Bottom-up Synthesis of Memory Mutations with Separation Logic

Kasra Ferdowsi
UC San Diego, San Diego, CA, USA

Hila Peleg
Technion, Haifa, Israel

Abstract

Programming-by-Example (PBE) is the paradigm of program synthesis specified via input-output pairs. It is commonly used because examples are easy to provide and collect from the environment. A popular optimization for enumerative synthesis with examples is Observational Equivalence (OE), which groups programs into equivalence classes according to their evaluation on example inputs. Current formulations of OE, however, are severely limited by the assumption that the synthesizer's target language contains only *pure* components with no side-effects, either enforcing this in their target language, or ignoring it, leading to an incorrect enumeration. This limits their ability to use realistic component sets.

We address this limitation by borrowing from Separation Logic, which can compositionally reason about heap mutations. We reformulate PBE using a restricted Separation Logic: Concrete Heap Separation Logic (CHSL), transforming the search for programs into a proof search in CHSL. This lets us perform bottom-up enumerative synthesis without the need for expert-provided annotations or domain-specific inferences, but with three key advantages: we (i) preserve *correctness* in the presence of memory-mutating operations, (ii) *compact* the search space by representing many concrete programs as one under CHSL, and (iii) perform a provably correct OE-reduction.

We present SObEq (*Side-effects in OBservational EQuivalence*), a bottom-up enumerative algorithm that, given a PBE task, searches for its CHSL derivation. The SObEq algorithm is proved correct with no purity assumptions: we show it is guaranteed to lose no solutions. We also evaluate our implementation of SObEq on benchmarks from the literature and online sources, and show that it produces high-quality results quickly.

2012 ACM Subject Classification Software and its engineering → Automatic programming

Keywords and phrases Program synthesis, observational equivalence

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.7

Supplementary Material *Software (Source Code)*: `https://github.com/peleghila/sobeq`

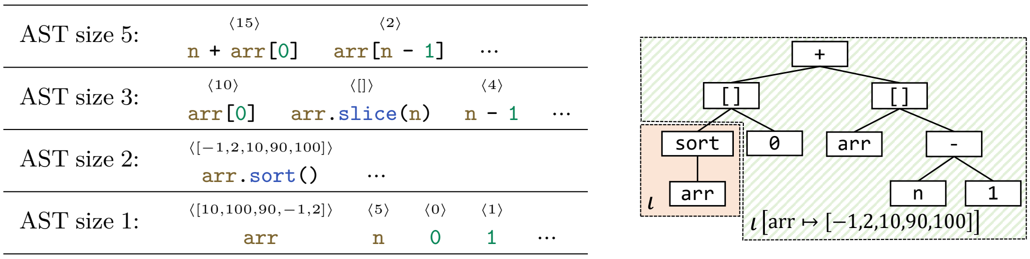
Funding This paper was published with the support of the Israel Science Foundation (ISF) #651/23.

Acknowledgements The authors thank Nadia Polikarpova for her help with formalisms, and Ilya Sergey and Yotam M. Y. Feldman for reading many drafts of this work and asking very good questions.

1 Introduction

Program Synthesis is the task of automatically generating a program that satisfies a given specification. A popular specification modality for synthesizing general-purpose programs is input-output examples, where a solution is defined as a program that, evaluated on each provided input, produces the provided output. This is commonly referred to as *Programming-by-Example* (PBE) [7, 16, 21, 23, 26]. PBE is notable because, unlike logic-based specifications [29, 49, 56], it does not require specialized knowledge to use, and values for it can be collected from a programming environment. It has been used to develop synthesizers for both end-users [23, 60] and programmers [12, 16, 17].

39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva; Article No. 7; pp. 7:1–7:35
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(a) **(b)**

Figure 1 (a) Classical OE bank of programs with their observed evaluation results used to enumerate pure programs. Size 2 contains `arr.sort()` that sorts in-place, making compositions with it incorrect. (b) AST of the target program, divided into the variable states in which each subexpression is evaluated.

Consider a synthesis specification for a JavaScript program where, given an array of integers `arr` of length `n`, we want to return the sum of the smallest and largest elements of the array. We might specify this problem with the example:

$$\{\texttt{arr} \mapsto [10,100,90,-1,2], \texttt{n} \mapsto 5\} \rightarrow 99$$

If we assume the value in `arr` is disposable, e.g., the program may sort it in place, the following is a solution that we would like to synthesize:

```
    arr.sort()[0] + arr[n-1]
```

To solve this synthesis task, we can select a set of *components* (operations, library functions, etc.) and search the space of all programs constructed from those components. There are many ways to search, including constructing a representation of the space that can be traversed [15, 23, 39] and encoding the problem for a solver [30, 56]. Such approaches place heavy constraints on which components can be included. But the simplest form of search is to enumerate the space of programs [13, 19, 25, 52] by applying the components, then test resulting programs. Because the space of programs is astronomical, this approach is prohibitively slow when implemented naively. Moreover, most mechanisms to prune the space are domain-specific and restricted by the manual effort for handling each component individually.

Bottom-up Enumeration. One technique that does not require any domain-specific effort is *bottom-up enumeration* with an *Observational Equivalence* (OE) reduction [2, 59]. OE uses components to combine smaller programs into increasingly larger ones. To prune its search space, it maintains a bank of programs that are representatives of the *equivalence classes* of observationally equivalent programs, i.e., programs that, on the given inputs, evaluate to the same outputs.

New programs are obtained by applying components from the component set to smaller sub-expressions already in the bank. For example, in Fig. 1(a) applying + to the representative of equivalence classes ⟨5⟩ and ⟨10⟩ will yield the program `n + arr[0]`, which will be banked as the representative of equivalence class ⟨15⟩. The bank lets the enumeration memoize the evaluation results of each program, and use them when evaluating larger programs, rather than re-evaluating each full program from scratch. The space is pruned by keeping only one program per equivalence class. Despite discarding many programs, this ensures no solution that existed in the un-reduced space is lost. Previous works use variations of this technique in synthesizers for C [59], Python [16, 17], JavaScript [45], Java [22], Selenium [36], and OCaml [2, 40].

Enumeration with heap mutations. The standard assumption of OE is that all components are *pure* and cause no side-effects [2, 17, 36, 40, 59, 60], and any mutations of the state in the target program are performed outside the synthesis task. Other synthesizers make assumptions on the intermediate states [16, 47], reducing the problem to synthesis of pure programs.

This is because OE has a fundamental limitation when handling mutating programs. Notice that JavaScript's `sort()` works in-place. In the example above, the call to `arr[n-1]` evaluates to a different value if evaluated before `arr.sort()` versus after it. A classical OE enumerator that performs its OE-reduction based on variable values in the inputs would consider `arr[n-1]` to be in the equivalence class $\langle 2 \rangle$. However, unlike its equivalent in languages like Python, `arr.sort()` also returns a self-reference, allowing the result to be composed into larger expressions, as in our target program. If the enumeration itself does not account for this, it will use the wrong memoized value computed using the unsorted value of `arr` in its candidates, yielding a result that does not actually satisfy the specifications. Moreover, the enumeration may see another program whose value is *really* $\langle 2 \rangle$, discard `arr[n-1]`, and never find this solution.

In other words, the classical bottom-up enumeration with OE described above is incorrect with memory mutations. This was also a problem in previous work: some synthesizers ignore the issue, leading to an incorrect enumeration [22, 45]. Peleg et al. [45], for example, allow JavaScript's in-place `sort()`, breaking their own purity assumption.

The key question addressed by this work, then, is how to correctly enumerate mutating programs without losing solutions while maintaining enough pruning to make the process tractable.

The Problem with Mutations. Classical OE considers only one memory state, the state both before and after evaluation of any expression. A composition with memory mutations, however, must consider a potentially different *after-state*. Fig. 1(b) shows this: sub-programs after a mutation are evaluated in a modified *before-state*, the after-state of the mutation. Since composition by the synthesizer simulates evaluation, composition with a mutating expression requires that the synthesizer to be aware of modified before-states. The synthesizer will need to generate complex expressions, e.g., `arr[n-1]` in Fig. 1(b), in a modified state.

Naively, we could store alongside each program not only its evaluation results, as in classical OE, but also its before- and after-state. While this would correct the equivalence relation and ensure programs are not erroneously discarded, the enumeration would still need to enumerate programs over and over: the program `n-1` that yields the result 4 would be enumerated both when `arr` is in its initial state and its modified state, and, as we do not know what the sequence of mutations in the target program will be (if any), in many other states as well. This is not only costly, as it creates many copies of the program in different equivalence classes, it is unnecessarily wasteful, since `n-1` is independent of the value of `arr`. We need an alternative equivalence reduction that both composes *valid* evaluation sequences, and constructs a *compact* program space.

Our approach. We introduce SObEq (*Side-effects in OBservational EQuivalence*), to our knowledge the first bottom-up proof-directed synthesis technique. SObEq is a bottom-up enumerative algorithm for expressions proven correct in the presence of mutating components.

A key insight of SObEq is that mutating operations only mutate a small portion of the state. If the variable state, usually considered as a variable valuation or a stack, is considered as a heap, then *Separation Logic*'s [44, 51] concept of *local reasoning* can be used to reason about programs in the synthesizer's space. The *Frame Rule* gives us the tools for a *compositional* application of operations, in much the same way that bottom-up synthesis already does for pure programs. Thanks to the Frame Rule, a compact representation of programs suffices, rather than considering the full variable valuation every time.

We define Concrete Heap Separation Logic (CHSL), a representation of the operational semantics of the language in the style of Separation Logic. For example, in CHSL we describe the mutating program `n++` evaluated on the example's input as the triple $\{\texttt{n} \mapsto 5\}\texttt{n++}\{\texttt{n} \mapsto 6;5\}$ where the precondition $\texttt{n} \mapsto 5$ describes `n`'s initial value, and the postcondition $\texttt{n} \mapsto 6;5$ describes its value after mutation and its result. Instead of searching for a program, SObEq searches for a CHSL derivation from inputs to outputs. Separation Logic has been used in program synthesis before [29, 49], including where information from the environment can be leveraged as a SL specification [20]. But previous work performed the proof search *deductively*, applying manually-crafted inference rules. The concrete heaps of CHSL allow SObEq to instead search the proof space bottom-up, composing larger programs from smaller ones, and relying only on program evaluation, like a classical OE search does. CHSL gives SObEq two key advantages over classical bottom-up synthesis.

Correct composition via logic. First, SObEq's bank stores CHSL triples rather than programs. To build larger programs, SObEq also applies each component to smaller triples in the bank, but unlike simple programs, triples can only be combined according to the rules of the logic. This means programs always compose in a way that forms a valid evaluation, and there is no risk of evaluating a program in the wrong state. Moreover, OE can be determined by simply ignoring the program and comparing the specifications surrounding each triple: two programs with the same pre- and postcondition are guaranteed to behave the same, and will also compose the same. This maintains SObEq's *correctness* in the presence of mutations.

Local reasoning. Second, thanks to the Frame Rule, CHSL can reason about the local effects of programs: triples do not need to store parts of the state that the program does not touch. For instance, $\{\texttt{n} \mapsto 5\}\texttt{n++}\{\texttt{n} \mapsto 6;5\}$ does not specify anything about `arr`, because `n++` will behave the same way in all concrete states where the value of `n` is 5. This representation is *compact*, with each triple describing the behavior in many, potentially infinitely many, concrete states. This makes it much easier for SObEq to find programs that use variables after a mutation occurs, like our example, compared to the naive solution of enumerating using the full before- and after-states. This local reasoning also allows us to prioritize more general programs: ones that provide the same result and the same mutations for more concrete states. Discarding the less general programs lets us compact the space even further.

Our implementation. We implemented SObEq as a synthesizer for JavaScript programs that can create non-pure expressions, including sequences of expressions. We evaluated our synthesizer on 63 benchmarks curated from the literature, from StackOverflow and from competitive programming website LeetCode. Compared to state of the art synthesis of general mutations, SObEq returns more concise and higher quality solutions, with less overfitting and without performing spurious operations, on both tasks that require and do not require mutations. Importantly, SObEq is both deterministic, meaning it will more reliably integrate with a larger tool, and proved correct, meaning that it will never discard all solutions to a task while exploring the space.

Contributions. The main contributions of this paper are as follows:

- A formulation of the PBE problem with memory mutations as the search for a derivation in a restricted Separation Logic, CHSL.
- The SObEq algorithm, a bottom-up enumerative algorithm that searches for a derivation with a proven-correct OE-reduction.
- An implementation of SObEq in a JavaScript synthesizer, and an evaluation on 63 benchmarks curated from the literature, StackOverflow and LeetCode, which shows that SObEq outperforms state of the art on a variety of synthesis tasks with and without mutations, and highlights the class of tasks for which a naive approach would fail.

`arr` ↦ [−1,2,10,90,100] ∗ `n` ↦ 5:	{`arr` ↦ [−1,2,10,90,100] ∗ `n` ↦ 5;100}
	`arr[n-1]`
		…		

`arr` ↦ [10,100,90,−1,2] ∗ `n` ↦ 5:	{`arr` ↦ [10,100,90,−1,2] ∗ `n` ↦ 5;2}
	`arr[n-1]`
		{`arr` ↦ [10,100,90,−1,2] ∗ `n` ↦ 5;−1}
		`arr[n-(1 + 1)]`
			…	

`arr` ↦ [−1,2,10,90,100]:	{`arr` ↦ [−1,2,10,90,100];[−1,2,10,90,100]}
	`arr`
		{`arr` ↦ [−1,2,10,90,100];−1}
		`arr[0]`
			…	

`arr` ↦ [10,100,90,−1,2]:	{`arr` ↦ [10,100,90,−1,2];[10,100,90,−1,2]}
	`arr`
	{`arr` ↦ [−1,2,10,90,100];[−1,2,10,90,100]}
	`arr.sort()`
		{`arr` ↦ [10,100,90,−1,2];10}
		`arr[0]`
		{`arr` ↦ [−1,2,10,90,100];−1}
		`arr.sort()[0]`
			…	

`n` ↦ 5:	{`n` ↦ 5;5}
	`n`
		{`n` ↦ 5;4}
		`n - 1`
			{`n` ↦ 6;5}
			`n++`
				…

emp:	{*emp*;0}
	`0`
		{*emp*;1}
		`1`
			{*emp*;−1}
			`-1`
				…

Figure 2 The precondition bank used to enumerate the program in Fig. 1(b). Each program is additionally labeled with its postcondition comprising an assertion and its result.

2 Overview

In this section, we present SObEq through the example introduced in Sec. 1. First, we present SObEq's notation for mutating programs, which is compact compared to the real space of programs. Next, we show how this helps us enumerate programs correctly. Finally, we show how ordering programs by generality helps us prune the space even further.

2.1 Mutating programs

We begin with a reminder of programs and their construction in Observational Equivalence [2, 59], then show how we modify them to correctly reason about side-effects.

Classical OE-reduction and the program bank. Bottom-up enumeration with an OE-reduction uses a component set $\mathcal{C}$ comprising functions, operators, variables, and literals, to search for a program specified by a vector of input-output examples, $\vec{\mathcal{E}}$, where each example is of the form $\iota \to \omega$. For each program we compute its equivalence class's label. In classical OE, we use its evaluation result, $[\![p]\!](\iota)$, for each ι in $\vec{\mathcal{E}}$. In our example, the expression `arr` has the *observed behavior* [10,100,90,−1,2] on the provided input, so its equivalence class is ⟨[10,100,90,−1,2]⟩.

To enumerate the space of programs, we use a *bank* to store representatives of equivalence classes, as seen in Fig. 1(a). For each function or operator $f \in \mathcal{C}$ with arity k, we collect all k-tuples of programs already in the bank and use them as arguments to apply f. This is usually done with some notion of iterations, e.g., as in Fig. 1(a), by increasing number of AST nodes. When enumerating programs of AST size 5, then, the enumerator considers the component array dereference (`[]`), which has an arity of 2. Among pairs of programs collected from the bank with a total size of 4 will be the program `arr`, representative of the equivalence class ⟨[10,100,90,−1,2]⟩, and the program `n - 1`, representative of ⟨4⟩. Applying array dereference yields `arr[n - 1]` with the observed behavior ⟨2⟩. Crucially, we do not have to evaluate the full program `arr[n - 1]` to get this value: we can use observed values [10,100,90,−1,2] and 4 and only compute the final step.

When the enumeration finds a program from an equivalence class with no representative in the bank—i.e., no program in the bank has the same label—it is added to the bank to be used when constructing larger programs. This is the case with `arr[n - 1]`, which is added to the bank at size 5. Alternatively, if a representative of the equivalence class already exists in the bank, we discard the new program. If a program's observed values are equal to the provided outputs, the program is a solution. Since we only use discovered equivalence classes (i.e., programs) to construct larger programs, the effect of OE's pruning compounds.

The problems with side-effects. We quickly run into issues when $\mathcal{C}$ contains functions or operators that can cause side-effects. The next step in solving our example is to enumerate `arr.sort()[0] + arr[n - 1]`. But, as we are about to see, using the classical approach this program will not be evaluated correctly, and it may even be pruned away.

We already saw how the subprogram $p_1 =$ `arr[n - 1]` was enumerated and labeled $\langle 2 \rangle$, and $p_2 =$ `arr.sort()[0]` will be enumerated similarly: at size 4, it will be composed from `arr.sort()` (size 2) and `0` (size 1), and added to the bank as a representative of $\langle -1 \rangle$.

The first source of trouble is that OE is a dynamic programming algorithm: when constructing the larger p_2 `+` p_1, the enumerator uses the memoized observed values to compute: $\langle -1+2 \rangle = \langle 1 \rangle$. However, since p_2 modifies `arr`, p_2 `+` p_1 *actually* evaluates to 99 on the input ι. Worse, this erroneous value means the synthesizer will consider p_2 `+` p_1, our solution, equivalent to p_1 `+` p_2 so it may be discarded. While a seemingly simple solution is to just evaluate the program in full each time, ensuring labels are correct, this is not only expensive, depending on the operation, but more importantly, insufficient. If the synthesizer sees the constant value 2, e.g., by enumerating `1 + 1`, this would cause the larger—and seemingly equivalent—`arr[n - 1]` to be discarded. OE hinges on the enumeration only discarding a program when it already has a suitable replacement, but `arr[n - 1]` evaluated *after* `arr.sort()` would still have no equivalent program in the bank.

Rethinking the program representation. Our problem was not distinguishing between the *same* program when evaluated using *different* values of `arr`. We also saw in Sec. 1 the need for distinction between programs that return the same value, but their *effect* on variables is different, e.g., `n` and `n++` when evaluated using the same value of `n`.

We replace programs with triples in Concrete Heap Separation Logic (CHSL): the precondition records the variables before its evaluation, and the postcondition comprises an assertion recording the variables after evaluation and the result of the evaluation, like so:

$$\{\texttt{arr} \mapsto [10{,}100{,}90{,}{-1}{,}2]\}\texttt{arr.sort()}\{\texttt{arr} \mapsto [-1{,}2{,}10{,}90{,}100];[-1{,}2{,}10{,}90{,}100]\}$$

From here on, we name the original array value arr_{orig} and its sorted value arr_{sort}, and use them in our examples and figures for brevity. In this representation, we can tell apart these two triples:

$$\{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr}\{\texttt{arr} \mapsto arr_{orig};arr_{orig}\} \qquad \{\texttt{arr} \mapsto arr_{sort}\}\texttt{arr}\{\texttt{arr} \mapsto arr_{sort};arr_{sort}\}$$

where the first is `arr` evaluated on the original array, returning the original array value, and the second is evaluated on a state where the array has been sorted, returning the sorted value. We use the second triple to construct larger programs evaluated after `sort()`.

Notice that since all values are concrete, CHSL does not need domain- or library-specific inference rules to connect the pre- and postcondition of the triple, but can rely entirely on evaluation.

Local reasoning. These pre- and postconditions do not encode fully-concrete states, they leave `n` unconstrained. This makes our representation of the space more compact: the triple $\{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr.sort()}\{\texttt{arr} \mapsto arr_{sort};arr_{sort}\}$ takes advantage of the fact that only constraining **`arr`** is sufficient to reason about the behavior of this program. We only need to constrain the *footprint* of the program.

CHSL includes the notion of the separating conjunction $*$ to indicate separate parts of the heap (or, in our case, variable store) that can be reasoned about locally. Thanks to aliasing restrictions of the synthesizer's target language detailed in Sec. 3.1, any two variables can be separated with $*$. E.g., the assertion $\texttt{arr} \mapsto arr_{orig} * \texttt{n} \mapsto 5$ describes the concrete initial state. Once we replace the programs in our bank with CHSL triples, we can now further borrow from Separation Logic when considering evaluation sequences.

2.2 Enumeration and heaps

A bottom-up enumeration composes larger programs from smaller ones by selecting a sequence of arguments to apply functions or operators f to. In a classical enumeration, our bank contains programs, so if we wish to apply `+` to two programs in it, this is only a matter of creating the new program p_1 `+` p_2. When executing p_1 `+` p_2 on some variable valuation, the runtime first evaluates p_1, then p_2, then their addition. The classical OE enumeration duplicates this by summing the evaluation result banked for p_1 to that banked for p_2. Without mutations, this is correct for all orderings of arguments to `+` as all the programs evaluate from the initial state and to the initial state, so their results can always be memoized in the bank.

Now that our bank comprises CHSL triples, however, the enumeration needs to apply f to triples, not programs. Here, we have to be more careful. Let us consider two programs in our space, `arr[0]` and `arr.sort()[0]` enumerated with classical OE in Fig. 1(a). In our new bank, they are now the triples:

$$tr_1 = \{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr[0]}\{\texttt{arr} \mapsto arr_{orig}; 10\}$$
$$tr_2 = \{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr.sort()[0]}\{\texttt{arr} \mapsto arr_{sort}; -1\}$$

First, we consider what applying `+` to a pair of triples means: applying it to (tr_1, tr_2) as arguments means trying to create `arr[0] + arr.sort()[0]`, or applying `+` to the program components of the triples. However, unlike the mutation-free case, not every pair of triples with those two programs can create this application. We expect a sequence of arguments to a component to describe a real evaluation of those arguments.

This is akin to creating a valid composition or sequence in Separation Logic, the postcondition of each argument must be equal to the precondition of the next one in the sequence; this means all mutations are accounted for. For (tr_1, tr_2), this is not a problem: $\texttt{arr} \mapsto arr_{orig}$ is the after-state of tr_1 and the before-state of tr_2, which means that their sequence mutates the state from the precondition of tr_1 to the postcondition of tr_2, and they can be used in the application of any component, e.g., `+`, with the values memoized in the triple, so we add the two precomputed results $10 + -1$, and get:

$$\{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr[0] + arr.sort()[0]}\{\texttt{arr} \mapsto arr_{sort}; 9\}$$

However, we cannot compose (tr_2, tr_1) in the same way: `arr.sort()` mutated the state, changing the value of `arr` to arr_{sort}, so tr_1, whose memoized result was computed where `arr` is arr_{orig}, cannot follow it. Requiring arguments to a component to form a valid sequence ensures that only valid executions are enumerated by the synthesizer. The enumerator will not try to compose (tr_2, tr_1) under any binary operator, but can—and will—select (tr_1, tr_2) as arguments to `+`. Once a new program is composed, the enumerator invokes the language interpreter to evaluate the composition from the values of its arguments, yielding a triple containing the result and compound effect of the arguments and operation. We denote this invocation the Eval inference rule.

If we partition our bank of programs by the program's precondition, as in Fig. 2, then the enumerator can always create correct sequences of arguments by construction, using a triple's postcondition to fetch all suitable next arguments according to their precondition.

Discovering more preconditions. Let us consider the last subprogram of our target program: `arr[n - 1]`. It is going to be added with `arr.sort()[0]`, which means that for that composition to be allowed, it must match the state after `arr` is sorted. Specifically, we consider the first argument of the array dereference, which will be needed to enumerate it:

$$tr_3 = \{\texttt{arr} \mapsto arr_{sort}\}\texttt{arr}\{\texttt{arr} \mapsto arr_{sort}; arr_{sort}\}$$

Importantly, this triple will never be enumerated if the enumerator only considers triples starting at a precondition describing the initial state, but without it we will never find the target program.

We have to ensure, therefore, that we also enumerate programs that have other preconditions. tr_3 is a necessary building block for such programs.

To this end, whenever a new assertion is discovered by the enumerator, e.g., when a mutating component is evaluated, creating a never-before-seen assertion in the postcondition, as enumerating $\{\texttt{arr} \mapsto arr_{orig}\}\texttt{arr.sort()}\{\texttt{arr} \mapsto arr_{sort}; arr_{sort}\}$ does, we get ready to enumerate programs that *begin* in this new assertion: we create tr_3, which any program starting at $\texttt{arr} \mapsto arr_{sort}$ will need to incorporate, and add it to the bank, which also adds $\texttt{arr} \mapsto arr_{sort}$ to the bank as a precondition. The next time the enumerator selects arguments to an operator, tr_3 will be available. This lets the enumerator create programs that begin at $\texttt{arr} \mapsto arr_{sort}$.

Programs with different footprints. Now that we have tr_3, the first argument to the array dereference, let us consider the second:

$$\{\texttt{n} \mapsto 5\}\texttt{n - 1}\{\texttt{n} \mapsto 5; 4\}$$

We enumerate this program at the initial value of `n`, i.e., without the need for its precondition to be discovered. However, when the enumerator tries to compose it with tr_3 into a sequence of arguments to `[]`, it runs into a problem: the two triples do not make a valid sequence. The assertion in the postcondition of the first argument, $\texttt{arr} \mapsto arr_{sort}$, is different than the precondition of the following argument, $\texttt{n} \mapsto 5$.

However, we clearly see that these two assertions deal with separate parts of the heap, and do not interfere with each other. This is where we draw on Separation Logic's powerful tool: the Frame rule. Since these are separate parts of the heap, we can use the separating conjunction to create a unified heap in the pre- and postcondition of both triples. The more constrained triples are now composable, so the enumerator can apply array dereference and Eval, yielding:

$$\{\texttt{arr} \mapsto arr_{sort} * \texttt{n} \mapsto 5\}\texttt{arr[n - 1]}\{\texttt{arr} \mapsto arr_{sort} * \texttt{n} \mapsto 5; 100\}$$

The full derivation is shown in Fig. 3. Then similarly, the same extension—now adding $\texttt{n} \mapsto 5$ to tr_2 and nothing to the second argument—is applied to allow the enumerator to compose the target program. This is an important strength of our technique: the same triple for `n - 1` can compose with any value for `arr`—we will only enumerate it once!

We notice that in this case, this will discover a new assertion not as a postcondition as we did when `sort()` mutated `arr` but as a precondition. Once the program is added to the bank, the enumerator will continue to enumerate more programs with this new assertion as their precondition.

2.3 Equivalence and mutation

So far, we showed how the enumerator composes programs, but we did not show how the OE-reduction works. Now that we have a new representation of programs, we can re-define how each triple is observed and labeled, so that the label vector suits our needs.

$$\textsc{Eval}\,\dfrac{\begin{array}{c}\textsc{Frame}\,\dfrac{\{\texttt{arr}\mapsto arr_{sort}\}\texttt{arr}\{\texttt{arr}\mapsto arr_{sort};arr_{sort}\}}{\{\texttt{arr}\mapsto arr_{sort}*\texttt{n}\mapsto 5\}\texttt{arr}\{\texttt{arr}\mapsto arr_{sort}*\texttt{n}\mapsto 5;arr_{sort}\}}\\ \downarrow\end{array}\quad \textsc{Frame}\,\dfrac{\textsc{Eval}\,\dfrac{\{\texttt{n}\mapsto 5\}\texttt{n}\{\texttt{n}\mapsto 5;5\}\quad \textsc{Frame}\,\dfrac{\{emp\}\texttt{1}\{emp;1\}}{\{\texttt{n}\mapsto 5\}\texttt{1}\{\texttt{n}\mapsto 5;1\}}}{\{\texttt{n}\mapsto 5\}\texttt{n - 1}\{\texttt{n}\mapsto 5;4\}}}{\{\texttt{arr}\mapsto arr_{sort}*\texttt{n}\mapsto 5\}\texttt{n - 1}\{\texttt{arr}\mapsto arr_{sort}*\texttt{n}\mapsto 5;4\}}}{\{\texttt{arr}\mapsto arr_{sort}*\texttt{n}\mapsto 5\}\texttt{arr[n - 1]}\{\texttt{arr}\mapsto arr_{sort}*\texttt{n}\mapsto 5;100\}}$$

Figure 3 Using Frame and Eval to enumerate `arr[n - 1]`. Recall arr_{sort} denotes the constant value $[-1,2,10,90,100]$ for brevity.

In classical OE, the observed value for each example is the program's evaluation result, but this is now insufficient: while in the precondition `arr` $\mapsto [10,100,90,-1,2]$ the program `arr[1]` evaluates to 100, if we discard all the following programs that evaluate to 100 we will also discard the `arr[n - 1]` that is evaluated on a sorted array. Likewise, `n` and `n++` with the same precondition are distinct in that they return the same value given the same precondition, but have a different effect on the state.

Generally, we can see that we can label a triple with its pre- and postcondition, including the result. Once those are identical, any programs that form a valid triple with them would be *interchangeable*, i.e., can be swapped for each other with no change to a larger evaluation. In Fig. 2, the precondition component is expressed by the precondition the triple is stored under in the bank, while the postcondition appears above each program.

Generality of equivalence classes. The equivalence class labels above define an *equivalence relation* for OE. However, in this setting, we can do even better: there are cases where we want to discard programs even though one is not equivalent to the other. Let us consider two triples:

$$tr_4 = \{\texttt{n}\mapsto 5\}\texttt{n - n}\{\texttt{n}\mapsto 5;0\} \qquad tr_5 = \{emp\}\texttt{0}\{emp;0\}$$

where emp is the unconstrained heap.

Both triples have the same evaluation result (0), and the same effect (no effect), but different assertions at the pre- and postconditions. But they are not completely different: if we select a *frame axiom* $R = \texttt{n}\mapsto 5$, we can bridge their difference: $\{emp*R\}\texttt{0}\{emp*R;0\}$ is equivalent to tr_4. Under this condition, we say that tr_5 is more general than tr_4: it describes a program that behaves the same over more concrete states. If we let the more general equivalence class *subsume* less general ones, we further compact our representation of the space while remainting correct. While we cannot tractably apply this additional reduction to the entire space, we can at least discard less general programs once more general ones exist in the bank.

Temporary values and variables. This tactic cannot stand on its own: naively employing it means $tr_6 = \{emp\}\texttt{5}\{emp;5\}$ subsumes $tr_7 = \{\texttt{n}\mapsto 5\}\texttt{n}\{\texttt{n}\mapsto 5;5\}$ in the same way. However, in Fig. 2 similar subsumptions do not happen: $\{emp\}\texttt{-1}\{emp;-1\}$ does not subsume $\{\texttt{arr}\mapsto arr_{sort}\}\texttt{arr[0]}\{\texttt{arr}\mapsto arr_{sort};-1\}$ and $\{\texttt{arr}\mapsto arr_{orig}*\texttt{n}\mapsto 5\}\texttt{arr[n - (1+1)]}\{\texttt{arr}\mapsto arr_{orig}*\texttt{n}\mapsto 5;-1\}$, which are still in the bank. Why?

We notice that if these subsumptions are allowed, the programs will no longer be interchangeable, and some will break entirely. Consider the application of `++`: while $\{\texttt{n}\mapsto 5\}\texttt{n++}\{\texttt{n}\mapsto 6;5\}$ uses tr_7 as its argument, `5++` using tr_6 does not compile. Likewise, replacing `arr` within `arr.sort()` with the array literal with `arr`'s value would produce a different mutation of the heap, indicting they should be separate programs. The difference, which must be encoded into our representation, is what is *reachable* from state variables, and if so, from *where*.

Our evaluation result therefore comprises two components: the *value* returned by evaluation, and the *location* of that value. tr_3, then, returns $arr_{sort}@arr$, and tr_7 returns $5@n$. This ensures the interchangeability within `++` or `sort`: the new program will not only compile, it will affect the same variable.

What, then, should be the location of a temporary variable, i.e., a variable that is not reachable from the stack? We must consider the location's purpose: to separate programs enough to preserve interchangeability when the heap is mutated. Two temporary values, on the other hand, are (inherently) interchangeable, so we do not want to over-separate them which would inflate the number of equivalence classes in the bank. We therefore give all temporary values the *same* location: $\bot$. This way, tr_4 and tr_5 both return $0@\bot$, preserving the subsumption, and tr_6 returns $5@\bot$ which separates it from tr_7.

Fig. 2 omits this for simplicity, but does still separate $\{emp\}$`-1`$\{emp; -1\}$ with result $-1@\bot$ from $\{$`arr` $\mapsto arr_{sort}\}$`arr[0]`$\{$`arr` $\mapsto arr_{sort}; -1\}$ with result $-1@arr[0]$ and from $\{$`arr`$\mapsto arr_{orig} *$`n`$\mapsto 5\}$`arr[n-(1+1)]`$\{$`arr`$\mapsto arr_{orig} *$`n`$\mapsto 5; -1\}$ with result $-1@arr[3]$.

Defining the Solution. We are almost finished: like classical OE, we continue iteratively enumerating programs into the bank until a solution is found. Our final step is to reconsider the definition of a solution to a synthesis specification.

When enumerating a program in a non-mutating space with an OE-reduction using the examples' inputs, a solution is a program that is labeled by (i.e., that evaluates to) the provided outputs. In SOBEQ this is insufficient: the enumerator can find a triple returning the correct value in *some* precondition. To be correct when evaluated on the concrete example inputs, we require that a solution's precondition describe the concrete initial state ι. In our example, `arr`$\mapsto arr_{orig} *$`n`$\mapsto 5$ trivially describes ι—it is equal to it—but so does `n`$\mapsto 5$.

Thus, to be a solution, a triple must: 1) return the values of the example outputs, and 2) have a precondition that describes ι.

SOBEQ is, to our knowledge, the first proof-directed synthesis algorithm that works bottom-up. It is also the first bottom-up enumerative synthesizer for general mutating components with a correctness guarantee: if a solution exists in the space spanned by $\mathcal{C}$, SOBEQ will find an equivalent solution.

3 Enumerative Synthesis and Heap Mutations

In this section we define the basic elements with which we construct SOBEQ: programs and their evaluation, and the synthesis task.

3.1 Concrete Heap Separation Logic

In this section, we introduce Concrete Heap Separation Logic (CHSL), a notation for operational semantics in the style of Separation Logic. Its syntax and two inference rules are shown in Fig. 4.

The heap. In CHSL, we use a heap notation to reason about the concrete values assigned to variables from the input. We can reason locally about each individual variable because SOBEQ's target language does not allow assignments, and so as long as nothing is aliased at the input—we can enforce this when generating the synthesis task—each variable is separate from all others and can be separated from them with a separating conjunction, $*$.

Input variables	Identifiers $x \in \mathcal{V}$
Literal values	$v \in \mathcal{K}$
Assertion	$a \in \mathcal{A} ::= emp \mid x \mapsto v * \mathcal{A}$
Assertions	$\vec{a},\vec{P},\vec{Q} \in \mathcal{A} \times \cdots \times \mathcal{A}$
Location	$l \in \mathcal{M} ::= \bot \mid x \mid x[n]$, n is an integer literal
Result	$r \in \mathcal{R} ::= v@l$
Results	$\vec{r} \in \mathcal{R} \times \cdots \times \mathcal{R}$
Postcondition	$\{\vec{Q};\vec{r}\}$

$$\frac{\{\vec{P_1}\}p_1\{\vec{P_2};\vec{r_1}\} \quad \{\vec{P_2}\}p_2\{\vec{P_3};\vec{r_2}\} \cdots \{\vec{P_k}\}p_k\{\vec{P}_{k+1};\vec{r_k}\} \\ c \in \mathcal{C}, arity(c) = k \qquad \forall i.(c(r_1^i,\ldots,r_k^i),P_{k+1}^i) \rightarrow (r^i,Q^i)}{\{\vec{P_1}\}c(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\}}\ \textsc{Eval}$$

$$\frac{\{\vec{P}\}p\{\vec{Q};\vec{r}\}}{\{\vec{P}*\vec{R}\}p\{\vec{Q}*\vec{R};\vec{r}\}}\ \textsc{Frame}$$

Figure 4 CHSL syntax and inference rules

CHSL assertions. CHSL reasons about concrete values assigned to variables. As such, a CHSL assertion comprises heaplets of the form $x \mapsto v$ where v is a concrete value.

This also means we can trivially transform a state (partial function) σ into an assertion describing an identical heap s.t. $x_1 \mapsto \sigma(x_1) * \cdots * x_n \mapsto \sigma(x_n)$ for all $x_i \in \mathrm{dom}(\sigma)$. We use the function interchangeably with its assertion equivalent, i.e., σ as a shorthand for the assertion above.

For a vector of states $\vec{\sigma}$, corresponding to the vector of examples $\vec{\mathcal{E}}$, CHSL likewise uses a vector of assertions. We therefore also define a vector separating conjunction s.t. $\vec{P}*\vec{R} = \langle P_i * R_i \rangle_i$.

Values with location. In the presence of mutations, we must also consider that some values are reachable from our heap. As shown in Sec. 2, this is important for components that modify their arguments (e.g. `arr.sort()`), especially components that return a self-reference and are therefore composable, as in `arr.sort().reverse()`, where both `sort` and `reverse` mutate `arr`.

To support this, a result $r \in \mathcal{R}$ comprises two components: a *value* and that value's *location*, $l \in \mathcal{M}$ that can be a variable, a variable at a *concrete* index, or $\bot$. We use $\bot$ to indicate a value not reachable from any variable, i.e., a temporary value. We decompose the result as $r = v@l$. As with variable values, the values in results are concrete.

Triples. A triple in CHSL comprises a precondition $\vec{P} \in \mathcal{A}^k$ (where $|\vec{\mathcal{E}}| = k$), the program p, and a postcondition. Our postconditions have two components: i) an assertion $\vec{Q} \in \mathcal{A}^k$, and ii) the evaluation results of p on $\vec{P}$. Vectors of assertions and results form a conjunction: $\{\vec{P}\}p\{\vec{Q};\vec{r}\}$ means all triples $\{P^1\}p\{Q^1;r^1\},\{P^2\}p\{Q^2;r^2\},\ldots,\{P^k\}p\{Q^k;r^k\}$ hold.

We call the subset of each assertion Q whose values are different from P the *effect* of the program on P. For example, in the triple $\{\texttt{x} \mapsto 1 * \texttt{y} \mapsto 2\}$`x++ + y`$\{\texttt{x} \mapsto 2 * \texttt{y} \mapsto 2;3\}$ the effect is $\texttt{x} \mapsto 2$.

We consider $\{\vec{P}\}p\{\vec{Q};\vec{r}\}$ to be a *valid* triple if it accurately tracks the evaluation of p on $\vec{P}$. In other words, assuming a (deterministic) small-step operational semantics for programs the behavior of which is defined by the interpreter of the language: $\rightarrow : (\mathbb{P},\mathcal{A}) \rightarrow (\mathcal{R},\mathcal{A})$, we say $\{\vec{P}\}p\{\vec{Q};\vec{r}\}$ is valid if $\forall 1 \leq i \leq k.(p,P^i) \rightarrow (r^i,Q^i)$.

Because SObEq will be *enumerating* triples, it is crucial that it can construct valid triples.

The Eval rule. When constructing a new triple, the Eval rule denotes the use of the interpreter to ensure the resulting triple is valid. Because an evaluation of an AST node will first evaluate its children in sequence, then use the results to evaluate the node's operation, Eval requires a sequence of triples. As in the classical Hoare Logic Sequence rule, Eval requires matching midconditions. However, since we often want to form a sequence from triples where $Q_i \neq P_{i+1}$, we use the Frame rule to find a larger composed heap where the midconditions do match.

The Frame rule. Generally, all triples in the space contain only their footprint in their pre- and postcondition. As the example in Fig. 3 shows, in this compact representation midconditions may not match. We recall that the frame rule allows us to reason about a larger heap by equally extending the pre- and postcondition with an additional assertion called the *frame axiom.*

Fortunately, in CHSL, frame axioms are very easy to find. To prepare a pair of arguments for Eval, then, one of two things is true: either for every i, $Q_1^i \cup P_2^i$ is still a partial function (i.e., if the same variable appears in both, it maps to the same value) and R_1^i,R_2^i are found by their difference, or the two triples cannot form a valid sequence.

Combining Eval and Frame. When the SObEq enumeration will apply a function or operator f to k child triples, this will combine the two rules. Since not all valid sequences of possible arguments to f have matching midconditions, we apply the Frame rule to the entire sequence: given a set of $k = arity(f)$ triples $tr_i = \{\vec{P_i}\}p_i\{\vec{Q}_i;\vec{r_i}\}$, Eval is applicable if there exist k frame axioms $\vec{R}_1,...,\vec{R}_k$ where $\vec{Q}_1 * \vec{R}_1 = \vec{P}_2 * \vec{R}_2, \vec{Q}_2 * \vec{R}_2 = \vec{P}_3 * \vec{R}_3, ..., \vec{Q}_{k-1} * \vec{R}_{k-1} = \vec{P}_k * \vec{R}_k$. Some of these $\vec{R}_i$ may be $e\vec{m}p$. Now that the midconditions of the triples match, Eval can be applied, yielding a triple that evaluates from the new shared precondition $\vec{P_1} * \vec{R_1}$, and computing the result and precondition for $f(p_1,...,p_k)$.

3.2 Problem definition

Since SObEq uses CHSL triples to solve a PBE problem, we must re-state the PBE problem for CHSL. In this section we define what it means for a CHSL triple to represent our target program, define a translation from a PBE task, and explain how additional specifications, constraining the program's effects on the state, can also be supported.

The Programming by Example (PBE) task is usually defined as follows: given a vector of input-output examples $\vec{\mathcal{E}} = \langle \iota_i \to \omega_i \rangle_i$, find a program p over a component set $\mathcal{C}$ where for every input ι_i, evaluating p on ι_i, denoted $[\![p]\!](\iota_i)$, is equal to the provided output ω_i. We first want to translate this into a goal of the shape $\{P\}\_\{Q;r\}$.

Examples in CHSL. We denote the vector of input states $\vec{I} = \langle \iota_i \mid \iota_i \to \omega_i \rangle_i$. The direct translation of $\vec{\mathcal{E}}$ to CHSL, then, is $\{\vec{P}\}\_\{\_;\langle \omega_i @\_ \rangle_i\}$, where $\exists \vec{R}. \vec{P} * \vec{R} = \vec{I}$. In other words, the precondition needs to *describe* the initial states (it is a footprint, so it may not describe all input variables), any postcondition will satisfy our specification, and the correct values can be at any location.

Constraining effect. Users may want to constrain specific mutations as well as the value. We let the user pair with each example any constraints on the effect, e.g., the value of `n` must be 9, or `arr` must end in its initial value [10,100,90,−1,2]. To do this, we let the user provide an *effect constraint* $q: \mathcal{V} \to \mathcal{K}$, constraining the target value for any variable, where if $x \notin \mathrm{dom}(q)$, x is unconstrained.

The SObEq task. A PBE user does not need to know about CHSL. Much like the original $\vec{\mathcal{E}}$, they provide a vector $\vec{\Phi}$ where each $\varphi \in \vec{\Phi}$ is a pair of example and effect constraint: $\varphi = (\iota \to \omega, q)$. If they are unconcerned with effects, this is identical to providing $\vec{\mathcal{E}}$.

We then turn $\vec{\Phi}$ into a synthesis goal in CHSL:

▶ **Definition 1.** *(SObEq synthesis goal) Given a specification $\vec{\Phi}$, our goal $\mathcal{G}_{\vec{\Phi}}$ is:*

$$\mathcal{G}_{\vec{\Phi}} = \{\vec{P}\}\_\{\langle q_i * \_ \rangle_i ; \langle \omega_i @\_ \rangle_i\}, \textit{ where } \exists \vec{R}. \vec{P} * \vec{R} = \vec{I}$$

The SObEq task is to find a triple $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$ that matches $\mathcal{G}_{\vec{\Phi}}$.

3.3 The program space

Now that we have SObEq's specifications, we define the space in which it performs the search. We consider the component set $\mathcal{C}$ as a union of three sets: $\mathcal{V} \cup \mathit{lits} \cup \mathcal{F}$, where $x \in \mathcal{V}$ are the variables in $\vec{I}$, $l \in \mathit{lits}$ are literals in the component set, and $f \in \mathcal{F}$ are the functions and operators that can be applied to subprograms ($\mathit{arity}(f) > 0$). The full program space of a synthesizer with a component set $\mathcal{C}$ is then every program using the components in $\mathcal{C}$. We denote this program set $\llparenthesis \mathcal{C} \rrparenthesis$. This set is infinite and highly redundant, e.g., contains both `x + 1` and `1 + x + 0`, and in our mutating domain, both `arr.sort()` and `arr.sort().sort()`.

Classical Observational Equivalence. In a problem domain with no side-effects, observational equivalence unifies programs with the same evaluation results on the example inputs. This is expressed in the equivalence relation $\equiv_{\vec{\mathcal{E}}}$ that deems two programs p_1, p_2 (observationally) equivalent if $\forall \iota \to \omega \in \vec{\mathcal{E}}. [\![p_1]\!](\iota) = [\![p_2]\!](\iota)$. The OE-reduced space $\llparenthesis \mathcal{C} \rrparenthesis^{OE}$ is then defined using $\equiv_{\vec{\mathcal{E}}}$. This idea was first suggested by Udupa et al. [59] and Albarghouthi et al. [2] and is in wide use today.

A classical bottom-up enumeration (CBE), keeps a bank of programs, and the vector of observed results is used to label the program's equivalence class. Only one program with each vector is added to the bank, meaning any observationally equivalent programs are discarded.

A program space with mutations. In the presence of mutations, the space of possible programs no longer comprises only programs evaluated in the initial states $\vec{I}$. As we saw in Sec. 2, composition in this space requires subprograms that are evaluated in a modified state.

The full program space when $\mathcal{F}$ includes mutations is constructed inductively, building larger valid triples from smaller ones via applications of Eval, and the Frame rule when necessary. We denote the possible values for each $x \in \mathcal{V}$ as $\mathcal{K}_x$. We define the full space of programs $\llparenthesis \mathcal{C} \rrparenthesis$:

▶ **Definition 2** (Full program space). *We define the full program space $\llparenthesis \mathcal{C} \rrparenthesis$ as $\bigcup_{i \geq 0} \llparenthesis \mathcal{C} \rrparenthesis_{(i)}$ where $\llparenthesis \mathcal{C} \rrparenthesis_{(i)}$ is defined inductively as follows:*

$$\begin{aligned}
\llparenthesis \mathcal{C} \rrparenthesis_{(0)} = & \{ \{ \vec{emp} \} l \{ \vec{emp}; \overrightarrow{v @ \bot} \} \mid l \in \mathit{lits}, v \in \mathcal{K} \text{ is the value of } l \} \cup \\
& \{ \{ \vec{\sigma} \} x \{ \vec{\sigma}; \langle v_i @ x \rangle_i \} \mid x \in \mathcal{V}, \vec{\sigma} = \langle x \mapsto v_i \rangle_i, \vec{v} = \mathcal{K}_x \times \cdots \times \mathcal{K}_x \} \\
\llparenthesis \mathcal{C} \rrparenthesis_{(n)} = & \llparenthesis \mathcal{C} \rrparenthesis_{(n-1)} \cup \{ \{ \vec{P} \} f(p_1, \ldots, p_k) \{ \vec{Q}; \vec{r} \} \mid f \in \mathcal{F}, \mathit{arity}(f) = k, \{ \vec{\sigma}_i \} p_i \{ \vec{\sigma}'_i; r_i \} \in \llparenthesis \mathcal{C} \rrparenthesis_{(n-1)}, \\
& \exists \vec{R}_1, \ldots, \vec{R}_k . \textsc{Eval} \text{ of } f \text{ on } \{ \vec{\sigma}_i * \vec{R}_i \} p_i \{ \vec{\sigma}'_i * \vec{R}_i; \vec{r}_i \} \text{ is } \{ \vec{P} \} f(p_1, \ldots, p_k) \{ \vec{Q}; \vec{r} \} \ \}
\end{aligned}$$

Finally, given $\llparenthesis \mathcal{C} \rrparenthesis$, we can define the OE-reduced space that SObEq will enumerate.

3.4 SObEq: observational equivalence with side effects

As we showed in Sec. 2, classical OE's $\equiv_{\vec{\mathcal{E}}}$ is no longer useful for the program space in Def. 2. We need our equivalence relation to encompass every aspect of the goal φ (Def. 1).

The generalized formalization of OE [45] defined two properties for equivalence, *interchangeability*: two equivalent programs can be used interchangeably within a larger program, and *consistency*: two equivalent programs will either both satisfy or both not satisfy the specification.

To satisfy consistency we must include everything that participates in Def. 1. The *value* returned by the expression, and the postcondition, which includes any constrained effect. We also need the state on which the program will be evaluated to check whether it describes $\vec{I}$.

As we saw in Sec. 2, to satisfy interchangeability we need the *location* of the value as well: while the literal [2,3,1] and a variable `x` whose valuation is [2,3,1] return the same value, they are not interchangeable under a larger program like `?.sort()`, where they would lead to different postconditions. Moreover, to be able to swap a program in, its postcondition must be the same.

Combined, then, our equivalence relation simply compares the triple excluding the program:

▶ **Definition 3.** *(SObEq equivalence relation) The equivalence relation for* $\vec{\Phi}=\langle(\iota_i\to\omega_i,q_i)\rangle_i$ *is*

$$\{\vec{\sigma}_1\}p_1\{\vec{\sigma}'_1;\vec{r}_1\}\equiv_{\vec{\Phi}}\{\vec{\sigma}_2\}p_2\{\vec{\sigma}'_2;\vec{r}_2\}\iff\vec{\sigma}_1=\vec{\sigma}_2\wedge\vec{\sigma}'_1=\vec{\sigma}'_2\wedge\vec{r}_1=\vec{r}_2$$

Or in other words, equivalence does not look at the program, only at its specifications.

Subsumption of triples. Finally, we consider the generality ordering of triples and (implicitly) of their equivalence classes, which will let us find more general programs.

In Sec. 2, we considered the two triples $tr_4=\{\mathtt{n}\mapsto 5\}\mathtt{n}$ - $\mathtt{n}\{\mathtt{n}\mapsto 5;0@\bot\}$ and $tr_5=\{emp\}\mathtt{0}\{emp;0@\bot\}$. While tr_4 and tr_5 are not observationally equivalent under Def. 3, as $emp\neq\mathtt{n}\mapsto 5$, tr_5 is *more general* than tr_4: it produces the same result ($0@\bot$) and the same effect ($\emptyset$) from more concrete states. If tr_5 can absorb tr_4, we can arrive at a more compact representation of the space. To this end, we formally define this generality: the partial order of programs that differ by only a more general pre- and postcondition.

▶ **Definition 4** (Generality ordering of triples)**.** *Given a specification* $\vec{\Phi}$*, we define*

$$\{\vec{P}_1\}p_1\{\vec{Q}_1;\vec{r}_1\}\sqsubseteq_{\vec{\Phi}}\{\vec{P}_2\}p_2\{\vec{Q}_2;\vec{r}_2\}\iff\exists\vec{R}.\{\vec{P}_1\}p_1\{\vec{Q}_1;\vec{r}_1\}\equiv_{\vec{\Phi}}\{\vec{P}_2*\vec{R}\}p_2\{\vec{Q}_2*\vec{R};\vec{r}_2\}$$

I.e., the more general way to achieve the same result and effect is considered larger.

While $\sqsubseteq_{\vec{\Phi}}$ is clearly not an equivalence relation (the existence of a frame axiom is not symmetrical) so it cannot be used to partition the space into classes, we notice that replacing a triple with one that is larger according to $\sqsubseteq_{\vec{\Phi}}$ does preserve the two properties of an OE-relation, *interchangeability* and *consistency*, in one direction. Originally [45], *interchangeability* states that swapping two equivalent programs within a larger program will yield an equivalent larger program, and *consistency* states that two equivalent programs either both satisfy a given specification φ or neither does. Under $\sqsubseteq$, however, a slightly weaker—but still useful—version holds.

Interchangeability under $\sqsubseteq_{\vec{\Phi}}$ means that given $f\in\mathcal{F}$, $arity(f)=k$, k triples $tr_j=\{\vec{P}_j\}p_j\{\vec{Q}_j;\vec{r}_j\}$, $1\leq j\leq k$, and k assertions $\vec{R}_j$ s.t. Eval can be applied on f and $\{\vec{P}_j*\vec{R}_j\}p_j\{\vec{Q}_j*\vec{R}_j;\vec{r}_j\}$ to infer $\{\vec{P}\}f(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\}$, and a triple $tr'_i=\{\vec{P}'_i\}p'_i\{\vec{Q}'_i;\vec{r'}_i\}$ s.t. $tr_i\sqsubseteq_{\vec{\Phi}}tr'_i$ for some $1\leq i\leq k$, then:

1. There exists $\vec{R'}_i$ s.t. Eval can still be applied on f when replacing $\{\vec{P}_i*\vec{R}_i\}p_i\{\vec{Q}_i*\vec{R}_i;\vec{r}_i\}$ with $\{\vec{P'}_i*\vec{R}'_i\}p'_i\{\vec{Q'}_i*\vec{R}'_i;\vec{r'}_i\}$ to infer $\{\vec{P'}\}f(p_1,\ldots,p'_i,\ldots,p_k)\{\vec{Q'};\vec{r'}\}$, and
2. $\{\vec{P}\}f(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\}\sqsubseteq_{\vec{\Phi}}\{\vec{P'}\}f(p_1,\ldots,p'_i,\ldots,p_k)\{\vec{Q'};\vec{r'}\}$

Consistency under $\sqsubseteq_{\vec{\Phi}}$ means that for goal $\mathcal{G}$, if $\{\vec{P}_1\}p_1\{\vec{Q}_1;\vec{r}_1\}\sqsubseteq_{\vec{\Phi}}\{\vec{P}_2\}p_2\{\vec{Q}_2;\vec{r}_2\}$, then if there exists $\vec{R}_1$ s.t. $\{\vec{P}_1*\vec{R}_1\}p_1\{\vec{Q}_1*\vec{R}_1;\vec{r}_1\}$ matches $\mathcal{G}$, then there exists $\vec{R}_2$ s.t. $\{\vec{P}_2*\vec{R}_2\}p_2\{\vec{Q}_2*\vec{R}_2;\vec{r}_2\}$ matches $\mathcal{G}$. It's important to notice that the $\Leftrightarrow$ of the equivalence version of *consistency* is now $\Rightarrow$: if $\{\vec{P}_1\}p_1\{\vec{Q}_1;\vec{r}_1\}$ is a solution, so is $\{\vec{P}_2\}p_2\{\vec{Q}_2;\vec{r}_2\}$, but since $\{\vec{P}_2\}p_2\{\vec{Q}_2;\vec{r}_2\}$ is more general $\vec{P}_2$ might describe $\vec{I}$ even if $\vec{P}_1$ did not.

▶ **Lemma 5.** Interchangeability *and* consistency *are preserved under* $\sqsubseteq$.

The proof of Lemma 5 is found in Sec. A.1.

With the program space $(\!|\mathcal{C}|\!)$ and the SObEq equivalence relation and generality ordering, we are ready to search for a program, i.e., to construct the OE-reduced space via enumeration.

4 The SObEq Enumeration

In this section, we describe the construction of the OE-reduced space. We first describe the structure of the space, then the enumeration, and finally how the enumeration can find equivalence classes that, by virtue of their generality, subsume other equivalence classes.

Classical Bottom-Up Enumeration. A bottom-up enumeration consists of applying each $f \in \mathcal{F}$ to k-tuples of child programs from the bank of previously seen programs, where $\mathit{arity}(f) = k$. Each newly enumerated $f(p_1,\ldots,p_k)$ is evaluated, and if an equivalence class labeled by its observed values is not yet represented in the bank, it is added to the bank. Enumeration is done according to some ordering, e.g., height, number of AST nodes, or probability score [7]. This is usually facilitated by the structure of the bank to allow for easier retrieval of relevant child-programs.

The precondition bank. In our setting, we want to bank discovered triples according to their precondition: a bank section $\vec{a}$ contains all discovered programs whose precondition is $\vec{a}$.

There are two reasons for this: first, when composing a k-tuple of arguments to f, finding a sequence Eval is enabled for can be done constructively, without considering all possible k-tuples. For a sequence of length $k-1$ with postcondition assertion $\vec{Q}_{k-1}$, we can preemptively rule out assertion vectors $\vec{a}$ with different values assigned to a variable appearing in both $\vec{Q}_{k-1}$ and $\vec{a}$, and will never form a valid sequence. In Algorithm 1 we call this construction CollectChildren.

Second, and even more importantly, this lets us track what assertions have been discovered by the enumeration. If the enumeration produces a triple $\{\vec{a}\}p\{\vec{a}';\vec{r}\}$ where $\vec{a}$ or $\vec{a}'$ have not yet been seen by the enumeration, then a new bank will be initialized for $\vec{a}$ (or $\vec{a}'$, resp.).

Initializing a new bank. A new assertion $\vec{a}$ can be discovered by the enumeration in one of two ways, as a pre- or a postcondition. For example, the derivation in Fig. 3 combines $\{\texttt{n} \mapsto 5\}\texttt{n - 1}\{\texttt{n} \mapsto 5; 4\}$ with $\{\texttt{arr} \mapsto arr_{sort}\}\texttt{arr}\{\texttt{arr} \mapsto arr_{sort}; arr_{sort}\}$ into $\{\texttt{arr} \mapsto arr_{sort} * \texttt{n} \mapsto 5\}\texttt{arr[n-1]}\{\texttt{arr} \mapsto arr_{sort} * \texttt{n} \mapsto 5;100\}$. Both the compound pre- and postcondition might be new for the enumeration, in which case they do not have a section in the bank.

If $\vec{a}$ is discovered as a precondition, we naturally add a section of the bank to bank the newly enumerated program. If $\vec{a}$ was discovered as a postcondition, then it may be a result of an effect, which means some of its atomic heaplets may not be in the bank yet. In this case, we create a section for $\vec{a}$ in the bank, but also add its atomic heaplets to the bank if they are new: we construct $\{a_x\}x\{a_x;r_x\}$ where $a_x = \langle x \mapsto a_i(x)\rangle_i$ and $r_x = \langle a_i(x)\rangle_i$ for each x in $\vec{a}$, and add it to the bank if a_x is not yet in the bank. In Algorithm 1, this is collectively denoted as InitBank($\vec{a}$).

At the beginning of the enumeration, the bank is trivially initialized with the empty assertion $\vec{emp}$ for literals and with $\langle x \mapsto \iota_i(x)\rangle_i$ for each variable $x \in \mathcal{V}$.

Enumerating terms. The main enumeration is not very different from the CBE enumeration: the enumerator selects $f \in \mathcal{F}$ with $\mathit{arity}(f) = k$, and composes as arguments for it all k-tuples of triples $\{\vec{P}_i\}p_i\{\vec{Q}_i;\vec{r}_i\}$ that Eval can be applied to from triples already in the bank.

The application of Eval then constructs a new triple $\{\vec{P}\}f(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\}$ that can then be *observed*: we use its pre- and postcondition assertions and its results as the label for its equivalence class. This label is searched for in the bank for precondition $\vec{P}$. If the bank for $\vec{P}$ has an equivalent program, $\{\vec{P}\}f(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\}$ is discarded, and if no equivalent program was previously seen, $\{\vec{P}\}f(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\}$ is added to the bank for $\vec{P}$.

We also have at our disposal $\sqsubseteq_{\vec{\Phi}}$ (Def. 4), which means we reduce the space even further.

Using more general triples. We want to use the $\sqsubseteq_{\vec{\Phi}}$ relation to let more general triples subsume less general ones in the course of enumeration, even when they are not strictly equivalent. However, this can happen in one of two cases: (i) the less general triple being enumerated after the more general triple is in the bank, and (ii) a more general triple being enumerated later.

Handling (i) is simple enough: when enumerating $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$, even if there is no representative for the equivalence class of $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$, the enumerator checks whether some $\{\vec{\sigma}_2\}p_2\{\vec{\sigma}_2';\vec{r}_2\}$

Algorithm 1 The SObEq enumeration

```
   Input   : Set of specifications $\vec{\Phi}=\langle(\iota_i \to \omega_i, q_i)\rangle_i$, variables $\mathcal{V}$ specified in example inputs
   Output  : Program $p$ composed of components $\mathcal{C}$ s.t. $p$ that satisfies $\vec{\Phi}$
   Globals: Precondition bank $bank$, mapping from assertion to an OE bank
1  initialAsserts := $\{\vec{emp}\} \cup \{\langle x \mapsto \iota_i(x)\rangle_i \mid x \in \mathcal{V}\}$
2  foreach $\vec{a} \in initialAsserts$ do InitBank($\vec{a}$);
3  repeat // until timeout
4      foreach $f \in \mathcal{F},\ arity(f)=k$ do
5          foreach $\{\vec{\sigma}_1 * \vec{R}_1\}p_1\{\vec{\sigma}'_1 * \vec{R}_1;\vec{r}_1\}\cdots\{\vec{\sigma}_k * \vec{R}_k\}p_k\{\vec{\sigma}'_k * \vec{R}_k;\vec{r}_k\} \in$
             CollectChildren($bank$,$k$) do
6              $\{\vec{\sigma}\}p\{\vec{Q};\vec{r}\}$ := Eval $(f,\{\vec{\sigma}_1 * \vec{R}_1\}p_1\{\vec{\sigma}'_1 * \vec{R}_1;\vec{r}_1\}\cdots\{\vec{\sigma}_k * \vec{R}_k\}p_k\{\vec{\sigma}'_k * \vec{R}_k;\vec{r}_k\})$
7              if $\exists \vec{R}.\{\vec{\sigma} * \vec{R}\}p\{\vec{Q} * \vec{R};\vec{r}\}$ matches $\mathcal{G}_{\vec{\Phi}}$ then return $p$;
8              if $\forall \vec{a} \in keys(bank).\exists \vec{R}.(\vec{a} * \vec{R} = \vec{\sigma} \wedge \vec{R} \subseteq \vec{Q}) \Rightarrow (\vec{a},\vec{Q} \setminus \vec{R},\vec{r}) \notin bank(\vec{a})$ then
9                  if $\vec{\sigma} \notin keys(bank)$ then InitBank($\vec{\sigma}$);
10                 $bank(\vec{\sigma})$ += $\{\vec{\sigma}\}p\{\vec{Q};\vec{r}\}$
11             end
12             if $\vec{Q} \notin keys(bank)$ then InitBank($\vec{Q}$);
13         end
14     end
15 until $timeout$;
```

s.t. $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \sqsubseteq_{\vec{\Phi}} \{\vec{\sigma}_2\}p_2\{\vec{\sigma}'_2;\vec{r}_2\}$ already exists in the bank, and if it does, discards $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$. This is efficiently done by changing the assertions to more general ones and searching the bank.

Handling (ii), however, is a larger feat. *In theory*, once we enumerate the more generic triple $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$, we can replace all less-general triples $\{\vec{\sigma}_2\}p_2\{\vec{\sigma}'_2;\vec{r}_2\}$ with $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$, propagating this change to any compositions that use p_2. This would be prohibitively costly on its own before the implications of Lemma 5: this replacement can weaken the pre- and postconditions of compositions, changing their place in the bank. This cascade might even weaken the precondition a triple that did not satisfy $\vec{\Phi}$ enough to describe $\vec{I}$ and become a solution. As such, we elect to only incorporate (i) into SObEq.

4.1 The enumeration algorithm

We can now compose the complete SObEq enumeration, seen in Algorithm 1.

Lines 1–2 of the algorithm initialize the enumeration by calling InitBank, described above, to create the basic programs for $\vec{emp}$ and for each of the variables' initial values.

Line 4 selects the next component for application; components with an arity of 0 are handled exclusively by InitBank. Next, on line 5 CollectChildren constructs all k-tuples of children that Eval is enabled for. Since it is a part of the construction process, CollectChildren returns each k triples already updated with the frame axioms $\vec{R}_1,...,\vec{R}_k$ to a unified composed heap.

Line 6 then applies Eval to construct the new triple. If the new triple is a solution to the task, it will be returned by line 7. If not, line 8 checks whether an equivalent or subsuming program already exists in the bank via its equivalence class label: if some existing key in the bank can be made equivalent to the current program via a frame axiom. For example, the program $\{\mathtt{n} \mapsto 5\}n-n\{\mathtt{n} \mapsto 5;0@\bot\}$ will be enumerated, and will then be tested twice: for $R = emp$, $(\mathtt{n} \mapsto 5,\mathtt{n} \mapsto 5,0@\bot)$ will be searched for in the bank, and also for $R = n \mapsto 5$, $(emp,emp,0@\bot)$ will be searched and found (for the program `0`), so $\{\mathtt{n}\mapsto 5\}n-n\{\mathtt{n}\mapsto 5;0@\bot\}$ is discarded. A precondition $\vec{a}$ in the bank is deemed relevant if there exists a frame axiom $\vec{R}$ s.t. $\vec{R}$ both matches $\vec{a}$ to $\vec{\sigma}$ and all values in $\vec{R}$ are unchanged in $\vec{Q}$ (i.e., $\vec{R}$ truly is a frame).

Every $\vec{a}$ in the bank lookup uses the relevant equivalence class label, with $\vec{a}$ as its precondition and subtracting the frame axiom from its postcondition assertion. If no triple is found, the new triple is added to the bank. If either the pre- or postcondition assertion do not exist in the bank, InitBank is called for them to generate triples for their variables.

SObEq as a base enumerator. Many synthesizers [5, 16, 36, 45] use a base OE enumeration of expressions to synthesize additional language constructs, e.g., conditionals, higher-order functions, and assignments. These techniques are parameterizable in their base enumerator, and so could leverage SObEq to expand their target language.

Correctness of SObEq. We show that Algorithm 1 is *correct*: that if a solution p exists in the unreduced space, SObEq's enumeration will find a solution observationally equivalent to p or more general than p.

▶ **Theorem 6** (Correctness of SObEq). *If a valid solution $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$ to goal $\mathcal{G}_{\vec{\Phi}}$ exists in $\llparenthesis\mathcal{C}\rrparenthesis$, then there exists $\{\vec{\sigma}''\}p'\{\vec{\sigma}''';\vec{r}'\}$ that matches $\mathcal{G}_{\vec{\Phi}}$ in the SObEq-reduced space, and it is reachable by SObEq's enumeration.*

In Sec. A.2 we prove Theorem 6 in two steps: first we define the (unreduced) space reachable via enumeration and show that any solution will still be in that space, then we apply the reduction of $\equiv_{\vec{\Phi}}$ to the space, defining the *SObEq-reduced program space* $\llparenthesis\mathcal{C}\rrparenthesis^S$, and prove that any solution will have an observationally equivalent or more general under $\sqsubseteq_{\vec{\Phi}}$ solution in $\llparenthesis\mathcal{C}\rrparenthesis^S$.

5 Implementation

We implemented a SObEq synthesizer for JavaScript programs in 3,700 lines of Scala code. The set of functions and operators $\mathcal{F}$ contains 54 components for integers, strings, arrays, and sets, including nine mutating operations: postfix increment for integers, the array functions `sort`, `reverse`, `splice`, `push`, `pop`, `shift`, and the set functions `add` and `delete`. JavaScript strings are immutable, so $\mathcal{F}$ includes no mutating components for strings. $\mathcal{F}$ also includes a sequence operator for any two expressions. We allow a task to provide additional string constants for *lits*, common in benchmarks from the literature [4, 7] or interaction models [16, 17].

Preserving immutability. Effect constraints in $\vec{\Phi}$ can specify that a variable has not changed in the postcondition of the solution. But effect constraints cannot distinguish between, e.g., `x` and `--(++x)` because we only reason about the pre- and postcondition, not the path between them. This is generally desirable as it marks `--(++x)` as redundant, but this will only happen when applying `--`. The user may want `x` to *always* be immutable.

This can be enforced by the enumeration. We add to our task specifications the ability to mark a variable as `immut`. Once the user specifies `x` as immutable, there is no reason to construct `++x` at all: a triple $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$ where $\langle\sigma_i(x)\rangle_i \neq \langle\sigma_i'(x)\rangle_i$ is discarded immediately, and $\vec{\sigma}'$ is not initialized in the bank, since any programs with the precondition $\vec{\sigma}'$ assume a mutation of `x`.

Goals without an output. Under CHSL it is possible to express a goal that has no specified output, only a specified effect. This is desirable when, e.g., specifying that a value is inserted into a list, but what the insertion returns is irrelevant. If a benchmark has no specified outputs, i.e., $\vec{\Phi} = \langle(\iota_i, q_i)\rangle_i$, we create the synthesis goal $\mathcal{G}_{\vec{\Phi}} = \{\vec{P}\}\_\{\langle q_i * \_\rangle_i;\_\}$. The lack of an output does not change anything in how programs are enumerated and the goal is matched in the same way.

6 Experimental Evaluation

We evaluate SObEq on our implementation. All our experiments used a server with Intel Xeon Gold 6338 2 GHz[1] and 128GB of RAM, with the JVM maximum heap size set to 110 GB.

Research Questions. We aim to answer the following research questions:

RQ1 How does SObEq compare to the state-of-the-art in syntax-guided synthesis of heap mutations?

RQ2 What is the overhead of SObEq compared to a classical OE enumeration?

RQ3 How effective is SObEq at pruning the space of programs?

RQ4 How does SObEq compare to enumerating with concrete states?

6.1 Benchmarks and baselines

We evaluated SObEq on a set of 63 benchmarks, curated from previous synthesizers [7, 16, 22, 52], programming exercises on LeetCode.com, and questions on StackOverflow.com. We discarded benchmarks that use types other than strings, arrays and sets, i.e., that would be inherently unsupported by our implementation. As SObEq supports a straight-line sequence of expressions, we also discarded benchmarks from Shi et al. [52] and Ferdowsifard et al. [16] if their solution used assignments to intermediate variables, loops, or conditionals.

We split the benchmarks into two sets: ▷ ***May*** **Mutate**: 45 benchmarks that have a non-mutating solution in $\mathcal{C}$. ▷ ***Must*** **Mutate**: 18 benchmarks that require mutation: benchmarks with a non-empty effect constraint or cannot be solved without mutation in $\mathcal{C}$. Of these, five are benchmarks from *May* with a constrained effect, ruling out non-mutating solutions. For RQ2, we also create ▷ ***Pure***: a version of *May* where all variables are marked as `immut`.

On average, *May* benchmarks have 2.8 examples (min 1, max 4) and 1.4 variables (min 1, max 3). *Must* benchmarks have 2.7 examples (min 1, max 5), 2.3 variables (min 1, max 4), and 1.2 variables specified in the effect (min 0, max 2). Five *Must* benchmarks have no specified outputs. More information about our benchmarks is provided in Sec. B.

Selecting a baseline. We considered several synthesizers as baselines for SObEq, ruling out synthesizers built for a single example [22], synthesizers that mutate by assigning synthesized pure expressions [16, 17, 59], and tools hinging on Java's rich type system [15, 61]. FrAngel [52] is the current state of the art in synthesis of general programs with heap mutations: it is specified with input-output examples, it has the same ability as SObEq to declare a variable as immutable, and its stochastic search carries no additional requirements for specification other than the ability to evaluate—just as SObEq does. This makes it the most suitable baseline.

FrAngel as a baseline. FrAngel targets Java collection implementations, which have many more methods than their JavaScript counterparts. This means FrAngel and SObEq can yield solutions that are entirely incomparable. To ensure an apples-to-apples comparison, we created new Java classes that mimic JavaScript types in Java, recreating the same $\mathcal{C}$ used by SObEq's implementation. We then defined our benchmark set over these types.

In addition, in order to force FrAngel to create a straight-line sequence of expression statements as SObEq does, we also removed from its enumeration loops, conditionals, and assign-

[1] This processor has 32 cores/64 threads, but our implementation is single-threaded.

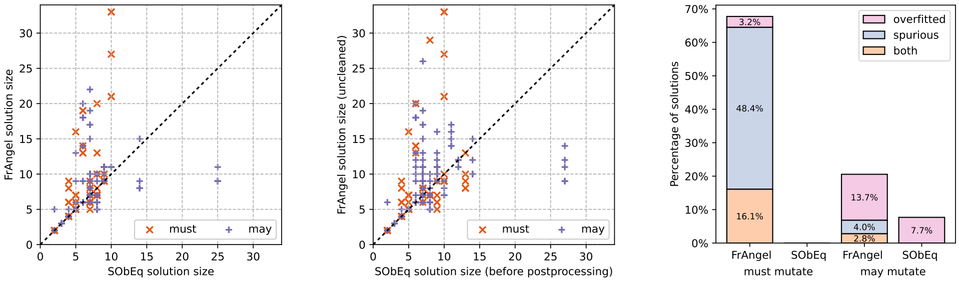


Figure 5 RQ1: Comparing SObEq's solutions to FrAngel's on size (left, center) and quality (right). Solution sizes are per-benchmark: each SObEq solution has up to five corresponding FrAngel values. Above the line means SObEq is smaller. We consider both the final results (left) and solutions before postprocessing/cleaning steps (center). Solution quality (right) indicates the percentage of solutions with overfitted or spurious operations, determined by manual inspection (lower is better).

ments. Notice that removing choices from its random sampling only improves FrAngel's odds of finding a solution; before this additional change, FrAngel's run times were considerably worse.

FrAngel also has the option to set inputs as immutable, which let us encode `immut` variables, and an option to set a target value for an input, which we used for encoding effect constraints in the benchmark. This means FrAngel can run on every benchmark in SObEq's suite.

Selecting timeouts. We consider two timeouts for our runs. First, we set our timeout at 60 minutes, extending the original 30 minute timeout used to evaluate FrAngel. However, since we aim SObEq to be used in interactive systems, and by other OE-based systems as the base enumerator, we also consider results at an *interactive timeout* or a timeout needed for the synthesizer to run in a user-facing tool and not create a *disruptive interruption* [43]. Building on recent interactive synthesizers, we choose 7 seconds as our interactive timeout.

6.2 RQ1: Comparison to FrAngel

We compare SObEq to state of the art FrAngel, using an equivalent component set as detailed in Sec. 6.1. FrAngel's algorithm is stochastic, so we ran FrAngel five times on each of the 63 benchmarks. We consider a benchmark *solved* if the synthesizer returns a solution that is correct on all examples. While some benchmarks from the literature contained a gold standard solution, we only used those as part of the analysis of result quality. We compare the quality of SObEq's solutions to FrAngel on three counts: the size of the solution, is it *overfitted*, and does it contain *spurious operations*. These are shown in Fig. 5. In addition, we compare the run times of the two tools at both an interactive 7s and a one-hour timeout.

Size of the solution. While the size of solutions is not, in itself, a good proxy for quality, it is telling in combination with overfitting and spurious operations. SObEq postprocessed solutions had an average of 6.2 AST nodes at an interactive timeout and 6.9 at a 1-hour timeout (max 14, 25, resp.) where *Must* benchmarks were generally smaller—5.7 nodes at the interactive timeout and 6.1 in an hour—compared to *May* benchmarks—6.4 and 7.1 nodes.

The left and center of Fig. 5 show the AST size for each SObEq solution compared to each of FrAngel's solutions for the same benchmark. Overall, FrAngel's solutions are much larger than SObEq's. There are two benchmarks where SObEq's solution is considerably larger than FrAngel's: *Is All Positive?* (originally from FrAngel), both SObEq's and FrAngel's solutions were overfitted. In *Get first name from name with comma* (originally from Probe [7]), FrAngel returned two overfitted solutions similar to SObEq's, both appearing closer to the diagonal line, and three times found a solution closer to the gold standard solution, which is shorter, though never the gold standard solution itself. (The two identical solutions and three identical solutions overlap in Fig. 5.) An interesting trend in SObEq's solutions was that the enumeration order favored extracting postfix increment to a sequence. E.g., instead of `buffer[idx++]++, idx` SObEq returns the larger but functionally identical `buffer[idx]++, idx++, idx`.

Overfitted solutions. We define overfitted solutions as solutions that do not generalize beyond the provided examples. This was determined by manual inspection of both SObEq and FrAngel solutions, and with reference to task descriptions and gold standard solutions, if they exist.

Two SObEq solutions were overfitted, and one additional benchmark was close to a gold standard solution for the task, but since all its inputs were of length 4, it replaced `str.length` in the gold standard solution with `4`. Altogether, 3 of SObEq's 54 solutions were overfitted (6%). Of 279 FrAngel solutions (up to five per benchmark, which sometimes differ) 47 were overfitted (17%). As the right of Fig. 5 shows, both tools were more prone to overfitting in *May* benchmarks.

We bring several examples of FrAngel's overfitted solutions here.

For example in the *Max and Min* benchmark, a simplified version of the example in Sec. 1, all four solutions (one run timed out) are overfitted and *different*. Two of them are:

```
return arr.join(arr.join(Str(""))).replaceAll(Str(""), Str("␣"))
  .concat(arr.join(Str(""))).indexOf(arr.slice(Int(0), Int(1)).join(Str("")));
```

where the result number (max plus min) is generated by `indexOf`, and

```
arr.reverse();
return arr.join(arr.sort().join(Str(""))).replaceAll(Str(""), Str("␣")).replace(
  arr.join(arr.join(Str(""))).replaceAll(Str(""), Str("␣")), Str("␣")
).length();
```

where it is generated by the length of a string.

We also consider FrAngel's results on the three benchmarks that SObEq overfitted on: For the *Is All Positive?* benchmark mentioned above for its large solutions, both SObEq and FrAngel overfit to the examples in two different ways. FrAngel overfit all give solutions, returning `true` only for arrays where the smallest element is 1, and SObEq looked for a `'-'` in an overfitted location in the printed array. For *get from name with comma*, four of five FrAngel solutions were *not* overfit. The fifth is:

```
return _arg_0.split(Str(",")).sort().slice(Int(1).subtract(_arg_1),_arg_1).join(Str(""));
```

where the input string is split into words but then sorted before extracting the `1-_arg_1`th word. This happens to be true because all example inputs have a decreasing lexicographic ordering of the words. In *Negative Index Of*, where SObEq string lengths to `4`, three of FrAngel's solutions are not overfitted, but two rely on string `<` comparisons to between variables and constants.

Spurious operations. We define spurious operations as terms in the AST that do not contribute to reaching the target values: e.g., sorting an already-sorted list, mutating a temporary value that is then discarded, etc. Of the 279 FrAngel solutions, 37 had spurious operations (13%). We bring a few examples of these here.

One of the solutions FrAngel found for the *buffer-add* benchmark was:

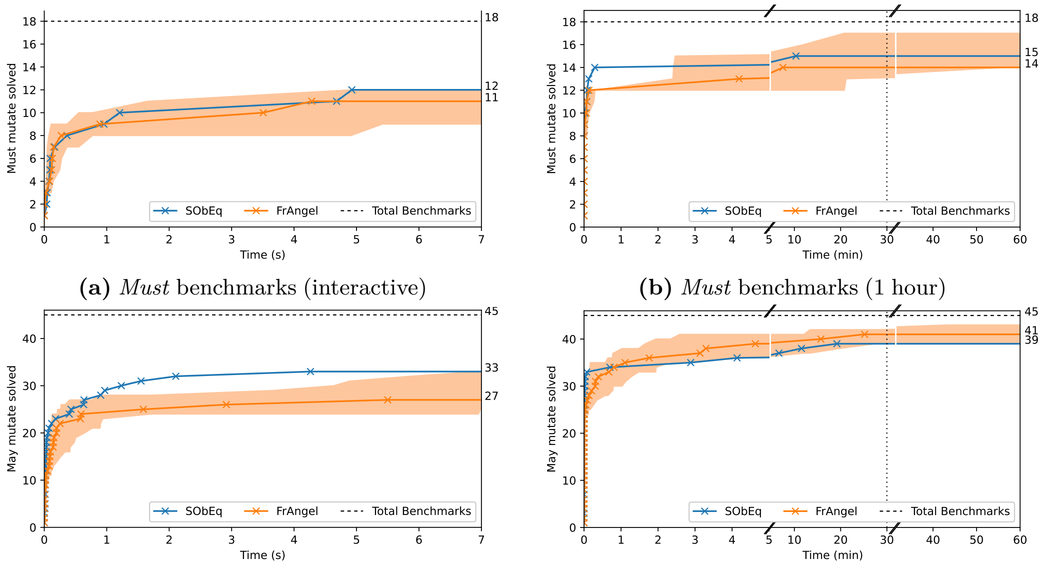
```
arr.splice(beg++, 1, toadd).reverse();
return end + 1;
```

Here, `splice` removes a sub-array from `arr`, replaces it with `toadd`, and returns it. The solution then calls `reverse()` on this temporary value, which is immediately discarded.

Likewise, in three of the five solutions to *rotate queue*:

```
mod.equals(inp.push(inp.shift()));
```

where input variable `mod` is compared to the result of the target program (shifting the first element out of the array and pushing it back in at the end).



(a) *Must* benchmarks (interactive)

(b) *Must* benchmarks (1 hour)

(c) *May* benchmarks (interactive)

(d) *May* benchmarks (1 hour)

Figure 6 RQ1: The number of benchmarks SObEq and FrAngel solve over time. FrAngel is stochastic: median run for each benchmark shown by line with best and worst runs as range.

For *set add-delete*, which asks for an in-place mutation of a the set `seen`, one solution is:

```
seen.delete(s2).compareTo(seen.add(s1).has(null));
```

where both `has` and `compareTo` are useless, computing a result that is not returned.

And, of course, this problem is not disparate from overfitting; the only successful run of FrAngel for *fill fibonacci window* found this solution:

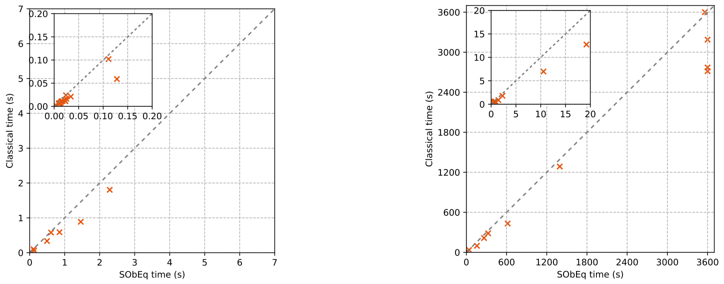
```
out.reverse().spliceReplace(
  Int(1),
  out.concat(out).length().subtract(
    Str("␣").concat(out.join(Str(""))).lastIndexOf((new IntArray(Int(1))).join(Str("")))
) ).shift();
```

where `reverse().spliceReplace(1,_)` is a consistent, if odd, way to remove the first element and add something after the second element by swapping the elements and then replacing the (now) second one. The second argument to `splice`, which is expected to compute the next fibonacci element, overfits to the examples, and `shift`ing on the discarded return value of `splice` is spurious.

SObEq, in contrast, had no spurious operations. Some are avoided thanks to OE: FrAngel can draw programs that perform round trip operations like reversing multiple times (3 of 5 solutions to the example from Sec. 1 do this) or sort a sorted list. Enumerating with OE discards these, as they return the heap to a state it was already at and return the same value.

Other spurious operations are avoided because of the implicit bias of a bottom-up search toward small programs: the program `inp.push(inp.shift())` already satisfies the input, so a bottom-up search would not try to compose it into a larger program adding on spurious operations. Likewise, the enumeration would try to compose `arr.splice(beg++, 1, toadd)` to `end + 1` before trying it with the larger `arr.splice(beg++, 1, toadd).reverse()`.

Runtimes. We compared SObEq's runtimes to FrAngel's best, worst and median performance on each benchmark in Fig. 6. In an interactive timeout, SObEq outperforms FrAngel for the majority of the 7 seconds. At the 7s limit, SObEq solves 45 benchmarks, 7 benchmarks more than FrAngel's median run, whereas FrAngel solves between 34 and 45 benchmarks. We also notice the main disadvantage of FrAngel, which is the variance introduced by the stochastic algorithm: a difference of 30% between its best and worst cases. At the 1-hour



(a) Interactive timeout

(b) 1-hour timeout

Figure 7 RQ2: Comparison with CBE: time to termination for each *Pure* benchmark with SObEq compared to a classical bottom-up enumeration. Points at (timeout,timeout) not shown.

timeout, SObEq solves 54 benchmarks, compared to FrAngel's 53–60 (median 55).

Fig. 6 separates benchmarks into sets to ensure performance is not at the expense of one set of benchmarks over the other, important for a general-purpose synthesizer. At an interactive timeout, SObEq performs on *Must* benchmarks as well as FrAngel's best run, both solving 12 of 18 benchmarks, and solves one more than FrAngel's median run of 11. For *May* benchmarks, SObEq solves 33 of 45, while FrAngel solves between 25 and 33 (median 27). After an hour, SObEq outperforms FrAngel's median run for *Must* benchmarks, solving 15 benchmarks to FrAngel's 14 and is within FrAngel's range for both benchmark sets.

To answer RQ1, **SObEq returns higher quality solutions than FrAngel at comparable runtimes, without the nondeterministic variance in both time and expected solution.**

6.3 RQ2: Overhead of SObEq

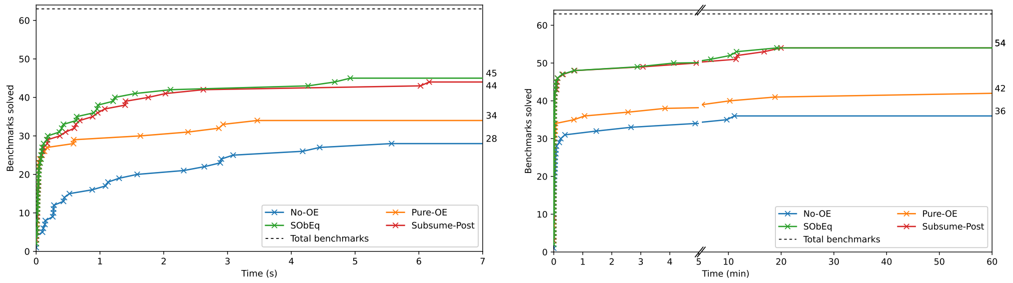
A more general algorithm inevitably carries an overhead on a (now) special case compared to specialized algorithms for that case. This is often because of additional work by an algorithm making fewer assumptions, but in OE this can also take the form of a more refined equivalence relation, which unifies fewer programs creating a bigger space to explore. We wish to measure this overhead in order to quantify the cost of the changes made to support mutations in SObEq.

If $\mathcal{C}$ has no mutating components, SObEq would not discover assertions outside the initial preconditions and their compositions. Such a search space is directly comparable to a classical bottom-up enumeration (CBE). This lets us measure the cost of the more fine-grained equivalence classes caused by using results, not just values, in equivalence class labels, and the additional overhead of creating a compound heap for sub-programs using different variables.

To measure the overhead, we restricted our $\mathcal{C}$ to only non-mutating components, and compared its performance on *Pure* benchmarks to an enumerator that uses only the initial *concrete* state and uses the classical value-based equivalence relation for an OE-reduction. This is equivalent classical OE with a height-based enumeration. By removing the mutating components for this evaluation, we avoid the correctness issues in classical OE, while controlling for performance differences caused by having different grammars.

In this restricted context CBE outperforms SObEq for most benchmarks (time comparison shown in Fig. 7). In an interactive timeout, CBE solved the benchmarks an average of 1.52 times faster (median 1.5). CBE was also able to solve one additional benchmark, *Max and Min*, within the 7s timeout whereas SObEq took 10.5*s*, and it is therefore not shown in Fig. 7. The biggest difference, where CBE was 2x faster or more (up to a maximum of 2.16 times faster), were all in benchmarks with runtimes of under 130ms for both enumerators.

In the full hour, CBE solved the benchmarks an average of 1.48 times faster (median 1.5), and finished with a two-benchmark lead: there are three benchmarks CBE solved that SObEq



(a) Interactive timeout **(b)** 1-hour timeout

Figure 8 RQ3: Run times of disabling different pruning mechanisms within SObEq: any OE, OE outside the initial state, and discarding programs according to $\sqsubseteq$.

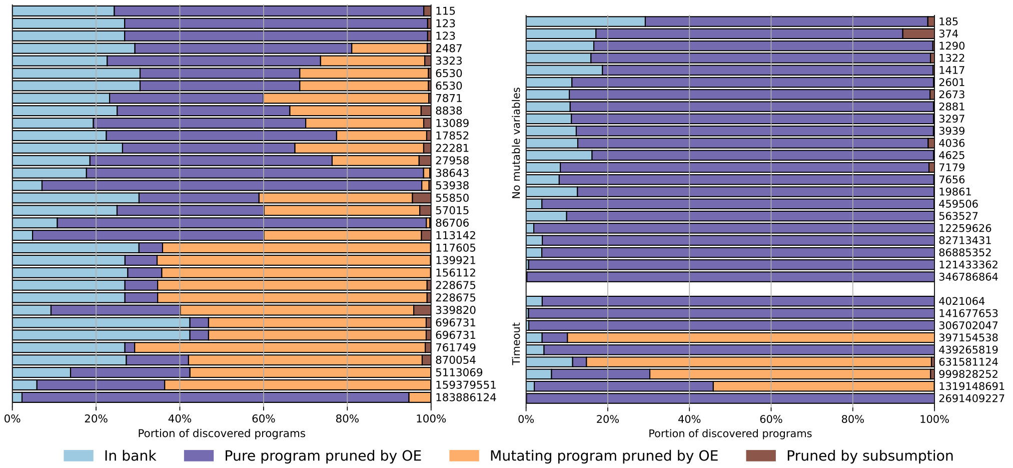


Figure 9 Handling of programs seen in the space of each benchmark (percentages plotted, no. programs in text). Benchmarks where mutation cannot occur and timeouts plotted separately.

did not, and one benchmark that SObEq solved and CBE did not.

For RQ2, **we find that the overhead of SObEq's more fine-grained equivalence classes and creating compound heaps is not negligible compared to CBE.**

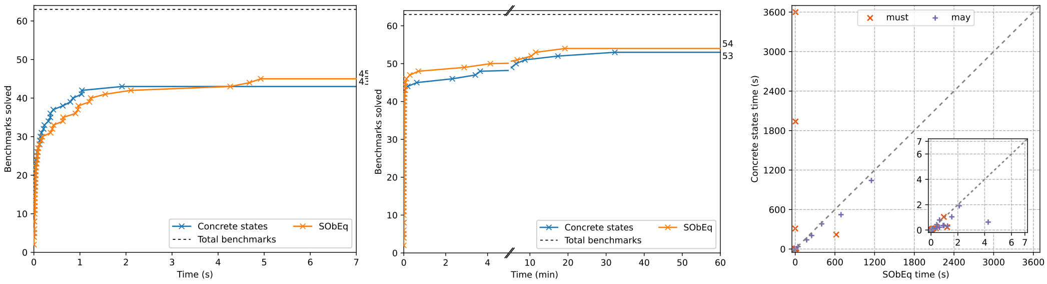
6.4 RQ3: SObEq's effectiveness in pruning

To test SObEq's effectiveness at pruning its space we compare SObEq to two ablation enumerations: **1.** *No-OE*: an enumeration that discards no programs, i.e, enumerates $(\!|\mathcal{C}|\!)^{reach}$. **2.** *Pure-OE*: an enumeration that only performs OE on pure programs, i.e., fully enumerates the space outside preconditions that cover the initial state. Additionally, to measure the effect of subsumption, we also add **3.** *Subsume-Post*: an enumeration with the SObEq OE-reduction, but no subsumption during enumeration, only as postprocessing of results. The results are shown in Fig. 8.

It is, of course, unsurprising that *No-OE* performs poorly: with no reduction of the space, few programs are feasible, and this version solves only 28 benchmarks within 7s, diverging completely after 10 minutes. *Pure-OE* is an interesting case: testing whether it should prune a program required computing its effect, an expensive operation that leads *Pure-OE* to be considerably slower. At the 1-hour limit, *Subsume-Post* does as well as SObEq. However, it lags a little behind SObEq for most of its run, having solved one fewer benchmark at 7s.

In summary, **both the application of OE to the entire space and the application of subsumption help SObEq's performance, particularly in an interactive timeout.**

We also looked at what happens to the discovered programs across runs of SObEq, shown



(a) Interactive timeout **(b)** 1-hour timeout **(c)** Per-benchmark comparison

Figure 10 RQ4: Comparing an enumeration with preconditions to an enumeration with concrete states. Points at (timeout,timeout) not shown in (c).

in Fig. 9, specifically to gauge how many programs are discarded and how many are subsumed by a more general program that already exists in the bank. We separate out runs that time out and benchmarks where all input variables are strings, i.e., no mutation is *possible*. In the latter, the number of mutating programs pruned is trivially 0, but subsumption still takes place.

On (weighted) average, only 3% of programs seen by SObEq are added to the bank (med 14%, min 0.1%, max 42% of the programs). 66% on average are pure programs pruned by equivalence (med 72%, min 2%, max 99.98%), and another 30% are mutating programs pruned by equivalence (med 1.5%, min 0%, max 90%). Finally, an average 0.2% of the programs are pruned by subsumption (med 0.4%, min 0.00002%, max 8%). **While the ratios vary, each of the pruning methods in SObEq is significant for some of the benchmarks.**

6.5 RQ4: Enumerating with preconditions vs. concrete states

Finally, we examine the effect of the *local reasoning* in SObEq's enumeration. Composing partial heaps takes effort, but how costly is it, and does it allow more benchmarks to be solved? To test this, we created a version of SObEq where instead of footprints, preconditions are always *full* concrete states. We compare this to SObEq's enumeration in Fig. 10.

For most benchmarks, both enumerations perform similarly with concrete states slightly faster. We expected to see this as an effect of the overhead of composition, but it is also caused by pure programs being banked in one main state, which changes the enumeration order favorably for *May* benchmarks. If we consider only *Must* benchmarks, SObEq overtakes the concrete state enumeration after 1.5s and stays ahead for the entire 1-hour run.

Three benchmarks are noteworthy in this experiment. The concrete state enumeration times out on our overview example (benchmark *Max and Min with n*) while SObEq solves it in under 5s. Two more, *partition k largest* and *get range loopy while*, SObEq solves in 90ms and 4.6s, resp., and enumerating with concrete states takes 5min and 32min, resp. All three share the need to use multiple variables *after* one is mutated, which means subprograms already enumerated with the unmutated variable do not make a valid sequence and need to be re-enumerated.

To conclude, **SObEq's local reasoning is faster for benchmarks that need to mutate, and is not significantly slower for other benchmarks.**

7 Related work

Syntax- and Component-Guided Synthesis. Syntax-Guided Synthesis (SyGuS) [3] is a form of the synthesis problem providing a specification along with a grammar defining its program space. It can also describe a form of synthesis where programs are constructed by syntax rules. The latter is essentially identical to Component-Based Synthesis, where the synthesizer is provided with a list of functions and operations spanning the program space. Both of these formulations are tackled by many synthesis strategies: enumerative [7, 9, 14, 22, 25, 28, 34, 35, 52, 54, 62], SMT [1, 6, 30, 50] and SAT [56, 58] based, by traversing constructed representations of the space such as VSAs [31, 33, 63], version spaces [53] and Petri-nets [15, 24].

Proof-directed synthesizers. A specific form of enumerative synthesis is one that attempts to find a proof for a proof goal, where the proof dictates a program. This idea is not new [57], and some deductive type-driven synthesizers describe their search as a proof search [24, 42]. Other works' goals are more expressive than types: refinement types [48], relational specifications [32], select/update operators [41], Separation Logic, and even natural language [10]. Importantly, proof searches are overwhelmingly *deductive*, searching the proof space top-down. To our knowledge, SObEq is the first synthesis proof search that enumerates the proof space bottom-up.

Synthesis with Separation Logic. Separation Logic proofs are a popular vehicle for proof-directed synthesis. They have been successfully used for parallelizing non-parallel code [8], but are most often used to synthesize heap-manipulating programs. SuSLik [49] performs a deductive search with inferences about the heap to generate programs performing pointer operations. Cypress [29] extends this with the ability to synthesize recursive auxiliaries, ROBoSuSLik [11] extends it with read-only borrows, and RusSOL [20] applies it to Rust programs, leveraging Rust's type system to simplify the required specification from formal assertions to functional annotations of Rust functions. All of these accept assertions as their specification, hinge on manually-crafted inference rules, and employ solvers. SObEq, in contrast, works on input-output examples and relies only the generic Eval rule in conjunction with an interpreter.

Observational equivalence. Observational Equivalence [2, 59] is a popular way to reduce the size of the space when enumerating bottom-up. Originally OE works on examples, but RESL [45] generalized it to other specifications on programs as well; this generalization also introduced the notions of *interchangeability* and *consistency* that SObEq's correctness relies on. SyMetric [18] relaxes OE to observational similarity by choosing representatives via clustering instead of equivalence, then repairing the resulting program. While this compacts the space like SObEq's subsumption, correctness of SyMetric's solutions hinges on the success of its repair step.

Many of the synthesizers that use OE only synthesize pure programs [2, 7, 40, 45, 46, 60] (we include [45] in this list, despite its misuse of JavaScript's `Array.sort`), others employ synthesis with OE to produce pure expressions that are then used in assignments or other larger mutating expressions [16, 17, 37, 38, 59], creating a top-level program with memory mutations.

Arborist [36] and Lens [47] relax the purity assumption in interesting ways. Arborist tackles the specific problem of dependent loops (e.g., *fold*) where the composition of smaller programs discovers new inputs and changes the equivalence classes. Arborist uses finite tree automata to unify programs without discarding them, which allows it to iteratively break up equivalence classes. Arborist finds a solution if its iterative refinement terminates. Lens take advantage of its target domain (straight line assembly code) to treat operations as having no arguments, making sequences the only composition. In this representation, OE values are full register valuations, and synthesis of state-mutating programs reduces to the OE enumeration used for pure programs. Both hinge on the ability to represent equivalence classes *without* discarding any programs using, implementing OE without a bank, which is only tractable under very strict assumptions on the set of components and on values.

Synthesis with heap mutations. The problem of synthesizing mutating snippets has been worked around in many ways. SL proof-directed approaches assume a limited target language of mutations available to the proof. Sketch[56], Transit [59], SnipPy [17], Simpl [55] and Cozy [37, 38] assume expressions are pure, then use them in specific mutating contexts validated by other means (e.g., SMT). SyPet [15] and EdSynth [61] rely on types for getting programs, and only then evaluate them. Likewise, FrAngel [52] draws random programs, then evaluates them.

LooPy's [16] problem statement bounds the intermediate mutations, and is to enumerate independent paths between these states. Similarly Lens [47] reduces all registers to 3-bit and constructs the full space of intermediate states. Cobalt [41] also separates pure expressions from mutations using provided API annotations. Toshokan [27] encodes the specifications for mutating methods as two versions of the function, one for changes to the state and the other the for result, akin to the two parts of SObEq's postcondition.

CodeHint [22] is, in some ways, the closest to our approach: it defines expressions as equivalent if they have the same value and side effects in the *current state*, or, in our terminology, an equivalence relation on both before- and after-states. How current states are used in enumeration is not defined in the paper, but their implementation uses programs enumerated on the initial state to populate their bank for new after-states, replicating the problem discussed in Sec. 2.1. Moreover, CodeHint's focus is its interactivity, and its interaction is designed around a single example, with any additional information only filtering an initial list of programs, not informing the OE enumeration. This is why we ruled it out as a baseline in Sec. 6.1.

SObEq sheds assumptions that expressions are pure and that intermediate states are known in advance. Moreover, SObEq has the formal guarantee in Theorem 6, unlike the two most general existing solutions: FrAngel, which randomly draws programs, and CodeHint, which performs OE only on its first example and mutated states in an unprincipled manner.

Other side effects like file or database writes have been a challenge in program synthesis. RbSyn [25] addresses database writes via a type-guided top-down enumeration from a syntax including *effect holes* that can be filled with annotated mutating methods; expressions not within effect holes have a purity assumption, limiting how effects are composed, unlike SObEq that can use mutating operations anywhere. CodeHint allows users to enable effects on the file system in their enumeration, even though they cannot be undone like memory mutations are.

References

1 Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 270–288, Cham, 2018. Springer International Publishing.

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. *arXiv preprint arXiv:1711.11438*, 2017.

5 Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the*

European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I 23, pages 319–336. Springer, 2017.

6 Haniel Barbosa, Andrew Reynolds, Daniel Larraz, and Cesare Tinelli. Extending enumerative function synthesis via SMT-driven classification. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 212–220. IEEE, 2019.

7 Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

8 Matko Botinčan, Mike Dodds, and Suresh Jagannathan. Proof-directed parallelization synthesis by separation logic. *ACM Trans. Program. Lang. Syst.*, 35(2), jul 2013. doi:10.1145/2491522.2491525.

9 José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. FlashFill++: Scaling programming by example by cutting to the chase. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi:10.1145/3571226.

10 João Costa Seco, Jonathan Aldrich, Luís Carvalho, Bernardo Toninho, and Carla Ferreira. Derivations with holes for concept-based program synthesis. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 63–79, 2022.

11 Andreea Costea, Amy Zhu, Nadia Polikarpova, and Ilya Sergey. Concise read-only specifications for better synthesis of programs with pointers. In Peter Müller, editor, *Programming Languages and Systems*, pages 141–168, Cham, 2020. Springer International Publishing.

12 Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the 2020 CHI conference on human factors in computing systems*, pages 1–12, 2020.

13 Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices*, 53(4):420–435, 2018.

14 Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices*, 52(6):422–436, 2017.

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

16 Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. LooPy: Interactive program synthesis with control structures. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA), 10 2021. doi:10.1145/3485530.

17 Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. Small-step live programming by example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, page 614–626. Association for Computing Machinery, 2020. doi:10.1145/3379337.3415869.

18 Jack Feser, Işıl Dillig, and Armando Solar-Lezama. Inductive program synthesis guided by observational program similarity. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):912–940, 2023.

19 John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices*, 50(6):229–239, 2015.

20 Jonáš Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, and Ilya Sergey. Leveraging Rust types for program synthesis. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023. doi:10.1145/3591278.

21 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*. ACM, 2016.

22 Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. CodeHint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 653–663, 2014.

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, page 317–330, New York, NY, USA, 2011. Association for Computing Machinery. `doi:10.1145/1926385.1926423`.

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

25 Sankha Narayan Guria, Jeffrey S Foster, and David Van Horn. RbSyn: type-and effect-guided program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 344–358, 2021.

26 Sankha Narayan Guria, Jeffrey S Foster, and David Van Horn. Absynthe: Abstract interpretation-guided synthesis. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1584–1607, 2023.

27 Kangjing Huang and Xiaokang Qiu. Bootstrapping library-based synthesis. In *International Static Analysis Symposium*, pages 272–298. Springer, 2022.

28 Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling enumerative and deductive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1159–1174, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3385412.3386027`.

29 Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. Cyclic program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 944–959, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3453483.3454087`.

30 Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224. IEEE, 2010.

31 Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. Question selection for interactive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1143–1158, 2020.

32 Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 407–426, 2013.

33 Tessa A Lau, Pedro M Domingos, and Daniel S Weld. Version space algebra and its application to programming by demonstration. In *ICML*, pages 527–534, 2000.

34 Woosuk Lee. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, 2021.

35 Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices*, 53(4):436–449, 2018.

36 Xiang Li, Xiangyu Zhou, Rui Dong, Yihong Zhang, and Xinyu Wang. Efficient bottom-up synthesis for programs with local variables. *Proceedings of the ACM on Programming Languages*, 8(POPL), 2024.

37 Calvin Loncaric, Michael D Ernst, and Emina Torlak. Generalized data structure synthesis. In *Proceedings of the 40th International Conference on Software Engineering*, pages 958–968, 2018.

38 Calvin Loncaric, Emina Torlak, and Michael D. Ernst. Fast synthesis of fast collections. In *PLDI 2016: Proceedings of the ACM SIGPLAN 2016 Conference on Programming Language Design and Implementation*, pages 355–368, Santa Barbara, CA, USA, June 2016.

39 David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 48–61, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1065010.1065018.

40 Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. Bottom-up synthesis of recursive functional programs using angelic execution. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, 2022.

41 Ashish Mishra and Suresh Jagannathan. Specification-guided component-based synthesis from effectful libraries. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):616–645, 2022.

42 Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 619–630, 2015.

43 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.

44 Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic: 15th International Workshop, CSL 2001 10th Annual Conference of the EACSL Paris, France, September 10–13, 2001, Proceedings 15*, pages 1–19. Springer, 2001.

45 Hila Peleg, Roi Gabai, Shachar Itzhaky, and Eran Yahav. Programming with a read-eval-synth loop. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 11 2020. doi:10.1145/3428227.

46 Hila Peleg and Nadia Polikarpova. Perfect is the enemy of good: Best-effort program synthesis. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

47 Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–310, 2016.

48 Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

49 Nadia Polikarpova and Ilya Sergey. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi:10.1145/3290385.

50 Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. cvc4sy: smart and fast term enumeration for syntax-guided synthesis. In *International Conference on Computer Aided Verification*, pages 74–83. Springer, 2019.

51 John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

52 Kensen Shi, Jacob Steinhardt, and Percy Liang. FrAngel: component-based synthesis with control structures. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

53 Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of datalog programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 515–527, 2018.

54 Calvin Smith and Aws Albarghouthi. MapReduce program synthesis. *Acm Sigplan Notices*, 51(6):326–340, 2016.

55 Sunbeom So and Hakjoo Oh. Synthesizing imperative programs from examples guided by static analysis. In *International Static Analysis Symposium*, pages 364–381. Springer, 2017.

56 Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 404–415, 2006.

57 Jamie Stark and Andrew Ireland. Towards automatic imperative program synthesis through proof planning. In *14th IEEE International Conference on Automated Software Engineering*, pages 44–51. IEEE, 1999.

58 Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S Păsăreanu. SyRust: automatic testing of Rust libraries with semantic-aware program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 899–913, 2021.

59 Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. TRANSIT: Specifying protocols with concolic snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 287–296, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491956.2462174.

60 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.

61 Zijiang Yang, Jinru Hua, Kaiyuan Wang, and Sarfraz Khurshid. EdSynth: Synthesizing api sequences with conditionals and loops. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 161–171. IEEE, 2018.

62 Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. Inductive program synthesis via iterative forward-backward abstract interpretation. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1657–1681, 2023.

63 Yongwei Yuan, Arjun Radhakrishna, and Roopsha Samanta. Trace-guided inductive synthesis of recursive functional programs. *Proceedings of the ACM on Programming Languages*, 7(PLDI):860–883, 2023.

A Proofs of Theorems and Lemmas

A.1 Proof of Lemma 5

Interchangeability under $\sqsubseteq_{\vec{\Phi}}$ means that given $f \in \mathcal{F}$ with $arity(f) = k$, and k triples $tr_j = \{\vec{P}_j\}p_j\{\vec{Q}_j;\vec{r}_i\}, 1 \leq j \leq k$, with k frame axioms $\vec{R}_j$ s.t. Eval can be applied on f and $\{\vec{P}_j * \vec{R}_j\}p_j\{\vec{Q}_j * \vec{R}_j;\vec{r}_j\}$ to infer $\{\vec{P}\}f(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\}$, and a triple $tr'_i = \{\vec{P}'_i\}p'_i\{\vec{Q}'_i;\vec{r'}_i\}$ s.t. $tr_i \sqsubseteq_{\vec{\Phi}} tr'_i$ for some $1 \leq i \leq k$, then

1. There exists $\vec{R'}_i$ s.t. Eval can still be applied on f when replacing $\{\vec{P}_i * \vec{R}_i\}p_i\{\vec{Q}_i * \vec{R}_i;\vec{r}_i\}$ with $\{\vec{P'}_i * \vec{R}'_i\}p'_i\{\vec{Q'}_i * \vec{R}'_i;\vec{r'}_i\}$ to infer $\{\vec{P'}\}f(p_1,\ldots,p'_i,\ldots,p_k)\{\vec{Q'};\vec{r'}\}$, and
2. $\{\vec{P}\}f(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\} \sqsubseteq_{\vec{\Phi}} \{\vec{P'}\}f(p_1,\ldots,p'_i,\ldots,p_k)\{\vec{Q'};\vec{r'}\}$

Proof: interchangeability. Because $tr_1 \sqsubseteq_{\vec{\Phi}} tr'_1$, there exists $\vec{R}'$ s.t. $\{\vec{P}'_i * \vec{R}'\}p'_i\{\vec{Q}'_i * \vec{R}';\vec{r'}_i\} \equiv_{\vec{\Phi}} \{\vec{P}_i\}p_i\{\vec{Q}_i;\vec{r}_i\}$. Therefore, let us select $\vec{R}'_i = \vec{R}_i * \vec{R}'$, which will keep the composed heap for the sequence exactly as it was, making all midconditions equal again. Since $\vec{r}_i = \vec{r}'_i$, Eval is still applicable.

While sufficient to prove (1), we also notice that it is possible to select a better $\vec{R'}_i$ if some $x \in \mathrm{dom}(\vec{P}_i)$ is removed from $\vec{P}_i$ and now exists only in the frame axioms. We denote $\vec{R'}_j$ for all tr_j new frame axioms without any such x. We now apply Eval to the sequence with the new frame axioms and, since all r_i remain the same, infer $\{\vec{P'}\}f(p_1,\ldots,p'_i,\ldots,p_k)\{\vec{Q'};\vec{r}\}$ where there exists $\vec{R}$ comprising the difference between $\vec{R}_i$ and $\vec{R}'_i$ that proves $\{\vec{P}\}f(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\} \sqsubseteq_{\vec{\Phi}} \{\vec{P'}\}f(p_1,\ldots,p'_i,\ldots,p_k)\{\vec{Q'};\vec{r}\}$. ◀

Consistency under $\sqsubseteq_{\vec{\Phi}}$ means that for goal $\mathcal{G}$, if $\{\vec{P}_1\}p_1\{\vec{Q}_1;\vec{r}_1\} \sqsubseteq \{\vec{P}_2\}p_2\{\vec{Q}_2;\vec{r}_2\}$, then if there exists $\vec{R}_1$ s.t. $\{\vec{P}_1 * \vec{R}_1\}p_1\{\vec{Q}_1 * \vec{R}_1;\vec{r}_1\}$ matches $\mathcal{G}$, then there exists $\vec{R}_2$ s.t. $\{\vec{P}_2 * \vec{R}_2\}p_2\{\vec{Q}_2 * \vec{R}_2;\vec{r}_2\}$ matches $\mathcal{G}$.

Proof: consistency. Since $\{\vec{P}_1 * \vec{R}_1\}p_1\{\vec{Q}_1 * \vec{R}_1;\vec{r}_1\}$ matches $\mathcal{G}$, we know that $\vec{P}_1 * \vec{R}_1 = \vec{I}$, $\vec{q} \subseteq \vec{Q}_1 * \vec{R}_1$, and $\forall v_i@l_i \in \vec{r}_1.\omega_i = v_i$.

Because $\{\vec{P}_1\}p_1\{\vec{Q}_1;\vec{r}_1\} \sqsubseteq \{\vec{P}_2\}p_2\{\vec{Q}_2;\vec{r}_2\}$, $\exists \vec{R}.\{\vec{P}_1\}p_1\{\vec{Q}_1;\vec{r}_1\} \equiv_{\vec{\Phi}} \{\vec{P}_2 * \vec{R}\}p_2\{\vec{Q}_2 * \vec{R};\vec{r}_2\}$, so $\vec{r}_1 = \vec{r}_2$, $\vec{P}_1 = \vec{P}_2 * \vec{R}$, and $\vec{Q}_1 = \vec{Q}_2 * \vec{R}$.

From this, $\forall v_i@l_i \in \vec{r}_2.\omega_i = v_i$, $\vec{P}_2 * \vec{R} * \vec{R}_1 = \vec{I}$, and $\vec{q} \subseteq \vec{Q}_2 * \vec{R} * \vec{R}_1$. In other words, there exists $\vec{R}_2 = \vec{R} * \vec{R}_1$ s.t. $\{\vec{P}_2 * \vec{R}_2\}p_2\{\vec{Q}_2 * \vec{R}_2;\vec{r}_2\}$ matches $\mathcal{G}$. ◀

A.2 Proof of correctness of SObEq

In this section, we prove Theorem 6 by breaking it into two theorems: first, a correctness theorem for the SObEq enumeration, then a correctness theorem for SObEq's reduction of the space.

A.2.1 Correctness of the SObEqenumeration

Def. 2 defines a space $(\!|\mathcal{C}|\!)$ where $(\!|\mathcal{C}|\!)_{(0)}$ starts with all possible values of each variable x, and, inductively, will cover all possible concrete states. Our enumeration, however, starts from $\langle x \mapsto \iota_i(x)\rangle_i$, i.e., only from the variable values in the inputs, and only enumerates programs using other preconditions as it discovers them. Many $\vec{\sigma} \in \mathcal{A} \times \cdots \times \mathcal{A}$ will simply not be reached by the enumeration.

We therefore define the *reachable* program space (first without, and later with the OE-reduction), and claim that it will not lose a solution that exists in the full program space.

▶ **Definition 7** (Reachable Program Space). *We define a reachable program space from some initial states $\vec{I}$ to be $\llparenthesis\mathcal{C}\rrparenthesis^{reach} = \bigcup_{i\geq 0}\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(i)}$, where*

$$\begin{aligned}
\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(0)} \;&= \{\{e\vec{m}p\}l\{e\vec{m}p;\overrightarrow{v@\bot}\} | l\in lits, v\in\mathcal{K} \textit{ is the value of } l\}\cup \\
&\quad \{\{\vec{\sigma}\}x\{\vec{\sigma};\vec{r}\} | x\in\mathcal{V}, \vec{\sigma}=\langle x\mapsto\iota_i(x)\rangle_i, \vec{r}=\langle\iota_i(x)@x\rangle_i\} \\
\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n)} = \llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n-1)} &\cup \{\{\vec{P}\}f(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\} | f\in\mathcal{F}, arity(f)=k, \{\vec{\sigma}_i\}p_i\{\vec{\sigma}'_i;\in\}\llparenthesis\mathcal{C}\rrparenthesis_{(n-1)}, \\
&\qquad \exists\vec{R}_1,\ldots,\vec{R}_k.\textsc{Eval} \textit{ of } f \textit{ on } \{\vec{\sigma}_i*\vec{R}_i\}p_i\{\vec{\sigma}'_i*\vec{R}_i;r_i\} \textit{ is } \{\vec{P}\}f(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\}\} \\
&\quad \cup\{\{\vec{a}_x\}x\{\vec{a}_x;\vec{r}\} | x\in\mathcal{V}, \vec{a}_x=\langle x\mapsto\sigma_i(x)\rangle_i, \vec{r}=\langle\sigma_i(x)@x\rangle_i, \\
&\qquad \{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r'}\}\in\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n-1)} \vee \{\vec{\sigma}'\}p\{\vec{\sigma};\vec{r'}\}\in\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n-1)}\}
\end{aligned}$$

▶ **Theorem 8.** *If a valid solution $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$ exists in $\llparenthesis\mathcal{C}\rrparenthesis$, then $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\in\llparenthesis\mathcal{C}\rrparenthesis^{reach}$.*

We prove Theorem 8 by showing that *all* programs that can be reached in an evaluation sequence from the initial state in$\llparenthesis\mathcal{C}\rrparenthesis$ also exist in $\llparenthesis\mathcal{C}\rrparenthesis^{reach}$.

We denote a program that can be part of a valid execution sequence starting at the start states $\vec{I}$ as *reachable.* In our program representation, this means $\vec{I}$ satisfies the preconditions of the sequence. Programs that are not reachable are irrelevant for us, as they cannot participate in any solution. We define a helper predicate for a program $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$ being reachable at iteration n as follows:

$$\begin{aligned}
\textit{reach}(\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\},n) \triangleq\; & \exists\vec{R}.\vec{\sigma}*R=\vec{I}\vee \\
& \exists\{\vec{\sigma}_0\}p_0\{\vec{\sigma}'_0;\vec{r_0}\},\ldots,\{\vec{\sigma}_i\}p_i\{\vec{\sigma}'_i;\vec{r_i}\}\in\llparenthesis\mathcal{C}\rrparenthesis_{(n-1)}. \\
& \exists\vec{R}_0,\ldots,\vec{R}_{i+1}.\forall 0<j\leq i.(\vec{\sigma}'_{j-1}*\vec{R}_{j-1})=(\vec{\sigma}_j*\vec{R}_j)\wedge \\
& \vec{\sigma}'_i*\vec{R}_i=\vec{\sigma}*\vec{R}_{i+1}\wedge \\
& \exists\vec{R}'.\vec{\sigma}_0*\vec{R}_0*\vec{R}'=\vec{I}
\end{aligned}$$

Proof. We show by induction over n that all programs in $\llparenthesis\mathcal{C}\rrparenthesis_{(n)}$ whose precondition holds for $\vec{I}$, or have some precondition $\vec{\sigma}$ and can form a valid execution trace starting a precondition that holds for $\vec{I}$, will also be in $\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n')}$ for some n'. I.e.,

$$\forall\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\in\llparenthesis\mathcal{C}\rrparenthesis_{(n)}.\textit{reach}(\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\},n)\implies\exists n'.\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\in\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n')}$$

This implies that for any solution $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\in\llparenthesis\mathcal{C}\rrparenthesis_{(n)}$, $\exists n'.\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\in\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n')}$ as well, which is what $Theorem$ 8 states.

For $\underline{n=0}$: For all $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\in\llparenthesis\mathcal{C}\rrparenthesis_{(0)}$, $p=c\in\mathcal{C}$. Since there is no $\llparenthesis\mathcal{C}\rrparenthesis_{(-1)}$, the second disjunct of *reach* is irrelevant. That leaves $\exists\vec{R}.\vec{\sigma}*R=\vec{I}$, where by definition $\{\vec{\sigma}\}c\{\vec{\sigma}';\vec{r}\}\in\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(0)}$: $\vec{\sigma}=\vec{\sigma}'$ is either $e\vec{m}p$ or the initial values of variable c.

For $\underline{n\geq 1}$: We begin by assuming the IH:

$$\forall\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\in\llparenthesis\mathcal{C}\rrparenthesis_{(n-1)}.\textit{reach}(\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\},n-1)\implies\exists n'.\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\in\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n')}$$

Then for all $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\in\llparenthesis\mathcal{C}\rrparenthesis_{(n)}$ where $\textit{reach}(\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\},n)$, there are two cases:

1. If $p=c$ (i.e., $\textit{arity}(c)=0$):
   - i. if $\vec{\sigma}=e\vec{m}p$ or $\vec{\sigma}=\langle x\mapsto\iota_i(x)\rangle_i$, then $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\in\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(0)}$.
   - ii. for other $\vec{\sigma}$, but $\textit{reach}(\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\},n)$, then from *reach* there exist a sequence of triples $\{\vec{\sigma}_j\}p_j\{\vec{\sigma}'_j;\vec{r_j}\}\in\llparenthesis\mathcal{C}\rrparenthesis_{(n-1)}$ and matching frame axioms $\vec{R}_j$ s.t. $\vec{\sigma}_i*\vec{R}_i=\vec{\sigma}*R_{i+1}$, $\exists\vec{R}'.\vec{\sigma}_0*\vec{R}_0*\vec{R}'=\vec{I}$, and $\vec{\sigma}'=\vec{\sigma}$ since c has no effect.
     By the IH, there exists an n' where all $\{\vec{\sigma}_j\}p_j\{\vec{\sigma}'_j;\vec{r}_j\}\in\llparenthesis\mathcal{C}\rrparenthesis^{reach}n'$.
     Since c is not at its original values, it is modified by some some $\{\vec{\sigma}_j\}p_j\{\vec{\sigma}'_j;\vec{r}_j\}\in\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n')}$, so by Def. 7, $\{\langle c\mapsto\sigma'_j(c)\rangle_i\}c\{\langle c\mapsto\sigma'_j(c)\rangle_i;\vec{r'}\}$ was added to $\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n'+1)}$.

In all cases, $\exists n'.\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \in \llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n')}$.

2. If p is of the form $f(p_1,\ldots,p_k)$ (i.e. $f \in \mathcal{F}, arity(f) = k$), then its children $\{\vec{\sigma}_j\}p_j\{\vec{\sigma}'_j;\vec{r}_j\} \in \llparenthesis\mathcal{C}\rrparenthesis_{(n-1)}, 1 \leq j \leq k$ and $\exists.\vec{R}_1,\ldots,\vec{R}_k$ s.t. $\textsc{Eval}$ can be applied to them. $\vec{\sigma} = \vec{\sigma}_1 * \vec{R}_1$.
   Since we know there is a frame axiom $\vec{R}_{i+1}$ s.t. $\vec{\sigma} * \vec{R}_{i+1} = \vec{\sigma}_i * R_i$, this means that the same one can be used for the sequence of arguments to f. This implies that for each $\{\vec{\sigma}_j\}p_j\{\vec{\sigma}'_j;\vec{r}_j\}$, $reach(\{\vec{\sigma}_j\}p_j\{\vec{\sigma}'_j;\vec{r}_j\}, n-1)$.
   And so, by the induction hypothesis, $\exists n_j.\{\vec{\sigma}_j\}p_j\{\vec{\sigma}'_j;\vec{r}_j\} \in \llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n_j)}$, and if we let $n'' = \max(n_j)$, by the definition of $\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n)}$, $\forall 1 \leq j \leq k.\{\vec{\sigma}_j\}p_j\{\vec{\sigma}'_j;\vec{r}_j\} \in \llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n'')}$. Since all the child sub-expressions of p exist in $\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n'')}$, p will be enumerated in $\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n''+1)}$, and so if we let $n' = n''+1$, $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \in \llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n')}$.

◀

A.2.2 Correctness of the SObEq reduction

Next, we show the SObEq enumeration (Algorithm 1) is correct: its reduction will not lose any solutions.

▶ **Definition 9** (SObEq-Reduced Program Space). *Given the specifications $\vec{\Phi}$ and the equivalence relation $\equiv_{\vec{\Phi}}$ (Def. 3) and order relation $\sqsubseteq_{\vec{\Phi}}$ (Def. 4) they induce, we define the reduction*

$$\begin{aligned} reduce(\vec{\Phi},\mathcal{S}_{-1},\mathcal{S}) = \big\{ s(ec) = \{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \,|\, ec \in \mathcal{S}/\equiv_{\vec{\Phi}}, \\ \forall \{\vec{P}\}p'\{\vec{Q};\vec{r'}\} \in \mathcal{S} \cup \mathcal{S}_{-1}.\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\} \not\sqsubseteq_{\vec{\Phi}} \{\vec{P}\}p'\{\vec{Q};\vec{r'}\}\big\} \end{aligned}$$

where $s(ec)$ selects a representative of equivalence class ec, and $/\equiv_{\vec{\Phi}}$ partitions the space into equivalence classes based on $\equiv_{\vec{\Phi}}$. We then define the space after the SObEq reduction to be $\bigcup_{i\geq 0}\llparenthesis\mathcal{C}\rrparenthesis^{S}_{(i)}$ where $\llparenthesis\mathcal{C}\rrparenthesis^{S}_{(i)}$ is also defined inductively:

$$\begin{aligned} &\llparenthesis\mathcal{C}\rrparenthesis^{S}_{(0)} = reduce(\vec{\Phi},\emptyset,\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(0)}) \\ &\llparenthesis\mathcal{C}\rrparenthesis^{S}_{(n)} = \llparenthesis\mathcal{C}\rrparenthesis^{S}_{(n-1)} \cup reduce\Big(\vec{\Phi},\llparenthesis\mathcal{C}\rrparenthesis^{S}_{(n-1)},\{ \\ &\qquad \{\vec{P}\}f(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\} \,|\, f \in \mathcal{F}, arity(f) = k, \{\vec{\sigma}_i\}p_i\{\vec{\sigma}'_i;\vec{r}_i\} \in \llparenthesis\mathcal{C}\rrparenthesis^{S}_{(n-1)}, \\ &\qquad \exists \vec{R}_1,\ldots,\vec{R}_k.\textsc{Eval} \text{ of } f \text{ on } \{\vec{\sigma}_i * \vec{R}_i\}p_i\{\vec{\sigma}'_i * \vec{R}_i;r_i\} \text{ is } \{\vec{P}\}f(p_1,\ldots,p_k)\{\vec{Q};\vec{r}\}\} \\ &\qquad \cup \big\{\{\vec{a}_x\}x\{\vec{a}_x;\vec{r}\} \,|\, x \in \mathcal{V}, \vec{a}_z = \langle z \mapsto \sigma_i(x)\rangle_i, \vec{r} = \langle \sigma_i(x)@x\rangle_i, \\ &\qquad \{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r'}\} \in \llparenthesis\mathcal{C}\rrparenthesis^{S}_{(n-1)} \vee \{\vec{\sigma}'\}p\{\vec{\sigma};\vec{r'}\} \in \llparenthesis\mathcal{C}\rrparenthesis^{S}_{(n-1)}\big\}\Big) \end{aligned}$$

Then to show that the SObEq enumeration using $\equiv_{\vec{\Phi}}$ and $\sqsubseteq_{\vec{\Phi}}$ is correct, we re-introduce the correctness theorem from [45, Theorem 7.2] for an OE-reduced enumeration in our terms:

▶ **Theorem 10** (Correctness of the reduced enumeration). *Given $\{\vec{\sigma}_1\}p_1\{\vec{\sigma}'_1;\vec{r}_1\} \in \llparenthesis\mathcal{C}\rrparenthesis^{reach}$ and a goal $\mathcal{G}_{\vec{\Phi}}$, if there exists $\vec{R}_1$ s.t. $\{\vec{\sigma}_1 * \vec{R}_1\}p_1\{\vec{\sigma}'_1 * \vec{R}_1;\vec{r}_1\}$ matches $\mathcal{G}_{\vec{\Phi}}$, then there exists $\{\vec{\sigma}_2\}p_2\{\vec{\sigma}'_2;\vec{r}_2\} \in \llparenthesis\mathcal{C}\rrparenthesis^{S}$ and a $\vec{R}_2$ s.t. $\{\vec{\sigma}_2 * \vec{R}_2\}p_2\{\vec{\sigma}'_2 * \vec{R}_2;\vec{r}_2\}$ also matches $\mathcal{G}_{\vec{\Phi}}$.*

First, we show programs are not lost, and equivalent programs in the space still exist:

▶ **Lemma 11.** *If there exists a triple $\{\vec{\sigma}_1\}p_1\{\vec{\sigma}'_1;\vec{r}_1\} \in \llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n)}$ then there is a triple $\{\vec{\sigma}_2\}p_2\{\vec{\sigma}'_2;\vec{r}_2\} \in \llparenthesis\mathcal{C}\rrparenthesis^{S}_{(n)}$ s.t. $\{\vec{\sigma}_1\}p_1\{\vec{\sigma}'_1;\vec{r}_1\} \sqsubseteq_{\vec{\Phi}} \{\vec{\sigma}_2\}p_2\{\vec{\sigma}'_2;\vec{r}_2\}$*

In other words, the reduced space has an *interchangeable* and *consistent* program under $\sqsubseteq_{\vec{\Phi}}$.

Proof. $\underline{n=0}$: this is true by the definition of $\llparenthesis\mathcal{C}\rrparenthesis^{S}_{(0)}$.

$\underline{n>0}$: We assume $\forall\{\vec{\sigma}_1\}p_1\{\vec{\sigma}'_1;\vec{r}_1\} \in \llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n-1)}.\exists\{\vec{\sigma}_2\}p_2\{\vec{\sigma}'_2;\vec{r}_2\} \in \llparenthesis\mathcal{C}\rrparenthesis^{S}_{(n-1)}.\{\vec{\sigma}_1\}p_1\{\vec{\sigma}'_1;\vec{r}_1\} \sqsubseteq_{\vec{\Phi}} \{\vec{\sigma}_2\}p_2\{\vec{\sigma}'_2;\vec{r}_2\}$.

For each $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}$ in $\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n)}$:

1. If $p = c$ (i.e., $arity(c) = 0$):
   - i. if $\exists\vec{R}.\vec{\sigma}*\vec{R}=\vec{I}$, then by our base case $\{\vec{\sigma}_2\}p_2\{\vec{\sigma}'_2;\vec{r_2}\}\in\llparenthesis\mathcal{C}\rrparenthesis^S_{(0)}$.
   - ii. if $\neg\exists\vec{R}.\vec{\sigma}*\vec{R}=\vec{I}$, then by definition of $\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n)}$, $\exists tr_1,...tr_k\in\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n-1)}.tr_i=\{\vec{\sigma}_i\}p_i\{\vec{\sigma}'_i;\vec{r}_i\}$ and $\exists\vec{R}_1,...,\vec{R}_{k+1}$ and $\vec{\sigma}=\langle(c\mapsto\vec{\sigma}_k*R_k)_i(c)\rangle_i$. By the induction hypothesis, there exist $tr'_1,...,tr'_k\in\llparenthesis\mathcal{C}\rrparenthesis^S_{(n-1)}$ s.t. $tr_i\sqsubseteq_{\vec{\Phi}}tr'_i$. Since the value of c changed since the initial state, it was changed by (i.e., cannot be in the frame of) some tr_i, so it was also changed by tr'_i. So by definition of $\llparenthesis\mathcal{C}\rrparenthesis^S_{(n)}$, the triple $\{\vec{\sigma}\}c\{\vec{\sigma};\vec{r}\}$ will be enumerated in $\llparenthesis\mathcal{C}\rrparenthesis^S_{(n)}$ at the latest.

2. If $p=f(p_1,...,p_k)$: there exist $tr_1,...,tr_k\in\llparenthesis\mathcal{C}\rrparenthesis^{reach}_{(n-1)},tr_i=\{\vec{\sigma}_i\}p_i\{\vec{\sigma}'_i;\vec{r}_i\}$ s.t. $\exists\vec{R}_1,...,\vec{R}_k$ to enable Eval, and $\vec{\sigma}_1*\vec{R}_1=\vec{\sigma}$. So by the IH there exist $tr'_1,...,tr'_k\in\llparenthesis\mathcal{C}\rrparenthesis^S_{(n-1)},tr'_i=\{\vec{\sigma}''_i\}p'_i\{\vec{\sigma}'''_i;\vec{r'}\}$ s.t. $r_i=r'_i$, and for each tr'_i there exists $\vec{R'}_i$ that is the frame axiom for $tr_i\sqsubseteq_{\vec{\Phi}}tr'_i$: $\vec{\sigma}''_i*\vec{R'}_i=\vec{\sigma}_i$ and $\vec{\sigma}'''_i*\vec{R'}_i=\vec{\sigma}'_i$. Therefore, we can use $\vec{R}_1*\vec{R'}_1,...,\vec{R}_k*\vec{R'}_k$ to apply Eval on $tr'_1,...,tr'_k$. Since $r_i = r'_i$, the result will still be $\vec{r}$, and will have the same effect, i.e., starting at $\vec{P'}=\vec{\sigma}_1*\vec{R}_1*\vec{R''}_1$ where $\vec{R''}_1$ is the minimal necessary frame axiom for eval ($\vec{R''}_1\subseteq\vec{R'}_1$) Eval will yield postcondition assertion $\vec{Q}\subseteq\vec{\sigma}'$.
   By Lemma 5, $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\sqsubseteq_{\vec{\Phi}}\{\vec{P}\}f(p'_1,...,p'_k)\{\vec{Q};\vec{r}\}$, and since all its components are in $\llparenthesis\mathcal{C}\rrparenthesis^S_{(n-1)}$ it will be enumerated by $\llparenthesis\mathcal{C}\rrparenthesis^S_{(n)}$.

◀

We are now ready to prove our theorem:

Proof of Theorem 10. Given a triple $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\in\llparenthesis\mathcal{C}\rrparenthesis^{reach}$ s.t. there exists $\vec{R}$ where $\{\vec{\sigma}*\vec{R}\}p\{\vec{\sigma}'*\vec{R};\vec{r}\}$ matches $\mathcal{G}_{\vec{\Phi}}$, we know from Lemma 11 that there is a $\{\vec{\sigma}''\}p'\{\vec{\sigma}''';\vec{r'}\}\in\llparenthesis\mathcal{C}\rrparenthesis^S$ s.t. $\{\vec{\sigma}\}p\{\vec{\sigma}';\vec{r}\}\sqsubseteq\{\vec{\sigma}''\}p'\{\vec{\sigma}''';\vec{r'}\}$. From Lemma 5 for *consistency*, we get that if $\{\vec{\sigma}*\vec{R}\}p\{\vec{\sigma}'*\vec{R};\vec{r}\}$ matches $\mathcal{G}_{\vec{\Phi}}$, then there exists $\vec{R}'$ s.t. $\{\vec{\sigma}''*\vec{R}'\}p'\{\vec{\sigma}'''*\vec{R}';\vec{r'}\}$ matches $\mathcal{G}_{\vec{\Phi}}$, or in other words, there exists a solution in $\llparenthesis\mathcal{C}\rrparenthesis^S$. ◀

B Benchmark information

Table 1 $|\vec{\Phi}|$: number of specifications provided for the benchmark; $|\mathcal{V}|$: total number of variables; $|q|$: number of variables with a sepcified effect.

				# vars by type						
	name	$\|\vec{\Phi}\|$	$\|\mathcal{V}\|$	int	string	list	set	return type	# `immut`	$\|q\|$
Must	ArraySwap.sy	2	2	1	0	1	0		1	1
	set-adddelete.sy	3	3	0	2	0	1		0	1
	fill_fibonacci_loopy_window.sy	3	1	0	0	1	0		0	1
	get_range_loopy_for.sy	5	4	3	0	1	0		3	1
	rotate_queue.sy	4	2	1	0	1	0		1	1
	leetcode-setdelete.sy	3	3	1	0	1	1	Bool	0	1
	buffer-add.sy	1	4	3	0	1	0	Int	0	2
	MaxAndMinWithN.sy	2	2	1	0	1	0	Int	0	1
	MoveFromAToB.sy	2	2	0	0	2	0	Int	0	2
	get_range_loopy_while.sy	5	4	3	0	1	0	Int	2	2
	MaxAndMin-keepAllElems.sy	2	1	0	0	1	0	Int	0	1
	RemoveMax.sy	3	1	0	0	1	0	Int	0	1
	partition-k-largest.sy	2	2	1	0	1	0	[Int]	0	1
	sum-two-arrays-destructive.sy	2	3	0	0	3	0	[Int]	1	2
	ArrayCounter-mustmut.sy	2	2	1	0	1	0	[Int]	0	1
	leetcode-arraypermutations-mustmut.sy	4	3	1	0	2	0	[Int]	1	1
	SortArrayGivenSwap-mustmut.sy	3	1	0	0	1	0	[Int]	0	1
	leetcode-adddominos.sy	2	2	0	1	0	1	{String}	0	
May	IsAllPositive.sy	3	1	0	0	1	0	Bool	0	
	IsIncreasing.sy	3	1	0	0	1	0	Bool	0	
	MaxAndMin.sy	2	1	0	0	1	0	Int	0	
	MaxDouble.sy	3	1	0	0	1	0	Int	0	
	count-total-words-in-a-cell.sy	3	1	0	1	0	0	Int	0	
	NegativeIndex.sy	2	2	0	2	0	0	Int	0	
	DigitsToString.sy	1	1	0	0	1	0	String	0	
	DelimitedStringConcat.sy	4	3	0	2	1	0	String	0	
	UppercaseFirstChar.sy	3	1	0	1	0	0	String	0	
	17212077.sy	3	1	0	1	0	0	String	0	
	38871714.sy	2	1	0	1	0	0	String	0	
	43606446.sy	3	1	0	1	0	0	String	0	
	clean-and-reformat-telephone-numbers.sy	3	1	0	1	0	0	String	0	
	exceljet1.sy	3	2	0	2	0	0	String	0	
	exceljet2.sy	3	1	0	1	0	0	String	0	
	exceljet3.sy	3	1	0	1	0	0	String	0	
	exceljet4.sy	3	1	0	1	0	0	String	0	
	extract-nth-word-from-text-string.sy	3	2	1	1	0	0	String	0	
	extract-text-between-parentheses.sy	3	1	0	1	0	0	String	0	
	extract-word-containing-specific-text.sy	2	1	0	1	0	0	String	0	
	extract-word-that-begins-with-specific-character.sy	3	1	0	1	0	0	String	0	
	get-first-name-from-name-with-comma.sy	3	2	1	1	0	0	String	0	
	get-last-word.sy	3	1	0	1	0	0	String	0	
	get-middle-name-from-full-name.sy	3	1	0	1	0	0	String	0	
	initials_small.sy	3	1	0	1	0	0	String	0	
	phone-10-long.sy	3	1	0	1	0	0	String	0	
	phone-5-long.sy	3	1	0	1	0	0	String	0	
	phone-6-long-repeat.sy	3	1	0	1	0	0	String	0	
	remove-text-by-position.sy	3	1	0	1	0	0	String	0	
	split-text-string-at-specific-character.sy	3	2	1	1	0	0	String	0	
	stackoverflow1.sy	3	1	0	1	0	0	String	0	
	stackoverflow2.sy	2	1	0	1	0	0	String	0	
	stackoverflow4.sy	3	1	0	1	0	0	String	0	
	stackoverflow6.sy	2	1	0	1	0	0	String	0	
	stackoverflow7.sy	2	2	0	2	0	0	String	0	
	strip-html-from-text-or-numbers.sy	3	1	0	1	0	0	String	0	
	FirstAndLast.sy	3	1	0	0	1	0	String	0	
	leetcode-arraypermutations-mut.sy	4	3	1	0	2	0	[Int]	1	
	leetcode-arraysum.sy	4	3	1	0	2	0	[Int]	0	
	ArrayCounter.sy	2	2	1	0	1	0	[Int]	0	
	SortArrayGivenSwap.sy	3	1	0	0	1	0	[Int]	0	
	RotateQueueOnce.sy	2	1	0	0	1	0	[Int]	0	
	leetcode-arraypermutation.sy	4	3	1	0	2	0	[Int]	0	
	RotateQueue.sy	3	2	1	0	1	0	[String]	0	
	AddArrayToListOrNull.sy	3	2	0	0	2	0	[String]	0	