# Perfect is the Enemy of Good: Best-Effort Program Synthesis

**Hila Peleg**
University of California, San Diego, USA
hpeleg@eng.ucsd.edu

**Nadia Polikarpova**
University of California, San Diego, USA
npolikarpova@eng.ucsd.edu

## Abstract

Program synthesis promises to help software developers with everyday tasks by generating code snippets automatically from input-output examples and other high-level specifications. The conventional wisdom is that a synthesizer must always satisfy the specification exactly. We conjecture that this all-or-nothing paradigm stands in the way of adopting program synthesis as a developer tool: in practice, the user-written specification often contains errors or is simply too hard for the synthesizer to solve within a reasonable time; in these cases, the user is left with a single over-fitted result or, more often than not, no result at all. In this paper we propose a new program synthesis paradigm we call *best-effort program synthesis*, where the synthesizer returns a ranked list of partially-valid results, *i.e.* programs that satisfy some part of the specification.

To support this paradigm, we develop *best-effort enumeration*, a new synthesis algorithm that extends a popular program enumeration technique with the ability to accumulate and return multiple partially-valid results with minimal overhead. We implement this algorithm in a tool called Bester, and evaluate it on 79 synthesis benchmarks from the literature. Contrary to the conventional wisdom, our evaluation shows that Bester returns useful results even when the specification is flawed or too hard: *i*) for all benchmarks with an error in the specification, the top three Bester results contain the correct solution, and *ii*) for most hard benchmarks, the top three results contain non-trivial *fragments* of the correct solution. We also performed an exploratory user study, which confirms our intuition that partially-valid results are useful: the study shows that programmers use the output of the synthesizer for comprehension and often incorporate it into their solutions.

**2012 ACM Subject Classification** Theory of computation → Program specifications; Software and its engineering → Automatic programming

**Keywords and phrases** Program Synthesis, Programming by Example

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2020.2

**Funding** This work has been supported by the National Science Foundation under Grant 1911149.

## 1 Introduction

Program synthesis has emerged as a promising technology for automating low-level programming tasks [24, 50, 54, 3]. For software developers, program synthesis can be an attractive alternative to online help forums when it comes to "opportunistic programming" [11], or hunting for code that will perform a small subtask needed in a larger development task. Using a *Programming by Example* (PBE) synthesizer [36, 21, 20, 19, 46, 25, 56], developers can specify the desired behavior with a set of input-output examples (or unit tests), and the synthesizer would generate a code snippet that satisfies each of the examples.

Although PBE techniques have made great strides in recent years and have been used successfully in end-user tools [23, 31, 29], they have not seen wide adoption in mainstream software development. We conjecture that one important reason is that existing synthesizers follow an "all-or-nothing" paradigm: they either return a program that is correct on *all*


*examples*, or fail. In practice, however, humans make mistakes, so examples might contains errors. Even if all the examples are correct, the program might just be too complex for the synthesizer to generate: no matter how much we improve the synthesizer, there will always be problems it fails to solve within the amount of time that the user is willing to wait. In these cases, all-or-nothing synthesis is utterly useless to the programmer: it either returns a single over-fitted result (that satisfies the erroneous specification) or, more often than not, no result at all. Iterative synthesizers [32, 39, 7] offer a partial remedy by allowing the user to refine a problematic specification, but they still waste user's time in the unsuccessful iterations.

We believe that turning PBE synthesizers into useful mainstream programming tools requires addressing two core *challenges*:

1) **Erroneous specifications:** How can we make the synthesizer robust to small errors in the specification?
2) **Hard problems:** How can we make the synthesizer helpful even if it cannot solve a problem completely?

### Switching paradigms

To address the two core challenges, we need to abandon the all-or-nothing view of synthesis and instead take the approach of successful code completion tools: an imperfect result is better than no result, as long as it is indicated as such. To this end, we propose a new PBE paradigm we dub *best-effort program synthesis*, in which the user provides examples, and the synthesizer returns a shortlist of partially-valid results, *i.e.* programs that satisfy at least some of the examples. Previous work has shown that *a*) partially-valid programs often share non-trivial fragments with the correct solution [46], and *b*) users prefer editing incorrect code to writing code from scratch [13]. Hence it is reasonable to assume that partially-valid results help the user move forward both when the specification contains errors (by generating a solution for the error-free subset of the examples) and when the problem is too hard (by generating a spacial-case program that can be used as a building block in the final solution).

### Efficient best-effort synthesis

A naive way to implement best-effort synthesis would be to use an existing synthesizer as a black box and re-run it again and again with different subsets of the specification, displaying any generated programs to the user. This is highly inefficient, however, especially when the original synthesis problem takes too long to solve: in this case, some specification subsets may still take too long. Ideally, we would like to deliver partially-valid results without requiring the synthesizer to do more work.

Our core *technical insight* is that a popular program search algorithm—bottom-up enumeration with observational equivalence reduction [55, 2]—can be extended to accumulate partially-valid results during search with minimal overhead. The extension is possible because this search algorithm is *monotonic* in the set of examples: the set of programs explored with the full specification includes all programs that would be explored with a partial specification. We formalize this monotonicity property and our extended *best-effort enumeration* algorithm in Section 3.

### Ranking partially-valid results

In general, there can be too many partially-valid results to display them all to the user, so a best-effort synthesizer needs a way to automatically select a manageable number of results (3–5) that are most likely to be useful to the programmer. It is common in program synthesis to introduce a *ranking function* for the generated programs and present top $N \geq 1$ ranked results to the user [23, 28, 43]. For the best-effort setting, we design a ranking function that incorporates both *syntactic* and *semantic* features of programs, such as simplicity and the number of examples satisfied. The details of the ranking are described in Section 4.

### Evaluating best-effort solutions

We implement our approach in a tool called Bester (Best-Effort Synthesis TERminal), which gives users access to a best-effort synthesizer from a Read-Evaluate-Print-Loop (REPL). We evaluate Bester on 79 benchmarks we collected from the 2017 SyGuS competition [4] and the Euphony benchmark suite [33]. Our evaluation shows that *i*) Bester can overcome errors in the specification and still return the correct solution in the top three results, *ii*) when a synthesis problem is hard and times out, Bester still returns useful fragments of the solution, and *iii*) Bester's ability to solve correct specifications is not impacted (Section 5). Moreover, Bester compares favorably to the naive approach of using a state-of-the-art synthesizer[1] as a black box and eliminating examples from the specification one by one.

We also performed a small exploratory user study of Bester, in which programmers used Bester to solve tasks in an unfamiliar programming language; the tasks were too hard for the synthesizer to solve completely within 40 seconds (Section 6). Our study shows that programmers make use of synthesis results for comprehension, both of the task and of the language, and that programmers often incorporate synthesis results into their solutions either by copy-pasting or by editing a partially-valid solution until it fully satisfies the examples.

### Main contributions

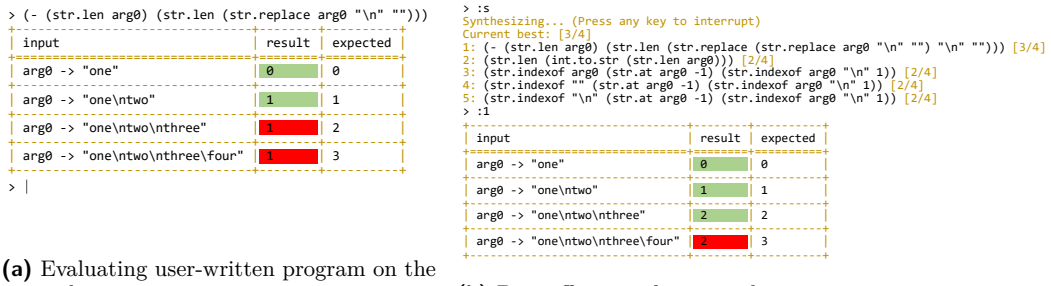
To summarize, this paper makes the following contributions:

1. *Best-effort program synthesis:* a new user interaction paradigm for PBE that is likely to yield helpful results even when the problem is ill-specified or too hard to solve completely.
2. *Best-effort enumeration:* an algorithm for efficiently collecting partially-valid solutions during enumerative synthesis.
3. A ranking function for partially-valid solutions that incorporates both syntactic and semantic properties of programs, and performs well in our experiments.
4. Bester: a prototype implementation of best-effort synthesis, shown both empirically and in an exploratory user study to be robust to specification errors and to produce useful program fragments on hard problems.

## 2 Overview

In this section, we consider a scenario that requires best-effort synthesis.

[1] We used CVC4 [44], the winner of the 2017–2019 SyGuS competitions in the PBE-Strings category.




**(a)** Evaluating user-written program on the examples.

**(b)** Best-effort synthesis results.

**Figure 1** The BESTER REPL interface.

## 2.1 A motivating example

Our example is derived from one of the benchmarks in the PBE-Strings track of the SyGuS (Syntax-Guided Synthesis) competition [5, 4]. In this competition, synthesizers are expected to generate programs in a simple language of S-expressions with built-in operations on integers (such as `+` or `-`) and strings (such as, `str.len` and `str.replace`). A benchmark in the PBE-Strings track is given by a set of input-output examples and a grammar that defines the space of candidate programs (*i.e.* the relevant subset of the SyGuS language). These benchmarks mimic small tasks performed by programmers, and some are directly derived from StackOverflow questions.

In this scenario, a programmer is attempting to solve a task that asks them to count the number of line breaks in a string. They are using a development environment enriched with a synthesizer: they have the option to invoke the synthesizer at any point during development and incorporate (fragments of) its output into their own code.

The programmer starts by providing a set of test cases (examples):

$$e_0 = \texttt{"one"} \to 0$$
$$e_1 = \texttt{"one\ntwo"} \to 1$$
$$e_2 = \texttt{"one\ntwo\nthree"} \to 2$$
$$e_3 = \texttt{"one\ntwo\nthree\four"} \to 3$$

We notice, though the user does not, that $e_3$ contains a typo in the string and would, given the expected program, only return 2 rather than 3.

The user then attempts to write a program to satisfy their test cases by computing the difference in length between the input string, `arg0`, and `arg0` with newlines removed:

```
(- (str.len arg0) (str.len (str.replace arg0 "\n" "")))
```

The user executes their tests, and only $e_0$ and $e_1$ pass, as shown in Figure 1a. They might not immediately realize that the reason for this behavior is the unexpected semantics of `str.replace` in the SyGuS language, which only replaces the first instance of the substring rather than all instances. Because $e_2$ fails as well as $e_3$, the typo in $e_3$ goes unnoticed.

At this point, the user decides to delegate solving the task to the synthesizer. Running the state-of-the-art synthesizer CVC4 [44] on this synthesis query yields the result:

```
(ite (str.contains (str.replace arg0 "\n" "") "\n")
  (ite (str.suffixof (str.at arg0 (str.len "\n")) arg0)
    (str.len "\n") (str.indexof arg0 "\n" 1))
  (ite (str.prefixof arg0 (str.replace arg0 "\n" arg0)) 0 1))
```

This program satisfies all the test cases provided to the synthesizer, but it is so complex that the user will most likely discard it without reading and be none the wiser about the typo in the tests or their misconception about the semantics of `str.replace`.

Running our tool Bester, on the other hand, produces a ranked list of synthesis results, as shown in Figure 1b. The first result in this list is:

```
(- (str.len arg0) (str.len (str.replace (str.replace arg0 "\n" "") "\n" "")))
```

which is relatively simple and in fact similar to the user's initial solution (except that it calls `str.replace` on the input string twice). Contrasting the outputs of the initial program and this result helps the user realize their misconception about `str.replace`, while the tool's failure to solve $e_3$ is likely to call their attention to the typo.

### Best-effort synthesis for hard specifications

Consider a slightly different specification our programmer could have provided, where examples $e_0, e_1, e_2$ are as before, but example $e_3$ is replaced with

$$e_3' = \texttt{"one\ntwo\nthree\nfour\nfive\nsix\nseven\neight"} \rightarrow 7$$

The programmer asks the (traditional) synthesizer for help, but after 30 seconds of waiting, their patience is exhausted, and they interrupt the synthesizer before it can produce any results. The reason this problem is taking so long to solve is that the SyGuS language contains no general solution that works for an arbitrary number of newlines, so the shortest program that satisfies $e_3'$ contains seven calls to `str.replace`; programs of this size present a challenge for state-of-the-art synthesizers. Once again, the user just wasted their time and is back to square one.

Although this particular example seems contrived, the general scenario where the user is unaware of the limitations of the synthesis algorithm and gives it more than it can handle, is very common. If the programmer is using Bester, however, and interrupts it after 30 seconds, they would get exactly the same set of results as in the previous scenario, shown in Figure 1b. This is because Bester always searches for solutions to all subsets of input examples simultaneously, and the solution for $\{e_0, e_1, e_2\}$ is much smaller—and hence will be discovered much earlier—than the solution for the full set of examples.

## 2.2 Background: Observational Equivalence Reduction

Before we explain how Bester is able to generate such partially-valid results efficiently, we must introduce the baseline synthesis technique we build upon: bottom-up enumeration with *observational equivalence reduction* [55, 2], or OE-reduction for short. Program synthesizers work by searching a space of candidate programs until they encounter one that satisfies the specification. The central challenge of program synthesis is the astronomically large size of the search space, so different synthesis techniques find different ways to reduce the space, *i.e.* exclude large chunks of the space from consideration.

For illustration purposes, in this section we will consider the program space defined by an artificially small grammar, shown in Figure 2a. This grammar allows using only two integer literals (`0` and `3`), one string literal (`" "`), a single variable (`input`), and three operations: `+`, `str.indexof`, and `str.substr`.

### Bottom-up enumeration

Bottom-up enumeration is a synthesis technique that maintains a *bank* of enumerated programs and constructs new programs by applying production rules to programs from the bank. Recall the grammar in Figure 2a. We begin enumeration with an empty bank, so in the first iteration we are limited to production rules that require no subexpressions—literals and variables; this yields the programs `0`, `3`, `" "`, and `input`, which are added to the bank. In the following iterations, production rules that require subexpressions are applied to the programs in the bank: for example, the rule $Int \to$ (`+` $Int$ $Int$) is applied to all pairs of $Int$ expressions, creating new programs `(+ 0 0)`, `(+ 0 3)`, `(+ 3 0)`, and `(+ 3 3)`, as seen in Figure 2b.

The enumeration is generally performed in the order of height: we first construct all programs of height 0, then height 1 and so on; each iteration constructs all programs of height $n + 1$ using the programs of heights up to $n$ stored in the bank. As a consequence, discarding even a few programs from the bank can drastically reduce the number of programs to be enumerated in future iterations.

### Equivalence reduction

A natural candidate for discarding from the bank is a redundant program, *i.e.* , a program that is functionally equivalent to another program in the bank. In our example, the program `(+ 0 3)` is functionally equivalent to the program `3`, and hence can be safely discarded. State-of-the-art bottom-up synthesizers [55, 2, 6] use a more aggressive notion of program equivalence called *observational equivalence*, which is also easier to check: two programs are considered equivalent if they evaluate to the same output for every input in the user-provided set of examples.

▶ **Example 1.** Let us assume two pairs of input-output examples

$e_0 =$ `"The Demolished Man"` $\to$ `"Demolished"`
$e_1 =$ `"The Stars My Destination"` $\to$ `"Stars"`

We follow the enumeration of programs with OE-reduction, summarized in Figure 2b.

First, we create an *input vector*, which in this case contains two inputs:

$$\langle \texttt{"The Demolished Man"}, \texttt{"The Stars My Destination"} \rangle$$

The algorithm evaluates each constructed program point-wise on the input vector, producing an *output vector*. Two programs are deemed observationally equivalent if their output vectors are equal.

**Height** 0**:** First we enumerate programs of height 0 (programs 1–4 in Figure 2b). The program `0` is a literal and evaluates to 0 on every input, resulting in the output vector $\langle 0, 0 \rangle$. Likewise the programs `3` and `" "` result in $\langle 3, 3 \rangle$ and $\langle$`" "`, `" "`$\rangle$ respectively. The program `input` (the input variable) yields the output vector $\langle$`"The Demolished Man"`, `"The Stars My Destination"`$\rangle$. Since all four output vectors are different, all four programs are added to the bank.

**Height** 1**:** Next, we enumerate programs of height $n + 1$ by applying production rules in the grammar to programs from the bank at heights up to $n$ (in this case, up to 0). The production rule for `str.indexof` requires two arguments of type string, and will be applied to all combinations of string programs of height 0. This will produce, among others, the program `(str.indexof " " " ")` with the output vector $\langle 0, 0 \rangle$. Notice that the bank already contains a program with this vector: the program `0`. The algorithm therefore discards `(str.indexof`

$$
\begin{array}{rcl}
\mathit{Start} & \to & \mathit{String} \\
\mathit{Int} & \to & 0 \mid 3 \\
& \mid & (\texttt{+}\ \mathit{Int}\ \mathit{Int}) \\
& \mid & (\texttt{str.indexof}\ \mathit{String}\ \mathit{String}) \\
\mathit{String} & \to & \texttt{" "} \mid \texttt{input} \\
& \mid & (\texttt{str.substr}\ \mathit{String}\ \mathit{Int}\ \mathit{Int})
\end{array}
$$

**(a)** A small grammar in the SyGuS format. Notice that the language is limited to the literal constants that appear here.

| # | program | output on $e_0$ | output on $e_1$ | equivalent to |
|---|---|---|---|---|
| 1 | `0` | 0 | 0 | |
| 2 | `3` | 3 | 3 | |
| 3 | `" "` | `" "` | `" "` | |
| 4 | `input` | `"The Demolished Man"` | `"The Stars My Destination"` | |
| 5 | ~~`(+ 0 0)`~~ | 0 | 0 | #1 |
| 6 | ~~`(+ 0 3)`~~ | 3 | 3 | #2 |
| 7 | ~~`(+ 3 0)`~~ | 3 | 3 | #2 |
| 8 | `(+ 3 3)` | 6 | 6 | |
| 9 | ~~`(str.indexof " " " ")`~~ | 0 | 0 | #1 |
| 10 | `(str.indexof " " input)` | $-1$ | $-1$ | |
| 11 | ~~`(str.indexof input input)`~~ | 0 | 0 | #1 |
| 12 | ~~`(str.indexof input " ")`~~ | 3 | 3 | #2 |

**(b)** An enumeration of the grammar by height.

**Figure 2** The enumeration in Example 1. Programs are generated from the grammar by height, first productions requiring only a terminal, and next productions requiring a subtree, taken from previously seen programs.

`" " " ")` and does not add it to the bank. In general, the algorithm maintains an *invariant* that the bank contains at most one representative of any observational equivalence class.

The same production rule also generates the program `(str.indexof input " ")`. This program seems helpful for solving the given examples; however, its output vector is $\langle 3, 3\rangle$, whose equivalence class already has a representative, the program `3`, so the program `(str.indexof input " ")` will be discarded. Unlike in the case of `(str.indexof " " " ")`, this seems an imprudent decision. However, it is in fact sound to do so *for these inputs*: so long as we do not care about differently structured inputs, `(str.indexof input " ")` and `3` are completely interchangeable. If the user introduces another example with a new input such as `"Virtual Unrealities"`, the new extended output vectors will be $\langle 3, 3, 3\rangle$ and $\langle 3, 3, 7\rangle$, and the two programs will no longer be equivalent.

## 2.3 Our approach

Next we describe how Bester modifies the baseline OE-reduction enumeration technique from the previous subsection to maintain a ranked list of partially-valid programs. If the search happens to encounter a program that fully satisfies the specification, it stops; otherwise, if the search is interrupted before a solution was found, Bester simply returns the current list of partially-valid results to the user. We refer to this modification of OE-reduction search as

*best-effort enumeration*; Section 3 details the search algorithm and its correctness.

### Searching for all example subsets

Recall the task from Section 2.1, where the user is trying to count line breaks in a string, but has an error in the example $e_3$. We would like to show the programmer the following partially-valid yet useful program $p^*$, which satisfies examples $\{e_0, e_1, e_2\}$:

```
(- (str.len arg0) (str.len (str.replace (str.replace arg0 "\n" "") "\n" "")))
```

Since we do not know a-priori which subset of examples would yield a useful result, we would like the synthesizer to simultaneously search for programs satisfying *all* non-empty subsets of $\{e_0, e_1, e_2, e_3\}$ (thus, including $\{e_0, e_1, e_2\}$).

Note that many synthesis techniques are not amenable to such simultaneous search: for example, in constraint-based synthesis [52, 26], a run of the synthesizer with the full set of examples would never construct $p^*$, because it does not satisfy $e_3$. We observe that unlike most synthesis techniques, the OE-reduction algorithm has the ability to maintain solutions for all example subsets with little to no overhead, thanks to a curious monotonicity property: *adding a new example never excludes programs from the enumeration.*

Let us illustrate this property on our running example. Consider a *hypothetical* run of an OE-synthesizer on the examples $\{e_0, e_1, e_2\}$, and assume that in this run $p^*$ is added to the bank. We conclude that $p^*$ is the first program the synthesizer constructed that produces the output vector $\langle 0, 1, 2\rangle$, and hence has been chosen as the representative of the $\langle 0, 1, 2\rangle$ equivalence class. Now consider the *actual* run of the synthesizer, on the full set of examples $\{e_0, e_1, e_2, e_3\}$; we argue that in this run $p^*$ must be chosen as the representative of the $\langle 0, 1, 2, 2\rangle$ equivalence class and cannot be discarded by OE reduction. To see why, assume a different program $p'$ is chosen as the representative; then $p'$ would have been enumerated before $p^*$ and would also return $\langle 0, 1, 2\rangle$ on the first three examples; but this contradicts our assumption that $p^*$ is the representative for $\langle 0, 1, 2\rangle$.

In other words, since each additional example *refines* the partition of the program space, the bank in the actual run must be a superset of the bank in the hypothetical run. Moreover, the output vector of each program in the bank is already computed as part of performing OE-reduction, and compared to the expected output vector; hence, performing a slightly more complex check for the purpose of identifying partially-valid results incurs only minimal overhead.

### Ranking best-effort candidates

A best-effort enumeration as described above might accumulate multiple results satisfying each subset of the examples. However, we cannot simply show them to the user in the order in which they are discovered: trivial programs such as a literal or variable satisfying one or two of the examples would be discovered immediately, but would often be a poor candidate. For instance, in the example from Section 2.1, the program `0` satisfies $\{e_0\}$, the program `(str.indexof arg0 "\n")` satisfies $\{e_4\}$ (the erroneous example), and the program `(ite (str.contains arg0 "\n") 1 0)` satisfies $\{e_0, e_1\}$. All of these will be discovered fairly early on in the enumeration.

Instead, the partially-valid programs in the bank need to be *ranked* so that a manageable number (no more than 5) of promising programs can be returned to the user. We have developed a simple ranking function for Bester that takes into account both syntactic and semantic properties of programs, and performs well empirically. Section 4 details our ranking

function and discusses other possible rankings. Intuitively, our ranking rewards programs that satisfy more examples, programs that use all of their inputs (the so called *relevancy requirement* inspired by other synthesis techniques [20, 27]), smaller programs, and programs where the incorrect outputs are close to the expected outputs. Among the programs listed above, `(str.indexof arg0 "\n")` and `0` both satisfy one example, but the former is preferred by our ranking because it uses its input.

## 3 Best-Effort Enumeration With Observational Equivalence

In this section, we detail the way an enumerative search with observational equivalence can be used to find and rank best-effort results to a synthesis query.

Let us consider the challenge in finding a best-effort solution. Since the set of user-provided examples $\mathcal{E}$ might be unsatisfiable, we wish to return a program that satisfies some $\mathcal{E}^* \subseteq \mathcal{E}$. However, we do not know in advance whether $\mathcal{E}$ is satisfiable, and if it is not, *which* $\mathcal{E}^*$ we are searching for a solution to.

We can address this challenge with minimal effort thanks to several properties of equivalence classes.

### Refined equivalence classes

Enumerative synthesis with observational equivalence adds only one representative from each equivalence class to its bank of programs based on an equivalence relation $\equiv_I$ defined as follows:

$$p_1 \equiv_I p_2 \iff \forall \iota \in I. [\![p_1]\!](\iota) = [\![p_2]\!](\iota)$$

where the equality of execution results considers outputs, exceptions, and side effects. In a PBE synthesis query, the inputs in $I$ are derived from the example set $\mathcal{E}$ such that $I = \{\iota \mid (\iota, \omega) \in \mathcal{E}\}$.

If the enumeration that has already added to the reduced program bank the program $p$ encounters a program $p'$ such that $p \equiv_I p'$, a decision is made which one will be the *representative of the equivalence class* $[p]$ that both $p$ and $p'$ inhabit. The representative is then kept in the program bank and the other program is discarded. In most synthesizers that perform the enumeration in layers (i.e., first programs of height 0, then of height 1, etc.), the first program encountered from each equivalence class is selected as its representative, as was shown in Figure 2b.

Now consider $\mathcal{E}' \subset \mathcal{E}$, a non-empty subset of examples, and its input set $I'$. It is easy to see that $\equiv_I$ is a *refinement* of $\equiv_{I'}$, since it is the intersection of $\equiv_{I'}$ and $\equiv_{I \setminus I'}$. This means that $\equiv_I$ refines the partition into equivalence classes made by $\equiv_{I'}$, or that for a program $p$ in the candidate program space, $[p]_{\equiv_I} \subseteq [p]_{\equiv_{I'}}$.

We notice that if selection of the representative is deterministic, then if $p$ was the representative of $[p]_{\equiv_{I'}}$, the less refined (and possibly larger) equivalence class, then $p$ will also be the representative of $[p]_{\equiv_I}$: representative selection has determined $p$ to be the representative against each of the programs in $[p]_{\equiv_I}$ when it was decided to be the representative of $[p]_{\equiv_{I'}}$.

This means that if $p$ was included in the bank of programs in a less refined enumeration with OE-reduction, $p$ will be in the program bank of a more refined enumeration, one with more examples.

**Algorithm 1** A best-effort enumeration

**Input:** $\mathcal{E}$ a user-provided example specification, $\mathcal{G}$ a grammar, $f$ a fitness function, $maxResults$ the maximum number of results to return to the user
**Result:** Top $maxResults$ synthesized programs

```
1  programBank ← ∅
2  resultCandidates ← PriorityQueue()
3  while timeout has not passed do
4  |  foreach prodRule ∈ 𝒢 do
5  |  |  k ← arity(prodRule)
6  |  |  foreach (arg_1, ..., arg_k) ∈ programBank^k do
7  |  |  |  if (arg_1, ..., arg_k) is suitable for prodRule then
8  |  |  |  |  newProg ← prodRule(arg_1, ..., arg_k)
9  |  |  |  |  if ∀p ∈ programBank. p ≢_I newProg then
   |  |  |  |  |  /* Found the representative of a new equivalence class,
   |  |  |  |  |     add to the bank                                    */
10 |  |  |  |  |  programBank ← programBank ∪ {newProg}
11 |  |  |  |  |  exec ← {(ι, [[newProg]](ι)) | ι ∈ I}
12 |  |  |  |  |  if exec ∩ 𝒢 ≠ ∅ then /* newProg partially satisfies 𝒢  */
13 |  |  |  |  |  |  resultCandidates.insertWithPriority(newProg, f(newProg, 𝒢))
14 |  |  |  |  |  end
15 |  |  |  |  |  if exec = 𝒢 then /* newProg fully satisfies 𝒢          */
16 |  |  |  |  |  |  break all loops
17 |  |  |  |  end
18 |  |  |  end
19 |  |  end
20 |  end
21 end
   /* Either timeout has passed and or a fully satisfying program was
      found.  We now return a list of options by rank.                 */
22 results ← List()
23 for i = 1 to min(maxResults, resultCandidates.size()) do
24 |  results.append(resultCandidates.getFront())
25 end
26 return results
```

Notice that, despite the use of an inputs *vector* in Section 2.2 (and in practical implementations), the operations are unordered. This means that it does not matter which of the examples are missing from $\mathcal{E}^-$ for the property to hold.

## 3.1 Finding best-effort solutions

Fortunately, since performing observational equivalence with $\mathcal{E}$ is a refinement of any strict, nonempty subset of $\mathcal{E}$, we can essentially test all nonempty subsets of $\mathcal{E}$ simultaneously. Representative selection ensures we will see all programs we would see enumerating a subset of the examples, so we can simply collect programs that satisfy any of the examples, instead of ones that satisfy *all* of them.

Lines 4 – 8 of Algorithm 1 are a simple bottom-up enumeration of the space, applying

each of the production rules to each of the programs previously added to the program bank, generating additional programs. Lines $9-10$ are the implementation of the OE-reduction, adding to the program bank only programs that are the first of their equivalence class to be encountered. Line 15 is the stopping condition for any PBE synthesizer: whether executing each input leads to its expected output. It is simply lines $12-14$ that "piggyback" on the enumeration with observational equivalence, collecting programs that satisfy any of the examples and create the best-effort search.

This means that when enumerating the example in Section 2.1, the program

```
(- (str.len arg0) (str.len (str.replace (str.replace arg0 "\n" "") "\n" "")))
```

is produced by the algorithm on line 8. In a regular observational equivalence reduction, the program will be added to the reduced program bank on line 10 for use in enumerating larger programs, and the next step would be to perform the check on line 15, testing whether it fully satisfies the specification. Since it satisfies 3 of the 4 examples, a simple enumeration would not return it and enumeration would continue searching for a single fully-satisfying program to show the user.

In a best-effort enumeration, the condition on line 12 admits programs that satisfy any nonempty subset of $\mathcal{E}$. The program is added to the list of best-effort results, of which the best results will be returned to the user.

The correctness proposition of observational equivalence [2] guarantees that if a program that satisfies $\mathcal{E}$ exists in the space, we will encounter exactly one such program, as other programs satisfying $\mathcal{E}$ are in its equivalence class and are not part of the reduced program space. However, if we consider any strict subset, this guarantee no longer holds: when partitioning the space of programs possible in the grammar based on observational equivalence for $\mathcal{E}$, any $\mathcal{E}' \subset \mathcal{E}$ is now represented by a number of equivalence classes in the program space instead of just one. In other words, more than one program satisfying $\mathcal{E}'$ may be encountered in the course of the enumeration.

This means there are two dimensions in which our goal is no longer unique: along an enumeration, we are looking for a program that satisfies one of exponentially many $\mathcal{E}' \subseteq \mathcal{E}$, and there can be many such programs for each $\mathcal{E}'$. However, since the results of a best-effort enumeration are intended for consumption by a user, we must limit ourselves to a small number of returned results. This means that in the course of an enumeration based on $\mathcal{E}$ programs that satisfy any nonempty subset of $\mathcal{E}$ are collected, and the *best* few are returned to the user. This is determined by a fitness function used to rank the programs in line 13 of Algorithm 1.

We will introduce our fitness function in the next section.

## 4 Fitness Function

As we have shown in Sections 2 and 3, more than one program can satisfy the same number of specifications. In this section, we discuss the considerations in constructing the fitness function used in our implementation of Bester, and suggest additional parameters that could be added for other synthesizers.

The composition of the function is:

$$f(p, \mathcal{E}) = 3 \cdot \mathit{satisfied}(p, \mathcal{E}) + 2 \cdot \mathit{relevancy}(p) + \mathit{distance}(p, \mathcal{E}) + \mathit{size}(p)$$

We now break down each of these elements.

### Examples satisfied

Since a program satisfying one example and a program satisfying all examples but one are not equally good, we use the portion of examples satisfied in our ranking of the program.

$$\mathit{satisfied}(p, \mathcal{E}) = \frac{|\{(\iota, \omega) \in \mathcal{E} \mid [\![p]\!](\iota) = \omega\}|}{|\mathcal{E}|}$$

This portion of the fitness function is the most strongly weighted, as we still give the most importance to the best effort, *i.e.* solving the largest portion of the specification.

### Relevancy

Given two programs that solve the same number of examples, we prefer one that uses more of its input. For example, let us assume a grammar with two input variables, `arg0` and `arg1`, and three programs that satisfy 2 of 3 examples in $\mathcal{E}$:

$p_1 =$ `true`
$p_2 =$ `(str.contains arg0 " ")`
$p_3 =$ `(str.prefixof arg1 arg0)`

Intuitively, we are certain we want $f(p_1)$ to be the lowest of the three, but in all likelihood, we also want to reward $p_3$ for using all available input from the user. This is a tactic employed by other synthesis tools such as [20, 27].

We define for all variables $\mathcal{V}$ available in the grammar:

$$\mathit{relevancy}(p) = \frac{|\{var \in \mathcal{V} \mid var \in p\}|}{|\mathcal{V}|}$$

### Distance from output

While we strongly reward a program for each satisfied example, we also wish to reward programs that do "better" with regard to the remaining examples.

Currently we include this element only for synthesis tasks that search for a string program. For strings, being closer to the expected output can be seen as returning a subset or superset of it, or constructing a close string. This is easily rewarded by using Levenshtein Distance [34] to measure the distance of the *unsatisfied* example results from the intended output. While this component may not be suitable for numeric types, for other structured types such as lists or trees, other such structured distance metrics can be employed in place of $LD$.

We denote $\mathcal{E}^- = \{(\iota, \omega) \in \mathcal{E} \mid [\![p]\!](\iota) \neq \omega\}$ to be the unsatisfied examples, and define:

$$\mathit{distance}(p, \mathcal{E}) = \begin{cases} \underset{(\iota,\omega)\in\mathcal{E}^-}{\text{avg}} (\{1 - \frac{LD([\![p]\!](\iota),\omega)}{\max(|\omega|,|[\![p]\!](\iota)|)}\}) & p \text{ is a string program and } |\mathcal{E}^-| > 0 \\ 0 & \text{o.w.} \end{cases}$$

While we include this in the fitness function, we do not weight it as high as some of the other components as we do still want to allow other logic that may help the user toward the correct answer, e.g., constructing a complement of the result in order to remove it, to rank well and be displayed.

### Program size

Finally, we incorporate the size of the program into the function. In a regular enumerative synthesizer, ranking by size is implicit, as programs of a lower height will be reached first. Since programs of a lower height are *simpler programs*, this tactic is employed in many synthesizers. In best-effort synthesis we may encounter programs of very different sizes that satisfy the same examples before we reach the timeout. We therefore add the height of the program into the ranking to prefer shorter ASTs.

Additionally, we would like to distinguish between programs of the same height. To do this, we use $terms(p)$, the nubmer of nodes in the AST of $p$. For example, $p_1 =$`(str.at arg0 (+ 1 1))` and $p_2 =$`(str.++ (str.++ " " " ") (str.substr arg0 1 1))` are both programs of height 2, but $terms(p_1) = 5$ whereas $terms(p_2) = 8$.

Since programs are eventually displayed to a user, given two programs of the same height that are indistinguishable by other parameters, we would like to show the user first the one that is easier to read, or the overall-smaller one.

Together, we define:

$$size(p) = \frac{1}{height(p) + 1} + \frac{1}{terms(p)}$$

### Including other data

In a domain where not all specifications are created equal, some may be ranked as more important than others. For instance, examples that detail an error scenario may be deemed more or less important than examples that specify a simple output value. Likewise, if not all specifications are examples [40], an importance ranking between different specification types can be used to decide which are more likely to be dropped.

Finally, we address the fact that our fitness function is not learned. In theory, a model could be trained to compute a fitness function according to desired program rankings, or to provide features for a fitness function (e.g., [8, 33] compute the probability of a program, which in their tool is used to speed up the search but could also be used for simple numerical ranking). However, the pool of programs is small, and creating a dataset of ranked best-effort programs large enough to train from, either manually or automatically, would be unreliable at best. In addition, our fitness function, both in selected features and in their weights, encodes in it what we consider to be the important aspects of a best-effort program, rather than numbers overfitted to a small dataset.

## 5 Empirical Evaluation

In this section we detail the empirical evaluation performed to validate our approach. Our experiments are based on the benchmarks of the SyGuS competition [4] and Euphony [33].

### Implementation

We implemented an enumerating, observational equivalence synthesizer for the SyGuS language in Scala, then augmented it for best-effort enumeration[2]. Best-effort solutions are accumulated as the enumeration progresses, and the top 5 results are returned. The

[2] `https://github.com/peleghila/bester`

enumeration loop of our synthesizer has a 40$s$ timeout, selected since it is a manageable length of task interruption for a human user [37].

### Benchmarks

We used a set of 79 synthesis queries from the 2017 SyGuS competition and the Euphony benchmarks. These benchmarks contain a selection of data wrangling and string transformation tasks: the SyGuS benchmarks are entirely string to string transformations but 19 of the Euphony benchmarks either have a non-string parameter or synthesize a numeric or boolean expression. Duplicate tasks between SyGuS and Euphony were removed from the original benchmark set, as well as benchmarks requiring recursion.

We initially divided them into two sets using a simple OE-based enumerating synthesizer (that does not collect best-effort results): 63 that can be solved within 40$s$, denoted "easy", and 16 that cannot, denoted "hard".

We then created a modified version of the benchmarks in the "easy" set by adding erroneous examples such as typos, off-by-one errors, etc. This was done manually and required great care in order to make sure that the additions are *i*) not consistent with the original target program, and *ii*) do not always create a new example set that is easily generalized. Of 37 modified benchmarks, two contain more than one erroneous example.

We note that while we introduced errors, it is near impossible to introduce *contradictions*, short of pairing the same input with two different outputs. Since most of the SyGuS and Euphony benchmarks include the conditional `ite` in their grammar, given enough time the inconsistency in the examples in many of the modified benchmarks can be overcome with case-splitting. The exception to this is a result that requires string constants not included in the grammar and that cannot be generated from the input.

For convenience, we use the simple OE synthesizer to make a distinction between the modified benchmarks:

1. "no-solution": benchmarks in which the synthesizer does not find a program that satisfies all examples within the 40$s$ timeout, and
2. "overfitted": benchmarks in which the synthesizer is able to find a solution to the given examples (this solution will usually be long and overfitted via multiple case splits).

Since the origin of many of our benchmarks is the `PBE-Strings` track of the SyGuS competition, we take as state-of-the-art the synthesizer/solver CVC4 [44], winner of the `PBE-Strings` track of the competition since 2017. We use `CVC4` 1.7, the most recent version available.

### Experimental setup

We generated gold-standard solutions for each of the original, unmodified 79 benchmarks. Our gold standard is more forgiving than the SyGuS competition, including both hand-written solutions for the task in the benchmark, as understood by the authors, and solutions from CVC4 that cover all examples, despite taking a different approach. Solutions by CVC4 were accepted as-is, in order to use it as a baseline, despite the fact that, as seen in Section 2.1, those solutions are at times overfitted and full of case-splits, but for every such case a hand-crafted gold-standard solution was also added.

All benchmarks were run on a Lenovo laptop with a i7 quad-core CPU @ 2.60GHz with 16GB of RAM.

### Research questions

**RQ1:** **Can Bester discard contradicting examples better than a naive search using a state-of-the-art tool?** To test this, we examine the result of running Bester on the "no-solution" portion of the modified benchmark set. We run Bester with a 40$s$ timeout, which is not enough for a simple enumerating synthesizer to find a satisfying program for these tasks. We then test whether a gold-standard program for the original benchmark was returned as the top-ranked result, and compare to the ability of CVC4 to find the gold-standard result when run first with the full example set and then with reduced example sets.

**RQ2:** **Can Bester rank a gold-standard result high when there is an overfitted, uninteded result for the example set?** To test this, we examine the "overfitted" portion of the modified benchmark set. We still ran Bester with a 40$s$ timeout, but since a fully satisfying result exists, these benchmarks terminate before the timeout. Though Bester will find a fully-satifying result to the examples, it will also return other best-effort results. We search for a gold-standard solution in the top 3 results for each task.

**RQ3:** **Can Bester find pieces of a gold-standard solution when the task is too hard for it to synthesize?** To test this, we search for pieces of gold-standard solutions in the top results when enumerating the "hard" benchmark set. This question is further examined in the user study in Section 6.

**RQ4:** **Does the best-effort enumeration in Bester interfere with its ability to solve a simple synthesis task?** In other words, can Bester solve the "easy" benchmark set, returning the gold-standard solution as the top-rated result?

## 5.1 Erroneous examples

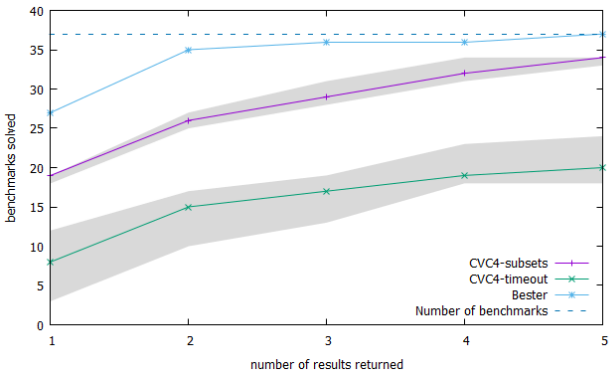
In RQ1 and RQ2, we wish to empirically quantify the effort of a user looking at a list of results. That the gold-standard solution appear *somewhere* on the list of programs shown as a result to a synthesis call is necessary but insufficient. Ideally, the user would have to look through as few programs as possible until they find the one they are looking for–and for confidence in the tool to be high, this should also be consistent.

Since CVC4 only returns one result that satisfies all examples, it will successfully synthesize none of the modified benchmarks by construction of the benchmark set. To test RQ1 and RQ2, we implemented a naive best-effort search using CVC4:

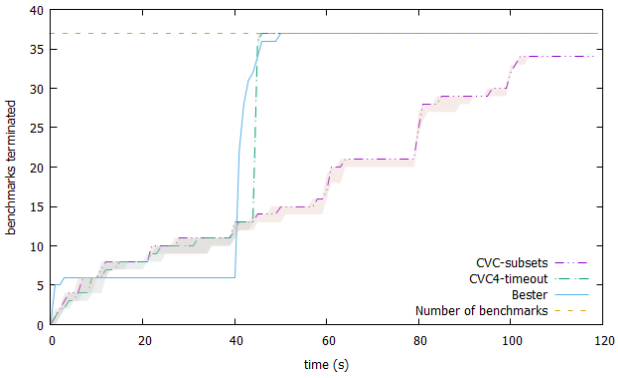
- CVC4-subsets runs on $\mathcal{E}$, and then on all subsets of size $|\mathcal{E}| - 1$ in a random order. Each such run is done with a 20$s$ timeout (a longer timeout would give Bester an unfair advantage in the measurements, and as can be seen in Figure 4b, 20$s$ is sufficient for CVC4 for most of the unmodified benchmarks), and results are accumulated in the order that they are discovered and deduplicated in-order.
- CVC4-timeout runs as CVC4-subsets, but with an additional overall timeout of 45$s$, in order to be comparable to Bester.

We ran the 37 modified benchmarks with Bester, CVC4-subsets, and CVC4-timeout. Since CVC4-subsets and CVC4-timeout depend on the random order of the dropped examples, we ran each 5 times and indicate the median and variance. The results are shown in Figure 3.

**RQ1:** *Can* Bester *discard contradictions in the example set?* Out of 31 benchmarks in the "no-solution" subset of the modified benchmarks, Bester returned the gold-standard solution first for 26, and the remaining 5 as the second solution. CVC4-subsets returned




**(a)** Number of benchmarks in which the gold-standard solution was returned for a given length of result list. More benchmarks in which a gold-standard solution was found in a shorter list is better. CVC4 runs include a random component, so indicated is the median over 5 runs, with the shaded area indicating the variance.




**(b)** Number of benchmarks that terminate within a given length of time. This is irrespective of correctness, as the tool must first terminate for its results to be judged by the user. CVC4 runs include a random component, so indicated is the median over 5 runs, with the shaded area indicating the variance. The first plateau for BESTER indicates the "overfitted" benchmark set, where a fully-satisfying but overfitted program is found within the timeout.

**Figure 3** Correctness and termination times on benchmarks containing at least one erroneous example.

the gold-standard solution within the top 3 for only 25 of the 31 "no-solution" benchmarks (over 5 runs, min 23, max 28), notably failing completely to synthesize a specification with more than one erroneous example, of which "no-solution" contains two. In addition, it only returned the gold-standard solution first for 17 of the benchmarks (min 16, max 18), with some gold-standard solutions being as low as fifth. Finally, CVC-TIMEOUT fails to return a gold-standard solution in the top 5 for 15 of the 31 benchmarks (min 13, max 18), and only returns the gold-standard solution first for 7 of them (min 2, max 9).

We therefore conclude that BESTER is effective at discarding contradictions from the specification and returning a desirable program to the user. Additionally, we conclude that our efficient best-effort implementation is more efficient than a naive approach using a state of the art synthesizer.

**RQ2:** *Can* BESTER *return a useful solution despite an overfitted program matching the examples?* Out of the remaining 6 "overfitted" modified benchmarks, BESTER shows 5 in the top three results and 4 in the top 2, exactly the same as CVC4-SUBSETS (min 4, max 6 and min 3, max 4, respectively). CVC4-TIMEOUT had 4 in the top three results (min 3, max 5)

and 3 in the top two (min 2, max 4).

We can also see the "overfitted" benchmarks in Figure 3b, as the first plateau between 3 and 40 seconds: overfitted programs are found quickly, and other program options collected along the way are also shown to the user, as opposed to enumerating a benchmark from "no-solution", which will continue until the timeout.

> We conclude that Bester performs as well as CVC4-subsets and CVC4-timeout at ranking the gold-standard solution in the top 3 when an overfitted solution exists. This is done more efficiently than a naive solution implemented with CVC4, which still pays the overhead of having to perform multiple runs.

## 5.2 Partially solving hard benchmarks

In RQ3, we examine the results of Bester on the "hard" set of benchmarks, which are benchmarks that a simple enumerating OE-reduction synthesizer cannot complete within $40s$. Bester also runs with a timeout of $40s$, but returns any best-effort results it finds. None of the results returned will be a gold-standard solution, but they may be part of a path to a solution. Therefore, to answer RQ3, we try to quanitify how much of each of the results returned by Bester can be used to construct a solution.

In order to do that, we must first define the way we measure this similarity.

### Tree similarity

In order to judge how much of a result returned by Bester is relevant to the user, we use a similarity metric between trees on the ASTs of the Bester result and the gold-standard solution. This metric essentially counts what non-trivial parts of the code can be copied out verbatim.

When computing $s(p_1, p_2)$, we look for maximal sub-expressions (or subtrees) $x$ within $p_1$ (denotes $x \in p_1$) that are also included in $p_2$. For each such $x$, if $height(x) > 0$ (i.e., $x$ is not a leaf node) we count $terms(x)$. Additionally, we reward the same term for using some identical children even if not *all* children are identical. For example, if there exist two trees, $t(x, y) \in p_1$ and $t(x, z) \in p_2$ (notice that $t$ is the same node type and $x$ is in the same location) we count the root $t$ in addition to $terms(x)$, i.e., add 1 to the accumulated similarity.

Equivalent programs that result in structurally different trees (e.g., `(str.++ "be seeing" (str.++ " " "you"))` vs. `(str.++ (str.++ "be seeing" " ") "you")`) were handled manually by first performing equivalence-preserving tree transformations on the gold standard and then computing the similarity.

Other similarity metrics were originally considered. Program repair projects often employ distance metrics between programs to choose between several possible repairs. Distance metrics for structured objects such as DiffX [1] for XMLs were applied to ASTs, and application-specific ones were crafted [15, 57]. However, the fragment mapping employed by such distances is more useful for describing insertion and deletion of code (e.g., wrapping a part of the tree in a conditional, removing a statement), whereas we are interested in pieces of code that can be used without modification.

### Usable parts of best-effort solutions

We ran Bester on the 16 benchmarks in the "hard" set. The results are shown in Table 1.

**RQ3**: *Can* Bester *return a useful best-effort solution for tasks that it cannot solve within the timeout?* On average, Bester results discover over 40% of the gold-standard solution to a

| | gold standard | | | top Bester solution | | | | closest solution to GS | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| benchmark | # GS | avg h | avg t | h | t | sim | % best GS | rank | h | t | sim | % best GS |
| 11604909 | 3 | 3.7 | 15.0 | 2 | 12 | 8 | 62% | 1 | 2 | 12 | 8 | 62% |
| 30732554 | 1 | 3.0 | 12.0 | 0 | 1 | 0 | 0% | 1 | 0 | 1 | 0 | 0% |
| 38871714 | 2 | 6.0 | 19.0 | 2 | 7 | 7 | 37% | 1 | 2 | 7 | 7 | 37% |
| 39060015 | 2 | 11.0 | 72.0 | 2 | 6 | 0 | 0% | 2 | 2 | 7 | 15 | 45% |
| 41503046 | 3 | 8.0 | 64.7 | 2 | 7 | 11 | 7% | 1 | 2 | 7 | 11 | 7% |
| 43606446 | 2 | 5.5 | 24.5 | 3 | 16 | 12 | 38% | 1 | 3 | 16 | 12 | 38% |
| 44789427 | 3 | 5.7 | 40.3 | 2 | 7 | 15 | 21% | 2 | 2 | 11 | 16 | 73% |
| bikes | 2 | 4.0 | 16.5 | 3 | 14 | 13 | 50% | 3 | 3 | 14 | 20 | 77% |
| count-total-words | 1 | 5.0 | 35.0 | 3 | 14 | 22 | 63% | 3 | 3 | 20 | 23 | 66% |
| exceljet2 | 1 | 7.0 | 43.0 | 2 | 11 | 14 | 33% | 1 | 2 | 11 | 14 | 33% |
| stackoverflow1 | 1 | 3.0 | 16.0 | 2 | 9 | 9 | 56% | 1 | 2 | 9 | 9 | 56% |
| stackoverflow2 | 1 | 6.0 | 28.0 | 3 | 20 | 24 | 86% | 1 | 3 | 20 | 24 | 86% |
| stackoverflow3 | 1 | 4.0 | 12.0 | 2 | 6 | 0 | 0% | 1 | 2 | 6 | 0 | 0% |
| strip-html | 1 | 4.0 | 15.0 | - | - | - | - | - | - | - | - | - |
| univ_2_short | 1 | 4.0 | 20.0 | 2 | 7 | 7 | 35% | 1 | 2 | 7 | 7 | 35% |
| univ_3_short | 1 | 4.0 | 14.0 | 0 | 1 | 0 | 0% | 1 | 0 | 1 | 0 | 0% |

**Table 1** Portions of the gold-standard solutions discovered by Bester for the tasks in the "hard" set. The first set of columns is information on the gold standard solutions available for a task: number and average size. The second set shows the program Bester ranked first: size, its similarity to the most similar gold-standard solution, and what percentage of the terms in the gold-standard solution is covered ($sim(p, gs)/terms(gs)$). For the closest solution to a gold-standard solution, the rank of the program in Bester's list is also indicated. t denotes terms, h denotes height (this is zero-based), sim denotes the similarity to most similar gold-standard solution.

task (or the most similar one, if there is more than one), or over 11 terms. When considering only the programs ranked first by Bester, 32% of the gold-standard solution is discovered with an average of almost 9.5 terms. In 3 of the benchmarks, the entire top-ranking Bester result was a sub-expression of the solution to the task.

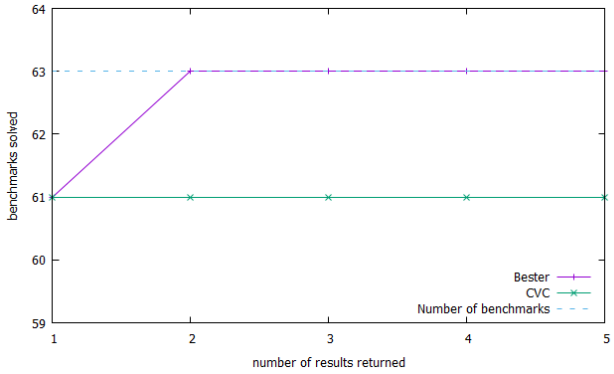
Notice that in some of the tasks (e.g., stackoverflow2) the similarity between the Bester result and its nearest gold-standard solution is greater than the number of terms in the Bester result. This is because an expression in the Bester result can repeat multiple times in the gold-standard solution.

In one of the 16 benchmarks, Bester did not find any program that satisfies at least one example, and so returned no programs. In 3 additional benchmarks, none of the programs returned had any non-trivial subtree in common with a gold-standard solution.

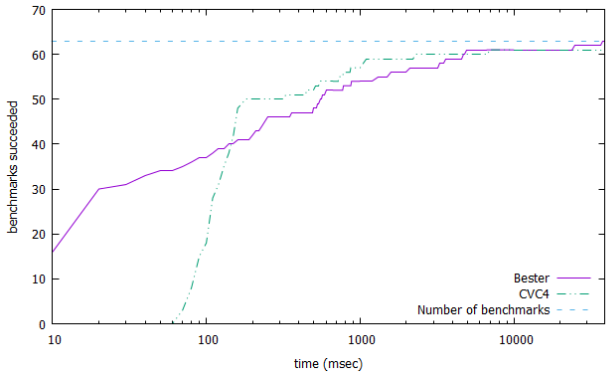
> Overall, we conclude that Bester generates results that can advance the uesr toward a solution even when they do not fully satisfy the specification. This will be further demonstrated in Section 6. Even though in some of the benchmarks none of the results had any usable components, these are still a minority (overall a quarter of the benchmarks) and the high similarity of those that did succeed indicates the approach can be of great use to a user.

## 5.3 Solving the original easy benchmarks

Since Bester ranks its results, taking into account but not relying solely on the number of examples satisfied (see Section 4), we must verify that the solutions to the original, unmodified benchmarks that can be solved by the simple OE synthesizer are still found.




**(a)** Number of benchmarks in which the gold-standard solution was returned for a given length of result list. CVC4 only returns a single result.




**(b)** Number of benchmarks that terminate within a given length of time, graph is logscale.

**Figure 4** Correctness and time to solution on "easy" benchmarks. CVC4, which was part of the baseline for correct results, is correct every task that terminates within the 40*s* timeout. CVC4 is faster, but the difference is not extreme.

To test RQ4, we ran Bester and CVC4 with a 40*s* timeout on the unmodified "easy" set of benchmarks. The results are in Figure 4.

**RQ4:** *Can* Bester *return the correct result for unmodified "easy" benchmarks?* Bester succeeds in returning a correct solution that is ranked first for 61 out of 63 of the benchmarks in the "easy" set, on par with the performance of CVC4. (Since CVC4 was used in the creation of the gold standard, it succeeds on every benchmark it terminates on within the 40*s* timeout.)

In the remaining two benchmarks, the gold standard solution is ranked second. In both of these benchmarks, the desired outputs are a substring of `arg0`, the input variable. Both also contain multiple examples where the input is unchanged. For both of these benchmarks, Bester ranks the program `arg0` before the target program, since it satisfies some of the examples and is very close to the correct output in the others, uses all the variables, and is very simple. This is rare, and when presented to a user, as in Figure 1a, the program would be accompanied by the number of benchmarks it solves, and we belive it will be easy for users to discard.

Additionally, Bester is not considerably slower than CVC4 on the benchmarks in "easy".

We conclude that implementing the best-effort enumeration in Bester does not harm its correctness on benchmarks that contain no error or contradition, and that its efficiency in such cases is not much worse than a state of the art synthesizer optimized for competitions.

## 6 An Exploratory User Study

In this section we detail the results of a small exploratory study in which 8 users were asked to use Bester to perform two tasks each. Tasks were selected from the benchmark suite presented in Section 5, from the "hard" set of benchmarks, i.e., benchmarks that could not be solved within the timeout by a simple enumerating synthesizer. Notice that these are not modified tasks, i.e., they are identical to their version in the Euphony benchmark set from which they both originated. After completing the tasks, we asked each user to answer a set of questions in a brief interview.

### Experiment setup

8 graduate students participated in the study. Users were presented with a brief task description (as it appears in a comment in the benchmark file), the examples in the benchmark, and the grammar at their disposal. As shown in Section 2.1, the semantics of some grammar elements can be misleading, particularly in edge cases.

Participants used a REPL for the target SyGuS language that is initialized with the limited grammar and the example set for the task. For each program entered, the REPL prints the output for every input in the example set. Satisfied examples (matching the example's expected output) are indicated in green. Screenshots of the REPL are shown in Figure 1. Participants could edit the program on their own or, at any point, call the synthesizer to find a program that would satisfy the examples. The Bester synthesizer runs either until a timeout of $40s$ or until interrupted by the user ("press any key" in Figure 1b). While synthesis ran, a number showing the maximum number of examples satisfied was shown and updated when new programs were found. The top 5 programs found by the synthesizer are presented to the user, and can be executed or copied. Participants could call the synthesizer multiple times in the course of one task, as a longer wait could possibly yield more results.

A task was concluded when a participant said they solved the task, when they gave up on the task, or when 20 minutes had elapsed.

Users were told that the tasks are underspecified, and they may resolve any ambiguity as they see fit. Correctness was judged according to semantic equivalence to one of the gold standard solutions for Section 5.

After performing the tasks, users were given a brief structured interview with questions about their use of the synthesizer and the helpfulness of the results. Each participant was paid $10.

### Study tasks

The two tasks given to the participants shared a SyGuS grammar, differing only in the available string literals.

**Task 1:** "stackoverflow1" in Table 1. Its comment in the benchmark file, provided to participants, was "function to replace substring".

Examples:

| arg0 | expected result |
|---|---|
| `"Trucking Inc."` | `"Trucking"` |
| `"New Truck Inc"` | `"New Truck"` |
| `"ABV Trucking Inc, LLC"` | `"ABV Trucking"` |

The available string literals were: `""`, `" "`, `"Inc"`, `"."`, `","`, and `"LLC"`.

**Task 2:** "41503046" in Table 1. Its comment in the task file was "find string in substring with lookup".

Examples:

| arg0 | expected result |
|---|---|
| `"Polygonum amphibium"` | `"Polygonum"` |
| `"Hippuris vulgaris"` | `"Hippuris"` |
| `"Lysimachia vulgaris"` | `"Lysimachia"` |
| `"Juncus bulbosus ssp.  bulbosus"` | `"Juncus bulbosus"` |
| `"Lycopus europaeus ssp.  europaeus"` | `"Lycopus europaeus"` |
| `"Nymphaea alba"` | `"Nymphaea"` |

The available string literals were: `""`, `" "`, and `"ssp."`.

### Research questions

In order to find out whether the *best-effort* paradigm can be useful to programmers, we attempt to answer the following questions:

**RQ1:** Did users apply any part of the results from Bester to their solution?
**RQ2:** Did users find the results from Bester helpful even though they do not satisfy every example?

## 6.1 Observed behavior

Participants completed task 1 in an average of 9.56 minutes and task 2 in an average of 11.35 minutes. The fastest solution was programmed in just under 5 minutes.

Of 8 users performing two tasks each, 7 successfully completed both tasks. One user failed to finish the first task within the 20 minute bound and successfully finished the second task. In addition, one user finished the second task with an incorrect result, and, as they were not satisfied with it and had time left, continued to rewrite it until they reached a correct result.

In 15 of the 16 task sessions, the users called the synthesizer at some point during the session. In task 1, 3 of the users ran the synthesizer a second time in the course of the session. In task 2, 2 of the users did so, and one ran the synthesizer a third time. One user performed task 1 without running the synthesizer at all.

Users waited for the synthesizer an average of 17.5$s$ per session while working on task 1 and 27.6$s$ per session while working on task 2, or an average of 14$s$ per individual run of the synthesizer for task 1 and an average of 18.4$s$ for task 2. Only twice did users allow their synthesis request to run until the 40$s$ timeout, both in the course of solving task 2.

7 of the 8 participants executed the top synthesis result once the synthesizer terminated. Only one user executed any result other than the top result—and they executed all results. 6 users later returned to an executed synthesis result using the REPL history and continued to edit it from there.

6 of the users used the mouse to highlight and copy a synthesized expression and paste it into their code. Two users also copied parts of a synthesized expression, but for the most part, the synthesis results that were copied by users were used in their entirety and placed within larger expressions.

Task 1 has two possible modes of solution: one using `str.substr` to slice the string up to the occurrence of `"Inc"` and using `str.replace` to replace undesirable substrings with `""`. Four users followed the synthesizer's lead in solving the task with `str.replace`, and another user attempted this and abandoned the direction.

Of the 8 users, 5 ran the synthesizer immediately upon being given task 1 (of the 3 who did not, one did not run the synthesizer at all), and 7 ran it immediately upon being given task 2.

Many of the participants struggled with the behavior of the `str.indexof` function which returns the index of a substring within a string. Unlike the simplified version included in the grammar in Figure 2a, the function takes an integer parameter which indicates at what index the search for the substring should begin. Many of the users assumed the index parameter to indicate which occurrence of the string should be returned. In the solution of task 2, users spent some time trying to get the second occurrence of `" "` under this assumption.

## 6.2 Interviews

In the interview conducted after the tasks were concluded, participants were asked about their decision to call the synthesizer (and to call it again in the course of the session, if they did so), about how they decided how long to wait for the synthesizer, and about the helpfulness of the results.

### Calling the synthesizer

Several users explained their call to synthesis as a way to search for a solution they were not seeing, or in hopes it will simply solve the task for them (or, in the case of one user, "just to see what it can do"). Some also recognized, particularly for task 2, that there may be at least a subproblem that can be solved by the synthesizer, providing them with "a start on the solution" or "a piece that can be reused".

However, many of the users explained their call to synthesis as a way to help them understand the problem: either by seeing if there was a generalization of the examples they were not considering, or to get a confirmation of their understanding, "make sure the model in [their] head was correct".

The user who performed task 1 without synthesis said they did not think there exists a simpler way to perform the task than the one they had in mind, so there was no need for synthesis.

Finally, many of the users explained that synthesized code was, to them, a good source of example programs on the inputs. Synthesized code gave them examples of *a*) the language syntax and useful available functions, *b*) the semantics of the functions, and the order of the arguments, *c*) function composition, and how different functions interact, and *d*) help dealing with what one of the users called "an unnatural collection of primitives".

### Waiting for the synthesizer

Most users who ran the synthesizer immediately at the start of the task attested that it seemed to them a good use of time to let it run as they were reading the task — it might find something and save them the effort. One user ran the synthesizer again (and to timeout) while they were thinking through a problem they had encountered, just in case.

Users could stop the synthesizer at any time. Three of the users said they used the printout of how many examples were solved by the best discovered program as an indication of

when to stop: "[as long as] it made some progress, it was fine". When the number plateaued, they "figured it solved part of the problem, but the rest isn't easy."

Frustration was also a deciding factor in willingness to wait. Users who were not having a hard time with the tasks and simply wanted some reference, terminated the synthesizer very quickly, and they just wanted to see the first results rather than be slowed down by waiting. Users who were more frustrated, especially those who entered task 2 frustrated from task 1, expressed being more willing to wait. The user who failed to finish task 1 and ran the synthesizer to timeout ($40s$) in task 2 said, "I really struggled, so even if the timeout was 10 minutes, it's worth it."

Only two of the 8 users explicitly named impatience as the criterion for deciding how long to wait for the synthesizer.

Half (4) the users re-ran the synthesizer within the course of the same task for one of the two tasks. All said it was in hopes that waiting longer would produce more or better results. One did so because they lost their train of thought and wanted to start over from a synthesized solution in order to remember what they were trying to do, and had forgotten they can call up the solutions from the last run of the synthesizer. This user also stated that, as they were struggling a bit, they were now more willing to wait for a result. Two users stated wanting to utilize time when they had stopped to think about what to do next, in case better solutions would be found. (One user who did not run it a second time said that "in hindsight, letting it run while I was thinking would have been good.") One user said they were curious as to whether there was a random component that would lead to different results.

### Helpfulness of the results

All participants stated the synthesized results were helpful to them in some way.

**Getting to a solution:** In each of the tasks, the synthesizer returned a different kind of a sub-solution. In task 1, it needed to be wrapped in more function applications to solve more cases, whereas task 2 required a case-split and the synthesizer returned a solution to one of the two cases. Some users viewed one as far more helpful than the other, though which one was not a constant. Some treated the solution to task 1 as "nearly solved the problem", whereas others saw the solution to task 1 as less helpful but the solution to task 2 as giving them the subprogram that they wanted, where "I could just steal that as a subcomponent".

**Comprehension of the language:** Participants who used the synthesizer to understand the language said synthesized results gave them "phrases" for later use and what constants were available; "here is some code, here's what it does." (In task 2, when they got used to the language, it was less helpful). Those who did not trust themselves with the language trusted synthesized code.

**Comprehension of the task:** Users also attested that synthesized results helped them better understand the task itself and in what way the examples generalized. This was particularly true in the second task which contained a case-split. Users said the result of the synthesizer classified the examples for them into the two cases of the split, or as one user said, "once I saw the response from the synthesizer, I knew exactly what the correct answer was."

## 6.3 Discussion

We return to our research questions:

**RQ1:** *Did users use* Bester *results in their code?* Participants of our study used both entire Bester results and subprograms of them in their solution code. In addition, in task 1,

several participants let the synthesizer direct the algorithm of their solution. We therefore answer this question in the **affirmative**.

**RQ2:** *Did users find the results of* Bester *helpful?* Participants of our study listed different ways in which the results of Bester were helpful to them, including (but not limited to) finding code that solves a subproblem. Synthesizer results were also widely used as a comprehension tool by the users. We therefore answer this question in the **affirmative**.

## 6.4 Threats to validity

Finally, we briefly discuss the threats to the validity of our conclusions from the study.

**Number of participants and number of tasks:** The study was conducted on 8 participants, performing only two tasks each, which is not enough to make any statistically significant claims. We therefore try to steer away from such conclusions, and instead observe and report usage patterns that occurred throughout user sessions.

**Selection of programming language:** While using the SyGuS language can be seen as an advantage of the study, mimicking a situation where users are not the most familiar with the language or API they are using, and therefore need the help of a synthesizer, it is also not the easiest programming language to read or write, and includes nontrivial semantics for some of its functions (as demonstrated both in Section 2.1 and in this section). This may lead to different results than a synthesizer for a programming language users are more comfortable reading and writing. All the participants in the study were familiar with the S-expression syntax and had some experience in using it, mitigating some of the comprehension difficulty if not the problematic semantics.

**Homogeneous participants:** Since students were recruited from a single department in a single institution, there is great similarity in their knowledge and ability. This may have resulted in similar behaviors in the course of the study.

**Inability to specify the synthesizer:** The Bester implementation used in the study was not fully-equipped for an iterative and interactive workflow, and users could not control the specifications the synthesizer attempted to solve. This also means users did not spend time on (or have a learning curve in) entering specifications or deciding what they should be. Within such a larger workflow, the observed behaviors may be different. However, we have tried to only draw conclusions about the usefulness of the results of a synthesizer iteration, rather than on the interactive incorporation of synthesis in the development workflow.

# 7 Related work

**Syntax-guided synthesis** [3] is the domain of program synthesis where the target program is derived from a set of syntax rules. [30, 55, 18, 56] all fall within this scope. FlashFill and FlashMeta [23, 42] are tools for automating string transformations and data wrangling tasks, whose DSL design centers the delicate balance between an expressive grammar, which is needed to find a solution, and a tractable enumeration. Padhi et al. [38] raise the issue of the overfitting of an over-expressive grammar, leading to programs such as the one shown in Section 2.1.

**The SyGuS competition** [5, 4] is held every year and allows solvers and synthesizers to compete for both performance and correctness on a large selection of benchmarks. The competition introduced a PBE track in 2016, and now has two PBE tacks, one for string tasks and one for bit-vector tasks. Both CVC4 [45] and EUSolver [6] have won the competition in the past.

**Programming by Example** is a popular technique in program synthesis that leverages either user-provided input-outputs [56, 36, 21, 25, 23, 24, 42, 58] or tests [20]. Most PBE techniques target exact specifications and do not handle **noise** in user input. Some notable exceptions are FlashFill [23] and RuleSynth [48], as well as Bayesian and neural program induction techniques [16, 17, 53]. None of these approaches, however, compute results for all subsets of examples, or deal with timeouts.

**Ranking** and returning multiple results are two common approach to handing ambiguous specifications in program synthesis; the two often—but not always—go hand-in-hand. The FlashX tool family [23, 42] uses a ranking function to select a single, most likely program among all the programs that satisfy all user-provided examples. This line of work has explored both hand-crafted [23] and learned [47] ranking functions. Recent work on guiding synthesis using learned probabilistic models [33] can also be seen as applying a learned ranking, but during synthesis rather than at the end. Our ranking function for Bester is hand-crafted, but is different from existing work in that it incorporates semantic features of programs in addition to syntactic ones, such as the number of examples satisfied, and the distance between the expected and actual outputs. Recent work on synthesizing lenses [35] proposed a novel approach to semantic ranking based on information theory. In the future we would like to explore whether best-effort synthesis can benefit from a more sophisticated ranking function along these lines. Unlike PBE tools, which use ranking to select a single result, code completion tools [28, 43] typically present a ranked list of results to the user, and most commonly rely on learned statistical models and syntactic features.

**Observational equivalence** Many enumerating synthesizers apply equivalence reductions as a form of pruning the program space [28, 36, 22, 21, 49]. Observational equivalence [2, 55], as a more agressive and therefore more optimizing form of equivalence, is used in many bottom-up synthesizers [56, 6, 51, 41].

EUSolver [6] specializes in solving benchmarks that require case-splitting by performing an OE-reduced enumeration searching for two subprograms that together cover the examples and a condition to decide between them. The enumeration performed by EUSolver is similar to that of Bester in that it is an enumeration over all the examples that also considers subsets of the examples, but only the first program covering a specific subset of the examples is used within the (single) result program, whereas Bester ranks all such programs and returns the highest ranking ones even if several of them cover the same subset of the examples.

**Interaction models for program synthesis** are a recent field of research, which has taken two main directions: Modifying specification mechanisms and output formats [40, 13] to make synthesis easier to use and better targeted to specific populations of users. Iterative program synthesis [32, 39, 7] focuses on allowing the user to refine the specification while running the synthesizer after each such refinement, essentially making explicit and improving upon what has been the implicit assumption of all synthesis tools. Bester is currently situated well within the first direction, but we believe it will also aid the users greatly in an iterative setting.

**MaxSAT** [10] and MaxSMT [9] are the formulation of the satisfiability problem in which certain clauses are marked as *hard constraints* and others as *soft constraints*, and the solver attempts to find an assignment that satisfies all hard constraints while maximizing the number of soft constraints satisfied. Viewing the world through this terminology, we see that previous work has viewed user-provided inputs as hard constraints, and even in work where other soft constraints are available to the user [14], examples are still considered hard constraints. In Bester, all examples are soft constraints and a ranking function is being maximized likening our paradigm to weighted MaxSAT. In Section 4 we suggest a case where

there could be additional weights between the specifications.

**Opportunistic programming** [12, 11] is the programming paradigm in which composite programming tasks are solved by hunting for and joining pieces of existing code from other sources. Projects such as ExampleStack [59] are intended to make the process of importing found code easier. The Bester user study in Section 6 demonstrates synthesis as another method that can provide *pieces* of the solution to the programmer.

## 8 Conclusion

We proposed a new program synthesis paradigm we call *best-effort program synthesis*, where the synthesizer returns a ranked list of programs that satisfy some part of the specification, rather than just one program that satisfies all of it or no program at all.

This paradigm is implemented in a *best-effort enumeration*, a new synthesis algorithm that extends a bottom-up enumeration with observational equivalence, and is able to accumulate multiple partially-valid results with minimal overhead. We implemented this algorithm in a tool called Bester, and evaluated it on 79 synthesis benchmarks from the SyGuS competition and the Euphony benchmark suite.

Our empirical evaluation showed that best-effort enumeration is more efficient and returns better results than a naive approach to best-effort program synthesis, and that Bester returned useful results even when the specification is flawed or too hard: *i*) for specifications containing an erroneous example, the top three Bester results contained the correct solution, and *ii*) for most hard benchmarks, the top three results contained non-trivial *fragments* of the correct solution. Our user study showed that users apply partially-valid results and *parts of* those results to their code. Additionally, we observed that programmers use the output of the synthesizer for comprehension and not only as a possible part of their solution.

## References


1. Raihan Al-Ekram, Archana Adma, and Olga Baysal. diffx: an algorithm to detect changes in multi-version xml documents. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 1–11. IBM Press, 2005.
2. Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *International Conference on Computer Aided Verification*, pages 934–950. Springer, 2013.
3. Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. *Dependable Software Systems Engineering*, 40:1–25, 2015.
4. Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2017: Results and analysis.
5. Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: Results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.
6. Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 319–336. Springer, 2017.
7. Shengwei An, Rishabh Singh, Sasa Misailovic, and Roopsha Samanta. Augmented example-based synthesis using relational perturbation properties. *Proceedings of the ACM on Programming Languages*, 4(POPL):56, 2019.
8. Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: Probabilistic model for code. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2933–2942, New York, New York, USA, 20–22 Jun 2016. PMLR. URL: `http://proceedings.mlr.press/v48/bielik16.html`.

**9** Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. $\nu$z-an optimizing smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199. Springer, 2015.

**10** Brian Borchers and Judith Furman. A two-phase exact algorithm for max-sat and weighted max-sat problems. *Journal of Combinatorial Optimization*, 2(4):299–306, 1998.

**11** Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598. ACM, 2009.

**12** Joel Brandt, Philip J Guo, Joel Lewenstein, and Scott R Klemmer. Opportunistic programming: How rapid ideation and prototyping occur in practice. In *Proceedings of the 4th international workshop on End-user software engineering*, pages 1–5. ACM, 2008.

**13** Sarah Chasins. *Democratizing Web Automation: Programming for Social Scientists and Other Domain Experts*. PhD thesis, EECS Department, University of California, Berkeley, Oct 2019. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-139.html`.

**14** Yanju Chen, Ruben Martins, and Yu Feng. Maximal multi-layer specification synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 602–612, 2019.

**15** Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 383–401, 2016. `doi:10.1007/978-3-319-41540-6\_21`.

**16** Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 990–998, 2017.

**17** Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 6059–6068. Curran Associates, Inc., 2018. URL: `http://papers.nips.cc/paper/7845-learning-to-infer-graphics-programs-from-hand-drawn-images.pdf`.

**18** Azadeh Farzan and Victor Nicolet. Modular divide-and-conquer parallelization of nested loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 610–624, New York, NY, USA, 2019. ACM. URL: `http://doi.acm.org/10.1145/3314221.3314612`, `doi:10.1145/3314221.3314612`.

**19** Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 422–436, 2017.

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

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

**22** Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: A type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 802–815, New York, NY, USA, 2016. ACM. URL: `http://doi.acm.org/10.1145/2837614.2837629`, `doi:10.1145/2837614.2837629`.

**23** Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

*Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1926385.1926423, doi:10.1145/1926385.1926423.

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

25 Sumit Gulwani. Programming by examples (and its applications in data wrangling). In Javier Esparza, Orna Grumberg, and Salomon Sickert, editors, *Verification and Synthesis of Correct and Secure Systems*. IOS Press, 2016.

26 Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73, 2011. URL: http://doi.acm.org/10.1145/1993498.1993506, doi:10.1145/1993498.1993506.

27 Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. In *Principles of programming languages*, page to appear, 2020.

28 Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *ACM SIGPLAN Notices*, volume 48, pages 27–38. ACM, 2013.

29 Jeevana Priya Inala and Rishabh Singh. Webrelate: integrating web data with spreadsheets using examples. *PACMPL*, 2(POPL):2:1–2:28, 2018. doi:10.1145/3158090.

30 Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. 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*, pages 145–164. ACM, 2016.

31 Vu Le and Sumit Gulwani. FlashExtract: a framework for data extraction by examples. In Michael F. P. O'Boyle and Keshav Pingali, editors, *Proceedings of the 35th Conference on Programming Language Design and Implementation*, page 55. ACM, 2014. doi:10.1145/2594291.2594333.

32 Vu Le, Daniel Perelman, Oleksandr Polozov, Mohammad Raza, Abhishek Udupa, and Sumit Gulwani. Interactive program synthesis. *CoRR*, abs/1703.03539, 2017. URL: http://arxiv.org/abs/1703.03539, arXiv:1703.03539.

33 Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *ACM SIGPLAN Notices*, volume 53, pages 436–449. ACM, 2018.

34 Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

35 Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing symmetric lenses. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi:10.1145/3341699.

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

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

38 Saswat Padhi, Todd D. Millstein, Aditya V. Nori, and Rahul Sharma. Overfitting in synthesis: Theory and practice. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, pages 315–334, 2019. doi:10.1007/978-3-030-25540-4_17.

39 Hila Peleg, Shachar Itzhaky, and Sharon Shoham. Abstraction-based interaction model for synthesis. In Isil Dillig and Jens Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 382–405, Cham, 2018. Springer International Publishing.

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

**41** Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 297–310. ACM, 2016.

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

**43** Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *ACM SIGPLAN Notices*, volume 49, pages 419–428. ACM, 2014.

**44** Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, pages 74–83, 2019.

**45** Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. Counterexample-guided quantifier instantiation for synthesis in smt. In *International Conference on Computer Aided Verification*, pages 198–216. Springer, 2015.

**46** Kensen Shi, Jacob Steinhardt, and Percy Liang. Frangel: Component-based synthesis with control structures. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290386.

**47** Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 398–414, 2015. doi:10.1007/978-3-319-21690-4_23.

**48** Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed K. Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. Synthesizing entity matching rules by examples. *PVLDB*, 11(2):189–202, 2017. URL: http://www.vldb.org/pvldb/vol11/p189-singh.pdf.

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

**50** Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013. URL: http://dx.doi.org/10.1007/s10009-012-0249-7, doi:10.1007/s10009-012-0249-7.

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

**52** Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review*, 40(5):404–415, 2006.

**53** Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. Learning to infer and execute 3d shape programs. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

**54** Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 54, 2014. URL: http://doi.acm.org/10.1145/2594291.2594340, doi:10.1145/2594291.2594340.

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

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

**57** Ke Wang, Rishabh Singh, and Zhendong Su. Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *ACM SIGPLAN Notices*, volume 53, pages 481–495. ACM, 2018.

**58** Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. Automated migration of hierarchical data to relational tables using programming-by-example. *Proc. VLDB Endow.*, 11(5):580–593, January 2018. doi:10.1145/3187009.3177735.

**59** Tianyi Zhang, Di Yang, Crista Lopes, and Miryung Kirnt. Analyzing and supporting adaptation of online code examples. In *Proceedings of the 41st International Conference on Software Engineering*, pages 316–327. IEEE Press, 2019.