We've Got You Covered: Type-Guided Repair of Incomplete Input Generators

ANONYMOUS AUTHOR(S)

Property-based testing is a popular technique for automatically testing semantic properties of a program, specified as a pair of pre- and post-conditions. The efficacy of this approach depends on being able to quickly generate inputs that meet the precondition, in order to maximize the set of program behaviors that are probed. For semantically rich preconditions, purely random generation is unlikely to produce many valid inputs; when this occurs, users are forced to manually write their own specialized input generators. One common problem with handwritten generators is that they may be *incomplete*, i.e., they are unable to generate some values meeting the target precondition. This paper presents a novel program repair technique that patches an incomplete generator so that its range includes every valid input. Our approach uses a novel enumerative synthesis algorithm that leverages the recently developed notion of *coverage types* to characterize the set of missing test values as well as the coverage provided by candidate repairs. We have implemented a repair tool for OCaml input generators, called Cobb, and have used it to repair a suite of benchmarks drawn from the property-based testing literature.

ACM Reference Format:
Anonymous Author(s). 2025. We've Got You Covered: Type-Guided Repair of Incomplete Input Generators. 1, 1 (July 2025), 41 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Property-based testing (PBT) is an increasingly popular methodology for automatically testing rich semantic properties of systems, with PBT frameworks targeting most mainstream programming languages, including Java [50, 59], JavaScript [15], Rust [61], Haskell [9], Python [26], Scala [62], and OCaml [57]. In recent years, PBT frameworks have been effectively applied in a number of real-world settings. Prominent examples include validating real-world commercial storage systems [7], ensuring the correctness of formal specifications against modern architecture and operating system artifacts [60], and generating executable specifications of automotive software components [46].

PBT frameworks require two key components from users: *executable properties* that capture the expected input-output behaviors of the system under test (i.e., pre- and post-conditions), and *test input generators* that generate random values of the input types. The values produced by an input generator are used to validate a system's behaviors, after filtering out any values that do not meet the stated precondition. A generator is simply a nondeterministic program that samples from a space of values, supplying a *family* of inputs against which programs are tested. As a simple example, the following generator for integer trees randomly chooses one of the two constructors of `int tree` using a nondeterministic choice operator, ⊕, and then recursively fills in any of its arguments:

```
let rec genTree (size : int) : int tree =
  if size <= 0 then Leaf
               else Leaf ⊕ Node(int_gen(), genTree (size - 1), genTree (size - 1))
```

Many PBT frameworks have some support for automatically deriving a default generator for an arbitrary algebraic datatype using a similar strategy– `genTree` is effectively what is produced by a `deriving Arbitrary` clause in QuickCheck, for example. Conceptually, a default generator naïvely samples from the space of possible values at random: `genTree n` produces trees of random integers of height at most `n`, for example. Unfortunately, many programs under test impose *sparse preconditions*

2025. ACM XXXX-XXXX/2025/7-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

on their inputs, i.e., a property that an arbitrary input is unlikely to satisfy: e.g., valid postal addresses, well-structured XML documents, red-black trees, or well-typed expressions. If we use `genTree` to test a function that expects valid binary search trees containing at least three elements, for example, we will have to throw away roughly 95% of the values it generates. As the precondition grows more restrictive, the overhead of simply filtering the results of a default generator becomes too great for most users, especially when testing is part of continuous integration [21]. When this occurs, the standard recourse is to manually write an input generator that produces the desired set of inputs. This process is unsatisfactory for end-users, however: a recent study of industrial users of PBT frameworks identified the need for handwritten generators to "be a source of friction for many participants" [20], with practitioners stating that writing generators by hand was a "tedious" and "high-effort" process.

An important challenge when writing generators tailored to a particular precondition is identifying which values *not* to enumerate– a generator that only produces a restricted set of values will miss valid parts of the input space, while one that is too permissive will waste work enumerating terms that are discarded by the testing framework. While PBT frameworks can report how many generated terms do not meet a precondition, signaling when a generator is too permissive, they do not provide similar feedback about the inputs an overly restrictive generator will fail to produce. To address this problem, Zhou et al. [70] recently proposed *coverage types*, a type system for reasoning about the values a generator *must* yield. Intuitively, a function that fails to type check against a particular coverage type $\overline{\tau} \longrightarrow [\nu{:}b \mid \phi]$ will fail to evaluate to at least one value that satisfies the predicate ϕ. Unfortunately, while coverage types can help developers identify when the range of a generator is missing certain values, it still falls to the developer to extend the generator so that its outputs cover those values. Simply using the default generator to augment the outputs of an incomplete generator suffers from the same problems as the naïve sample and filter approach: as our experiments in Section 6.2 show, this strategy fails to meaningfully extend the coverage of an incomplete generator in most scenarios. Thus, a more targeted approach is needed.

In this paper, we propose an approach that frees the developer from this obligation by *automatically repairing* an incomplete generator so that it is complete with respect to a user-specified property. Our approach uses a novel program synthesis algorithm which leverages coverage types to build patches that are guaranteed to fill in any gaps in a generator's coverage. In contrast to the traditional type-guided program synthesis setting, in which valid solutions are defined by the *absence* of unwanted/unsafe behaviors, the success of our repairs is defined by the sorts of behaviors they *add*. This qualitative difference manifests in meaningful ways in the design of our algorithm: in contrast to the safety specifications used by traditional deductive synthesis techniques, a top-level specification of the set of missing values provides limited guidance on how coverage duties should be distributed among the subexpressions of an incomplete generator. On the other hand, it is straightforward to combine partial solutions that only contribute a piece of the missing coverage to build a complete solution. Our algorithm leverages this capability to construct "minimal" solutions, i.e., ones that augment the existing generator with just enough new behaviors to fill in any coverage gaps– for almost all the incomplete generators in our experimental evaluation, 100% of the values produced by their repaired counterparts satisfy the target precondition. As we shall see, our approach can also be used to solve sketch-based synthesis problems [64, 66], wherein users provide a generator template comprised of only the control flow structure the final solution should use, and then rely on our repair algorithm to generate program fragments that complete this skeleton in a way that satisfies the target coverage property.

Fig. 1 depicts the high-level workflow of our repair algorithm and its two main phases. The first phase sets up the repair problem, which the second phase then solves. Our system takes two inputs:

```
let rec gen0or2 n =
  [0] ⊕ [2]
```

```
let rec genInts n =
  if n == 0 then [int_gen()]
  else
   [int_gen()] ⊕
   int_gen() :: genInts(n-1)
```

```
let rec genSomeEvens n =
  if n == 0 then [2 * int_gen()]
  else
   (* [2 * int_gen()] ⊕ *)
   2*int_gen() :: genSomeEvens(n-1)
```

$\{l : \mathit{il} \mid l = [0] \lor l = [2]\}$	<:	$\{l : \mathit{il} \mid 0 < \mathrm{len}(l) \leq n + 1\}$	:>	$\{l : \mathit{il} \mid \mathrm{all\_evens}(l) \land \mathrm{len}(l) = n + 1\}$
$[l : \mathit{il} \mid l = [0] \lor l = [2]]$	:>	$[l : \mathit{il} \mid 0 < \mathrm{len}(l) \leq n + 1]$	<:	$[l : \mathit{il} \mid \mathrm{all\_evens}(l) \land \mathrm{len}(l) = n + 1]$

Fig. 2. Three sized generators for non-empty lists of even numbers, followed by refinement and coverage types for their bodies. The direction of the subtyping relation on the types in each column is included. We use *il* as an alias for **int list**.

an incomplete generator and a target coverage type specifying the inputs that the generator needs to cover. The first phase begins by characterizing the current and missing coverage of the input generator using coverage types. It then builds a program sketch containing typed holes; the typing context of each hole captures all the local variables that can be used to complete that hole. The algorithm's second phase uses this information to complete the sketch, employing an enumerative synthesis procedure to find terms that can be used to patch each of its holes.

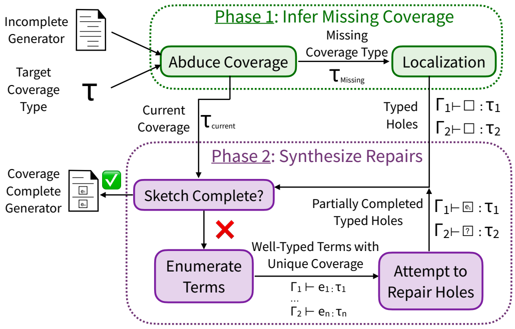
Incomplete Generator
Target Coverage Type
τ
Phase 1: Infer Missing Coverage
Abduce Coverage
Missing Coverage Type
τ_{Missing}
Localization
Current Coverage
τ_{current}
Typed Holes
$\Gamma_1 \vdash \square : \tau_1$
$\Gamma_2 \vdash \square : \tau_2$
Phase 2: Synthesize Repairs
Coverage Complete Generator
Sketch Complete?
Partially Completed Typed Holes
$\Gamma_1 \vdash \boxed{e} : \tau_1$
$\Gamma_2 \vdash \boxed{?} : \tau_2$
Enumerate Terms
Well-Typed Terms with Unique Coverage
$\Gamma_1 \vdash e_1 : \tau_1$
...
$\Gamma_2 \vdash e_n : \tau_n$
Attempt to Repair Holes

Fig. 1. Overview of our proposed pipeline.

In summary, we make the following contributions:

- Show how coverage types can be used to diagnose and formally specify values missing from the range of an incomplete test input generator.
- Present a novel synthesis algorithm that leverages coverage types to intelligently repair incomplete test generators.
- Implement this approach in a tool, Cobb, and demonstrate its efficacy by using it to automatically repair a suite of incomplete generators for a rich class of datatypes and semantic properties drawn from the property-based testing literature.

2 Background and Overview

Before presenting the technical details of our approach, we begin with a brief review of coverage types and then walk through an end-to-end example of our repair procedure. Fig. 2 presents three generators for lists of integers: all three are examples of *sized* generators [11], which use a parameter, in this case n, to bound the number of recursive calls and thus ensure termination. The first generator in Fig. 2, gen0or2, uses the nondeterministic choice operator ⊕ to randomly produce a singleton list containing 0 or 2, genInts yields all non-empty lists of integers whose length is less than n+1, and genSomeEvens generates lists of even numbers with *exactly* n+1 elements.

We will use these three generators to illustrate the difference between coverage types [70] and more standard refinement types [28]. Immediately under each generator are a refinement type, $\{l : \texttt{int list} \mid \phi\}$, and a coverage type $[l : \texttt{int list} \mid \phi]$. The *qualifiers* ϕ of both types capture properties of the range of the generator above them. Although the two types are syntactically similar, their semantic interpretation features an important difference: each refinement type describes a *superset* of the actual range of the generator above it, while each coverage type encodes a *subset* of its actual outputs. This relationship is captured by the subtyping relation for each kind of type. The refinement type in the middle column is a supertype of the types on either side of it, so it can also be assigned to both `gen0or2` and `genSomeEvens`. Coverage types invert this subtyping relationship: all three coverage types in the figure can be assigned to `genInts`, since each type describes a subset of the values it "covers".

Importantly, users will get a type error when checking a generator against a coverage type whose formula can be satisfied by values that fall outside its range: while we can type `gen0or2` against $\{l : \texttt{int list} \mid \texttt{len}(l) = 1 \land 0 \leq \texttt{hd}(l) \leq 2\}$, we cannot type it against a similar coverage type, $[l : \texttt{int list} \mid \texttt{len}(l) = 1 \land 0 \leq \texttt{hd}(l) \leq 2]$, because `[1]` is not one of its outputs. As another example, suppose that a user wants a sized generator for all non-empty lists that contain even integers, a property that is captured by the following type signature:

$$\texttt{n} : \{\texttt{n} : \texttt{int} \mid \texttt{n} \geq 0\} \rightarrow [l : \texttt{int list} \mid \neg\texttt{empty}(l) \land \texttt{len}(l) \leq \texttt{n} + 1 \;\land\; \texttt{all\_evens}(l)] \qquad (\tau_{\mathsf{Ev}})$$

The parameter `n` of this function type has a standard refinement type that stipulates that the function expects a non-negative argument. The return type is a coverage type stipulating that the range of the function includes every non-empty list of even numbers containing *at most* `n+1` elements. Notably, τ_{Ev} is not a valid type for `genSomeEvens`— this generator will never return a list with fewer than `n+1` elements— but it is possible to extend this generator so that it can output such lists by uncommenting the expression on line 4. We will now describe our approach for automatically generating these sorts of patches for incomplete generators.

Our algorithm expects two inputs: a *coverage type* capturing the set of values the repaired generator must output, e.g., τ_{Ev} and an *incomplete generator* to repair. To illustrate the details of our approach, our walkthrough will use the following generator, which is an even more incomplete version of `genSomeEvens` from Fig. 2:[1]

```
let rec genEvens_inc (n : int) : int list =
  if n == 0 then err (* base case *) else err (* recursive case *)
```

Phase 1: Characterizing Missing Coverage. Given these inputs, our repair algorithm begins by identifying any target values not covered by $\texttt{genEvens}_{\texttt{inc}}$. It does so by inferring a pair of coverage types, $[l : \texttt{int list} \mid \phi_{\mathsf{cur}}]$ and $[l : \texttt{int list} \mid \phi_{\mathsf{need}}]$, that capture a) the current coverage of the input generator, and b) the coverage that it is missing, respectively. Intuitively, b) provides a semantic characterization of the term(s) we need to synthesize. In the case of $\texttt{genEvens}_{\texttt{inc}}$, the current coverage is $[l : \texttt{int list} \mid \bot]$, as the function is not guaranteed to output *any* values, and the missing coverage type is simply the return type of τ_{Ev}, i.e.,

$$[l : \texttt{int list} \mid \neg\texttt{empty}(l) \land \texttt{len}(l) \leq \texttt{n} + 1 \;\land\; \texttt{all\_evens}(l)] \qquad (\tau_{\mathsf{EvNeed}})$$

If we had used `genSomeEvens` instead, τ_{EvNeed} would be:

$$[l : \texttt{int list} \mid \texttt{n} > 0 \land \texttt{len}(l) = 1 \land \texttt{all\_evens}(l)]$$

Since the **else** branch of `genSomeEvens` appends an even integer to the head of a non-empty list returned by a recursive call, it always builds a list with at least two elements.

From here, our algorithm builds a *sketch* [66] of the complete generator by inserting typed holes into each control flow path where a patch can be inserted to add coverage. Our motivating example results in the following sketch with two holes:

```
let rec genEvens_sk (n : int) : int list =
  if n == 0 then □_1 : τ_EvNeed (* base case *) else □_2 : τ_EvNeed (* recursive case *)
```

[1] The `err` expression always throws an exception, so $\texttt{genEvens}_{\texttt{inc}}$ does not produce any outputs.

Our algorithm also attaches a typing context to each hole in the sketch; intuitively, the typing context of a hole summarizes the control flow at that program point. The typing contexts for the holes in our sketch are

$$\mathsf{n} : \{\, \mathsf{n}{:}\mathsf{int} \mid \mathsf{n} = 0 \,\} \vdash \square_1 \;:\; \tau_{\mathsf{EvNeed}} \qquad\qquad \mathsf{n} : \{\, \mathsf{n}{:}\mathsf{int} \mid \mathsf{n} > 0 \,\} \vdash \square_2 \;:\; \tau_{\mathsf{EvNeed}}$$

Phase 2: Synthesis. The next phase of our algorithm synthesizes well-typed terms, $\Gamma_i \vdash e_i \;:\; \tau_{\mathsf{EvNeed}}$ for these holes; replacing each $\square_i$ in our sketch with e_i will produce a generator with the target coverage type. Observe that using the analogous refinement type as the target type of these terms

$$\{\mathsf{l}{:}\texttt{int list} \mid \neg\texttt{empty}(l) \wedge \texttt{len(l)} \leq \mathsf{n} + 1 \;\wedge\; \texttt{all\_evens(l)}\} \qquad (\tau_{\mathsf{EvBad}})$$

admits numerous solutions that are incongruous with our intended use of $\texttt{genEvens}_{\mathsf{sk}}$ as a test generator. Using τ_{EvBad} as the target type for the first hole would allow $\square_1$ to be filled with any singleton list containing an even number, including `[0]`, `[4]`, `[n]`, `[2*n]`, `[4*int_gen()]`. The first three of these expressions are consistent with the bias used by many program synthesizers, Occam's razor, which prioritizes the "smallest" program among candidate solutions [5, 23, 67].

A larger challenge when repairing an incomplete generator is that a description of the behaviors a patch must add does not provide much guidance on how to decompose those behaviors into independently solvable subproblems. To see why, consider how we type check `genInts` against the coverage type below it. Neither of the subexpressions of the $\oplus$ expression in its **`else`** branch check against this type, because neither individually covers all the values it stipulates — indeed, if either did so, the other expression would be redundant! In general, when typing an expression of the form $\mathsf{e}_1 \oplus \mathsf{e}_2$ against a coverage type $[\mathsf{l}{:}\mathsf{b} \mid \phi]$, we cannot simply independently check e_1 and e_2 against that type. Instead, we need to come up with types $[\mathsf{l}{:}\mathsf{b} \mid \phi_1]$ and $[\mathsf{l}{:}\mathsf{b} \mid \phi_2]$ to check e_1 and e_2 against, and then check that the combined coverage of those types is sufficient, i.e. $[\mathsf{l}{:}\mathsf{b} \mid \phi_1 \vee \phi_2] <: [\mathsf{l}{:}\mathsf{b} \mid \phi]$. When type checking $\mathsf{e}_1 \oplus \mathsf{e}_2$, we can use e_1 and e_2 to help infer $[\mathsf{l}{:}\mathsf{b} \mid \phi_1]$ and $[\mathsf{l}{:}\mathsf{b} \mid \phi_2]$, but a top-down, type-directed synthesis algorithm does not have either e_1 and e_2 in hand; it is responsible for generating both terms from a types. Unfortunately, there are many possible ways to partition the coverage responsibilities of a $\oplus$ expression between its subexpressions, each of which results in a different set of synthesis goals, and it is not obvious how to choose between these partitionings.

As a consequence, our synthesis procedure instead adopts a bottom-up approach: iteratively generating a set of partial solutions that can be combined to construct a complete answer. Our algorithm maintains a pool of candidate terms that it uses to generate new terms; this pool grows as the algorithm proceeds. At each iteration of the loop, the algorithm uses a *syntactic* cost function to prioritize the generation of certain terms. Section 4.3 provides more detail on our cost function, but intuitively, smaller terms and terms with more coverage, like generators and recursive calls, have lower cost. Fig. 3 provides some examples of terms at different cost levels for our running example. After generating all the terms at the current cost threshold, our algorithm infers a coverage type for each expression, and uses this *semantic* information to prune out any terms that are unsafe, not useful, or redundant. In the case of terms containing a recursive call, for example, type checking ensures that the first argument to each recursive call is structurally decreasing, ensuring that a generator using such a term will terminate. Our algorithm also uses the inferred types to safely discard any terms that do not provide

$$\Sigma_0 \equiv \{\, \texttt{[0]}, \texttt{[n]}, 1, 3, \texttt{[ ]}, \textbf{Leaf}, \texttt{[int\_gen()]}, \ldots \}$$

$$\Sigma_1 \equiv \{\, \texttt{2*int\_gen()}, \texttt{genEvens(n-1)}, \ldots \}$$

$$\Sigma_2 \equiv \{\, \texttt{0 :: genEvens(n-1)}, \texttt{n :: genEvens(n-1)}, \texttt{int\_gen() :: genEvens(n-1)}, \ldots \}$$

$$\Sigma_3 \equiv \{\, \texttt{genEvens(n-1) ++ genEvens(n-1)}, \texttt{2*int\_gen() :: genEvens(n-1)}, \ldots \}$$

Fig. 3. Example sets of enumerated terms, where the cost of the elements of Σ_i is less than cost of the elements of Σ_{i+1}.

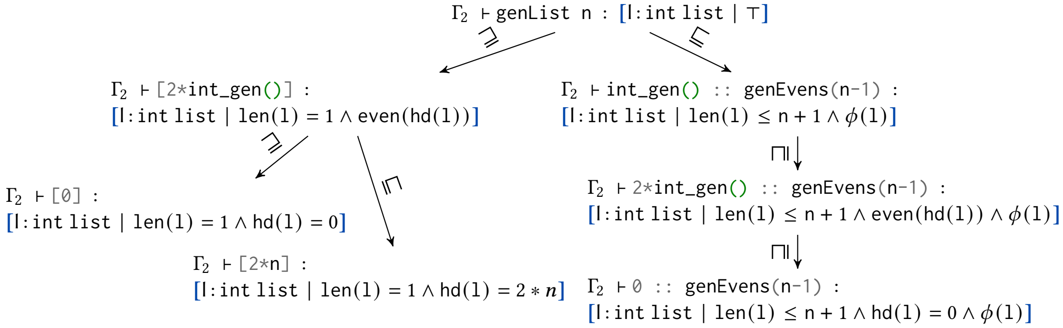
$\Gamma_2 \vdash \texttt{genList n} : [\texttt{l:int list} \mid \top]$
$\sqsupseteq$
$\sqsubseteq$
$\Gamma_2 \vdash \texttt{[2*int\_gen()]} :$
$[\texttt{l:int list} \mid \texttt{len(l)} = 1 \land \texttt{even(hd(l))}]$
$\Gamma_2 \vdash \texttt{int\_gen() :: genEvens(n-1)} :$
$[\texttt{l:int list} \mid \texttt{len(l)} \le \texttt{n} + 1 \land \phi(\texttt{l})]$
$\sqsupseteq$
$\sqsupseteq$
$\sqcap\!\!\mid$
$\Gamma_2 \vdash \texttt{[0]} :$
$[\texttt{l:int list} \mid \texttt{len(l)} = 1 \land \texttt{hd(l)} = 0]$
$\Gamma_2 \vdash \texttt{2*int\_gen() :: genEvens(n-1)} :$
$[\texttt{l:int list} \mid \texttt{len(l)} \le \texttt{n} + 1 \land \texttt{even(hd(l))} \land \phi(\texttt{l})]$
$\sqcap\!\!\mid$
$\Gamma_2 \vdash \texttt{[2*n]} :$
$[\texttt{l:int list} \mid \texttt{len(l)} = 1 \land \texttt{hd(l)} = 2 * n]$
$\Gamma_2 \vdash \texttt{0 :: genEvens(n-1)} :$
$[\texttt{l:int list} \mid \texttt{len(l)} \le \texttt{n} + 1 \land \texttt{hd(l)} = 0 \land \phi(\texttt{l})]$

Fig. 4. A subset of the join semi-lattice built for $\square_2$ in $\texttt{genEvens}_{\texttt{sk}}$, where $\Gamma_2 \equiv \texttt{n} : \{\, \texttt{n:int} \mid \texttt{n} > 0 \,\}$ and $\phi(\texttt{l}) \equiv \neg\texttt{empty(tail(l))} \land \texttt{len(tail(l))} \le (\texttt{n} - 1) + 1 \land \texttt{all\_evens(tail(l))}$.

new coverage. If the pool of candidates already contains the term int_gen(), for example, there is no reason to add int_gen()+1 or int_gen()+int_gen() to it: all of these expressions generate the same terms, and thus have the exact same coverage type. We only add terms that satisfy these sorts of semantic conditions to the pool of enumerated terms.

The final step in our algorithm's enumeration loop checks if a valid completion for any of the holes in the sketch has been found. To do so, it maintains a set of enumerated terms with the same base type as the hole, under the typing context for that hole. This set is partially ordered by the subtyping relation on the types inferred for elements by our type inference algorithm, TyInfer:

$$e_1 \sqsubseteq e_2 \equiv \texttt{TyInfer}(\Gamma_k,\ e_1) <: \texttt{TyInfer}(\Gamma_k,\ e_2)$$

Fig. 4 shows an example of part of this poset for $\square_2$ in $\texttt{genEvens}_{\texttt{sk}}$. As we have seen, if the terms $\texttt{e}_1$ and $\texttt{e}_2$ have the coverage types τ_1 and τ_2 where $\tau_1 <: \tau_2$, then $\texttt{e}_2$ is only guaranteed to generate a subset of the outputs of $\texttt{e}_1$. Thus, this poset tracks the relative coverages of the candidate solutions (as determined by TyInfer) our algorithm has enumerated so far. The top element in this poset is the default generator for our target type, capable of enumerating every list of integers. Its two children only produce a subset of its outputs; they are sibling nodes because neither is a subtype of the other, i.e., neither's outputs subsumes the other's. Importantly, this poset forms a join-semilattice: given any two terms, we can build a term that covers both sets of inputs by joining them together via our nondeterministic choice operator: $\texttt{e}_1 \sqsubseteq (\texttt{e}_1 \oplus \texttt{e}_2) \sqsupseteq \texttt{e}_2$. Our implementation of this poset does not need to maintain these sorts of elements, as it can always use $\oplus$ to reconstruct them on demand: as a consequence, there is no need for Fig. 4 to explicitly include

$$\texttt{n} : \{\, \texttt{n:int} \mid \texttt{n} > 0 \,\} \vdash \texttt{[0]} \oplus \texttt{[2*n]} : [\texttt{l:int list} \mid \texttt{len(l)} = 1 \land \texttt{hd(l)} = 0 \lor \texttt{len(l)} = 1 \land \texttt{hd(l)} = \texttt{2*n}]$$

To check if it has found a solution for a hole, our algorithm first walks down this lattice looking for an element with the same type as the hole, returning that element as the solution if so. The poset corresponding to Fig. 4 for $\square_1$ contains a such direct solution, for example:

$$\texttt{n} : \{\, \texttt{n:int} \mid \texttt{n} = 0 \,\} \vdash \texttt{[2*int\_gen()]} : \tau_{\texttt{EvNeed}} \qquad (\text{p}_1)$$

If a direct solution is not available, our algorithm attempts to build a solution by joining together all the elements that would be immediate subchildren of a hypothetical expression with the target coverage type. While there is no immediate solution to $\square_2$ in Fig. 4, it does contain two expressions that would be direct children of such a node: [2*int_gen()] and 2*int_gen() :: genEvens(n-1). The join of these expressions provides precisely the coverage required by $\square_2$:

$$\texttt{n} : \{\, \texttt{n:int} \mid \texttt{n} > 0 \,\} \vdash \texttt{[2*int\_gen()]} \oplus \texttt{2*int\_gen() :: genEvens(n-1)} : \tau_{\texttt{EvNeed}} \qquad (\text{p}_2)$$

Variables		$x, f, u, \ldots$
Data constructors	$d ::=$	$() \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{O} \mid \mathsf{S} \mid \mathsf{Cons} \mid \mathsf{Nil} \mid \mathsf{Leaf} \mid \mathsf{Node}$
Constants	$c ::=$	$\mathbb{B} \mid \mathbb{N} \mid \mathbb{Z} \mid \ \ldots \ \mid d\,\overline{c}$
Operators	$op ::=$	$d \mid + \mid == \mid < \mid \mathsf{mod} \mid \mathsf{nat\_gen} \mid \mathsf{int\_gen} \mid \ldots$
Values	$v ::=$	$c \mid op \mid x \mid \lambda x{:}t.e \mid \mathtt{fix}\ f(x{:}t) : t \coloneqq e$
Terms and	$e, s ::=$	$v \mid \mathsf{err} \mid \mathtt{let}\ x \coloneqq e\ \mathtt{in}\ e \mid \mathtt{let}\ x \coloneqq op\ \overline{v}\ \mathtt{in}\ e \mid \mathtt{let}\ x \coloneqq v\ v\ \mathtt{in}\ e$
		$\mid \mathtt{match}\ v\ \mathtt{with}\ \overline{d\ \overline{y} \to e}$
Incomplete Terms		$\mid \square\ :\ [v : b \mid \phi]$
Base Types	$b ::=$	$\mathit{unit} \mid \mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid b\ \mathit{list} \mid b\ \mathit{tree} \mid \ \ldots$
Basic Types	$t ::=$	$b \mid t \to t$
Method Predicates	$mp ::=$	$\mathit{emp} \mid \mathit{hd} \mid \mathit{mem} \mid \ldots$
Literals	$l ::=$	$c \mid x$
Propositions	$\phi ::=$	$l \mid \bot \mid \top_b \mid op(\bar{l}) \mid mp(\overline{x}) \mid \neg\phi \mid \phi \land \phi \mid \phi \lor \phi \mid \phi \implies \phi \mid \forall u{:}b.\ \phi \mid \exists u{:}b.\ \phi$
Refined Types	$\tau ::=$	$[v : b \mid \phi] \mid \{v : b \mid \phi\} \mid x{:}\tau \to \tau$
Type Contexts	$\Gamma ::=$	$\emptyset \mid \Gamma, x{:}\tau$

Fig. 5. $\boldsymbol{\lambda}^{TG}$++ syntax.

Replacing the holes in $\mathsf{genEvens}_{\mathsf{sk}}$ with p_1 and p_2 results in a complete generator that is identical to a version of genSomeEvens from with its fourth line uncommented.

We pause here to highlight the distinguishing features of our algorithm: the first is its use of the coverage type τ_{EvNeed} to precisely characterize the behaviors a repair needs to add in order to make $\mathsf{genEvens}_{\mathsf{inc}}$ complete. While τ_{EvNeed} provides a semantic specification for the top-level synthesis problem, it does not provide much guidance on how to decompose that problem into independently solvable subgoals, e.g., when patching $\square_2$. Thankfully, the nondeterministic nature of input generators enables the second key feature of our bottom-up synthesis algorithm, its ability to use ⊕ to combine partial patches into a complete solution, a capability that it used to generate p_2.

3 Language

To formalize our type-based approach to test generator synthesis and repair, we use $\boldsymbol{\lambda}^{TG}$++, a slightly modified version of $\boldsymbol{\lambda}^{TG}$, a core calculus for input generators introduced by Zhou et al. [70]. This section reviews the key features of that original calculus, highlighting our extension along the way. The syntax of $\boldsymbol{\lambda}^{TG}$++ is shown in Fig. 5. The language is a call-by-value lambda-calculus with pattern-matching, inductive datatypes, and recursive functions. Programs are written in monadic normal-form (MNF) [25], a variant of A-Normal Form (ANF) [17] that allows nested let-bindings. $\boldsymbol{\lambda}^{TG}$++ is equipped with generators for numeric types– nat_gen and int_gen– which can evaluate to any number in their range with nonzero probability. These built-in generators suffice to express additional nondeterministic behaviors: the ⊕ choice operator, for example, can be defined as:

$$e_1 \oplus e_2 \equiv \mathtt{let}\ \mathsf{n} \coloneqq \mathsf{nat\_gen}\ ()\ \mathsf{mod}\ 2\ \mathtt{in}\ \mathtt{match}\ \mathsf{n}\ \mathtt{with}\ 0 \longrightarrow e_1 \mid \_ \longrightarrow e_2$$

Like its predecessor, $\boldsymbol{\lambda}^{TG}$++ does not include operators to bias how often values are produced, e.g., QuickCheck's frequency; including such an operator would not fundamentally impact the guarantees we provide for synthesized generators. $\boldsymbol{\lambda}^{TG}$++ is equipped with a completely standard small-step operational semantics, $e \hookrightarrow e'$, that mirrors that of $\boldsymbol{\lambda}^{TG}$.

The only addition $\boldsymbol{\lambda}^{TG}$++ makes to $\boldsymbol{\lambda}^{TG}$ is an additional syntactic category of *incomplete terms* s; these terms may contain one or more typed *holes*, $\square : [v : b \mid \phi]$. Semantically, holes can evaluate to any value satisfying ϕ (EHole), and thus act as a kind of semantic placeholder for a complete

$$\frac{v \models \phi}{\square\ :\ [v : b \mid \phi] \hookrightarrow v}\ \textsc{EHole}$$

Typing $\boxed{\Gamma \vdash s : \tau}$

$$\frac{\Gamma \vdash^{\mathbf{WF}} [\nu : b \mid \phi]}{\Gamma \vdash \square \;:\; [\nu : b \mid \phi] : [\nu : b \mid \phi]}\ \textsc{THole} \qquad \frac{\Gamma \vdash v : \tau_v \quad \overline{\Gamma, \overline{y{:}\tau_y} \vdash d_i(\overline{y}) : \tau_v} \quad \overline{\Gamma, \overline{y{:}\tau_y} \vdash e_i : \tau_i} \quad \overline{\Gamma \vdash^{\mathbf{WF}} \tau_i}}{\Gamma \vdash \mathtt{match}\ v\ \mathtt{with}\ \overline{d_i\ \overline{y} \longrightarrow e_i} \;:\; \bigvee_i \tau_i}\ \textsc{TMatch}$$

$$\frac{\Gamma,\ x{:}\{\nu : b \mid \phi\},\ f{:}x{:}\{\nu : b \mid \nu \prec x\ \wedge\ \phi\} \rightarrow \tau\ \vdash\ e : \tau \quad \Gamma \vdash^{\mathbf{WF}} x{:}\{\nu : b \mid \phi\} \rightarrow \tau}{\Gamma \vdash \mathtt{fix}\ f(x{:}b) : \tau \coloneqq e \;:\; x{:}\{\nu : b \mid \phi\} \rightarrow \tau}\ \textsc{TFix}$$

Fig. 6. Selected $\boldsymbol{\lambda}^{TG}$++ typing rules.

patch. Syntactically, our algorithm use holes to identify program points at which repairs can be inserted. Given an incomplete program s with j holes, we write $s[\overline{e}]$ to denote the complete program where the i^{th} hole has been replaced by e_i. The output of our repair algorithm is a *syntactically* complete $\boldsymbol{\lambda}^{TG}$ program, i.e. it does not contain any holes, that is also *semantically* complete, i.e. it can produce all inputs satisfying the target property.

3.1 Type System

$\boldsymbol{\lambda}^{TG}$++ inherits the type system of its predecessor; like $\boldsymbol{\lambda}^{TG}$, $\boldsymbol{\lambda}^{TG}$++ has three categories of types: *base types*, *basic types*, and *refined types*. Base types (b) include primitive types, e.g., `unit` and `bool`, and inductive datatypes, e.g., `int list` and `bool tree`. Basic types (t) extend base types with function types. As in other refinement type systems, refined types (τ) qualify base types with predicates in a decidable fragment of first-order logic (FOL). In $\boldsymbol{\lambda}^{TG}$++, however, type refinements have two distinct modalities: as we saw in Section 2, the qualifiers of coverage types ($[\nu : b \mid \phi]$) identify a *subset* of the values a nondeterministic expression must be able to evaluate to, while the qualifiers of refinement types ($\{\nu : b \mid \phi\}$) characterize a *superset* of the values an expression may evaluate to. In order to express rich shape properties over inductive datatypes, we allow propositions to reference *method predicates*, boolean-valued functions on inductive datatypes like *emp*, *hd*, and *mem*. Using such predicates, it is straightforward to generate verification conditions that can be handled by an off-the-shelf theorem prover like Z3 [12]. In order to ensure that type checking is decidable, our type system restricts refinements to effectively propositional (EPR) sentences (i.e., prenex-quantified formulae of the form $\exists^*\forall^*\varphi$ where φ is quantifier-free). Following $\boldsymbol{\lambda}^{TG}$, our type system allows function parameters to be qualified by refinements that specify when it is safe to apply a test generator, while a generator's return type is qualified using a coverage type that characterizes the values it is guaranteed to produce.

Fig. 6 presents selected typing rules for $\boldsymbol{\lambda}^{TG}$++. The newly added typing rule for holes, THole, reflects the intuition that a hole is an oracle that can produce any value satisfying the qualifier of its annotated type. The sole premise of THole is a $\vdash^{\mathbf{WF}}$ judgment that ensures that a hole is annotated with a well-formed type, e.g., that ϕ does not include any free variables.[2] The typing rule for `match` expressions, TMatch, reflects that the coverage provided by pattern matching is the union ($\vee$) of the coverages of its branches, each of which may contribute a different set of values. This is in contrast to how branching control flow structures are treated in standard refinement type systems, where each branch can be independently checked against the type of the overall expression. The type system of $\boldsymbol{\lambda}^{TG}$++ enforces the same high-level properties as that of $\boldsymbol{\lambda}^{TG}$: the typing rule for recursive functions, TFix, for example, uses a well-founded relation on the first argument of a function to ensure that it is terminating. The remaining typing rules are identical to those of $\boldsymbol{\lambda}^{TG}$ and are similar to other refinement type systems [28].

[2] The full set of typing rules and their associated auxiliary judgements are included in the supplementary material.

Algorithm 1: The high-level coverage repair algorithm (Repair)

```
Inputs : $s$: incomplete program, $\Gamma$: typing context for $s$, $[\nu : b \mid \psi]$: target coverage type for $s$
Output : Coverage complete repaired program $e$ such that $\Gamma \vdash e : [\nu : b \mid \psi]$
1 $[\nu : b \mid \psi_{\mathsf{cur}}] \leftarrow$ TyInfer$(\Gamma, s)$;    ▷ Infer initial coverage of $s$
2 $[\nu : b \mid \psi_{\mathsf{need}}] \leftarrow$ Abduce$(\Gamma, [\nu : b \mid \psi_{\mathsf{cur}}], [\nu : b \mid \psi])$;    ▷ Abduce missing coverage
3 $(s', \overline{\Gamma_j \vdash \Box_j : [\nu : b \mid \psi_j]}) \leftarrow$ Localize$(\Gamma, s, [\nu : b \mid \psi_{\mathsf{need}}])$;    ▷ Identify repair locations
4 return Synthesize$(\Gamma, s', \overline{\Gamma_j \vdash \Box_j : [\nu : b \mid \psi_j]}, [\nu : b \mid \psi_{\mathsf{cur}} \lor \psi_{\mathsf{need}}])$;    ▷ Synthesize patches for holes
```

For the purposes of automatic test generator repair, the key property enforced by this type system is that a well-typed term e with the type $[\nu : b \mid \phi]$ can evaluate to every value satisfying ϕ:

Theorem 3.1 (Type Soundness [70]). *A well-typed test generator of type* $\vdash f : \overline{x_i : \{\nu : b_i \mid \phi_i\}} \to [\nu : b \mid \phi]$, *when applied to well-typed arguments* $\vdash \overline{v_i : \{\nu : b_i \mid \phi_i\}}$, *can evaluate to every value satisfying* $\phi[\overline{x_i \mapsto v_i}]$: $\forall v.\ \phi[\overline{x_i \mapsto v_i}, \nu \mapsto v] \implies f\ \overline{v_i} \hookrightarrow^{*} v$.[3]

$\boldsymbol{\lambda}^{TG}$++ is also equipped with a decidable bidirectional typing algorithm whose type synthesis (TyInfer) and type checking subroutines will play key roles in the repair algorithm we now present.

4 Input Generator Repair

Our top-level repair algorithm, shown in Algorithm 1, closely follows the workflow depicted in Fig. 1. Most of its functionality is delegated to three key subroutines (Abduce, Localize, and Synthesize); this section presents the important details these subroutines, focusing in particular on Synthesize. Repair takes the body of the target generator s (potentially with user-provided holes), a typing context Γ, and the target coverage type $[\nu : b \mid \psi]$. The algorithm is additionally parameterized over several ingredients that it uses to construct repairs: a collection of typed components that Synthesize uses to enumerate terms, a syntactic cost function used to prioritize which terms to enumerate, an upper bound on the cost of enumerated patches, the set of method predicates used in the types of those components and by Abduce to characterize missing coverage, and axioms characterizing the semantics of those method predicates. To avoid cluttering our discussion, we leave these parameters implicit in the definition of Repair and its subroutines.

Repair begins by inferring two coverage types, $[\nu : b \mid \psi_{\mathsf{cur}}]$ (line 1) and $[\nu : b \mid \psi_{\mathsf{need}}]$ (line 2). The former characterizes the current coverage of s, and the latter describes the coverage that s lacks. Next, Repair uses Localize (line 3) to construct a sketch s' that contains holes $\overline{\text{at each location in } s}$ where coverage should be added, as well as a context and type for each hole, $\Gamma_j \vdash \Box_j : [\nu : b \mid \psi_j]$. Repair then constructs the final generator by using Synthesize to patch each hole in s' (line 4).

4.1 Inferring Missing Coverage

The Abduce subroutine infers a coverage type that captures a set of values missing from the range of a generator. Notably, Abduce may return a coverage type that is more general than necessary, i.e., it may represent a superset of the values needed to complete a generator. To motivate why, consider the incomplete generator for length-bounded lists of integers shown in Fig. 7a, `genIntList`. To the right of `genIntList` are qualifiers for the coverage types that Repair will use to characterize the target (ψ), current (ψ_{cur}), and missing (ψ_{need}) coverage of `genIntList`. This generator always returns a list with *exactly* `n` members, so it will fail to type check against a coverage type qualified with ψ, which stipulates that `genIntList` should return every list containing *at most* `n` integers. To perform this check, our type checker infers a type for `genIntList` usingTyInfer, which produces

[3] The proof of Theorem B.1 is included in the supplementary material.

```
if n == 0
 then [ ]
 else
  let h = int_gen() in
  let t = genIntList(n-1) in
   h :: t
```

(a) genIntList

$$\psi \equiv \texttt{len}(\texttt{l}) \leq \texttt{n}$$

$$\psi_{\texttt{cur}} \equiv \begin{array}{rcl} \texttt{n} = 0 & \Longrightarrow & \texttt{empty}(\texttt{l}) \\ \wedge\ \texttt{n} \neq 0 & \Longrightarrow & \exists h.\exists t.\texttt{len}(t) \leq \texttt{n} - 1 \wedge \texttt{hd}(\texttt{l}) = h \wedge \texttt{tl}(\texttt{l}) = t \end{array}$$

$$\psi_{\texttt{need}} \equiv \texttt{len}(\texttt{l}) = 0 \wedge \texttt{len}(\texttt{l}) \leq \texttt{n}$$

(b) Qualifiers of the target, inferred and abduced coverage of genIntList.

Fig. 7. An incomplete generator for length-bounded lists of integers and coverage type qualifiers capturing its target $[\texttt{l}:\texttt{int list} \mid \psi]$, current $[\texttt{l}:\texttt{int list} \mid \psi_{\texttt{cur}}]$ and missing $[\texttt{l}:\texttt{int list} \mid \psi_{\texttt{need}}]$ coverage.

a coverage type with the qualifier $\psi_{\texttt{cur}}$. Note that TyInfer always infers the most precise type it can, so the complexity of $\psi_{\texttt{cur}}$ is commensurate with the definition of genIntList, e.g., the number of its control flow paths and the components it uses. Thus, while we could capture the missing coverage of genIntList by taking the intersection of ψ and $\psi_{\texttt{cur}}$, i.e., $\psi \wedge \neg\psi_{\texttt{cur}}$, the resulting type is overly complex and does not account for the coverage of the components used by Synthesize to construct repairs. Instead, Repair uses Abduce to find a simpler, but still precise characterization of the missing coverage that also aligns with the space of possible patches explored by Synthesize. In the case of genIntList, for example, Abduce identifies a qualifier $\psi_{\texttt{need}}$ which succinctly captures the coverage that needs to be added to genIntList.

Abduce is parameterized over a finite set of atomic formulas, and explores candidate solutions of the form $\bigvee(\bigwedge \overline{\phi} \wedge \bigwedge \overline{\neg\phi}) \wedge \psi$, where ϕ are drawn from this set. If this set contains $\texttt{len}(\texttt{l}) = 1$, $\texttt{empty}(\texttt{l})$, $\texttt{all\_evens}(\texttt{l})$, and $\texttt{n} = 0$, the set of qualifiers considered by Abduce for genEvens includes:

- $\texttt{len}(\texttt{l}) = 1 \wedge \texttt{all\_evens}(\texttt{l})$: this covers all singleton lists of even elements,
- $\texttt{n} \neq 0 \wedge \texttt{len}(\texttt{l}) \neq 1 \wedge \texttt{all\_evens}(\texttt{l})$: this covers all non-singleton even lists where the size parameter is non-zero,
- $(\texttt{len}(\texttt{l}) = 1 \vee \texttt{empty}(\texttt{l})) \wedge \texttt{all\_evens}(\texttt{l})$: this covers even lists with zero or one elements.

From this solution space, Abduce adapts an existing learning-based specification inference algorithm [69] to find a coverage type that captures the missing outputs of the target generator:[4]

Theorem 4.1 (Abduce is Sound). *Given a typing context* Γ, *the current coverage* $[v:b \mid \psi_{\texttt{cur}}]$, *and the target coverage* $[v:b \mid \psi]$, Abduce$(\Gamma, [v:b \mid \psi_{\texttt{cur}}], [v:b \mid \psi])$ *produces a* $\psi_{\texttt{need}}$ *of the form* $\bigvee(\bigwedge \overline{\phi} \wedge \bigwedge \overline{\neg\phi}) \wedge \psi$ *such that* $\Gamma \vdash [v:b \mid \psi_{\texttt{cur}} \vee \psi_{\texttt{need}}] <: [v:b \mid \psi]$. *Moreover,* $\psi_{\texttt{need}}$ *is a minimal solution in the solution space considered by* Abduce:[5]
$$\neg\exists\,\psi' \in \{\psi' \mid \psi' = \bigvee(\bigwedge \overline{\phi} \wedge \bigwedge \overline{\neg\phi}) \wedge \psi\}.\ \Gamma \vdash [v:b \mid \psi_{\texttt{cur}} \vee \psi'] <: [v:b \mid \psi]\ \wedge\ \psi_{\texttt{need}} \Longrightarrow \psi'.$$

4.2 Localization

The Localize subroutine inserts holes into a generator s, to produce a sketch, s', and a set of the locations in s', $\overline{\Gamma_j \vdash \Box_j\ :\ [v:b \mid \phi_j]}$, for the subsequent Synthesize phase to repair. Intuitively, Localize builds s' by inserting holes at the end of every control flow path in s, recording the typing context and missing coverage at that point.[6] Localize leaves any existing holes in s untouched, adding them to the set of repair locations; it also replaces any errs with holes, as these terms contribute no useful coverage. Fig. 8 shows the output of Localize on an incomplete generator for BSTs. Each of the four holes in the resulting sketch is accompanied by a typing context that extends the initial context Γ with hole-specific control flow constraints and local variables, e.g., the extended context for $\Box_1$ in the **then** branch of the generator in Fig. 8 is $\Gamma_1 \equiv \Gamma, \_ : \{v:\texttt{bool} \mid \texttt{n} = 0\}$.

[4] The full definition of Abduce is included in the supplementary material.

[5] The proof of Theorem 4.1 is included in the supplementary material.

[6] The full definition of Localize is included in the supplementary material.

```
if n == 0
 then Leaf
 else if lo + 1 < hi then
   let x = int_range lo hi in
    err
  else □_0 : τ_0
```

```
if n == 0
 then Leaf ⊕ □_1 : τ_need
 else if lo + 1 < hi then
   let x = int_range lo hi in
    □_2 : τ_need
  else □_0 : τ_0 ⊕ □_3 : τ_need
```

$$\left\{\begin{array}{r} \Gamma, \{\mathtt{n} = 0\} \vdash \Box_1 \ : \ \tau_{\mathsf{need}}, \\ \begin{array}{l}\Gamma, \{\mathtt{n} \neq 0\}, \{\mathtt{lo} + 1 < \mathtt{hi}\}, \\ \ \mathtt{x} : [\mathtt{x} : \mathtt{int} \mid \mathtt{lo} \leq \mathtt{x} \leq \mathtt{hi}]\end{array} \vdash \Box_2 \ : \ \tau_{\mathsf{need}}, \\ \Gamma, \{\mathtt{n} \neq 0\}, \{\mathtt{hi} \leq \mathtt{lo} + 1\} \vdash \Box_0 \ : \ \tau_0, \\ \Gamma, \{\mathtt{n} \neq 0\}, \{\mathtt{hi} \leq \mathtt{lo} + 1\} \vdash \Box_3 \ : \ \tau_{\mathsf{need}} \end{array}\right\}$$

Fig. 8. An incomplete generator for BSTs and the sketch and set of holes that Localize produces from it, where $\Gamma \equiv \mathtt{n}{:}\{\mathtt{n} : \mathtt{int} \mid \mathtt{n} \geq 0\}$, $\mathtt{lo}{:}\mathtt{int}$, $\mathtt{hi}{:}\{\mathtt{hi} : \mathtt{int} \mid \mathtt{lo} {\leq} \mathtt{hi}\}$, genBST:... We use $\{\phi\}$ as shorthand for $\_ \ : \ \{\_ : \mathtt{bool} \mid \phi\}$.

Algorithm 2: Synthesize repairs (Synthesize)

```
Inputs :Γ: typing context, s: A sketch with j holes, $\overline{\Gamma_j \vdash \Box_j \ : \ [\nu : b \mid \psi_j]}$: typing contexts and types
        for the j holes in s, $[\nu : b \mid \psi]$: target coverage
Output :A repaired generator $s[\overline{e_j}]$, where $\Gamma \vdash s[\overline{e_j}] \ : \ [\nu : b \mid \psi]$
$\overline{Exp_j} \leftarrow \emptyset$; $\overline{Cand_j} \leftarrow \emptyset$; $\overline{e_j} \leftarrow \mathtt{err}$; $\alpha \leftarrow 0$;
while $\Gamma \nvdash s[\overline{e_j}] \ : \ [\nu : b \mid \psi] \ \wedge \ \alpha \leq$ MaxCost do
    foreach $\Gamma_i \vdash \Box_i \ : \ [\nu : b \mid \psi_i] \in \overline{\Gamma_j \vdash \Box_j \ : \ [\nu : b \mid \psi_j]}$ do
        if $e_i \neq \mathtt{err}$ then continue;                          ▷ Skip if $\Box_i$ has already been repaired
        for $e \in$ genExp$(Exp_i, \alpha)$ do
            $\tau \leftarrow$ TyInfer'$(\Gamma_i,\ e)$;
            if $\Gamma_i \vdash \tau \equiv [\nu : b \mid \bot] \vee \exists e' \in Exp_i.\ \Gamma_i \vdash e' : \tau' \wedge \Gamma_i \vdash \tau' \equiv \tau$ then
                continue;                        ▷ Discard e if unsafe or provides no useful coverage
            $Exp_i \leftarrow Exp_i \cup \{e\}$;
            if $\Gamma_i \vdash [\nu : b \mid \psi_i] <: \tau$ then
                $Cand_i \leftarrow Cand_i \cup \{e\}$;
                $Repairs \leftarrow \{e \mid \exists Cand \subseteq Cand_i.\ e = \bigoplus_{e' \in Cand} e' \ \wedge \ \Gamma_i \vdash$ TyInfer'$(\Gamma_i,\ e) \equiv [\nu : b \mid \psi_i]\}$;
                if $Repairs \neq \emptyset$ then $e_i \leftarrow \min Repairs$;   ▷ Use a precise solution for $\Box_i$ if one exists
    $\alpha \leftarrow \alpha + 1$;
if $\Gamma \vdash s[\overline{e_j}] \ : \ [\nu : b \mid \psi]$ then return $s[\overline{e_j}]$;
foreach $\Gamma_i \vdash \Box_i \ : \ [\nu : b \mid \psi_i] \in \overline{\Gamma_j \vdash \Box_j \ : \ [\nu : b \mid \psi_j]}$ do
    if $e_i \neq \mathtt{err}$ then continue;                          ▷ Skip if $\Box_i$ already has a precise solution
    $r \leftarrow \min \{e \in Exp_i \mid$ TyInfer'$(\Gamma_i, e) <: [\nu : b \mid \psi_i]\}$;      ▷ Find term with minimal excess coverage
    $Repairs \leftarrow \{e \mid \exists Exp' \subseteq \{e' \in Exp_i \mid \Gamma_i \vdash r \sqsubset e'\}.\ e = \bigoplus_{e' \in Exp'} e' \ \wedge$ TyInfer'$(\Gamma_i, e) <: [\nu : b \mid \psi_i]\}$;
    if $Repairs \neq \emptyset$ then $e_i \leftarrow \min Repairs$ else $e_i \leftarrow r$;      ▷ Use the more precise patch for $\Box_i$
return $s[\overline{e_j}]$
```

4.3 Synthesizing Patches

The final subroutine of our algorithm is Synthesize, shown in Algorithm 2, which generates patches for the holes in the sketch built by Localize. As we saw in Section 2, while the type produced by Abduce provides a top-level goal for Synthesize; this type does not provide guidance on how coverage should be apportioned to subexpressions, e.g., in the presence of non-deterministic choice. For that reason, Synthesize uses a bottom-up approach, adopting an inductive-synthesis-style algorithm [1, 3, 42] to enumerate a pool of candidate repairs for each hole.

Synthesize maintains two sets of terms for each hole $\Gamma_i \vdash \Box_i \ : \ [\nu : b \mid \psi_i]$: a general pool of all type-safe terms enumerated so far, Exp_i, and a set of candidate patches $Cand_i$ that cover a portion of the hole's missing outputs. As discussed in Section 2, $Cand_i$ is equipped with a partial order that uses the coverage types of its elements, a property that Synthesize leverages when extracting candidate patches. Synthesize maintains hole-specific sets of terms because the validity of a repair depends on the context into which it is inserted: if a patch uses local variables or its safety depends on a particular set of path conditions, for example. The algorithm also tracks whether it has found a meaningful repair for each hole, e_i; these are initially set to err (line 1). Synthesize uses a loop to iteratively *enumerate* terms below the current cost α (lines 2-14), *filtering* any enumerated terms that are not useful (lines 7-8), and then attempts to *extract* a *precise* patch for each hole from the (partial) solutions it has found (lines 12-13). This loop terminates when either the current set of patches are sufficient to ensure the current sketch has the desired coverage or a limit on the cost of enumerated terms has been reached (line 2). If a coverage-complete solution is in hand when the loop ends, it is returned (line 15); this may happen even if some holes have not been repaired, e.g., if patching a user-provided hole ensures coverage completeness. If the current set of patches do not make the sketch coverage complete, Synthesize then *extracts* the *best* patch it can for each unrepaired hole, attempting to minimize any excess coverage (lines 16-21). We will now describe each of the four main pieces of Synthesize in more detail.

Enumerate. Synthesize uses the genExp subroutine to generate new terms at the current cost threshold (line 5). genExp is parameterized over a set of *seeds* and *components* that are used to construct candidate repairs. The seeds form the initial set of candidate repairs and includes constants, e.g., 0, false and [], the default generators for the base types, e.g., int_gen and genTree, any variables that are in scope for a particular hole. The components are used to construct new terms from previously generated expressions, and include, e.g., built-in operators, datatype constructors, and functions. Seeds and components are both equipped with type signatures characterizing their coverage guarantees. The name of the generator being repaired is available via a hole's typing context, e.g., genBST in Fig. 8, enabling patches to make use of recursive calls. The signature of this recursive component uses a refinement type to ensure that all recursive calls use strictly smaller arguments, the signature of genEvens, for example, is:

$$\texttt{genEvens} : n' : \{n' : \texttt{int} \mid n' \geq 0 \land n' < n\} \rightarrow [l : \texttt{int list} \mid \neg\texttt{empty}(l) \land \texttt{len}(l) \leq n' + 1 \land \texttt{all\_evens}(l)]$$

genExp is parameterized over a cost function that it uses to prioritize certain elements in the search space, a common strategy in the program synthesis literature [1, 22, 23]. Cost functions are required to be monotonic— a term never has a smaller cost than any of its subterms— and stateless— a cost of a term is independent of the terms genExp has already seen. The cost function used in our implementation of Synthesize prefers terms with the following properties:

Recursive Calls We assign a low cost to recursive calls, as they typically provide a large amount of coverage, tailored to the current repair.

Same Type as Target A valid patch must produce values of the target coverage type, so we prioritize components that construct expressions with the same base type as the target over those that do not.

Seed Generators Default type generators like int_gen() often provide useful coverage and are prioritized, alongside variables, over constant expressions.

Diverse Terms A naïve enumeration strategy will produce many terms that have repeated uses of the same components and seeds, e.g. Cons(x, Cons(y, [])). Recursive calls are likely to produce these sorts of terms in a more general way, so we prioritize expressions comprised of diverse subterms.

Filter. A key challenge in enumerative approaches to program synthesis is keeping the set of generated terms from becoming intractably large. Accordingly, Synthesize curates its pool of terms by discarding expressions that are redundant or unlikely to contribute to a solution (lines 6-8).

While `genExp` employs a purely *syntactic* cost function to prioritize terms, `Synthesize` uses the *semantics* of a term when deciding whether it should be pruned. This semantic information is encoded in the coverage type inferred by `TyInfer'` (line 6). `TyInfer'` is a more restrictive version of `TyInfer` which aggressively rejects function applications in order to filter potentially unsafe terms. `TyInfer'` does so by strictly limiting where type subsumption may be applied, so that the inferred coverage types of any function arguments do not include values that violate the refinement types of its parameters. `TyInfer'` implements the following modified version of the typing rule for function applications:

$$\frac{\begin{array}{c}\Gamma \vdash v_1 \Rightarrow a : \{v{:}b \mid \phi_1\} \rightarrow \tau_1 \quad \Gamma \vdash v_2 \Rightarrow c : [v{:}b \mid \phi_2] \quad \mathsf{Disj}(\neg(\phi_1), \phi_2) = \emptyset \\ \Gamma' = a : [v{:}b \mid v = c \wedge \phi_1], x : \tau_1 \quad \Gamma, \Gamma' \vdash e \Rightarrow \tau \quad \tau' = \mathsf{Ex}(\Gamma', \tau) \quad \Gamma \vdash^{\mathbf{WF}} \tau'\end{array}}{\Gamma \vdash \mathtt{let}\ x \coloneqq v_1\ v_2\ \mathtt{in}\ e \Rightarrow \tau'}\ \textsc{SynAppBase'}$$

This stronger rule ensures, e.g., that `Synthesize`'s pools of terms only include recursive calls whose size argument is strictly decreasing.

If `TyInfer'` successfully infers a type for a term $\Gamma_k \vdash e : \tau$, `Synthesize` then decides whether to include e in Exp_k, using τ to judge whether it supplies any meaningful new coverage (line 7). It prunes any e that may not produce any values at all, i.e., when τ is equivalent to $[v{:}b \mid \bot]$. `Synthesize` also discards e if it has already enumerated a *coverage equivalent* term: if two terms cover the same outputs, we only need to keep around the cheaper one, similarly to how other synthesizers use observational equivalence to prune enumerated terms [1, 35]. Any terms that are not filtered are then added to Exp_k (line 9); finally, `Synthesize` additionally checks whether e provides part of the target coverage, adding it to the poset of partial solutions if so (line 11).

Extract a Precise Repair. After enumerating all the useful terms for the current cost bound, `Synthesize` attempts to extract a *precise* patch, i.e., one that supplies exactly the coverage required by the current hole $\square_i$, by using $\oplus$ to combine the partial solutions in $Cand_i$ (lines 12-13). In order to do efficiently, our implementation of `Synthesize` leverages the fact that $Cand_i$ is a join semi-lattice ordered using the inferred types of its elements. The insight is that we can efficiently check if $Cand_i$ contains a complete repair by only examining the elements that are direct supertypes of the target coverage type for a hole $\square_i$, since:

$$\Gamma_i \vdash \bigoplus_{e \in Cand'} e : [v{:}b \mid \psi_{\mathsf{need}}] \Rightarrow \Gamma_i \vdash \bigoplus_{e \in Cand_i} e : [v{:}b \mid \psi_{\mathsf{need}}]$$

where

$$Cand' \equiv \left\{ e \in Cand_i \;\middle|\; \begin{array}{l} [v{:}b \mid \psi_{\mathsf{need}}] <: \mathtt{TyInfer'}(\Gamma_i, e) \\ \wedge\ \nexists e' \in Cand_i.\ [v{:}b \mid \psi_{\mathsf{need}}] <: \mathtt{TyInfer'}(\Gamma_i, e') <: \mathtt{TyInfer'}(\Gamma_i, e) \end{array} \right\}$$

Example 4.2. To see how we leverage this to extract a patch from a poset of candidate solutions, we will show how `Synthesize` builds a precise solution for the hole $\mathtt{n} : \{\mathtt{n}{:}\mathtt{int} \mid \mathtt{n} \geq 0\} \vdash \square : [\mathtt{l}{:}\mathtt{int\ list} \mid \mathtt{len(l)} \leq \mathtt{n}]$. We begin immediately after the term $\mathtt{e}_5$ has been added to the lattice at the top of Fig. 9. We first see if $\mathtt{e}_5$ itself is a precise patch by checking if its inferred type is equivalent to the target coverage $[\mathtt{l}{:}\mathtt{int\ list} \mid \mathtt{len(l)} \leq \mathtt{n}]$. Since this fails, we instead temporarily insert a "dummy" node with the target type into the lattice, producing the lattice at the bottom of Fig. 9. Observe that the inserted node is a direct parent of $\mathtt{e}_3$, $\mathtt{e}_4$, and $\mathtt{e}_5$, and that furthermore the join of these three expressions has the coverage type $[\mathtt{l}{:}\mathtt{int\ list} \mid \mathtt{empty(l)} \vee \mathtt{len(l)} = \mathtt{n} \vee \mathtt{len(l)} < \mathtt{n}]$. Since this type is equivalent to the type of the target hole, we have found a valid patch, which we adapt as the solution for this hole.[7]

[7] Note that this extraction strategy crucially depends on using the precise type inferred by `TyInfer'` to order elements in the lattice. Using the subtyping relation on any valid type (via subsumption) would allow an element's children to provide more coverage than it does: under such a strategy, `int_gen()` could be a child of any element, for example!

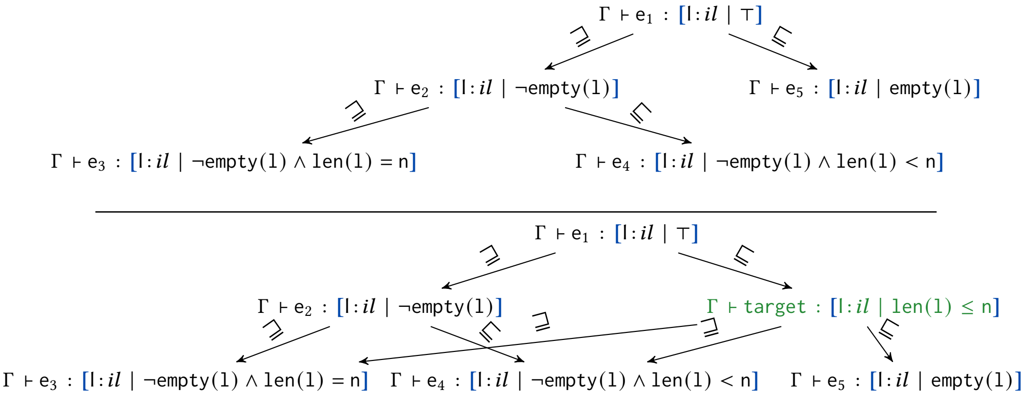
$\Gamma \vdash e_1 : [l{:}il \mid \top]$
$\sqsupseteq$
$\sqsubseteq$
$\Gamma \vdash e_2 : [l{:}il \mid \neg\mathsf{empty}(l)]$
$\Gamma \vdash e_5 : [l{:}il \mid \mathsf{empty}(l)]$
$\sqsupseteq$
$\sqsubseteq$
$\Gamma \vdash e_3 : [l{:}il \mid \neg\mathsf{empty}(l) \land \mathsf{len}(l) = n]$
$\Gamma \vdash e_4 : [l{:}il \mid \neg\mathsf{empty}(l) \land \mathsf{len}(l) < n]$
$\Gamma \vdash e_1 : [l{:}il \mid \top]$
$\sqsupseteq$
$\sqsubseteq$
$\Gamma \vdash e_2 : [l{:}il \mid \neg\mathsf{empty}(l)]$
$\Gamma \vdash \mathsf{target} : [l{:}il \mid \mathsf{len}(l) \leq n]$
$\sqsupseteq$
$\sqsubseteq$
$\sqsupseteq$
$\sqsupseteq$
$\sqsubseteq$
$\Gamma \vdash e_3 : [l{:}il \mid \neg\mathsf{empty}(l) \land \mathsf{len}(l) = n]$
$\Gamma \vdash e_4 : [l{:}il \mid \neg\mathsf{empty}(l) \land \mathsf{len}(l) < n]$
$\Gamma \vdash e_5 : [l{:}il \mid \mathsf{empty}(l)]$

Fig. 9. A lattice of repairs before and after inserting target, where $\Gamma \equiv n : \{n : \mathsf{int} \mid n \geq 0\}$ and $il \equiv$ **int list**

Extract an Imprecise Repair. If `Synthesize` hits its cost bound without finding precise solutions that complete the input sketch, it then extracts the best *imprecise* patch it can for each unrepaired hole (lines 16-20). This process proceeds similarly to the one for precise repairs, with a few key tweaks. Since we know we will not find a precise solution for the current hole, we identify the element from the complete pool of enumerated terms that has the minimal excess coverage (line 18). This term corresponds to the parent of a (hypothetical) element with the target type in our lattice; we identify it by once again inserting a "dummy" node into Exp_i.[8] `Synthesize` then tries to refine this solution by combining subsets of its children in the lattice using ⊕ (line 19); `Synthesize` uses the smallest such combination with sufficient coverage, if one exists (line 20).

Theorem 4.3 (`Repair` is Sound and Total). *Given a program s that is well typed under typing context Γ, $\Gamma \vdash s\ :\ b$, and target coverage type $[\nu{:}b \mid \psi]$,* `Repair` *returns a coverage complete generator,* $\Gamma \vdash \mathtt{Repair}(s,\ \Gamma,\ [\nu{:}b \mid \psi])\ :\ [\nu{:}b \mid \psi]$.

5 Implementation

Cobb, our prototype implementation of the above approach, consists of about 3k lines of OCaml [37], and uses a modified version of Poirot [70] as its coverage type checker; this type checker uses Z3 [12] as its backing SMT solver. Cobb ingests and outputs sized generators in a DSL that closely mimics $\boldsymbol{\lambda}^{TG}$++. We have implemented this language as a shallowly embedded DSL in OCaml, and repaired generators can be directly executed using OCaml's QCheck framework [57]. Cobb is parameterized over a set of base types, components, and method predicates. It currently supports a number of standard OCaml primitive operations and datatypes, and includes built-in method predicates for expressing properties of these types, e.g., `empty` and `sorted`.

The guarantees provided by the `Repair` algorithm are *possibilistic*: the coverage guarantees of weighted and fair implementations of ⊕ are the same. In practice, however, users often prefer generators that bias the choices of ⊕. If only one of its choices includes a recursive call, for example, a fair implementation of ⊕ will bias a generator towards smaller values: `genTree` in the introduction generates **`Leaf`** nodes half of the time, for example. Thus, Cobb adopts the commonly used approach of using the bound of a sized generator to bias uses of ⊕ operators [32]: after synthesis, Cobb applies a syntactic transformation to adjust the weights of ⊕ in which only a single choice has a recursive call, weighting that choice according to the current bound. In practice, this means that Cobb produces generators that are initially biased towards recursive calls, but which are more likely to take the other choice as `size` decreases.

[8] In the worst case, this will be the default generator for the target type, as this is always included in our set of seeds.

6 Evaluation

Using Cobb, we have investigated five key questions about our approach to generator repair:

RQ1 Is our approach able to automatically find complete repairs for different kinds of generators, covering a diverse set of properties and datatypes, in a reasonable amount of time?

RQ2 Is Cobb effective when used as a sketch-based synthesizer? Can it produce a generator from a skeleton that contains only the desired control flow structure of the target generator?

RQ3 How does our approach compare to alternative repair strategies that exclusively prioritize either safety or completeness?

RQ4 How sensitive is our approach to the set of components used?

RQ5 How do our statically repaired generators compare to alternative complete input generation approaches that rely on run-time constraint solving?

All of our experiments were run on a 2020 M1 13-inch MacBook Pro with 8 GB of memory.

6.1 Synthesis of Coverage Complete Generators (RQ1, RQ2)

Our first set of experiments evaluate the ability of Cobb to automatically repair an incomplete input generator, and considers a diverse set of data types (e.g., lists, trees, and lambda terms) and target preconditions (e.g., sorted lists, balanced trees, and well-typed lambda terms) (**RQ1**). To build the incomplete generators used in our experiments, we took coverage-complete generators drawn from the existing PBT literature [30, 32, 69], and made them incomplete by replacing one or more of their branches with `err`. We construct multiple variants of each generator by removing different combinations of branches, including *sketches* of each generator which replace all its branches with `err` (**RQ2**). The coverage type specification used in each benchmark is a direct translation of the precondition of the function under test. Table 1 presents the results of using Cobb to repair each of these variants. The variants are (roughly) ordered by the amount of the functionality they lack, with the sketch acting as the final variant of each generator. The required repairs range from the relatively trivial— the first two sized list variants only require inserting an empty list (`[ ]`), for example— to the more substantial: repairing the red-black tree sketch requires synthesizing multiple recursive calls and applications of datatype constructors with very specific arguments. Each benchmark uses one of three sets of components based on its base type, i.e., list, tree, or red-black tree. Each set includes all the components needed by at least one of the original generators targeting that base type.

For almost every variant, Cobb was able to find a repair that was equivalent to the term that had been replaced by `err`, modulo some syntactic differences (e.g., order of operations, normal form), including for every sketch (**RQ2**). A notable exception is the sixth variant of the red-black tree generator, shown in Fig. 10: while the original generator directly constructs a black node and its subtrees, Cobb finds a smaller, but semantically equivalent repair which makes a recursive call with the correct color/invariant arguments. Our cost function biases recursive terms earlier in the synthesis process because they produce similar coverage to that of our goal. In this case, the coverage supplied by the original branch can be fully realized by flipping the color in the recursive call and updating the size invariant.

```
let lt = rbtree_gen (inv - 2) false (h - 1) in
let rt = rbtree_gen (inv - 2) false (h - 1) in
Rbtnode (false, lt, int_gen(), rt)
```

```
rbtree_gen (inv - 1) true h
```

Fig. 10. The relevant part of the original and Cobb-repaired versions of the sixth red-black tree variant.

For all these benchmarks, Cobb was able to find a complete generator within a reasonable amount of time (**RQ1**), with the time taken roughly correlated to the complexity of the target property and the functionality that was removed. Most of the repair time is spent on calls to Z3, with `Abduce` and `Synthesize` dominating the total runtime. Longer abduction times generally correspond to a more

Table 1. The results of using Cobb to repair incomplete generators. Benchmarks are annotated with their source: QuickChick [32] (*), Lampropoulos et al. [30] (*) and Zhou et al. [69] ($^\diamond$). The middle set of columns characterize the complexity of the problem and the solution: the number of holes in the initial sketch (#Holes) and the size of the AST of the term that is synthesized (Repair Size). The last set of columns describe the effort required to find a repair: the number of terms enumerated (#Terms), the number of SMT queries (#Queries), the time it took to infer the missing coverage (Abduction), the time spent generating the final solution (Synthesis), and the total time needed to find a coverage complete generator (Total).

Benchmark	#Holes	Repair Size	#Queries	#Terms	Abduction (s)	Synthesis (s)	Total Time(s)
Sized List* 1	1	1	31	3	0.18	0.4	0.61
2	1	1	30	3	0.15	0.32	0.51
3	1	14	105	22	0.18	1.67	1.89
4	2	2	38	6	0.2	0.39	0.63
5	1	20	122	22	0.21	2.02	2.27
6	2	15	112	25	0.17	1.7	1.9
7	2	21	128	25	0.3	2.04	2.38
8	1	20	121	22	0.19	2.14	2.37
sketch	2	21	127	25	0.37	2.03	2.43
Duplicate List* 1	1	1	36	4	0.11	0.66	0.8
2	1	18	110	67	0.07	4.09	4.19
sketch	2	19	121	71	0.17	4.48	4.68
Unique List$^\diamond$ 1	1	1	30	3	0.1	0.29	0.43
2	2	7	68	16	0.08	1.08	1.18
sketch	3	8	71	19	0.11	1.09	1.22
Sorted List* 1	1	1	35	4	0.12	0.36	0.51
2	2	16	291	226	0.09	4.47	4.59
sketch	3	17	298	230	0.11	4.46	4.61
Even List 1	1	11	123	35	0.38	1.91	2.34
2	1	11	199	48	0.33	4.76	5.14
3	1	17	202	58	0.38	5.42	5.84
4	2	22	301	83	0.48	6.3	6.82
5	1	33	261	58	0.48	7.42	7.98
6	2	28	304	93	0.51	7.06	7.61
sketch	2	40	355	93	0.65	8.86	9.56
Sized Tree* 1	1	1	37	3	0.36	0.56	0.99
2	1	1	36	3	0.32	0.55	0.95
3	1	18	165	17	0.36	6.31	6.75
sketch	2	25	188	20	0.5	6.88	7.45
Complete Tree* 1	1	1	34	3	0.18	0.36	0.6
2	1	18	218	45	0.17	3.39	3.65
sketch	2	19	225	48	0.43	3.3	3.78
BST* 1	1	1	78	5	49.06	2.73	56.21
2	1	1	85	6	49.28	3.27	56.88
3	1	1	77	5	47.92	3.13	55.17
4	1	33	13312	11136	42.72	591.28	637.67
sketch	3	57	13401	11146	169.45	595.26	765.49
Red-Black Tree* 1	1	1	156	5	33.16	2.03	35.85
2	1	1	154	6	32.3	2.07	35.19
3	1	14	232	45	30.23	3.92	35.14
4	1	34	362	95	32.52	8.67	41.95
5	1	31	328	87	29.92	7.1	38.32
6	1	17	219	51	30.58	2.72	33.68
sketch	4	108	673	232	45.59	16.53	62.58

complex specification and more method predicates, while longer synthesis times correspond to a larger space of candidate patches. As expected, repairing the sketch of each benchmark takes the most time, as they are missing the most coverage. The BST sketch, for example, requires Cobb to explore one of the largest search spaces of all our experiments, with the final repair synthesizing a pair of recursive calls with non-trivial arguments. On the other end of the spectrum, the target coverage type of the red-black tree benchmarks enforces several non-trivial invariants, resulting in

Table 2. Results for the Simply Typed Lambda Calculus case study. The columns are the same as Table 1.

Benchmark	#Holes	Repair Size	#Queries	#Terms	Abduction (s)	Synthesis (s)	Total Time(s)
STLC 1	1	9	99	9	1.81	5.55	8.29
2	1	30	694	276	0.76	83.76	86.34
3	1	59	4460	1555	0.61	610.18	611.49

some of the longest abduction times. Most of the remainder of the total time is spent type checking the completed generator; these times are consistent with those reported by Zhou et al. [70].

Case Study: Simply-Typed Lambda Calculus Terms. As a final experiment, we also investigated Cobb’s performance on a more challenging problem: repairing a generator for well-typed simply typed lambda calculus (STLC) terms [29, 51]. On its own, the reference generator is already quite complex, featuring multiple inductive datatypes and auxiliary functions [70]. The specifications of the generator and these auxiliary functions are similarly intricate, requiring 15 method predicates. We developed three variants of the reference generator using the same methodology as our previous set of experiments. We additionally bound the space of candidate repairs in each experiment, by manually limiting the set of components used by Cobb to those occurring in the expression that was deleted from the reference implementation. Table 2 shows the results of this experiment.

Despite the challenges inherent in this benchmark, Cobb was able to find complete repairs for all three STLC variants, with the two simpler variants each taking less than two and a half minutes to repair. The final variant required a more substantial repair that involved multiple recursive calls and sophisticated reasoning, e.g., the patch must randomly divide the maximum number of applications allowed in recursively generated subterms. While searching for this patch, Cobb enumerates more than 1500 terms and issues almost 4500 SMT queries. While Cobb is able to successfully find this patch, it takes almost 10 minutes to do so, with the bulk of the time being spent querying Z3. We suspect that Cobb will need further optimizations to scale up to larger sets of components and to mitigate the inherent complexity of the verification conditions handed off to Z3.

Discussion. Taken together, these two sets of experiments provide evidence that Cobb’s runtime scales reasonably well with the complexity of both the repair and synthesis tasks, suggesting the potential of our approach in future applications that depend on generating data that meets some desired property. Importantly, the cost of performing a repair is paid once: a repaired generator can be run normally, without any need to invoke an SMT solver.

6.2 Comparison with Alternative Repair Strategies (RQ3)

At a high-level, Cobb balances two competing concerns when searching for a patch, trying to find a repair that limits the number of ‘useless’ inputs that fail to meet the target precondition, while simultaneously ensuring it does not omit any ‘interesting’ values that do. This set of experiments compares Cobb to alternative strategies that exclusively prioritize one of these concerns (**RQ3**).

Completeness-Focused Repair. Our first set of experiments compares Cobb against an approach that only prioritizes completeness when searching for repairs. This admits an easy implementation: we simply fill in each hole inserted by `Localize` with the default generator for the base type of the hole, e.g., `genTree` or `int_gen`. This results in generators that are at least as complete as those found by Cobb, at the cost of potentially producing more ’useless’ inputs. Thus, to compare the two strategies, we track how many values a repaired generator produces that violate the target precondition. We use each generator to produce 20k values, recording how many of these outputs satisfy the target precondition. Following prior work, we constrain the size parameter used in each experiment, adopting similar bounds to those works [29, 32, 63]; the discussion in Section 6.5

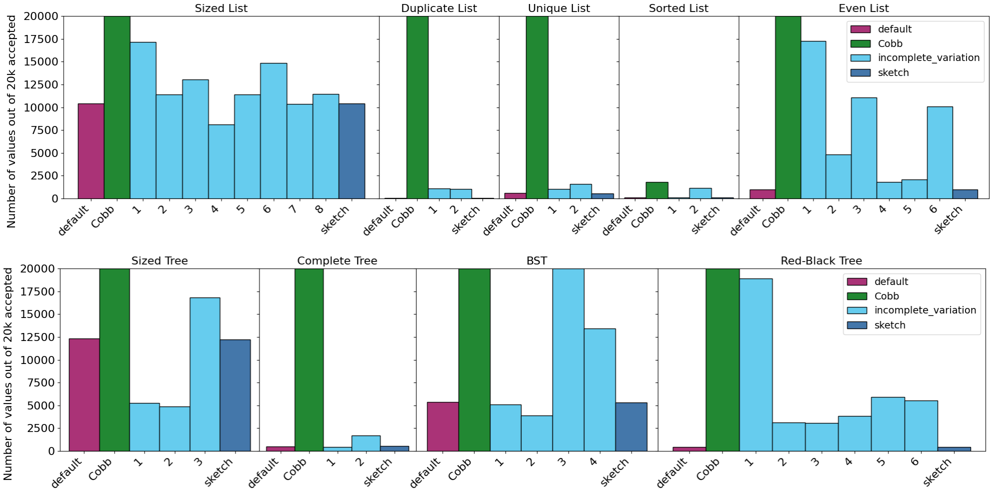

Fig. 11. Number of values satisfying the target precondition produced by the default, Cobb-repaired, and completeness-focused-repaired generators for the sketches in Table 1.

provides more details on the bounds used. These experiments also address the feasibility of directly using a default generator to compensate for a generator's missing coverage.

Fig. 11 presents the results of this experiment for each of the list and tree benchmarks from Table 1, respectively. Each graph also reports the number of valid outputs produced by a default generator; this serves as a rough proxy for the restrictiveness of the target precondition. From left to right, each group of columns in the figures report the number of valid inputs produced by the default generator (purple), the generator repaired by Cobb (green),[9] and the repaired version of each variant in Table 1 produced by a completeness-focused repair strategy (cyan), ending with the repaired sketch (blue). Unsurprisingly, the last variant performs comparably to the default generator: applying the completeness-focused repair strategy to `genEven`$_{\text{inc}}$, for example, results in a function that is effectively equivalent to the default **`int list`** generator.

While the generators produced by Cobb are consistently more likely to produce valid values than their completeness-focused counterparts, Fig. 11 shows that the latter strategy can be effective in certain cases, especially when pitted against the default generator. As expected, one such scenario is when the target property is relatively permissive, as is the case for our sized list and sized tree benchmarks. These generators only need to produce a value within the expected size; as the default generators show, roughly half of all randomly generated values satisfy this property.

Conversely, when the target specification is more restrictive, the alternative repair strategy is less effective. The complete tree benchmark falls into this category, as the subtrees of a randomly generated tree are unlikely to have a uniform depth. Similarly, the target preconditions used by our unique and duplicate list benchmarks are considerably tighter than that of the sized list benchmark: both require a list containing *exactly* `size` elements. In both cases, the coverage provided by the repaired generators produced by the alternative strategy is mostly limited to when the size parameter is small, although uniqueness of list elements is a slightly more forgiving property.

A similar phenomenon occurs in the BST and red-black tree benchmarks, albeit in a more nuanced way. Both of these benchmarks feature semantically rich specifications, while still being somewhat more permissive than the previous three examples. Notably, the completness-focused

[9] Fig. 11 uses the generator produced from the sketch for the generator produced by Cobb. Since the Cobb-repaired generators are semantically equivalent, so are their results on these benchmarks.

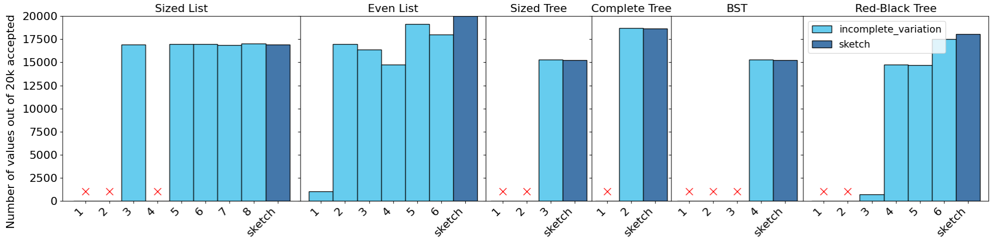

Fig. 12. Number of tests produced by Cobb-repaired generators from Table 1 that fall outside the range of repaired versions that used a safety-focused strategy. The safety-repaired versions of the three omitted generators were coverage complete.

repair strategy is effective for some of these benchmarks, in particular the third BST variant and the first red-black tree variant. For these two examples, the required repairs fall into execution paths which are exercised very rarely, so the default generator used in the repair is not given many opportunities to inject an invalid value into the output of the repaired generator. In the case of the third BST variant, for example, the repair is inserted into a branch in which bounds force the BST to be empty, i.e., lo + 1 = hi, a scenario that depends on a very particular sequence of nondeterministic choices. In the case of the red-black tree, the patch is similarly only executed when the generator is called with very specific values, namely when the input black height is precisely zero and the color argument is black. These sorts of corner cases are sometimes explicitly listed in handwritten generators in order to improve the likelihood they will occur; identifying and prioritizing these sorts of corner cases in repairs is an interesting direction for future work.

On most of these benchmarks, the generators repaired by Cobb almost always produce valid inputs, with the sorted list generator being the notable exception. This generator is unique among our benchmarks as it is the only benchmark in which the reference generator includes an err expression that is hit with some frequency. While errors are fine from the perspective of our coverage type system— the right sequence of nondeterministic choices always allows the generator to avoid them— because the original sorted list generator does not implement any kind of backtracking [29], these errors impact the number of sorted lists the repaired generator produces. As a result, the repaired generators in this experiment only have a reasonable probability of yielding a valid output for smaller lists.

Safety-Focused Repair. As the previous experiment showed, a repair strategy that only prioritizes completeness is likely to produce generators that output *useless*, i.e., invalid values. This set of experiments investigates whether a repair strategy that only considers safety will yield generators that are likely to omit *interesting*, i.e., valid values. Before discussing our results, observe that it is not obvious how to measure the incompleteness of a generator: even an incomplete generator can produce an infinite number of valid inputs. Our strategy is to instead compare the relative completeness of two generators, in this case by quantifying the number of outputs produced by a Cobb-repaired generator that could never be produced by a one repaired using a safety-focused strategy. In detail, we first ascribe a standard refinement type to the return type of each safe generator: intuitively, the qualifier of this type overapproximates the range of the generator. Negating this qualifier characterizes the set of outputs that fall outside the range of a safety-repaired generator; any valid outputs of a Cobb-repaired generator that satisfy this negated property could never have been produced by its safe counterpart.

As an example, one way to repair genIntList from Fig. 7a so that it is safe with respect to our target precondition is:

```
  let rec genIntList_safe (n : int) : int list = if n == 0 then [ ] else [ ]
```

We can ascribe the following refinement type to $\mathtt{genIntList}_{\mathtt{safe}}$, capturing the fact that it always returns the empty list:

$$\{\mathtt{n}:\mathtt{int} \mid \mathtt{n} \geq 0\} \rightarrow \{\mathtt{l}:\mathtt{int\ list} \mid \mathtt{len}(\mathtt{l}) = 0\}$$

Thus, any value of the type $\{\mathtt{l}:\mathtt{int\ list} \mid \mathtt{len}(\mathtt{l}) \neq 0\}$ must fall outside the range of $\mathtt{genIntList}_{\mathtt{safe}}$.

To carry out our experiment, we applied a safety-focused repair strategy that replaces Synthesize with the repair that an off-the-shelf, safety-guided synthesizer [54] would generate for each hole. The hole corresponding to the base case of the sized list benchmark yields the following synthesis goal, for example:

$$\mathtt{n}:\{\mathtt{n}:\mathtt{int} \mid \mathtt{n} = 0\} \vdash \square_1 \ :\ \{\mathtt{l}:\mathtt{int\ list} \mid \mathtt{len}(\mathtt{l}) \leq \mathtt{n}\}$$

To approximate the relative completeness of Cobb versus a safety-focused repair approach, we first examined each safety-repaired generator and came up with a standard refinement type for its outputs. We then generated 20k values from the Cobb-repaired generator and used the qualifier of this type to identify how many of these outputs fall outside the range of the safety-repaired generator. Fig. 12 present the results of this experiment for each of the list and tree variant from Table 1. The taller the bar, the more (relatively) complete the Cobb-repaired generator.

On these benchmarks, the completeness of the safety-focused repair strategy tends to be an all-or-nothing proposition: in the case of the sized list benchmark, for example, the safety-focused strategy's prioritization of minimal terms yields generators that always return a `[]` term for six of the nine variants; this is precisely the repair needed by three of these, however. In contrast, the safety-focused strategy tends to be more effective when the set of valid repairs are strongly constrained by the arguments of a generator: the length of the output lists in the unique and duplicate benchmarks is completely determined by its size parameter, for example. The safe strategy is also effective when the required repair is a constant: the required repair in the first two variants of depth tree benchmark is a single **`Leaf`** constructor, for example.

In contrast, the safe repair strategy tends to perform poorly when the target precondition admits a number of safe repairs that are relatively cheap: a cost function that employs Occam's razor, for example, prioritizes small repairs that use variables or constants. Even when the target property is quite restrictive, a safety-focused strategy is biased towards repairs that produce values of the right "shape", but whose contents do not vary much. When repairing the first hole of $\mathtt{genEvens}_{\mathtt{sk}}$, for example, this strategy is biased towards patches like `[0]`; this term is unlikely to explore code paths in the function under test that depend on the contents of its list. Crucially, prioritizing completeness is not simply a matter of equipping a safety-focused generator with a different *syntactic* cost function: the patch for the second hole of $\mathtt{genEvens}_{\mathtt{sk}}$ required synthesizing and joining two terms that were both individually safe; the complete repair for the red-black tree sketch similarly joins together multiple safe terms. Finding the right combinations of terms to join requires *semantic* characterizations of both the missing coverage *and* the coverage provided by a candidate patch.

Discussion. An important takeaway from both of these experiments is that while completeness- and safety-focused repair strategies can yield useful repairs in certain situations, their efficacy is highly dependent on the particular problem, and there are many scenarios in which neither approach performs well. When the target property is weak, a completeness-focused repair can improve on the default generator, while a safety-focused strategy tends to work well when the specification is very stringent. Neither approach tends to do well when the property falls somewhere between these two extremes, e.g., our BST and red-black tree examples. Prioritizing the minimal coverage-complete repair enables Cobb to produce repaired functions that generate useful inputs without omitting any interesting values.

Table 3. Results of adding superfluous components to synthesis benchmarks from Table 1. The columns show: #Components (total, removed), #Queries (total, % change), #Terms (total, % change), and Synthesis Time (seconds, % change).

Benchmark	#Components (removed)	#Queries (% change)	#Terms (% change)	Time (sec, % change)
Sorted List	3 (-5)	154 (-48.3%)	77 (-66.5%)	2.44 (-45.8%)
Unique List	2 (-6)	71 (+0.0%)	14 (-26.3%)	1.12 (+4.7%)
Sized List	4 (-4)	99 (-22.0%)	13 (-48.0%)	1.73 (-12.2%)
Duplicate List	3 (-5)	96 (-20.7%)	22 (-69.0%)	5.05 (+14.5%)
Even List	5 (-3)	301 (-15.2%)	43 (-53.8%)	7.76 (-12.5%)
Complete Tree	4 (-3)	95 (-57.8%)	17 (-64.6%)	1.52 (-53.1%)
BST	3 (-4)	916 (-93.2%)	597 (-94.6%)	79.93 (-86.8%)
Sized Tree	4 (-3)	182 (-3.2%)	17 (-15.0%)	5.18 (-23.9%)

6.3 Sensitivity to Set of Components (RQ4)

As with any component-based synthesizer [2, 36, 58], the ability of Cobb to effectively find a solution and the quality of that solution depends on the set of components available to it. This set of experiments investigates the sensitivity of Cobb to the set of components it uses.

Extraneous Components. All of the benchmarks in Section 6.1 used a common set of components based on the base type of the target generator, so all of the list and tree benchmarks included strictly more components than were needed to make the coverage complete. In order to measure how this inclusion of extra components impacted Cobb's ability to find a solution, this experiment examines how much overhead such components added to the synthesis time over a set that contains exactly the components needed to precisely repair a sketch.

Table 3 presents the results of these experiments. For most of these experiments, the extra components imposed relatively modest overhead to the performance of Cobb that is roughly commensurate with the number of components added.[10] There are a couple of reasons for this: Cobb adopts the standard technique of using types to filter out any ill-typed terms [16, 36, 49] and prioritizes lower-cost terms, limiting the number of additional terms the extra components cause Cobb to enumerate before it finds a precise solution. In addition, the bulk of the SMT queries in `Synthesize` are limited to terms in the candidate pool, which does not include many of the newly added terms. The one exception is our BST benchmark, where the extraneous components lead to considerably more enumerated terms and SMT queries. Here, the full set of components generates several integer terms with similar costs and slightly different coverages. Intuitively, because the solution uses one of these components, Cobb has to perform more comparisons to identify which of these variants to include in the final patch. In addition, the recursive call of `genBST` requires multiple arguments with the same base types as the extraneous components, resulting in several additional terms comprised of applications of the recursive call to different arguments.

Insufficient Components. Our second set of experiments investigates the quality of the solution found by Cobb when it cannot construct a precise repair, and effectively probe the ability of Cobb's "best effort" extraction mechanism to improve upon the completeness-focused repair strategy from the previous experiment. For this investigation, we examined each of the 44 benchmarks from Table 1 and then removed a key component used by the original (complete) generator.

On almost half of the deliberately sabotaged versions of these benchmarks (21/44) , Cobb was still able to produce a different result than the completeness-focused repair. For many of these, though, the practical difference with the default repair was not that significant: without access to the –

[10] The slight *improvement* in the synthesis time of two of our smallest benchmarks can be attributed to the sensitivity of the underlying SMT solver to perturbations in input queries [4].

```
let rec even_list_gen s =
 if sizecheck s then
-  (int_gen() * 2) :: []
+  int_gen() :: []
 else if bool_gen() then
-  (int_gen() * 2) :: []
+  int_gen() :: []
 else
-  (int_gen() * 2) :: even_list_gen (s-1)
+  int_gen() :: even_list_gen (s-1)
```

(a) Repair for Even List sketch with no * component

```
let rec rbtree_gen inv color h =
 if sizecheck h then
  if color then Rbtleaf
  else if bool_gen () then
-   Rbtleaf
+   rbtree_gen h true h
   else
     Rbtnode (true, Rbtleaf, int_gen (), Rbtleaf)
   (* ... rest omitted for space ... *)
```

(b) Repair for Red-Black Tree 3 with no Rbtleaf component

Fig. 13. Diffs of a generator with a perfect repair and the variant Cobb finds when it lacks a component needed by that repair.

component in the duplicate list benchmark, for example, Cobb generates the repair `x :: list_gen()` instead of `x :: duplicate_list_gen (s-1) x`; the former is only a marginal improvement over `list_gen()`. On the other hand, Cobb was able to produce more meaningfully useful repairs for the two benchmarks shown in Fig. 13. If it is not supplied with the multiplication operator, Cobb builds the repair shown in Fig. 13a for the sketch of the even list generator — while not perfect, this generator at least produces lists with the required shape. Fig. 13b presents the repair found by Cobb for a red-black tree variant that has a hole in one of its base cases when it is not provided the leaf constructor. Here, Cobb somewhat cleverly makes a recursive call using `h` — which is guaranteed to be less than its recursive argument `inv` at this hole — relying on the coverage provided by the other base cases to repair the hole. As these two examples demonstrate, Cobb is capable of finding non-trivial repairs even when it is not able to construct a precise solution, but the utility of those repairs also depends on the set of components available to it.

Discussion. These experiments show that, as with other bottom-up synthesis techniques, the set of components available to Cobb impacts both its performance when constructing a repair and the quality of the solution it finds— hence the restricted set of components in the STLC case study in Section 6.1. On the one hand, the results of the first experiment indicate that while Cobb's current strategy for prioritizing and filtering enumerated terms makes it reasonably robust to extraneous components, it is still more effective when equipped with a more targeted set of components. On the other hand, the second experiment demonstrates that Cobb's ability to construct useful repairs when it cannot find a precise solution also depends on the components it is provided. Taken together, these experiments suggest that incorporating more advanced techniques from the program synthesis community [6, 8, 35, 36, 48] for curating the set of components used by Cobb is an important future direction.

6.4 Comparison with Dynamic Test Input Generation (RQ5)

The ultimate goal of Cobb is to use symbolic reasoning to statically ensure that the set of values enumerated by a generator aligns with the complete set of values that meet the precondition of a function under test. One alternative approach is to instead use a theorem prover to dynamically generate these values during testing [29, 63]. To evaluate these alternative styles of test generation, this experiment compares Cobb with Luck [29], a tool which queries a constraint solver to produce values that satisfy a user-defined predicate written in a DSL. Fig. 14 reports the time needed for Luck to generate 1k and 10k red-black trees, respectively, against the time needed for Cobb to generate the same number of trees. As a baseline, the figure also reports the time needed by to

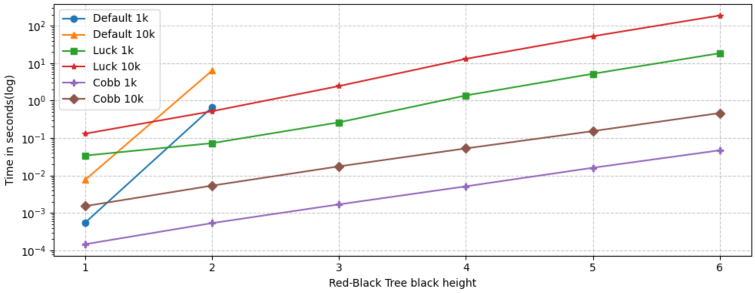

Fig. 14. Time needed to generate 1k and 10k valid red-black trees using Cobb-repaired generators, Luck [29], and a default 'randomly generate and filter' approach.

produce 1k and 10k valid red-black trees by running the default generator in 'generate and filter' loop. We use an upper time limit of 5 minutes for all the experiments; only the baseline approach exceeds this bound.

While not a perfect apples-to-apples comparison— among other things,[11] these generators are implemented in different languages and frameworks— these experiments confirm the conclusions of Lampropoulos et al. [29] that run-time constraint checking imposes (at least) an order of magnitude amount of overhead over a generator that does not solve constraints at runtime. One takeaway from this experiment is that the overhead of dynamic constraint solving quickly matches the overhead required by our static repair approach– it takes Cobb roughly a minute to repair the red-black tree sketch, which is roughly the amount of time needed to generate 10k red-black trees with a black height of at least 5. Given that a repaired generator can be run without any additional constraint solving, the overhead required by the static repair approach seems reasonable for settings in which generators are repeatedly used, e.g., when using PBT in a CI setting [20].

6.5 Discussion and Limitations

While our sized generators are complete for an arbitrary size bound, the experiments in Sections 6.2 and 6.4 use a more limited range of bounds when generating values, as is common in the literature. All of our list benchmarks use QCheck's built-in generator for natural numbers, `nat_gen()`. This generator produces integers between 0 and 10000, and its distribution of outputs is skewed towards smaller values. The bound in our tree benchmarks limits the height of the tree, which bounds the number of nodes in a tree at $O(2^{n+1} - 1)$. Simply using `nat_gen()` for these benchmarks can generate very large trees, so these benchmarks instead use a height between 0 and 12, chosen at random; this range is also used in prior works [29].

As mentioned in Section 5, Cobb only guarantees that a value can be generated with non-zero probability, and says nothing more about the likelihood that it will be generated. As a consequence, Cobb does not ensure that a repaired generator is fair, i.e., that every value can be produced with uniform probability. In our experiments, the distribution of a repaired generator's outputs largely depends on the structure of the original generator. Reasoning about the fairness of repaired generators and repairing unfair generators is an interesting direction for future work.

[11] Unlike Luck, Cobb-repaired generators are not guaranteed to produce unique values, although the latter's use of `int_gen()` to produce (signed 63-bit) integers means they are statistically unlikely to produce duplicate trees.

7 Related Work

Generating Data Meeting Sparse Preconditions. A number of works have considered how to effectively generate data satisfying a sparse precondition. The proposed solutions can be roughly categorized as either *dynamic* and *static* approaches. *Dynamic approaches* attempt to directly ensure the validity of inputs as they are being generated [10, 19, 29, 41, 63], typically by relying on run-time constraint solving. Like Cobb, Target [63] uses refinement types to define the space of valid inputs. To generate values, however, Target queries an SMT solver for a model satisfying the type qualifier, and then converts the model into a value in the target language. To generate additional values, the SMT query is updated to explicitly exclude any models that have already been found. Another particularly popular strategy is to directly leverage the definition of the target precondition, lazily concretizing the value being generated in a way that ensures the constraint is satisfied and backtracking when constraints become unsatisfiable [10, 19, 29, 41], similar to the idea of narrowing in logic programming languages. The completeness of dynamic approaches is typically tied to the completeness of the underlying constraint solver: as long as the solver can return any satisfying value, so can the input generator. The need to solve constraints at run time imposes considerable overhead however; as discussed in Section 6.4, dynamic approaches can be orders-of-magnitude slower than their static counterparts, particularly when the target property is complex.

Cobb instead adopts a *static approach.* Static generation techniques avoid run-time constraint solving, and instead seek to construct input generators that are sound and complete *by construction.* Closely related to Cobb is a line of work that automatically builds generators for the QuickChick PBT framework [32] by compiling inductively defined relations into efficient generators [31, 52]. This pipeline uses a translation validation approach [53] to ensure the correctness of the resulting generators, producing formal proofs of their soundness and completeness in the Coq/Rocq proof assistant. Unlike Cobb, which is agnostic to how the target property is defined, these works impose a strict requirement that the target precondition be defined as an inductive proposition in Coq/Rocq, although recent work has considered how this restriction can be somewhat relaxed by, e.g., composing different inductive relations into a single unified definition [56]. This restriction is used to produce generators that closely follow the definition of the proposition itself; Cobb, in contrast, is able to synthesize and repair arbitrary programs that supply the desired coverage.

Generating a Good Distribution of Data. An orthogonal problem to coverage is the question of the *distribution* of outputs produced by a generator: a generator for trees that produces `Leaf` nodes 90% of the time is less useful than one whose outputs are uniformly distributed, for example. A few tools have been proposed for statically reasoning about the distribution of a generator's outputs: one such example is Feat [14], is a library for writing enumerators of datatypes that are guaranteed to produce a uniform distribution of the values of an algebraic datatype. The Dragen tool [44, 45], in contrast, uses a mathematical model to statically estimate the distribution of constructors produced a QuickCheck generator built from its `frequency` combinator, and uses those estimates to adjust the arguments of `frequency` to achieve a more desirable distribution. Extending Cobb to account for the distribution of a generator is an interesting direction for future work.

Program Synthesis. Automatically deriving programs from logical specifications of their behavior has been a goal of the programming synthesis community for almost half a century [39]. The overwhelming majority of program synthesis techniques use specifications that overapproximate the set of desired behaviors [2, 13, 43, 55], including those that also use refinement types to specify program behaviors [24, 27, 54]. The specifications used by Cobb, in contrast, stipulate an underapproximation of the desired behaviors. This impacts Cobb's repair algorithm, which combines partial solutions which do not individually satisfy the target specification, to build a complete solution. The sets of input-output examples used by inductive synthesis, or programming-by-example (PBE), techniques [1, 3, 18, 35, 36, 42, 49, 64, 68] also underapproximate the target program's behavior, although they do so much less comprehensively than coverage types. While similar to the bottom-up term enumeration strategy employed in PBE systems, Cobb's `Synthesize` procedure is able to take advantage of the more complete approximation provided by coverage types to, e.g., recursively call the generator being repaired before its full definition is known [42, 68].

Automated Program Repair. The goal of automated program repair (APR) is to automatically patch a buggy program with minimal user effort [34]. Most APR approaches use test suites to identify buggy behaviors: a valid patch is one that causes a program that was failing some tests to instead pass its suite. A notable exception is the work of Logozzo and Ball [38], which defines a "good" repair as one that decreases the number of statically detected assertion failures in a program without introducing any new ones. A major challenge in APR is finding repairs that generalize beyond a particular failing test [65], a problem that Cobb avoids thanks to the strong correctness specifications provided by coverage types. Like Cobb, several APR techniques rely on program synthesis to generate candidate repairs. Nguyen et al. [47], for example, employ symbolic execution to identify path constraints that cause tests to succeed or fail. These constraints are used as a safety specification for the target repair, which is then generated using component-based synthesis. An alternative strategy is to use angelic execution [33, 40] to identify concrete values that can be used to help a program pass a failing test; finding a patch that generates these values is an instance of a PBE problem. As previously discussed, the program synthesis techniques employed by both strategies use specifications that are fundamentally different from Cobb's.

8 Conclusion

When using a property-based testing framework to automatically test a program that has a restrictive or sparse precondition, users are typically forced to manually write a function that effectively generates values of interest. Alongside the additional burden this imposes on users of PBT frameworks, this process is also error-prone, as generators can be both unsound, producing values that do not meet the target precondition, and incomplete, incapable of producing all the values that meet the precondition. This paper presents a technique for detecting and repairing incomplete test input generators, leveraging coverage types to characterize the set of missing test values and the coverage provided by candidate repairs. Our repair technique uses a novel coverage-type guided enumerative synthesis algorithm to generate candidate repairs, employing a lattice structure to store partial solutions so that they can be efficiently queried and combined to build a complete repair. We have implemented a repair tool for OCaml input generators, called Cobb, and have used it to repair a diverse suite of benchmarks drawn from the PBT literature. Our experiments demonstrate that Cobb can also be effective as a sketch-based synthesis tool for test input generators, suggesting its potential for further reducing a possible point of friction for users of PBT frameworks.

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044* (Saint Petersburg, Russia) *(CAV 2013)*. Springer-Verlag, Berlin, Heidelberg, 934–950.

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. doi:10.1109/FMCAD.2013.6679385

[3] Rajeev Alur, Pavol Černý, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 163–179.

[4] Daneshvar Amrollahi, Mathias Preiner, Aina Niemetz, Andrew Reynolds, Moses Charikar, Cesare Tinelli, and Clark Barrett. 2025. Towards SMT Solver Stability via Input Normalization. arXiv:2410.22419 [cs.LO] https://arxiv.org/abs/2410.22419

[5] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 227 (Nov. 2020), 29 pages. doi:10.1145/3428295

[6] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 227 (Nov. 2020), 29 pages. doi:10.1145/3428295

[7] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 836–850. doi:10.1145/3477132.3483540

[8] José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proc. ACM Program. Lang.* 7, POPL, Article 33 (Jan. 2023), 30 pages. doi:10.1145/3571226

[9] Koen Claessen. 2020. *QuickCheck*. https://hackage.haskell.org/package/QuickCheck

[10] Koen Claessen, Jonas Duregård, and Michał H. Pałka. 2014. Generating Constrained Random Data with Uniform Distribution. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 18–34.

[11] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. doi:10.1145/351240.351266

[12] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. doi:10.1007/978-3-540-78800-3_24

[13] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 689–700. doi:10.1145/2676726.2677006

[14] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: functional enumeration of algebraic types. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 61–72. doi:10.1145/2364506.2364515

[15] FastCheck 2022. *fast-check: Property based testing for JavaScript and TypeScript*. https://dubzzz.github.io/fast-check.github.com/

[16] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 229–239. doi:10.1145/2737924.2737977

[17] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 237–247. doi:10.1145/155090.155113

[18] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed synthesis: a type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 802–815. doi:10.1145/2837614.2837629

[19] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 225–234. doi:10.1145/1806799.1806835

[20] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the 46th ACM/IEEE International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA.

[21] Harrison Goldstein, Jeffrey Tao, Zac Hatfield-Dodds, Benjamin C. Pierce, and Andrew Head. 2024. Tyche: Making Sense of PBT Effectiveness. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*

(Pittsburgh, PA, USA) *(UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 10, 16 pages. doi:10.1145/3654777.3676407

[22] Sumit Gulwani. 2010. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming* (Hagenberg, Austria) *(PPDP '10)*. Association for Computing Machinery, New York, NY, USA, 13–24. doi:10.1145/1836089.1836091

[23] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 317–330. doi:10.1145/1926385.1926423

[24] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.* 4, POPL, Article 12 (Dec. 2019), 28 pages. doi:10.1145/3371080

[25] John Hatcliff and Olivier Danvy. 1994. A Generic Account of Continuation-Passing Styles. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 458–471. doi:10.1145/174675.178053

[26] Hypothesis 2022. *Hypothesis*. https://github.com/HypothesisWorks/hypothesis/tree/master/hypothesis-python

[27] Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. 2020. Digging for Fold: Synthesis-Aided API Discovery for Haskell. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 205 (nov 2020), 27 pages. doi:10.1145/3428273

[28] Ranjit Jhala and Niki Vazou. 2021. Refinement Types: A Tutorial. *Found. Trends Program. Lang.* 6, 3-4 (2021), 159–317. doi:10.1561/2500000032

[29] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 114–129. doi:10.1145/3009837.3009868

[30] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage guided, property based testing. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 181 (Oct. 2019), 29 pages. doi:10.1145/3360607

[31] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2018. Generating Good Generators for Inductive Relations. *Proc. ACM Program. Lang.* 2, POPL (2018), 45:1–45:30. doi:10.1145/3158133

[32] Leonidas Lampropoulos and Benjamin C. Pierce. 2022. *QuickChick: Property-Based Testing in Coq*. Software Foundations, Vol. 4. Electronic textbook. Version 1.3.1, https://softwarefoundations.cis.upenn.edu.

[33] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 593–604. doi:10.1145/3106237.3106309

[34] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (Nov. 2019), 56–65. doi:10.1145/3318162

[35] Sihyung Lee, Seung Yeob Nam, and Jiyeon Kim. 2022. Program Synthesis Through Learning the Input-Output Behavior of Commands. *IEEE Access* 10 (2022), 63508–63521. doi:10.1109/ACCESS.2022.3183091

[36] Woosuk Lee and Hangyeol Cho. 2023. Inductive Synthesis of Structurally Recursive Functional Programs from Non-recursive Expressions. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 2048–2078. doi:10.1145/3571263

[37] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, KC Sivaramakrishnan, and Jérôme Vouillon. 2024. *The OCaml system release 5.2: Documentation and user's manual.* Ph. D. Dissertation. Inria.

[38] Francesco Logozzo and Thomas Ball. 2012. Modular and verified automatic program repair. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 133–146. doi:10.1145/2384616.2384626

[39] Z. Manna and R. Waldinger. 1979. Synthesis: Dreams => Programs. *IEEE Trans. Softw. Eng.* 5, 4 (July 1979), 294–328. doi:10.1109/TSE.1979.234198

[40] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. doi:10.1145/2884781.2884807

[41] Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. 2007. Korat: A Tool for Generating Structurally Complex Test Inputs. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, USA, 771–774. doi:10.1109/ICSE.2007.48

[42] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up synthesis of recursive functional programs using angelic execution. *Proc. ACM Program. Lang.* 6, POPL, Article 21 (Jan. 2022), 29 pages. doi:10.1145/3498682

[43] Ashish Mishra and Suresh Jagannathan. 2022. Specification-guided component-based synthesis from effectful libraries. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 147 (Oct. 2022), 30 pages. doi:10.1145/3563310

[44] Agustín Mista and Alejandro Russo. 2019. Generating random structurally rich algebraic data type values. In *Proceedings of the 14th International Workshop on Automation of Software Test* (Montreal, Quebec, Canada) *(AST '19)*. IEEE Press, 48–54. doi:10.1109/AST.2019.00013

[45] Agustín Mista, Alejandro Russo, and John Hughes. 2018. Branching processes for QuickCheck generators. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3242744.3242747

[46] Wojciech Mostowski, Thomas Arts, and John Hughes. 2017. Modelling of Autosar Libraries for Large Scale Testing. In *Proceedings 2nd Workshop on Models for Formal Analysis of Real Systems, MARS@ETAPS 2017, Uppsala, Sweden, 29th April 2017 (EPTCS, Vol. 244)*, Holger Hermanns and Peter Höfner (Eds.). 184–199. doi:10.4204/EPTCS.244.7

[47] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 772–781.

[48] Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. 2021. {BUSTLE}: Bottom-Up Program Synthesis Through Learning-Guided Exploration. In *International Conference on Learning Representations*. https://openreview.net/forum?id=yHeg4PbFHh

[49] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. doi:10.1145/2737924.2738007

[50] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, New York, NY, USA, 329–340. doi:10.1145/3293882.3330576

[51] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test* (Waikiki, Honolulu, HI, USA) *(AST '11)*. Association for Computing Machinery, New York, NY, USA, 91–97. doi:10.1145/1982595.1982615

[52] Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. 2022. Computing correctly with inductive relations. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 966–980. doi:10.1145/3519939.3523707

[53] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '98)*. Springer-Verlag, Berlin, Heidelberg, 151–166.

[54] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. doi:10.1145/2908080.2908093

[55] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 72 (Jan. 2019), 30 pages. doi:10.1145/3290385

[56] Jacob Prinz and Leonidas Lampropoulos. 2023. Merging Inductive Relations. *Proc. ACM Program. Lang.* 7, PLDI, Article 178 (June 2023), 20 pages. doi:10.1145/3591292

[57] QCheck 2024. *QCheck*. https://c-cube.github.io/qcheck/

[58] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-modal program inference: a marriage of pre-trained language models and component-based synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 158 (Oct. 2021), 29 pages. doi:10.1145/3485535

[59] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1410–1421. doi:10.1145/3377811.3380399

[60] REMS 2020. Rigorous Engineering of Mainstream Systems. https://www.cl.cam.ac.uk/~pes20/rems/index_introduction.html. https://www.cl.cam.ac.uk/~pes20/rems/index_introduction.html

[61] RustCheck 2021. *Crate for PBT in Rust*. https://github.com/BurntSushi/quickcheck

[62] ScalaCheck 2021. *ScalaCheck*. https://scalacheck.org/

[63] Eric L. Seidel, Niki Vazou, and Ranjit Jhala. 2015. Type Targeted Testing. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*. Springer-Verlag, Berlin, Heidelberg, 812–836. doi:10.1007/978-3-662-46669-8_33

[64] Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: Component-Based Synthesis with Control Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 73 (jan 2019), 29 pages. doi:10.1145/3290386

[65] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 532–543. doi:10.1145/2786805.2786825

[66] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. University of California at Berkeley, USA.

[67] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of data completion scripts using finite tree automata. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 62 (Oct. 2017), 26 pages. doi:10.1145/3133886

[68] Yongwei Yuan, Arjun Radhakrishna, and Roopsha Samanta. 2023. Trace-Guided Inductive Synthesis of Recursive Functional Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 141 (June 2023), 24 pages. doi:10.1145/3591255

[69] Zhe Zhou, Robert Dickerson, Benjamin Delaware, and Suresh Jagannathan. 2021. Data-driven abductive inference of library specifications. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 116 (Oct. 2021), 29 pages. doi:10.1145/3485493

[70] Zhe Zhou, Ashish Mishra, Benjamin Delaware, and Suresh Jagannathan. 2023. Covering All the Bases: Type-Based Verification of Test Input Generators. *Proc. ACM Program. Lang.* 7, PLDI, Article 157 (jun 2023), 24 pages. doi:10.1145/3591271