predicates. We then describe the challenge of preserving OE when synthesizing completions to a sketch, and our approach to it. Finally, we observe a limitation of OE that poses an obstacle to handling data-dependent loops (e.g. `reduce()`).

[1]The syntactic predicates proposed by [Peleg et al. 2018b] were never implemented in a synthesizer.

$$T^0 = \texttt{input.sort((a,b) => ?)}$$
$$S = \left\{ \begin{array}{c} \texttt{['aba','efg','bcd']} \longrightarrow_{T^0} \texttt{['aba','bcd','efg']}, \\ \mathit{retain}_{\texttt{a[a.length - 1]}}, \quad \mathit{exclude}_{\texttt{a[0]}} \end{array} \right\}$$

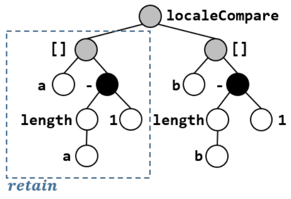


Fig. 5. The running example described in Example 7.1. During the enumeration described in Section 7.2, parts of the desired program are discarded (black nodes) and some are never constructed (grey nodes). Consequently, the *retain* element of the specification cannot be realized without an observer for it.

Running example We demonstrate the key aspects of our approach on the following example:

Example 7.1. a user is using RESL to solve the competitive programming exercise "Sort by Last Char". They use the Arena to create the sketch T^0=`input.sort((a,b) => ?)`, and provide the specification S shown in Figure 5, which is then sent to the synthesizer along with a vocabulary $\mathcal{V}$.

The `sort` method takes a comparator that encodes the relationship between elements using the sign of its return value. One correct solution for this task is `input.sort((a,b) => a[a.length - 1].localeCompare(b[b.length - 1]))`. The AST of this target program is also shown in Figure 5.

7.1 Bottom-Up Synthesis with Observational Equivalence

We now explain the need for bottom-up synthesis for the RESL synthesizer, introduce observational equivalence [Albarghouthi et al. 2013; Udupa et al. 2013], the existing state-of-the-art in pruning the search space for bottom-up synthesis, and our correctness theorem.

Why bottom-up An enumerating synthesizer can construct programs top-down, starting from the root and expanding, or bottom-up, constructing larger programs from smaller ones. A RESL synthesis query contains two elements which need to be handled by the synthesizer, the specification that can include non-example predicates, and the extended vocabulary that can include any language element the user pleases. We considered both approaches, and found that when handling the new specification predicates, bottom-up enumeration allows us to prune the space more effectively, and that it is crucial to handling extended vocabularies.

Top-down synthesizers rely on semantic rules for propagating the example through an AST node, creating specification for its children. The ability of the RESL Arena to extend the vocabulary means that a top-down synthesizer would need to contain such rules for every function and operator in the language and any imported API. In contrast, bottom-up synthesis relies on executing expressions, which means an unfamiliar function can simply be executed.

Additionally, top-down enumeration can only test programs for syntactic elements once they are fully generated: excluded expressions can only be eliminated once the entire excluded tree has been enumerated, meaning it will be enumerated again and again within many other programs. Retained expressions can be added to the vocabulary to optimize their creation (as they can be in bottom-up enumeration), but since they can be non-trivial ASTs, their semantics are not part of the synthesizer, which means the synthesizer cannot deduce whether or not they fit as a missing child, and will always have to try them.

Synthesis with observational equivalence In a bottom-up enumeration, we compose smaller programs to create bigger programs. This means the most effective form of program pruning is to discard programs as early as possible, reducing the number of larger programs that would be constructed. An effective way to reduce the size of the search space is to skip programs that are equivalent to ones already seen. However, checking whether two programs are equivalent over *all* inputs is generally undecidable. Observational equivalence redefines functional equivalence by limiting it to a few "important" inputs on which programs need to be discernible. In PBE, the inputs from input-output examples provided to the synthesizer are designated as "important".

For example, given the PBE specification for *isUpper* ⟨`'Kirk'`→*false*, `'sulu'`→*false*, `'KHAN'`→*true*⟩, the programs `input.length` and `4` are observationally equivalent, since they both map the input vector ⟨`'Kirk'`, `'sulu'`, `'KHAN'`⟩ to $\langle 4, 4, 4\rangle$. Under that assumption, substituting `4` for `input.length` is correct since it is correct for all values in the specification. If this does not align with user intent, they can always provide an additional example with a string of different length. Unlike equivalence reductions based on semantic information (e.g. [Feser et al. 2015; Smith and Albarghouthi 2019]), observational equivalence makes decisions solely based on execution values, making it lightweight to maintain, requiring only the ability to execute programs. A more formal definition appears in the extended version of this paper.

The correctness of observational equivalence We now describe the requirements for the correctness of observational equivalence. We build upon the correctness proposition by Albarghouthi et al. [2013] and generalize it. Notice that the correctness standard is that of a *partial specification*. The full definitions and proofs appear in the extended version of this paper.

Let us denote $(\!|\mathcal{V}|\!)$ as the full space of programs that can be constructed from vocabulary $\mathcal{V}$. Additionally, let us denote $(\!|\mathcal{V}|\!)^{OE}$ as the OE-reduced space of programs constructed from $\mathcal{V}$ in an enumeration over a given set of inputs.

Theorem 7.2. *Given a program $m \in (\!|\mathcal{V}|\!)$ and a specification (set of predicates) S, if every $p \in S$ holds for m then there exists $m' \in (\!|\mathcal{V}|\!)^{OE}$ that also satisfies every $p \in S$.*

This theorem states that as long as there is at least one program satisfying all specifications, we will find one. It does not matter *which one*, as the specification is the only criterion for correctness.

Remaining challenges To synthesize the completion to the synthesis query $(\mathcal{V}, S)$ in Example 7.1 with an OE-reduced enumeration, we must address two challenges. First, since $\mathcal{V}$ now contains inner-context variables `a` and `b`, in order to enumerate bottom-up, we need to find out the running-values of the inner context variables. Essentially, since `a` and `b` are loop variables, they are assigned multiple values, and we need them all for OE to be correct. Second, we must ensure the OE-reduction does not make the general predicate specification S unsatisfiable.

The following subsections will address these challenges, beginning with the predicates.

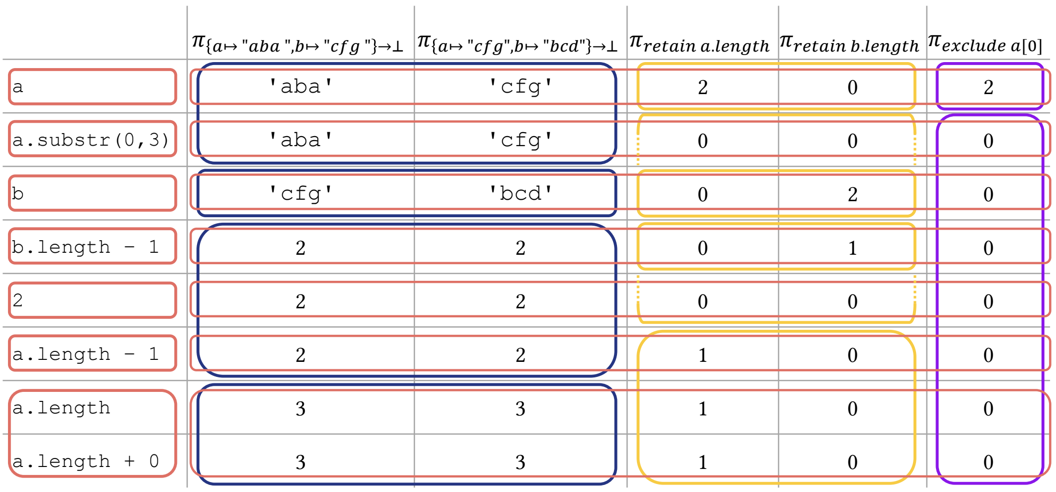
7.2 Synthesis with General Predicates

Section 4 details the predicates supported by RESL.

In Figure 5, the specification includes *exclude* and *retain* in addition to the one example predicate: the solution should contain `a[a.length - 1]` and should *not* contain `a[0]`.

General predicates and the OE-reduction Predicates in the specification can interfere with the standard OE-reduction. In our example, assume the vocabulary contains the constants `1` and `2`. Programs are enumerated in ascending order of height, so the program `2` is encountered long before `a.length - 1`. When evaluated on the valuations for `input`, `a`, and `b`, both programs yield the value vector $\langle 2, \ldots, 2\rangle$ (all strings are of length 3) and would therefore be deemed equivalent.

Under an OE-reduction, only one of these programs will be kept. Since it is usually the shortest program, `a.length - 1` will be discarded. As shown in Section 7.1, when dealing with values alone, this decision is correct, as any program that contains `a.length - 1` as a subexpression is observationally equivalent to the same program that uses `2` instead. However, since the new specification contains the requirement that `a[a.length - 1]` be a part of the solution, this form of enumeration will not be able to realize it.



	$\pi_{\{a\mapsto\text{"}aba\text{"},b\mapsto\text{"}cfg\text{"}\}\to\bot}$	$\pi_{\{a\mapsto\text{"}cfg\text{"},b\mapsto\text{"}bcd\text{"}\}\to\bot}$	$\pi_{retain\ a.length}$	$\pi_{retain\ b.length}$	$\pi_{exclude\ a[0]}$
`a`	`'aba'`	`'cfg'`	2	0	2
`a.substr(0,3)`	`'aba'`	`'cfg'`	0	0	0
`b`	`'cfg'`	`'bcd'`	0	2	0
`b.length - 1`	2	2	0	1	0
`2`	2	2	0	0	0
`a.length - 1`	2	2	1	0	0
`a.length`	3	3	1	0	0
`a.length + 0`	3	3	1	0	0

Fig. 6. Partition into equivalence classes based on observers for examples, *retain* and *exclude*. We modify Example 7.1 slightly to show *retain* of two smaller expressions, `a.length - 1` and `b.length - 1`, rather than the full `a[a.length - 1]`. (Examples used here for synthesizing a completion to a sketch are a subset of the correct example set, which is explained in Section 7.3, and since their output is not needed it is denoted $\bot$.)

Figure 5 shows the expected solution. The box denotes the specified required expression. Both `a.length - 1` and `b.length - 1` are deemed equivalent to `2`, and thus discarded (denoted with a black circle in the figure) from the enumeration, meaning no expression containing them can be constructed (denoted with a grey circle). This makes the rest of the enumeration will then fail to satisfy the *retain* of `a[a.length - 1]` (the boxed expression). The synthesizer will therefore produce no solution for S_1, even though a target program that satisfies it occurs in the unreduced space.

Observers To address this challenge, we extend the existing definition of observational equivalence, replacing the hard-coded use of execution values in determining equivalence with generic *observers*, a customized discriminator between programs that is defined per family of predicates.

Definition 7.3 (Observer). Let every predicate p be associated a function $\pi_p : \mathcal{P} \to K$, called its *observer*. It maps programs to *observations* from some range K, by which observational equivalence is to be determined. Intuitively, if $\pi_p(m^1) \neq \pi_p(m^2)$, then m^1 and m^2 are not observationally equivalent. K can consist of any number of possible outcomes—computed values, errors, side effects, etc.

For the input-output predicates $\iota \to o$, the observers are $\pi_{\iota\to o}(m) = [\![m]\!](\iota)$; these will be futher generalized in Section 7.3.

We require two properties of observers:

(O1) *Interchangeability*: For any two programs m_1, m_2, if $\pi_p(m_1) = \pi_p(m_2)$, then for any sketch T^0, it also holds that $\pi_p(T^0 \triangle m_1) = \pi_p(T^0 \triangle m_2)$.

(O2) *Consistency*: For any two programs m_1, m_2, if $\pi_p(m_1) = \pi_p(m_2)$, then $m_1 \models p \iff m_2 \models p$.

O1 ensures that two programs that are observed to be the same will be interchangable within any larger program under the same observer. This trivially holds for example predicates $\iota \to o$, and was the basis for the corretness proposition in [Albarghouthi et al. 2013]—as long as the target language does not admit destructive updates. This assumption is not specific to our framework; it is required for any operational semantics-based form of OE.

Figure 6 shows the values of observers for some predicates for Example 7.1: examples (whose outputs are unknown, as they were propogated into a sketch), *retain*, and *exclude*). The observers for examples (blue boxes) separate programs based on their execution values, as a standard OE-reduction would, whereas the yellow and purple boxes separate programs based on the *retain* and *exclude* predicates respectively. The red boxes indicate the final separation into equivalence classes. For example, the observer values for *retain* allow us to separate `2` and `a.length - 1` that are merged in a standard OE-reduction.

We provide guidelines for defining observers for specification predicates in the extended version of this paper, and show the correctness of OE holds so long as the observers adhere to the two required properties O1 and O2. We now extend our available observers beyond the one for examples.

Observing predicates In Figure 6, we can see that plain observational equivalence (the blue boxes only) found `2` and `a.length-1` equivalent, and would place them in the same equivalence class. To separate them, we replace the outputs vector used for equivalence with the results of observers, one for each predicate in the specification.

The observer for *retain*, like the observer for examples, is aimed at not losing partial programs that do not yet satisfy the predicate. Programs that satisfy a *retain* predicate can be constructed from programs that do not satisfy it, which means that to satisfy them we must ensure that we do not lose their subprograms in the course of the enumeration.

Likewise, for *exclude* we are interested in this behavior, as it ensures we can create something that is equivalent to the excluded expression by all other predicates, if such a program exists in the space.

Input-output observers already include this behavior—the result of the predicate is not part of the observer, and intermediate values needed to construct the final output value on which the predicate is tested are separated into different equivalence classes.

To replicate this behavior for *retain* and *exclude*, we define its observer like so:

Definition 7.4 (Retain and exclude observers). For $p{=}retain_E$ and $p{=}exclude_E$, retaining or excluding subtree E, we assign each node in E an ordinal number, where the root is 1 and all other nodes are assigned a value in the range $2..terms(E)$. We define $\pi_p(m) = 1$ if $E \subseteq m$, $\pi_p(m) = i$ if m is **equal** to the subtree at node i, and 0 otherwise.

Consider one of the retained expressions in Figure 6, `a.length`. The expression `a` is a subexpression of `a.length`, so the observer encodes its position in the AST. The expression `2` is not a subexpression of `a.length` or vice versa, so the observer value is 0. For `a.length-1`, for which `a.length` is a subexpression, the observer value will be 1, encoding that it is included. This separates `2` and `a.length-1`, which are equivalent under examples alone, into two equivalence classes.

Observing *require* and *prohibit* requires two values: the observer $\pi_{require_{(\tau,\iota)}}(m)$ includes both the value of its predicate $require_{(\tau,\iota)}(m)$ and the type of program m when evaluated with input ι. Observing $prohibit_{(\tau,\iota)}$ is preformed in the same way.

Since the purpose of observers is to create greater separation between programs, they increase the number of equivalence classes and number of programs in the space. While this is unfortunate, it is necessary for correctness, and our experiments (Section 8.1) show this increase is tractable.

The correctness proofs for these observers are in the extended version of this paper.

Negatively-Stable Predicates Moreover, some general predicates are amenable to further optimization, allowing for a reduction of the space to counteract the separation caused by the observer. E.g., our specification also contains $exclude_{\texttt{a[0]}}$, denoting that `a[0]` may not appear in the resulting completion, ruling out overfitted candidates. Once a program contains `a[0]` and fails the predicate, any program composed from it will also contain `a[0]` and also fail the predicate. We call such predicates *negatively-stable* predicates, and prune by discarding any candidate program that fails to satisfy at least one negatively-stable predicate in the specification. This is safe to do, because any larger program that would have been constructed using it would also violate the specification. Of the predicate families available in the RESL Arena, *exclude* and *prohibit* are negatively-stable, whereas examples, *retain* and *require* are not.

The correctness of further pruning the space according to negatively-stable predicates is proved in the extended version of this paper.

7.3 Synthesis inside a Sketch

In this subsection, we explain how to extend OE for synthesizing completions to sketches. So far, we assumed that our synthesizer is capable of providing values for the inner variables `a` and `b` in Example 7.1, and now we must revisit this assumption. We first explore the problem that arises from bottom-up synthesis in an inner context, and then describe our solution.

A naive solution would be to use the results of the completed program $T^0 \triangle C$ as the values for the OE-reduction. The problem is that subprograms of the correct completion are meaningless as a completion themselves, and can cause the synthesizer to fail even when the correct solution exists in the full space (i.e., be incorrect according to Theorem 7.2). To illustrate, in Example 7.1, `a.length` and `b.length` are both subexpressions of the sort comparator we wish to synthesize. However, each is meaningless as a completion on its own: they will only return 0 or a positive number that does not take the second argument into account at all, resulting in a nonsensical sort. In addition, if all strings in the array are non-empty, both will result in the *same* nonsensical sort order, leading observational equivalence to find them equivalent. In a statically-typed language, sub-expressions may also be of an incorrect type for the hole, making this approach unfeasable from the get-go.

Even though each completion C enumerated is not a full program, OE is still necessary to keep the process tractable. Since C cannot be assigned into the sketch, we need to evaluate each candidate completion in its apropriate inner context. In Example 7.1 this requires values for `a` and `b`. These values are not part of the input valuation in the examples in S. Moreover, since the hole of T^0 is inside the `sort` function, `a` and `b` obtain multiple values per execution, which may differ between different completions C, as the execution of `sort` depends on the return value of the sorting function. We must choose values for the variables that represent states that are reachable when executing with the inputs from S, because not representing a reachable state may cause us to miss programs.

Observers and sketches In order to enumerate the completions, we add the new variables to the execution context, so that we can perform OE over the new *extended* vocabulary for the scope of the hole $\mathcal{V} \cup V$, where $V = ext(T^0)$ (as defined in Section 6). In the case of a sketch that simply adds a new variable (e.g., `let x = `*`expr`*`; ?`), we can assign a value to this variable by executing the partial program up to the declaration. In this case there will be exactly one value of `x` for each value of `input`, and we can trivially add it to the input valuation.

In the general case, we define *context extension* for an input valuation ι inside a sketch T^0 (denoted $extend(\iota, T^0)$), which is a set of new valuations in which all variables in V have values, and is a subset of the possible set of valuations under which a completion C will be evaluated over when executing $[\![T^0 \triangle C]\!](\iota)$. The full definition is found in the extended version of this paper. Ideally, $extend(\iota, T^0)$ is not only a subset, but includes all values over all possible executions of any $T^0 \triangle C$. We call this a *complete extension.* To prove OE inside sketches correct, extensions must be complete.

For example, evaluating `sort`, the function is invoked for every pair of elements tested by the implementation. This means the actual set of pairs depends on the return values of previous calls. To get a complete extension, we consider *all* element pairs, as that will cover every possible return value of the function parameter and therefore every possible sorting order attempted. Then, when the observer is computed on the completion, this means separating comparator functions by their values rather than the sort order they induce.

We define several useful complete extensions for JavaScript:

$$extend(\iota, \texttt{let x = rhs; ?}) = \left\{\iota \cup \{x \mapsto [\![rhs]\!](\iota)\}\right\}$$

$$extend(\iota, \texttt{lhs.map((e,idx,arr) => ?)}) = extend(\iota, \texttt{lhs.filter((e,idx,arr) => ?)}) = \left\{\iota \cup \{arr \mapsto [\![\texttt{lhs}]\!](\iota), idx \mapsto j, e \mapsto [\![\texttt{lhs[j]}]\!](\iota)\} \;\middle|\; j \in 0{:}|lhs|\right\}$$

In JavaScript, `idx` and `arr` can be omitted from the declaration of the function parameter, and in other languages they might not be available at all. However, the definition of *extend* stays the same, only omitting the dropped parameters from the final set, which means some elements may converge. This extension can be used for other higher-order functions that perform independent iterations on the input array's elements, such as `groupBy`, `maxBy`, etc. We also define:

$$extend(\iota, \texttt{lhs.sort((a,b) => ?)}) = \left\{\{a \mapsto [\![\texttt{lhs[j]}]\!](\iota), b \mapsto [\![\texttt{lhs[k]}]\!](\iota)\} \cup \iota \mid j, k \in 0 : |lhs|, j \neq k\right\}$$

Depending on the implementation of `sort` there may be a smaller complete extension, but since at worst this is an overapproximation, it will not cause us to lose programs.

If scopes adding context variables are nested, extend is computed by first computing the extension of the outer scope, then for each member of the extension the process is repeated for the inner scope. E.g., for the sketch `input.map(l => l.filter(e => ?))`, the extension for the hole will be:

$$I = extend(\iota, \texttt{input.map(l => ?)})$$

$$extend(\iota, \texttt{input.map(l => l.filter(e => ?))}) = \bigcup_{\iota' \in I} extend(\iota', \texttt{l.filter(e => ?)})$$

Applying OE where it matters Once we have the ability to *extend* the input context, yielding all reachable valuations for new variables, the OE-reduction is then performed by evaluating the completion on the extended inputs. To do this, we expand the definition for observers of input-output examples from that given in Section 7.2:

Definition 7.5 (Input-output example observer). For an input-output predicate $\iota \rightarrow_{T^0} o$ we define the observer $\pi_{\iota \rightarrow_{T^0} o}$ as a set of valuations mapped to their execution results:

$$\pi_{\iota \rightarrow_{T^0} o}(m) = \{\sigma \mapsto [\![m]\!](\sigma) \mid \sigma \in extend(\iota, T^0)\}$$

Notice that while the observer is evaluated on the completion, separating programs based on all possible values for the variables, the predicate $\iota \rightarrow_{T^0} o$ evaluates and tests the completed program $T^0 \triangle C$. In our example, values for `a` and `b` allow us to perform the OE-reduction while enumerating the body of the function parameter, so long as we check for the specification on the full completed program.

Higher-order functions as non-user sketches The ability to fill in sketches through synthesis is useful not only when the user explicitly specifies them, but also when higher-order operations are part of the synthesis vocabulary. In a bottom-up enumeration, a function application is set as the root node of children that had already been enumerated, but since the vocabulary for the function parameter is different (namely one that includes the function's arguments, and perhaps excludes all variables outside the function's scope) it needs to be enumerated separately. Additionally, it would not do to simply enumerate all functions for k arguments once and reuse them, since without values for the new variables this enumeration would not be able to prune its space, and with values from a different context the pruning will be incorrect. In such cases, the synthesis loop is used to essentially enumerate sketches: observational equivalence is employed for all sub-components except for the function bodies, which remain as holes. For example, when enumerating `map`, some list `lhs` to be mapped will be drawn from the expressions that have already been enumerated, and applying `map` will create the sketch `lhs.map(?)`. These holes are then filled by an inner synthesis sub-task that generate sketch completions, as elaborated in this section. This is the technique used to synthesize list and dictionary comprehensions by Ferdowsifard et al. [2020].

Table 1. The impact of non-example predicates on the size of the search space. $S_1 = \mathcal{E}$, $S_2 = \mathcal{E} \cup NSP$, $S_3 = \mathcal{E} \cup nonNSP$, $S_4 = \mathcal{E}^- \cup NSP \cup nonNSP$. $\mathcal{E}$ is a set of $\iota \rightarrow o$ predicates. $\mathcal{E}^- \subset \mathcal{E}$ is insufficient to reach the target program. For each specification set, progs measures the number of programs enumerated until the result r was reached. Enumeration order is fixed, so changes in progs attest reduction or inflation of the space. Notice that the space enumerated for S_1 is the original OE-reduced space from [Albarghouthi et al. 2013]. Completed online signifies the popularity of the benchmark task on CODEWARS.COM. $h(r)$ is a shorthand for $height(r)$ (recall that it is zero-based), and $t(r)$ for $terms(r)$.

		completed			S_1		S_2		S_3		S_4		
benchmark name	difficulty	online	$h(r)$	$t(r)$	$\lvert\mathcal{E}\rvert$	progs	$\lvert S_2\rvert$	progs	$\lvert S_3\rvert$	progs	$\lvert\mathcal{E}^-\rvert$	$\lvert S_4\rvert$	progs
isUppercase	1	13,296	2	4	4	2352	6	53	5	2458	2	5	56
keepHydrated	1	36,151	2	4	3	12414	5	4398	4	12481	2	5	3694
removeFirstLastLetter	1	60,170	3	7	5	15218	7	12081	7	22543	2	6	2369
getLastChar	2	428	2	6	4	123	5	79	5	124	2	4	52
isNegativeZero	2	2,419	2	5	3	110	5	80	4	113	2	5	83
isSquare	2	82,712	3	7	6	8674	8	7444	7	8702	4	7	7212
nerdyString	2	970	2	5	2	4574	3	3088	3	4816	1	3	1525
numDecimalDigits	2	3,510	2	4	3	7083	4	3034	4	7373	2	4	2993
rotateString	3	3,057	2	7	2	358	3	226	3	413	1	3	96
decToHex	4	29,969	2	4	3	839	5	330	5	860	2	6	316

7.4 Extending reduce Sketches

A notoriously hard problem in program synthesis is generating loops where there is data dependency between the iterations. Many synthesizers solve the problem partially or using heuristics (examples of this are brought in the extended version of this paper). RESL takes the approach of Feser et al. [2015]; Smith and Albarghouthi [2016] and many others, and falls back on a full enumeration in such cases.

Observers and context extension provide a more formal way to examine this problem. Consider `arr.reduce((acc,elem) => ?)`, where the context of the hole contains the accumulator `acc` whose values are generated by previous iterations. In the previous subsection, we saw a similar problem with `sort`, which we solved by overapproximating the element pairs. However, in this case, the only sound overapproximation will include all the values generated by all programs, which is infinite (for nontrivial vocabularies). One can think of finding all necessary values of the accumulator as a chicken-and-egg problem: we need the values for the accumulator in order to enumerate the programs effectively, but we need the programs to obtain the values. We explore this problem and possible partial solutions in depth in the extended version of this paper.

8 EMPIRICAL EVALUATION

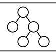
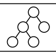
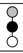
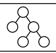
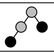
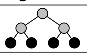
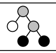
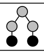
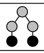
In this section, we detail our empirical evaluation. We tested two questions: (1) Is the change in the number of equivalence classes with different predicate types still tractable? (2) Is the OE-reduction on completions (rather than completed programs) in higher-order function sketches necessary?

Synthesizer Implementation We implemented an enumerating synthesizer for JavaScript programs in Scala, using the J2V8 library [j2v [n.d.]] to execute JavaScript expressions. The available predicates are (1) $\iota \rightarrow_{T^0} o$, (2) $retain_E$, (3) $exclude_E$, and (4) requiring or prohibiting subexpression types as well as concrete values (defined in the extended version of this paper). The first program encountered in a new equivalence class is kept as its representative.

8.1 Effects of Predicate Type on the OE-Reduction

We first measured the effect of predicate selection on the number of OE classes (and thus the number of programs seen when enumerating). Non negatively-stable predicates trivially increase the number of equivalence classes, but we wished to quantify this increase. We also explored the

Table 2. Performing the OE-reduction on the completion vs. the full program for sketches of the form `input.`$f(\overline{v}$ `=> ?)`, for $f \in \{$`map`, `groupBy`, `filter`, `sort`$\}$. $\mathcal{E}$ is the set of examples given to the task. $\overline{v}$ contains the new variables in the inner context. *terms(res)* is the number of AST nodes in the result. *extend*$(\mathcal{E})$ is the set of valuations for the inner variables. "Programs remaining" are programs that were not discarded by OE-reduction on the full program, and are a subset of "programs enumerated until target", where OE-reduction was performed on the completion. The right column shows the AST of the target program, and the colors denote what happens to it in full-program OE: black nodes denote discarded subtrees, and grey nodes are subtrees that were never constructed. Thus a grey or black root means that the target program was lost.

					OE on completion		OE on full program	
	benchmark name	$\lvert\mathcal{E}\rvert$	$\lvert\overline{v}\rvert$	*terms(res)*	$\lvert$*extend*$(\mathcal{E})\rvert$	programs enumerated until target	programs remaining	progress toward target program
map	plusOneTimesTen	3	2	5	9	412	412	
	noX	3	1	5	7	72192	72192	
groupBy	groupPeopleByAge	1	1	2	3	101	89	
	groupByXCoord	1	1	3	3	7916	6778	
	groupByOddEven	2	1	5	6	34954	26022	
filter	noLongWords	3	1	4	8	6821	2	
	noTeen	3	1	6	7	6392	2	
	dedupInOrder	3	3	5	22	60253	10	
sort	sortNumbers	4	2	3	33	117	13	
	sortByProperty	3	2	5	24	2579	2	
	sortByStringLength	2	2	5	14	7895	2	

effect of negatively-stable predicates discarding programs from the space. Finally, we explored the effect of mix-and-matching predicates.

Experimental setup We tested 10 numeric and string problems that do not require loops from the competitive programming website CodeWars.com. All problems were synthesized within an empty sketch. Each task was synthesized with four specifications: an example specification $S_1 = \mathcal{E}$, S_2 with the addition of negatively-stable predicates to S_1, S_3 with the addition of non-NSP to S_1, and finally S_4 that uses $\mathcal{E}^- \subset \mathcal{E}$, which is insufficient to synthesize the target program on its own, in addition to all non-example predicates in S_2 and S_3. The results are shown in Table 1. Benchmark names link to their content and sample benchmarks appear in the extended version of this paper.

Note that while in all cases but one $|S_4| > |S_1|$, past work on predicates [Peleg et al. 2018b] found that non-example predicates are easier for users to form than additional examples.

Results Adding NSP to the specification (S_2) reduced the number of equivalence classes by 45% on average (14% to 98%, med 36%). The 98% reduction in "isUppercase" is due to the *prohibit* enforcing that (for an input in $\mathcal{E}^-$) *type*(v) is not a `Number`.

Adding non-NSP to the specification (S_3) inflated the number of equivalence classes constructed by 8.4% on average (0.3% to 48.1%, med 3.4%). The top 3 increases are probably due to *retain*: "removeFirstLastLetter" (48.1%), "rotateString" (15.4%) and "isUppercase" (4.5%). However, *retain* is used in 7 of the 10 benchmarks, and in the overwhelming majority of cases, its effect is barely felt.

An assortment of predicates (S_4) resulted in an average 61% reduction (17% to 98%, med 64%) in the number of equivalence classes from S_1. Surprisingly, even though each S_4 includes at least one non-NSP, an OE-reduction with S_4 still had a fewer equivalence classes than with S_2 in all but two benchmarks: an average 23.8% reduction (med 10.1%, and max 80% fewer).

Conclusion We conclude that all our predicates are feasible for synthesis, and computational demands should not limit the user. Even the "wasteful" *retain* only causes a large increase in equivalence classes in one of the benchmarks, despite being used in almost all of them. Also, encouraging users to mix-and-match the specification is as beneficial as restricting them to NSP.

8.2 Applying OE to the Completion vs. the Completed Program

Next, we gauged the importance of performing the OE-reduction on the completion when synthesizing a solution to a sketch.

Experimental setup We synthesized the solution to 11 problems curated from the list comprehension sections of competitive programming and instructional sites, selected for having a solution of the form `input.`f`(?)` where f is `map`, `groupBy` (imported from lodash [lod [n.d.]]), `filter`, or `sort`, with no additional higher-order functions in the function parameter. Each problem was provided as a specification of input-output examples $\mathcal{E}$ and the sketch `input.`f`(?)`. The JavaScript implementations of `map` and `filter` allow for several different signatures for the function parameter, each introducing a different number of new variables (V); this number is indicated.

We enumerated the space until the target completion and compared the number of equivalence classes in an OE-reduction on the extended inputs to the number of equivalence classes in an OE-reduction on the completed program and original inputs. In addition, we checked which components of the target program were discarded as equivalent when the reduction is performed on the full program. The results are shown in Table 2. Benchmark names are links to the benchmarks.

Results When `f` is `map` the number of equivalence classes in both enumerations is equal, because the result of `map` is exactly the execution values of the completion. When `f` is `groupBy`, OE on the full program caused some type coercion: `groupBy` returns a JavaScript Object, whose keys are coerced to strings. In 2 of 3 `groupBy` benchmarks, this caused the space that was OE-reduced on the full program to be slightly smaller than that reduced on the completion with no elements of the target program lost, but in "groupByXCoord", a subexpression of the result was discarded, so the target program was never constructed. For `filter`, the drop in size of the space from OE-reduction on the completion to the full program is three orders of magnitude, and in every benchmark most or all subexpressions of the target program were discarded when reducing on the full program. For `sort`, 2 of 3 benchmarks behaved similarly, but one succeeded when reducing on the full program, aided by the fact the elements are numbers and the target program $a - b$ is very small.

Conclusion We conclude that performing the OE-reduction on the synthesized completion, using extended inputs, is *crucial* to the success of the synthesis task. The loss of programs to type coercion upon assignment and unexpected behaviors when executing on intermediate values (itself a feature of JavaScript's fault-avoidance, manifesting as an inability to test the program in stricter languages) is too high for even the pretense of correctness.

9 USER STUDY

RESL offers programmers a model for working within an unfamiliar ecosystem, e.g., an unfamiliar language. To evaluate our approach we conducted a controlled user study where experienced programmers who are JavaScript novices used JavaScript with and without RESL. In this section, we evaluate the effect RESL had on the success of the development process.

The study consisted of a control interface (REPL) and one of two treatments:

(T1) RESL: Using a fully-featured Arena that queries a synthesizer

(T2) REDL (Read-Eval-Debug Loop): The RESL Arena without synthesis, allowing only editing the program, adding and modifying examples, and viewing debug information on programs.

T2 isolates the influence of debug information and evaluation on multiple inputs in the performance of the users. This way, comparison between T1 and T2 shows the effect of synthesis itself.

Implementation We implemented a full Read-Eval-Synth Loop for JavaScript programs. The RESL Arena was implemented as a a web front-end querying a synthesis server. The Arena allows entering programs, viewing debug information on sub-expressions, and controlling the predicates and sketch of a synthesize operation, as detailed in Section 5. Details of the synthesizer implementation appear in Section 8. REDL was implemented by disabling all synthesis-related features in the Arena.

Synthesis queries started with $|\mathcal{V}| = 33$ (7 constants and 26 functions), extended with $\mathcal{V}(C) \cup ext(T^0)$ as specified in Section 6. Functions taking a callback such as `map` or `filter` were excluded from $\mathcal{V}$ to reduce the size of the space, but could be introduced by the user, as could any function in vanilla JavaScript. To keep interaction sessions feasible, queries timed out at 20 seconds, a manageable task interruption [Oulasvirta and Saariluoma 2006]. The synthesis server ran on a *c5n.4xlarge* AWS instance, with 16 cores of 3.0 GHz Intel Xeon Platinum, 42 GiB RAM, on Amazon Linux 2.

9.1 Experimental Setup

Our study consisted of 19 experienced industry programmers (4-40 years of programming experience, average 14.2, median 13) working in 8 different companies, recruited by answering a notice for programmers who cannot program in JavaScript *at all*. Participants claimed proficiency in Python (58%), C++ (42%), C# (31%), Java (21%), C (16%), SQL (10%), Go (5%), Bash (5%), Matlab (5%), and Labview (6%) (numbers do not add up to 100% as most listed multiple languages).

Each participant was randomly assigned to one of four groups:

(1) REPL (control), then RESL (T1): 4 participants
(2) RESL (T1), then REPL (control): 6 participants
(3) REPL (control), then REDL (T2): 4 participants
(4) REDL (T2), then REPL (control): 5 participants

Each participant was asked to solve four JavaScript competitive programming tasks from CODEWARS.COM, detailed below, two questions per tool, in one of two orders. Before using each tool, participants were shown a short demo. For REPL, this included executing a one-liner, assigning to a variable in the global scope, and the program history. For REDL, this included adding and editing inputs in the Arena, executing a one-liner, the output and expected output views, and viewing subexpression debug information. For RESL, participants were shown the example from Section 2.1, which included adding inputs, executing a one-liner, viewing the outputs and debug information, creating a sketch, and for the synthesis step (step 5 in Figure 1) using retain, as well as excluding the expression `[1,input.toString().length]` and prohibiting subexpressions of type Boolean to additionally demonstrate *exclude* and *prohibit*.

Tasks were shown to participants one by one, and participants moved to the next task either when they believed their answer is correct or when they gave up the task. No timeout was enforced.

Users had full access to the MDN [mdn [n.d.]], Mozilla's JavaScript documentation, including its internal search, and had the MDN entries for `map`, `filter`, and `sort` initially open. No other websites, including search engines, were allowed; the rationale being that the purpose of documentation in the study was to introduce JavaScript basic concepts and the function usage, rather than searching online for a partial or full solution. The tasks tested are used in competitions and interviews and as such are discussed online and have full solutions available.

Problem set We selected four tasks that are too difficult for our synthesizer to solve in full. The tasks are taken from the competitive programming website codewars.com, containing two numeric problems and two string problems, and are from the first three difficulty levels (of 8 available) on the site. Each problem was presented to the users as a one-liner task, increasing its difficulty—`sqdigit` appears on the site once with a difficulty level of 2 and once as a one-liner task with a difficulty level of 3. All tasks included at least one example in the task description.

The tasks are "Title Case" (`title`, difficulty: 3), "numbers divisible by given number" (`divisible`, difficulty: 1), "Alphabetically ordered" (`ordered`, difficulty: 2), and "Square every digit" (`sqdigit`, difficulty: 3). Task names are links to the original task.

The implementation and anonymized study session transcripts will be released upon publication.

9.2 Research Questions

(1) *Does RESL reduce the editing load on the programmer?* Ideally, a RESL user will have to write less code, easing their foray into an unfamiliar ecosystem. We test this question via two metrics: the number of edit iterations (i.e., programs entered into the Arena or evaluated in the REPL) the user performed during the session, including syntactically incorrect programs but excluding assigning to variables; and the portion of the code in a RESL session not written by the user.

(2) *Does RESL assist in bridging knowledge gaps when solving sub-tasks?* Programmers are adept at breaking a problem up into subproblems made up of familiar concepts [Wirth 1971]. Since RESL cannot solve the entire task for any of the tasks users must divide it into sub-tasks themselves. To a JavaScript novice, each of these involves translating a familiar idea into unfamiliar operations. We test this question by measuring the time spent consulting documentation.

(3) *Does RESL reduce programmer frustration?* We choose to examine this question not with a survey but with an empirical measure of frustration: how many of the sessions in each group were abandoned by the user.

(4) *Does RESL speed up a programmer's time to solution?* We test this question by comparing the times of RESL sessions to REPL sessions in two slices. Since knowledge transfer from session to session needs to be accounted for, we perform two between-subjects comparisons of the two halves of the within-subjects experiment (e.g., users who performed `sqdigit` second will be compared to each other and not to users who performed `sqdigit` fourth).

(5) *Are RESL users correct?* Using RESL (or just a REPL) is an iterative process driven by the programmer, who is responsible for deciding when the target program has been reached. We test whether RESL users can reach a correct program and whether they do so better than REPL users.

(6) *Does using RESL improve the user's knowledge of JavaScript?* We perform a between-subjects comparison of sessions of users who used REPL second, either after REDL or RESL. We consider differences in time, edits, need for documentation, abandoning the session, and correctness.

9.3 Results

RQ1: Does RESL reduce the number of edit iterations and the portion of the code written by the user? The average and median number of user edit iterations, as well as the percentage of the result that was synthesized (measured in terms) are shown in Table 3. The number of edits was reduced by a third or more between REPL and RESL in 3 out of the 4 tasks. In all three tasks, REDL sessions had a small reduction of edits compared to REPL, but the reduction from REDL to RESL was even larger, attributing the reduction to the synthesis features of RESL. In addition, the lower number of edit iterations also corresponds to large portions of the final program being produced by the synthesizer in RESL sessions. In `title`, as much as 93% and 96% of the final program was synthesized.

Table 3. Editing load for the programmer: average and median number of iterations where the user edited the program when performing each task. For RESL, the number of synthesize calls and portion of the result that was synthesized are also indicated. In three of four tasks, RESL reduces the editing load. In `divisible`, the easiest task, the editing load of solving the task manually was light for all users. Many RESL users wrote the program in full leading to median 0% synthesized.

	RESL						REDL		REPL	
	edit iterations		synth calls		% synth of final		edit iterations		edit iterations	
	avg	med	avg	med	avg	med	avg	med	avg	med
ordered	5	4	2.6	3	48%	63%	14	13	14.2	12
sqdigit	8.80	8	4.4	4	47%	54%	12	10	10.4	13
title	8.6	10	7.4	4	41%	13%	13	14	21.6	15.5
divisible	7.6	5	0.6	0	9%	0%	7	4	5.2	4

Table 4. Average and median times (seconds) participants spent browsing documentation. In RESL and REDL, also the time spent viewing debug information. Documentation times are greatly reduced in RESL.

	RESL				REDL				REPL	
	docs		debug		docs		debug		docs	
	avg	med	avg	med	avg	med	avg	med	avg	med
ordered	137	27	38	0	601	498	90	66	219	190
sqdigit	34	21	39	0	174	145	32	2	218	205
title	349	377	70	60	705	405	51	26	563	367
divisible	158	14	6	0	135	93	13	9	2	32

In `divisible`, the easiest of the tasks, there is no discernible difference between the groups. Generally, there was far less advantage to using the synthesizer in this task, due to a combination of a task that was easy enough that most users solved it quickly without use of the synthesizer, and necessary program elements only available via $\mathcal{V}(C)$. The only user who made use of the synthesizer for this task was one who was unfamiliar with the JavaScript modulo operator.

We answer question 1 in the affirmative: **users of RESL take fewer edits to arrive at a target program, and write less of it themselves.**

RQ2: Does RESL bridge knowledge gaps and reduce the need for documentation? Times users spent browsing documentation and exploring debug information appear in Table 4. In all tasks but `divisible`, RESL users consulted documentation far less than REDL and REPL users. Several RESL users went as far as to use no documentation at all within a given task (2 in `ordered`, 1 in `sqcode`, and 3 in `divisible`). This is compared to no REDL users, and 2 REPL users solving `divisible`. Additionally, REDL was an improvement over REPL in only one task, and only marginally, so we attribute this improvement to synthesis rather than the debug information.

We therefore answer question 2 in the affirmative: **RESL bridges the knowledge gap of novice users in place of the documentation.**

RQ3: Does RESL reduce task abandonment? We measure frustration empirically by looking at when users abandoned a task unfinished. Abandoned sessions are indicated in white in Figure 7. While some REPL users abandoned their task, no RESL user abandoned a task.

Additionally, we see that REDL actually frustrated users more, not less, than REPL, from which we conclude synthesis, not debug information, is responsible for users persisting. REDL users were actually more likely to pinpoint a problem (e.g., "I see that comparing two arrays is doing some sort of `Object.is`, rather than a value comparison") and then decide to abandon the task because they cannot find the solution to it.

We therefore answer question 3 in the affirmative: **RESL users do not quit before finishing a task.**

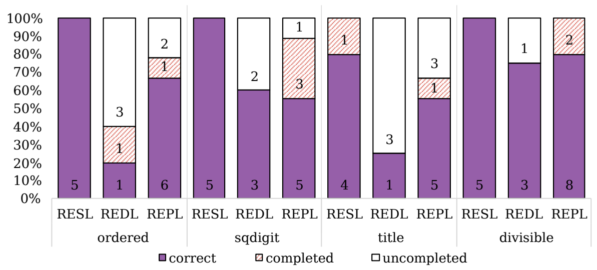


Fig. 7. Number and cumulative percentage of users who correctly completed a task (solid), completed a task (dashed), and abandoned the task (white) in each of the tools. No RESL users abandoned a task.

Table 5. Median time to perform task (seconds) for RESL, REPL, and completed REPL sessions, by task order in the session. For each task, row 1 shows times from sessions where the task was first or second, and row 2 sessions where the task was third or fourth.

	place	RESL	REPL	REPL completed
ordered	1	1061	1136	1136
	3	319	179	172
sqdigit	2	592	491	491
	4	689	240	240
title	1	1590	1671	1434
	3	925	1162	940
divisible	2	576	252	252
	4	154	166	166

RQ4: Does RESL speed up time to solution? The lengths of sessions (in seconds) appear in Table 5. The table contains both data for all REPL sessions and *completed* REPL sessions, excluding sessions abandoned by the user. Abandoned sessions also include sessions that were immediately abandoned, but typically were longer than finished sessions.

We perform a between-subjects comparison of the tasks when performed in the first half of the experiment (i.e., first or second), and those performed in the second half (i.e., third or fourth). This normalizes for JavaScript experience users acquired as the experiment progressed; times for the same task were generally shorter when performed later in the session.

Of the 8 sets examined (4 tasks and two possible positions in the sessions for each task), RESL was faster for 3 of the 8 ((`ordered`, 1), (`title`, 3), and (`divisible`, 4)). In one additional set, (`title`, 1), (as seen in Figure 7, `title` is the most abandoned task by REPL users), RESL users were not faster than REPL users who completed the task, but faster than the median REPL session.

While some of a RESL session is spent waiting for a synthesize call to return (up to 20 seconds per synthesis call), the total time is still more in half the cases. We therefore conclude **RESL does not conclusively change the time to solution**, but this may change with a faster synthesizer.

RQ5: Are RESL users correct? We define correctness of a solution as passing all examples in the task and a few additional tests for edge cases (e.g., empty strings or lists). The portions of correct answers appear in Figure 7 as the "correct" (solid color) bars.

In 3 of the 4 tasks, all RESL users arrived at the right answer. In `title`, the most challenging of the four, one RESL user finished the session with an incorrect program. This is compared to the high percentages of both uncompleted and completed but incorrect sessions in both REDL and REPL.

REPL users were more likely to declare they had finished when the solution was incorrect, whereas REDL users were more likely to give up. As noted in RQ3, this is likely related to REDL users discovering a problem using debug information, but failing to solve it on their own. Unfortunately, not enough REDL sessions were completed to allow us to analyze their correctness with any certainty, so can only hypothesize that the correctness of RESL users is the result of debug information.

Though we do not know which of the components is responsible, we answer question 5 in the affirmative: **RESL users arrive at a correct program far more often than REPL users.**

RQ6: Does RESL improve knowledge of JavaScript? To answer this question we performed a between-subjects analysis of the REPL portion of the session in groups (2) and (4)—users who used the REPL second, after RESL or after REDL.

The total time to solution, number of edit iterations, and correctness do not change between the two groups in three out of the four tasks. In `title`, the most challenging of the four tasks, users who started with RESL had considerably longer REPL sessions than users who started with REDL, and they performed an order of magnitude more edits. Time spent browsing documentation increased considerably for all tasks in sessions of users who started with RESL.

However, users who used the REPL after RESL were far less likely to abandon the task they were performing, and those who abandoned their task did so after almost twice as long (also accounting for the longer session times and larger number of edits).

We conclude from this several things: (i) using RESL for two tasks does not translate into the same level of knowledge as solving two tasks manually; (ii) having seen JavaScript functions in RESL sessions does not translate into less need for documentation when approaching tasks alone; and (iii) previous successes from using RESL ***do*** translate into persistence when approaching new tasks unaided. Although (iii) is an encouraging result for novices using RESL, we still answer question 6 in the negative: **users do not learn JavaScript from using RESL.**

9.4 Discussion

Even though some of our metrics were not improved (though not worsened) by RESL, the contribution of RESL to the users was very valuable on several fronts.

Reducing the mental load The RESL Arena includes two main components intended to reduce the mental load on the user: synthesis and the ability to view debug information on subexpressions.

Synthesis is directly related to reducing effort, as it can fill in some of the code. Conveying intent through specifications for synthesis proved easier than through documentation search queries, and was also faster.

Debug information was originally introduced to allow users to understand the synthesized programs (which are unfamiliar code). In practice, most synthesized programs were easily readable to programmers, so this use was less frequent than expected. REDL users used it much more frequently, both to pinpoint runtime errors in their own code, and to locate bugs that stem from lack of familiarity with JS, but this did not reduce their frustration; if anything, it caused more frustration over finding them but being unable to fix them.

Knowledge Transfer The results of RQ6 indicate that RESL is not a good pedagogical tool. This is not wholely unexpected, as in pedagogical terms synthesized steps are “bottom-out hints” (hints that simply tell a student what the next step is) [Suzuki et al. 2017; Vanlehn 2006].

While users did not learn more JavaScript from using RESL, we note two important things about its use. First, RESL enables programmers to tackle small tasks in a language they are not familiar with, when learning the language is not the objective. Second, successes using RESL had a lingering effect, reducing the drop-out rate of users later on, when they were programming with a REPL. Cautiosly generalizing from this, we may expect users who used RESL for a one-time task and must return to the code to be less frustrated and more persistant than ones who tackle such tasks alone.

Usage of predicates and sketches RESL provides an expressive specification mechanism through predicates and sketches. Both features were used across RESL sessions, as appears in the extended version of this paper. Most users created a sketch using the Arena at least once, marking both trivial and complex expressions to be replaced (avg 3 terms). In addition, 7 of 10 RESL users used non-example predicates (of the other 3, one did not use the synthesizer at all). Most users used *retain*, retaining expressions of 3-11 terms. *exclude* was not used at all; we believe this is because users perceived it as a “dangerous” action. This may change with more use. Type constraints were most frequently used in the `title` task, a string manipulation task and therefore susceptible to JavaScript’s type coercion, of the kind shown in Section 8.2. This variation shows the need for a large toolbox of specifications, so that the user may choose the most useful one per the specific scenario.

9.5 Threats to Validity

Lack of experience with RESL Participants were not only using JavaScript for the first time, but also using RESL for the first time. This means they had no opportunity to take in the features and recommended usage before the test session, as opposed to a REPL which most had used before. In other words, they constructed their mental model of RESL during the course of the experiment. Some users attested to this after their session, claiming they would have used synthesis more had they understood or felt comfortable with it. This also meant users were learning on the go what size sub-tasks RESL could handle, and made some overly-optimistic calls to the synthesizer.

Size of study A within-subjects study, while helping normalize the great variance in the experience of participants, means that each of the tasks for RESL and REDL had only 4 or 5 participants. With this size of study we are not able to show statisitical significance for most results.

Unclear cause of correctness The conditions of our user study do not allow us to draw conclusions about whether correctness of REPL users is a result of synthesis or examples. While both REDL and RESL users must input at least one example into the tool in order to execute their programs, and in RESL to use synthesis, REDL users input far fewer examples into the tool (on some tasks, even fewer than REPL users). Because the conditions are not level, the study setup could not be used to disprove tests are the results of correctness in REPL. While even with fewer examples used results may have strengthened the hypothesis of [Peleg et al. 2018b] that tests and intermediate execution values are a greater contributor to correctness than synthesis, the large number of uncompleted REDL sessions means we do not have sufficient data to try to make this comparison.

Differences of experience Though group assignment was random, user expertise and experience as programmers may be distributed unevenly. Experience in weakly typed or functional languages may have helped users decompose the task better or search the documentation more effectively.

10 RELATED WORK

Syntax-guided synthesis [Alur et al. 2015] is the problem formulation where the synthesizer is presented with a specification and a grammar defining the candidate program space. It is also used to describe synthesis where the target program is constructed by syntax rules. [Chasins and Newcomb 2016; Farzan and Nicolet 2019; Itzhaky et al. 2016; Udupa et al. 2013; Wang et al. 2017a] all fall within this scope. The RESL synthesizer accepts a vocabulary and a specification, adhering strictly to the definition by Alur et al. [2015]. Additionally, RESL's vocabulary is not fixed, but extended with $\mathcal{V}(C)$ with code entered by the user. This also allows us to circumvent the overfitting problem raised by Padhi et al. [2019] by extending an initial small vocabulary only as necessary.

Sketching The RESL Synthesize operation attempts to replace a subexpression in an incomplete program with a new expression. In this aspect, it is an instance of sketching (inspired by Sketch [Solar-Lezama et al. 2008] but separate from it), the paradigm of giving the synthesizer a program skeleton with a missing piece or pieces, a frequent technique to reduce the candidate space and guide the search [Hua and Khurshid 2017; Hua et al. 2018; Lubin et al. 2020; Srivastava et al. 2010, 2013; Wang et al. 2018]. Previous work using sketching made use of sketching to reduce the search space via a well-known sketch. Bornholt et al. [2016] and Chasins and Phothilimthana [2017] create a sketch internal to the algorithm and complete it. Galenson et al. [2014] use user-written sketches as filters on a list of candidates returned by the synthesizer. Making a hole in RESL extracts a sketch dynamically from an existing statement, as part of the interaction with the user. To our knowlege we are the first to support this.

Examples and other predicates RESL also extends Programming by Example, a frequent technique in program synthesis leveraging either user-provided input-outputs [Feser et al. 2015; Gulwani 2011, 2012, 2016; Osera and Zdancewic 2015; Polozov and Gulwani 2015; Wang et al. 2017a; Yaghmazadeh et al. 2018], tests [Feng et al. 2017b], or guided abstractions of user-prodvided examples [Wang et al. 2017b]. RESL also uses the granular predicates on programs *retain* and *exclude* [Peleg et al. 2018a,b]. Unlike [Peleg et al. 2018a], the RESL session is not monotonic, as predicates cannot be accumulated when the user is permitted to edit the program.

Interaction models for synthesis Few synthesis projects explore usability. Sketch-n-Sketch [Chugh et al. 2016; Hempel et al. 2019; Mayer et al. 2018] provides an interaction model for repair driven by user changes to visual objects in the output, and interleaves repairs and program edits. Rousillon [Chasins et al. 2018] introduces a Programming by Demonstration interaction model for web scraping that guides the user in demonstrating in an order that allows loops to be generalized from a single demonstration. Like them, RESL was created with the user of the tool firmly in mind, but is aimed at programmers rather than domain-specific end-users, choosing a REPL as an interaction model to enrich.

CodeHint [Galenson et al. 2014] is a debug-time synthesis tool that is called from the IDE to complete a missing next statement. CodeHint's interaction centers around a type-driven specification that can then be refined by the user using additional type information, a sketched structure that the result must meet, and boolean expressions on the program state that must hold. Like CodeHint, RESL is also aimed at programmers, but augments a REPL workflow rather than the IDE at debug-time. While CodeHint's sketches and boolean expressions are both filters on a pre-computed list of expressions, RESL incorporates both into the synthesis engine, as shown in Section 7, which ensures no programs will be lost as a result of the additional filtering.

Ellis et al. [2019] is a REPL-driven synthesizer; however, their interaction loop is between search the algorithm and the interpreter, rather than with the user.

Program repair Automatic program repair [Gazzola et al. 2019] attempts to fix a defective program based on a specification or a failing test [Le et al. 2017; Long and Rinard 2015; Xiong et al. 2017]. A major component of repair is *fault localization*, identifying the location where repair should be applied [Mechtaev et al. 2016; van Tonder and Le Goues 2018]. While RESL can be viewed as an inline repair tool, its fault localization is not automatic, but rather human-driven. Some repair tools [D Le et al. 2017; Kneuss et al. 2015; Long and Rinard 2015] rely on program synthesis in order to generate the repair, rather than predefined mutations. For example, in D Le et al. [2017], like in RESL, The repair program space is defined by the original repaired expression.

Reduction of the candidate space Reducing the number of programs a synthesizer considers is one of the major challenges of program synthesis. Gulwani [2016] and Farzan and Nicolet [2019] restrict both the DSL used for synthesis and how its operators are composed, making DSL design a key component of the solution. [Wang et al. 2017b,c] reduce finding a program in a DSL to finding an accepting path in a tree automaton of the examples, then reduce the space further via an abstraction refinement loop. Polikarpova and Sergey [2019] use separation logic proof steps to limit the space to programs that follow the proof. Feng et al. [2017a]; Guo et al. [2019] use a Petri-net of type signatures to limit the space to reachable paths in the net.

ACKNOWLEDGMENTS

This work was supported by Israel Science Foundation grants 243/19 and 2740/19, BSF grant no. 2018675, and by the National Science Foundation under Grant 1911149.

REFERENCES

[n.d.]. eclipsesource/J2V8: Java Bindings for V8. https://github.com/eclipsesource/J2V8. Accessed: 2020-4-22.
[n.d.]. JavaScript | MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript. Accessed: 2020-4-22.

[n.d.]. Lodash. https://lodash.com/. Accessed: 2020-4-22.

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *International Conference on Computer Aided Verification*. Springer, 934–950.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-guided synthesis. *Dependable Software Systems Engineering* 40 (2015), 1–25.

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 319–336.

James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing synthesis with metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 775–788. https://doi.org/10.1145/2837614.2837666

Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. 2013. It's alive! continuous feedback in UI programming. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 95–104.

Sarah Chasins and Julie Newcomb. 2016. Using SyGuS to Synthesize Reactive Motion Plans. *Electronic Proceedings in Theoretical Computer Science* 229 (11 2016), 3–20. https://doi.org/10.4204/EPTCS.229.3

Sarah Chasins and Phitchaya Mangpo Phothilimthana. 2017. Data-driven synthesis of full probabilistic programs. In *International Conference on Computer Aided Verification*. Springer, 279–304.

Sarah E Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 963–975.

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 341–354. https://doi.org/10.1145/2908080.2908103

Xuan Bach D Le, Duc-Hiep Chu, David Lo, Claire Goues, and Willem Visser. 2017. S3: Syntax-and Semantic-Guided Repair Synthesis via Programming by Examples. https://doi.org/10.1145/3106237.3106309

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. 2019. Write, execute, assess: Program synthesis with a repl. In *Advances in Neural Information Processing Systems*. 9165–9174.

Azadeh Farzan and Victor Nicolet. 2019. Modular Divide-and-conquer Parallelization of Nested Loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 610–624. https://doi.org/10.1145/3314221.3314612

Molly Q Feldman, Yiting Wang, William E Byrd, François Guimbretière, and Erik Andersen. 2019. Towards answering "Am I on the right track?" automatically using program synthesis. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. 13–24.

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas Reps. 2017a. Component-Based Synthesis for Complex APIs. In *Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2017*.

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. 2017b. Component-based synthesis for complex APIs. *ACM SIGPLAN Notices* 52, 1 (2017), 599–612.

Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. In *UIST*. forthcoming.

John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 229–239.

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*. ACM, 653–663.

Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 1 (2019), 34–67.

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

Sumit Gulwani. 2012. Synthesis from examples: Interaction models and algorithms. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*. IEEE, 8–14.

Sumit Gulwani. 2016. Programming by Examples (and its applications in Data Wrangling). In *Verification and Synthesis of Correct and Secure Systems*, Javier Esparza, Orna Grumberg, and Salomon Sickert (Eds.). IOS Press.

Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program synthesis by type-guided abstraction refinement. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28.

Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 281–292.

Jinru Hua and Sarfraz Khurshid. 2017. EdSketch: Execution-driven sketching for Java. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. ACM, 162–171.

Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 12–23.

Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. 2016. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 145–164.

Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. 2015. Deductive program repair. In *International Conference on Computer Aided Verification*. Springer, 217–233.

Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: semantics-based repair of Java programs via symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 376–379.

Sorin Lerner. 2020. Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3313831.3376494

Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 166–178.

Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program sketching with live bidirectional evaluation. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article Article 127 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276497

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A Hammer. 2019. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 619–630.

Antti Oulasvirta and Pertti Saariluoma. 2006. Surviving task interruptions: Investigating the implications of long-term working memory theory. *International Journal of Human-Computer Studies* 64, 10 (2006), 941–961.

Saswat Padhi, Todd Millstein, Aditya Nori, and Rahul Sharma. 2019. Overfitting in Synthesis: Theory and Practice. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 315–334.

Hila Peleg, Shachar Itzhaky, and Sharon Shoham. 2018a. Abstraction-Based Interaction Model for Synthesis. In *Verification, Model Checking, and Abstract Interpretation*, Isil Dillig and Jens Palsberg (Eds.). Springer International Publishing, Cham, 382–405.

Hila Peleg, Sharon Shoham, and Eran Yahav. 2018b. Programming not only by example. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1114–1124.

Nadia Polikarpova and Ilya Sergey. 2019. Structuring the synthesis of heap-manipulating programs. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 72.

Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices* 50, 10 (2015), 107–126.

Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 326–340.

Calvin Smith and Aws Albarghouthi. 2019. Program Synthesis with Equivalence Reduction. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*. 24–47. https://doi.org/10.1007/978-3-030-11245-5_2

Armando Solar-Lezama. 2008. *Program synthesis by sketching*. ProQuest.

Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching concurrent data structures. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 136–148.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. 2010. From program verification to program synthesis. In *ACM Sigplan Notices*, Vol. 45. ACM, 313–326.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. 2013. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer* 15, 5-6 (2013), 497–518.

Ryo Suzuki, Gustavo Soares, Elena Glassman, Andrew Head, Loris D'Antoni, and Björn Hartmann. 2017. Exploring the design space of automatically synthesized hints for introductory programming assignments. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. 2951–2958.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. *ACM SIGPLAN Notices* 48, 6 (2013), 287–296.

Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 151–162.

Kurt Vanlehn. 2006. The behavior of tutoring systems. *International journal of artificial intelligence in education* 16, 3 (2006), 227–265.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017a. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 452–466.

Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2018. Solver-Based Sketching of Alloy Models Using Test Valuations. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 121–136.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017b. Program Synthesis Using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL, Article 63, 30 pages. https://doi.org/10.1145/3158151

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017c. Synthesis of Data Completion Scripts Using Finite Tree Automata. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 62, 26 pages. https://doi.org/10.1145/3133886

Niklaus Wirth. 1971. Program development by stepwise refinement. *Commun. ACM* 14, 4 (1971), 221–227.

Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. 2017. Precise Condition Synthesis for Program Repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, Vol. 00. 416–426. https://doi.org/10.1109/ICSE.2017.45

Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated Migration of Hierarchical Data to Relational Tables Using Programming-by-example. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 580–593. https://doi.org/10.1145/3187009.3177735