Synthesizing Data Structure Refinements from Integrity Constraints

Shankara Pailoor
University of Texas at Austin
Austin, Texas, USA
spailoor@cs.utexas.edu

Yuepeng Wang
University of Pennsylvania
Philadelphia, Pennsylvania, USA
yuepeng@seas.upenn.edu

Xinyu Wang
University of Michigan, Ann Arbor
Ann Arbor, Michigan, USA
xwangsd@umich.edu

Isil Dillig
University of Texas at Austin
Austin, Texas, USA
idillig@cs.utexas.edu

Abstract

Implementations of many data structures use several correlated fields to improve their performance; however, inconsistencies between these fields can be a source of serious program errors. To address this problem, we propose a new technique for automatically refining data structures from integrity constraints. In particular, consider a data structure D with fields F and methods M, as well as a new set of auxiliary fields F' that should be added to D. Given this input and an integrity constraint Φ relating F and F', our method automatically generates a refinement of D that satisfies the provided integrity constraint. Our method is based on a *modular* instantiation of the CEGIS paradigm and uses a novel inductive synthesizer that augments top-down search with three key ideas. First, it computes *necessary preconditions* of partial programs to dramatically prune its search space. Second, it augments the grammar with promising new productions by leveraging the computed preconditions. Third, it guides top-down search using a *probabilistic* context-free grammar obtained by statically analyzing the integrity checking function and the original code base. We evaluated our method on 25 data structures from popular Java projects and show that our method can successfully refine 23 of them. We also compare our method against two state-of-the-art synthesis tools and perform an ablation study to justify our design choices. Our evaluation shows that (1) our method is successful at refining many data structure implementations in the wild, (2) it advances the state-of-the-art in synthesis, and (3) our proposed ideas are crucial for making this technique practical.

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

Keywords: Programming Languages, Program Synthesis, Data structure refinement

ACM Reference Format:
Shankara Pailoor, Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2021. Synthesizing Data Structure Refinements from Integrity Constraints. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453483.3454063

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

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454063

1 Introduction

It is well known that implementations of most data types use auxiliary fields to improve performance. For example, a program that needs to perform frequent bidirectional look-ups may explicitly store two different mappings $M : \tau_1 \rightarrow \tau_2$ and $M^{-1} : \tau_2 \rightarrow \tau_1$ even though one can be derived from the other. Similarly, programmers may tore the same information as both a linked list and a hash map to efficiently implement different types of functionality. As a final example, Figure 1 shows a class called `SpdySession` from the Netty project [5] that maintains two auxiliary fields called `als` and `ars`. Even though these fields can be derived from the core data structure called `actStream`, doing so would hurt the application's performance.

While it is quite common to maintain such auxiliary states for performance reasons, correctly maintaining all related copies of the same data can be a source of bugs and security vulnerabilities. For instance, several prior efforts study violations of consistency requirements in data structures and propose techniques for mitigating them [12–15, 27, 28]. In addition, The Common Vulnerabilities and Exposures (CVE) database reports several security vulnerabilities that

are caused by violations of integrity constraints between their fields [1–4].

Motivated by this problem, this paper proposes a new synthesis-based approach for automatically refining implementations of data types given integrity constraints between their fields. In particular, given a data structure D consisting of fields F and methods M as well as a set of new fields F' to be added to D, our approach generates a new data structure D' with fields $F \cup F'$ and a set of updated methods M' such that (1) D' is functionally equivalent to D, and (2) the implementation of D' satisfies a user-provided *integrity constraint* $\Phi(F, F')$ that expresses consistency requirements between these fields. Specifically, our method takes as input a Java class D and an integrity constraint (expressed in Java as a consistency checking function) and produces a new class D' that is a refinement of D. Thus, programmers can first focus on implementing functionally correct software and then use our approach to automatically refine their implementation in a way that ensures consistency.

In order to solve this problem, our approach needs to synthesize code that correctly updates the new fields such that the integrity constraints are satisfied. However, this synthesis task is quite challenging for several reasons: First, since these new fields may be complex user-defined data structures, the update logic is often non-trivial and requires synthesizing arbitrary Java code, including loops or introduction of new program variables. Second, data structures often contain multiple methods that update the relevant fields; thus, the synthesis problem is not localized to a single function. Finally, the update logic may require invoking other functions from libraries or the surrounding code base; hence, the synthesizer needs to consider an enormous number of building blocks from which to generate code.

In this paper, we address these challenges using a *modular* instantiation of the counterexample-guided inductive synthesis (CEGIS) paradigm powered by a novel inductive synthesis engine. In particular, our method is modular in the sense that it decomposes the data structure refinement problem into independent synthesis subproblems over a single procedure. For each subproblem, we use an inductive synthesizer based on top-down enumerative search to find code snippets that satisfy a given set of (counter-)examples and then check correctness of the synthesized code using a verifier. However, to make inductive synthesis tractable in this setting, we leverage the following three key ideas:

1. **PCFG construction via static analysis:** Our method performs static analysis of the integrity checking function and the surrounding code base to identify important syntactic and semantic clues that are useful for guiding the underlying search procedure. In particular, our method leverages the results of static analysis to construct a probabilistic context free grammar (PCFG) and uses this PCFG to prioritize promising candidates during the search.
2. **Pruning via necessary preconditions:** Our approach dramatically prunes its search space using the notion of *necessary preconditions*. In particular, given a *partial* program P with missing statements or expressions, our method computes a formula ϕ that *must* be satisfied by every completion of P. Thus, if this formula is falsified by any of the counterexamples, our method can prune the partial program without sacrificing completeness.
3. **Deduction-guided grammar augmentation:** In addition to pruning the search space, necessary preconditions are also useful for identifying promising code snippets that are likely to appear in the desired solution. In particular, our synthesis procedure augments the grammar with new productions so that promising code snippets need not be synthesized from scratch. Our method also updates probabilities in the PCFG in order to prioritize programs that use these promising code snippets.

Based on the above ideas, we designed and implemented a new system called Volt for automatically synthesizing data structure refinements from integrity constraints. To evaluate Volt, we identified data structures with correlated fields in widely-used Java projects and used Volt to derive the full data structure from a version without the auxiliary fields. Our evaluation shows that Volt can successfully synthesize 92% of these refinements with an average running time of 264.2 seconds per benchmark. We also compare Volt against two state-of-the-art synthesis tools for Java and show that Volt dramatically outperforms these tools. Finally, we also evaluate three different ablations of Volt and show that our proposed ideas are indeed necessary for making this approach practical.

In short, this paper makes the following key contributions:

- We propose automatically refining data structures from integrity constraints in order to correctly implement efficient data structures that utilize many correlated fields.
- We propose a modular algorithm that decomposes the overall data structure refinement task into procedure-level synthesis tasks.
- We propose a novel inductive synthesis algorithm that uses three key ideas, namely (1) PCFG construction using static analysis, (2) pruning via necessary preconditions, and (3) dynamic PCFG augmentation and update.
- We conduct an experimental evaluation on 25 real-world Java programs and compare our method against state-of-the-art synthesis/repair tools as well as against several ablations of Volt itself.

2 Overview

In this section we provide a high level overview of our technique with the aid of the motivating example shown in Figure 1. The code snippet in Figure 1 belongs to a data structure called `SpdySession` used in Netty [5], a framework for developing high-performance protocol servers in

```
1 public class SpdySession {
2   private Map<Integer, StreamState> actStreams;
3   private AtomicInteger als, ars; // new fields
4   ...
5   public SpdySession() {
6     actStreams = new HashMap<>();
7     // als = new AtomicInteger();
8     // ars = new AtomicInteger();
9   }
10
11  int numActiveStreams(bool remote) {...} //
        New
12
13  public void acceptStream(int sId, ...) {
14    StreamState newS = new StreamState(...);
15    StreamState oldS = actStreams.put(sid,
          newS);
16    // if (oldS == null) {
17    //   if (this.isRemoteInitiatedId(sId)) {
18    //     ars.addAndGet(1);
19    //   } else {
20    //     als.addAndGet(1);
21    //   }
22    // }
23  }
24
25  public void removeStream(int sId, ...) {
26    StreamState state =
          removeActiveStream(sId);
27    // if (state != null) {
28    //   if (this.isRemoteInitiatedId(sId)) {
29    //     ars.subAndGet(1);
30    //   } else {
31    //     als.subAndGet(1);
32    //   }
33    // }
34    ...
35  }
36  ...
37 }
```

Figure 1. SpdySession class from the Netty project. The auxiliary fields `als` and `ars` keep track of the number of local and remote streams in `actStream` respectively.

Java. `SpdySession` tracks active streams using a ConcurrentHashMap called `actStreams` (line 2) where keys correspond to stream identifiers and values are the stream states (encapsulated in another data structure called `StreamState`).

In one of the Github commits, Netty developers added a new method called `numActiveStreams` (line 11) to the class `SpdySession`. This new method takes as input a boolean argument called `remote` and returns the number of remote or local streams depending on the value of the boolean. A naive implementation of `numActiveStreams` would iterate over the hashmap (`actStreams`) to count the number of

```
1 public boolean check(SpdySession s) {
2   int remote = 0, local = 0;
3   for (Integer id : s.actStream) {
4     if (s.isRemoteInitiatedId(id)) remote++;
5     else local++;
6   }
7   if (s.als.get() != local || s.ars.get() !=
        remote)
8     return false;
9   return true;
10 }
```

Figure 2. Integrity checking function.

remote/local streams; however, this is obviously quite inefficient. Thus, Netty developers add two new fields called `als` and `ars` to keep track of the number of local and remote streams respectively and modify the implementation of `SpdySession` to correctly update these fields. In particular, the commented out (green) lines in Figure 1 are the new code added to the original code.

Volt usage scenario. Our proposed approach is useful in a scenario like this for automatically updating the implementation of `SpdySession` in a correct-by-construction manner. To use Volt, the user only needs to provide the original implementation of `SpdySession`, the names of the new fields (`ars` and `als`), and an integrity constraint expressed as a function called `check` in Figure 2. This `check` function iterates over the `actStream` hashmap and counts the number of remote and local streams. If the remote (resp. local) stream count is equal to `ars` (resp. `als`), `check` returns true; otherwise, it returns false.

Given this integrity checking function, Volt automatically generates the refined data structure in under one minute. The new implementation involves changes to 5 of the 42 methods in `SpdySession` and is verified to both preserve existing functionality and satisfy the provided integrity constraint.

Challenges. We now highlight some of the challenges involved in automating data structure refinement. First, when adding new fields to a data structure, we may need to synthesize new code in multiple methods. For instance, in the `SpdySession` example, 10 of the 42 methods in the original data structure manipulate streams, and 5 of these 10 need to be modified in order to correctly maintain the new `als` and `ars` fields. Second, modifications to existing methods are non-trivial: they require adding several lines of code with nested conditionals (e.g., see lines 16-22) and calling functions like `addAndGet` that are implemented elsewhere in the code base. Finally, observe that modifications to each method are not identical and cannot be simply "copy-pasted".

The Volt solution. We solve these challenges using a modular application of the CEGIS paradigm coupled with

a novel inductive synthesizer. In particular, Volt performs synthesis in a *modular* way by independently updating the implementation of each method using CEGIS. The inductive synthesizer used in Volt is based on top-down enumerative search but takes advantage of three important observations that allow it to succeed in this setting:

Observation #1: The integrity checker and existing methods in the code base provide important syntactic clues to the synthesizer. For instance, looking at the implementation of `check` from Figure 2, one would expect that `isRemoteInitiated` would be useful during synthesis, as the return value of `check` depends on that of `isRemoteInitiated`. In addition, performing deeper static analysis of existing methods in the code base can also provide important clues. For example, `addAndGet` and `subAndGet` methods update variables of type `AtomicInteger`, and since the new fields `als` and `ars` also have the same type, these two methods may appear as part of the solution. To leverage such useful clues during synthesis, our method attaches probabilities to productions in the context-free grammar. Then, our search algorithm expands non-terminals according to these probabilities, thereby biasing search towards program that use various clues present in the integrity checker and the surrounding code base.

Observation #2: We can deductively prune away partial programs using necessary preconditions. To gain some intuition, suppose that the inductive synthesizer generates the following partial implementation of `acceptStream` during search:

```
1  public void acceptStream(int sId, ...) {
2    StreamState newS = new StreamState(...);
3    StreamState oldS = actStreams.put(sid, newS);
4    ??_s ;
5    this.als = new AtomicInteger();
6  }
```

Here, $??_s$ is a hole that can be filled by any arbitrary statement. This partial program is infeasible because, no matter how we complete the hole, the resulting program will not satisfy the integrity constraint. In particular, the code snippet above always sets the value of `this.als` to 0 (the last line); however, this will violate the integrity constraint if `acceptStream` is ever called with a local stream (as it is added to `actStreams` on the second line).

Volt can prune such infeasible partial programs by computing a *necessary precondition* for correctness. A necessary precondition ϕ for a partial program P is a constraint on P's inputs that must be satisfied by any completion P in order for P to satisfy the integrity constraint. Thus, if the current set of counterexamples includes an input that violates ϕ, we can conclude that the partial program is *infeasible* – i.e., no completion of the holes can satisfy the specification.

Going back to our example, the necessary precondition computed by Volt for the partial program above logically implies that `sid` must correspond to a remote (not local) stream. Thus, if the current counterexamples include a local stream, Volt can immediately reject this partial program. Since there are approximately 756k completions of $??_s$ (up to size 5), the computation of necessary preconditions allows our technique to dramatically reduce the search space.

Observation #3: We can use the feedback from the feasibility checking engine to dynamically augment the PCFG. Consider the following partial implementation for `acceptStream`:

```
1  public void acceptStream(int sId, ...) {
2    StreamState newS = new StreamState(...);
3    StreamState oldS = actStreams.put(sid,
         newS);
4    if (??_e) {
5      this.ars.addAndGet(??_e);
6    } else {
7      this.als.addAndGet(??_e);
8    }
9  }
```

where lines 4–8 correspond to a *sketch* – i.e., synthesized code that does not contain any unknown statements or left-hand-side expressions. Even though this sketch is infeasible, its necessary precondition is consistent with many (but not all) of the counterexamples due to a missing edge case. We refer to such sketches as *partially feasible*. Our key observation is that such partially feasible program sketches often appear as sub-fragments of the desired solution. For example, observe that lines 4–8 above actually appear as a code snippet within the correct implementation of `acceptStream` shown in Figure 1 (lines 17–21). Based on this observation, our method augments the PCFG with new non-terminals that correspond to partially feasible program sketches. Since the addition of the new non-terminals is driven by necessary preconditions, we refer to this method as *deduction-guided grammar augmentation*.

3 Problem Formulation

We formulate the data structure refinement problem with respect to a Java-like statically-typed object-oriented language. A program in this language is a collection of data structures, and a data structure is of the form $D = (F, M)$ where F is a set of fields and M is a mapping from method signatures to their bodies. Without loss of generality, we assume that every data structure D has a unique method called D_0 that corresponds to its initialization method (constructor). Fields and variables in D are either of type `int` or $\texttt{Ref}(D')$ where `int` denotes integers and $\texttt{Ref}(D')$ denotes the address of some heap-allocated data structure of type D'. The program heap $\mathcal{H} : \texttt{Ref} \times \texttt{Field} \rightarrow \texttt{Value}$ maps reference and field pairs to values.

Definition 3.1. (State) A state σ for a data structure $D = (F, M)$ is a pair $(o, \mathcal{H})$ where o is an instance of D and heap $\mathcal{H}$ determines the values of o's fields.

We say that a state σ is an *initial state* for data structure $D = (F, M)$ if it is obtained immediately after calling the constructor D_0 of D.

Definition 3.2. (State equality modulo fields) Given a pair of data structure states $\sigma = (o, \mathcal{H})$ and $\sigma' = (o', \mathcal{H}')$, we say that σ and σ' are *equivalent modulo fields* F, written $\sigma \equiv_F \sigma'$, iff for every $f \in F$, we have:

1. $\mathcal{H}(o, f) = \mathcal{H}'(o', f)$ if f has type `int`
2. If f has type $\texttt{Ref}(D)$ for $D = (F', \_)$, then $\sigma_1 \equiv_{F'} \sigma_2$ where $\sigma_1 = (\mathcal{H}(o, f), \mathcal{H})$ and $\sigma_2 = (\mathcal{H}'(o', f), \mathcal{H}')$

In other words, the $\equiv_F$ relation indicates deep equality with respect to fields F.

Definition 3.3. (State refinement) Let σ and σ' be two states for data structures (F, M) and (F', M') respectively where $F \subseteq F'$. We say σ' is a refinement of σ, written $\sigma' \preceq \sigma$, iff $\sigma \equiv_F \sigma'$.

Intuitively, a data structure state σ' refines another state σ if they are equivalent modulo the fields defined in σ.

Definition 3.4. (Action) Let $D = (F, M)$ be a data structure. An action α on D is a pair $(m, \overline{x})$ where $m \in \mathsf{Domain}(M)$ and $\overline{x}$ is a list of arguments for m.

In other words, an action corresponds to invoking a method of the data structure.

Definition 3.5. (Transition) Let $D = (F, M)$ be a data structure, σ a state of D, and $\alpha = (m, \overline{x})$ an action on D. We write $\sigma \xrightarrow{\alpha} \sigma'$ to denote that σ' is the resulting state after executing method m on state σ with arguments $\overline{x}$. We refer to $(\sigma, \alpha, \sigma')$ as a transition of D.

Informally, a transition represents a change in the data structure's state after calling a method.

Definition 3.6. (State transition system) The state transition system T_D for a data structure D is a tuple $(S, I, A, \rightarrow)$ where S is the set of all possible states of D, $I \subseteq S$ are the set of initial states, A is the set of all possible actions on D, and $\rightarrow$ is the transition relation for D (i.e., $\sigma \rightarrow \sigma'$ iff $\exists \alpha \in A$ such that $(\sigma, \alpha, \sigma')$ is a transition of D).

We write $\rightarrow^*$ to denote the reflexive transitive closure of $\rightarrow$, and we say that σ is a *reachable state* of D if there exists a $\sigma_0 \in I$ such that $\sigma_0 \rightarrow^* \sigma$.

Definition 3.7. (Method refinement) Let $D = (F, M)$ and $D' = (F', M')$ be two data structures with corresponding transition systems $T_D = (S, I, A, \rightarrow)$ and $T_{D'} = (S', I', A', \rightarrow')$. The implementation of m in D' refines that of m in D, written $m_{D'} \preceq m_D$, if, for every action $\alpha = (m, \overline{v})$, and reachable state $\sigma_b \in S$, we have: If $\sigma'_b \preceq \sigma_b$ and $(\sigma_b, \alpha, \sigma_a) \in \rightarrow$, $(\sigma'_b, \alpha, \sigma'_a) \in \rightarrow'$, then $\sigma'_a \preceq \sigma_a$.

Intuitively, the implementation of a method in D' is a refinement of the corresponding method in D if it preserves the refinement relation between the states of the data structures when called with the same arguments.

Definition 3.8. (Data structure refinement) We say a data structure $D' = (F', M')$ is a refinement of $D = (F, M)$, written $D' \preceq D$, if the following conditions are satisfied: (1) $F \subseteq F'$ and (2) $\mathsf{Domain}(M) = \mathsf{Domain}(M')$ and (3) For every method $m \in \mathsf{Domain}(M)$, $m_{D'} \preceq m_D$.

In other words, a data structure D' refines D if the set of fields of D' is a superset of those of D and, for every method m in D and its corresponding method m' in D', m' refines m.

Definition 3.9. (Integrity constraint) Let D be a data structure with fields $F \cup F'$. An integrity constraint Φ_c for fields F' is a function that takes as input a data structure state σ of D and returns a boolean value such that:

$$\forall \sigma, \sigma' \in \mathsf{Reachable}(D).$$
$$\Phi_c(\sigma) \wedge \Phi_c(\sigma') \wedge \sigma \equiv_F \sigma' \implies \sigma \equiv_{F'} \sigma'$$

We say the integrity constraint holds on state σ, denoted $\sigma \models \Phi_c$, if Φ_c returns true on input σ.

In other words, the integrity constraint can be used to check whether the values of auxiliary fields F' are correct based on the values of other fields. For instance, the `check` function from Figure 2 conforms to our definition: given any value for field `actStream`, there is only one possible value for the fields `als` and `ars` for which `check` returns true.

Definition 3.10. (Integrity constraint satisfaction) We say that a data structure D satisfies integrity constraint Φ_c, denoted $D \models \Phi_c$, iff, for every reachable state σ of D, we have $\sigma \models \Phi_c$.

Problem statement. Given a data structure definition $D = (F, M)$ and an integrity constraint Φ_c , our goal is to synthesize a new data structure $D' = (F, M')$ such that (1) $D' \preceq D$ and (2) $D' \models \Phi_c$.

4 Data Structure Refinement Algorithm

In this section we present our algorithm for synthesizing data structure refinements from integrity constraints. We start by introducing some terminology and then give an overview of our modular refinement procedure. Afterwards, we describe the novel aspects of our inductive synthesis approach in more detail, including necessary precondition computation for pruning partial programs as well as deduction-guided grammar augmentation.

4.1 Preliminaries

Figure 3 shows the context-free grammar (CFG) used by our synthesis algorithm for generating method bodies. It contains loops, conditionals, assignments, stores, loads, method calls, etc. Some of the non-terminals $\mathcal{N}$ in this CFG (e.g., L^τ, E^τ) are parametrized by types to prevent enumeration of ill-typed programs during synthesis.

$$
\begin{array}{rcl}
\text{Stmt } S & \rightarrow & A \mid S_1; S_2 \mid \texttt{if}(B)\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}(B)\ S \\
\text{Atom } A & \rightarrow & L^\tau \leftarrow E^\tau \mid E^\tau.m(L_1^{\tau_1}, \cdot\cdot, E_n^{\tau_n}) \mid \text{new } m(L_1^{\tau_1}, \cdot\cdot, E_n^{\tau_n}) \\
\text{LHS } L^\tau & \rightarrow & L^{\tau'}.f \mid L^{\tau[]}[I] \mid v^\tau \\
\text{Index } I & \rightarrow & E^{\texttt{int}} \mid v^{\texttt{int}} \\
\text{Expr } E^\tau & \rightarrow & \otimes^\tau(E_1^{\tau_1}, \cdot\cdot, E_n^{\tau_n}) \mid v^\tau \mid c^\tau \mid E^{\tau_i}.f \mid E^{\tau[]}[I] \\
\text{Pred } B & \rightarrow & E^{\texttt{bool}} \mid \ \oplus(B_1, \cdot\cdot, B_n)
\end{array}
$$

Figure 3. CFG used by the synthesis algorithm. Here, $\otimes^\tau$ denotes an n-ary operator that produces a result of type τ, and $\oplus$ is a boolean connective.

Given grammar $\mathcal{G}$ with non-terminals $\mathcal{N}$, terminals $\mathcal{T}$, and productions $\mathcal{R}$, we say that a string $\mathcal{P} \in (\mathcal{T} \cup \mathcal{N})^*$ is a partial program iff $S \Rightarrow^* \mathcal{P}$ (i.e., $\mathcal{P}$ can be derived from the start symbol) and we refer to non-terminals in $\mathcal{P}$ as *holes*. A string $\mathcal{P}$ is said to be a *complete program* if $\mathcal{P}$ does not contain any non-terminals (i.e., $\mathcal{P} \in \mathcal{T}^*$). In this paper, we use the term *sketch* to refer to a partial program that does not contain non-terminals L^τ and S. In other words, a sketch is a left-hand-side complete partial program (modulo array indices). Finally, we say that $\mathcal{P}'$ is an *expansion* of $\mathcal{P}$ if $\mathcal{P}'$ can be obtained from $\mathcal{P}$ by substituting some non-terminal N in $\mathcal{P}$ with α for some production $N \rightarrow \alpha$ in the grammar.

Example 4.1. We provide a few examples to illustrate our terminology:

- $\texttt{if}\ (B)\ \ L^\tau \leftarrow E^\tau\ \ \texttt{else}\ \ S$ (Partial Program)
- $\texttt{if}\ (B)\ \ x^{\texttt{int}} \leftarrow E^{\texttt{int}}\ \ \texttt{else}\ \ x^{\texttt{int}} \leftarrow E^{\texttt{int}}$ (Sketch)
- $\texttt{if}\ (a^{\texttt{int}} > 1)\ \ x^{\texttt{int}} \leftarrow y^{\texttt{int}}\ \ \texttt{else}\ \ x^{\texttt{int}} \leftarrow z^{\texttt{int}}$ (Complete)

As mentioned earlier, our synthesis algorithm associates a probability with each production r in the grammar. Specifically, for a given production $r = (N \rightarrow \alpha) \in \mathcal{G}$, $\mathbb{P}(r) \in [0, 1]$ corresponds to the probability of expanding non-terminal N using production r. A context-free grammar $\mathcal{G}$ augmented with such a probability distribution $\mathbb{P}$ over the productions of $\mathcal{G}$ is called a *probabilistic context free grammar (PCFG)*. Also, given a partial program P, we define $\Pr(\mathcal{P})$ to be the probability of obtaining $\mathcal{P}$ using productions in the PCFG. More formally,

$$\Pr_{\mathcal{G}}(\mathcal{P}) = \sum_{\Delta \in \text{Derivs}(\mathcal{P}, \mathcal{G})} \prod_{r \in \Delta} \mathbb{P}(r)$$

where $\text{Derivs}(\mathcal{P})$ denotes all derivations for partial program $\mathcal{P}$ and a derivation is a sequence of productions $r_1, \ldots, r_n$.

4.2 Modular Refinement Algorithm

We now describe our modular data structure refinement procedure (summarized in Algorithms 1 and 2). The Refine procedure (Alg. 1) takes as input the original data structure $D = (F, M)$, a new set of fields F', and an integrity constraint $\Phi_c(F, F')$ (encoded as a boolean method), and it returns a refined data structure $D' = (F \cup F', M')$ such that $D' \preceq D$ and $D' \models \Phi_c$.

1: **procedure** Refine(D, F', Φ_c)
input: Data structure $D = (F, M)$; new fields F'
input: Integrity constraint Φ_c
output: D' such that $D' \preceq D$ and $D' \models \Phi_c$
2: $M' \leftarrow \varnothing$
3: **for all** $m \in \text{Domain}(M)$ **do**
4: $\mathcal{P} \leftarrow$ InsertHoles($M[m]$)
5: $S \leftarrow$ UpdateMethod($M[m], \mathcal{P}, \Phi_c$)
6: **if** $S = \perp$ **then return** $\perp$
7: $M' \leftarrow M' \cup \{(m, S)\}$
8: **return** $(F \cup F', M')$

Algorithm 1. Top-level refinement procedure

1: **procedure** UpdateMethod(S, $\mathcal{P}$, Φ_c)
input: Original body S for some method m in D
input: Partial program $\mathcal{P}$ for new body of m in D'
input: Integrity constraint Φ_c
output: New method body $m_{D'} = S'$ such that (1) $m_{D'} \preceq m_D$, and (2) $\{\Phi_c\}S'\{\Phi_c\}$ is a valid Hoare triple.
2: $C \leftarrow \varnothing$; ▷ Counterexamples
3: $\mathcal{G} \leftarrow$ InitPCFG(Φ_c, f);
4: **while** true **do**
5: $(S', \mathcal{G}') \leftarrow$ Synthesize($\mathcal{P}, C, \mathcal{G}, S, \Phi_c$);
6: **if** $S' = \perp$ **then return** $\perp$; ▷ No feasible solution
7: $S'' \leftarrow$ Instrument(S, S', Φ_c);
8: $C' =$ Verify(S'');
9: **if** $C' = \varnothing$ **then return** S'; ▷ Verification successful
10: $C \leftarrow C \cup C'$; $\mathcal{G} \leftarrow \mathcal{G}'$;

Algorithm 2. CEGIS loop for updating a method

At a high level, the Refine procedure constructs D' in a modular way by updating each method $m \in M$ independently. Specifically, for each method m, Refine invokes a procedure called InsertHoles (line 3) which uses lightweight static analysis to identify code fragments that modify F and inserts statement holes (i.e., nonterminals S) at all relevant program points. Hence, the output of InsertHoles is a partial program $\mathcal{P}$ which is then refined into a complete program using the UpdateMethod procedure at line 5. If UpdateMethod is successful (i.e., returns $S \neq \perp$), then the new implementation $m_{D'} = S$ of m in D' is guaranteed to satisfy the refinement relation $m_{D'} \preceq m_D$ as well as the integrity constraint (i.e., $\{\Phi_c\}m_{D'}\{\Phi_c\}$ is a valid Hoare triple).

The UpdateMethod procedure (Alg. 2) synthesizes $m_{D'}$ using a CEGIS loop but first initializes a PCFG $\mathcal{G}$ by assigning probabilities to each production in the grammar (line 3). We describe how to initialize the PCFG in more detail in Section 4.4. Then, in each iteration of the CEGIS loop, our algorithm invokes the Synthesize method (line 5) to generate a candidate implementation S' that satisfies all counterexamples C

encountered so far. A key novelty of our inductive synthesis procedure is that, in addition to producing a candidate implementation, it also produces a new grammar $\mathcal{G}'$ that can be leveraged in subsequent iterations of the CEGIS loop.

Next, given a candidate implementation S' for $\mathcal{P}$, UpdateMethod checks whether S' satisfies the conditions (1) $S' \preceq S$ and (2) $\{\Phi_c\}S'\{\Phi_c\}$. To do so, it invokes the Instrument procedure to construct the code snippet shown in Figure 4 which is then verified using an off-the-shelf assertion checker. Observe that the instrumented code in Figure 4 works as follows: First creates an instance d of the original data structure D by calling GetInstance (Figure 5), which simply invokes the constructor of D followed by an invocation of an arbitrary sequence of D's methods. Then, it initializes d' to be an arbitrary object of type D' and stipulates that (1) d and d' obey the refinement relation and (2) d' satisfies the integrity constraint. Next, it invokes the methods m, m' for d, d' with the same arguments, and finally asserts that d, d' continue to satisfy the refinement relation and that d' obeys the integrity constraint. If the assertions in this instrumented program can be verified using an off-the-shelf assertion checker (line 8 of Algorithm 2), UpdateMethod returns S' as a valid refinement of S (line 9). Otherwise, it adds the counterexamples returned by the verifier to C and continues the CEGIS loop with updated grammar $\mathcal{G}'$ and additional counterexamples C'.

Remark. Note that the GetInstance call in the instrumented code in Figure 4 ensures that d is a *reachable* state of data structure D. Furthermore, based on Def. 3.9 of integrity constraint, this d uniquely determines d' (modulo memory addresses). Thus, any counterexample corresponds to a reachable state of D' and ensures that we do not reject valid solutions despite using a modular strategy. Lastly, note that after calling GetInstance we can safely assume the integrity constraint holds on d'. This is because if Φ_c does not hold prior to m', then one of the methods invoked in GetInstance must violate Φ_c. However, we verify (inductively) that none of the methods are the first to break the integrity constraint and so it is safe to assume Φ_c holds. This is known in the verification literature as circular compositional reasoning [32, 35].

We state and prove the following theorems under the assumption that the verification oracle is sound and complete.

Lemma 4.2. *Suppose that* UpdateMethod *rejects a candidate body S' for method m. Then, there does not exist a solution D' to the synthesis problem where $M'[m] = S'$.*

This lemma is important for the completeness of the end-to-end approach (Theorem 4.4) because it states that UpdateMethod does not reject valid solutions.

Theorem 4.3. (Soundness) *Suppose that* Refine(D, F', Φ_c) *returns D'. If $D' \neq \bot$ then $D' \preceq D$ and $D' \models \Phi_c$.*

```
d = GetInstance(D); d' = ⋆
assume(d ≡_F d'); assume(Φ_c(d'));
args = ⋆; m(d, args); m'(d', args);
assert(d ≡_F d'); assert(Φ_c(d'));
```

Figure 4. Code generated by the Instrument procedure. ⋆ denotes a random value.

```
function GetInstance(D)
  d = new D()
  while ⋆ do m = randMethod(D); m(d, ⋆)
  return d
```

Figure 5. Returns reachable instance of data structure D

```
1:  procedure Synthesize(P, C, G, S, Φ_c)
    input: Partial program P; counterexamples C
    input: Original method body S for a method m; PCFG G
    input: Integrity constraint Φ_c
    output: Candidate implementation S' and new PCFG G'
2:    W = {P}
3:    while W ≠ ∅ do
4:      P' ← SelectBest(W, G)
5:      C_SAT ← Deduce(P', S, C, Φ_c)
6:      if IsComplete(P') ∧ |C_SAT| = |C| then return (P', G)
7:      if |C_SAT| > 0 ∧ IsSketch(P') then
8:        for all h ∈ Holes(P) do
9:          P_i ← Impl(P', h)
10:         G ← AugmentGrammar(G, P_i, C_SAT)
11:     if |C_SAT| = |C| then W ← W ∪ Expand(P')
12:   return (⊥, G)
```

Algorithm 3. Inductive synthesis algorithm

Theorem 4.4. (Completeness) *Let* Refine(D, F', Φ_c) *return D'. If $D' = \bot$ then there does not exist a $D' = (F', M')$ such that $D \preceq D'$ and $D' \models \Phi_c$.*

4.3 Inductive Synthesis Algorithm

In this section, we describe our inductive synthesis procedure summarized in Algorithm 3. This algorithm uses top-down enumerative search with deduction-based pruning and grammar augmentation. In particular, it maintains a worklist W of partial programs (initialized to $\{\mathcal{P}\}$ at line 2) and iteratively explores partial programs until it finds a complete program that satisfies all the counterexamples (line 6). In each iteration of the loop, it invokes the SelectBest function (line 4) to identify the most promising partial program in the worklist according to the PCFG. Here, the best program is defined as follows:

$$SelectBest(W, \mathcal{G}) = \left(\underset{\mathcal{P} \in W}{\text{argmax}}\ \Pr_{\mathcal{G}}(\mathcal{P})\right)$$

Next, given a partial program $\mathcal{P}'$, Synthesize invokes the Deduce function (line 5) to check whether $\mathcal{P}'$ is infeasible.

1: **procedure** DEDUCE($\mathcal{P}$, S, C, Φ_c)
input: Partial program $\mathcal{P}$
input: Original method body S for method m
input: Counterexamples C; Integrity Constraint Φ_c
output: Set of satisfied counterexamples C_{SAT}
2: $C_{SAT} \leftarrow \varnothing$
3: $S' \leftarrow$ APPROXIMATE(Inline($\mathcal{P}$), $\varnothing$)
4: $\Phi \leftarrow \text{WP}(S', \Phi_c)$
5: **for all** $C \in C$ **do**
6: **if** $SAT(\Phi[C])$ **then** $C_{SAT} \leftarrow C_{SAT} \cup \{C\}$
7: **return** C_{SAT}

Algorithm 4. Checking feasibility of partial programs

In particular, DEDUCE returns a set $C_{SAT} \subseteq C$ such that $\mathcal{P}'$ is consistent with the specification for every input $C \in C_{SAT}$. Thus, if $C_{SAT} \neq C$, this means that all completions of $\mathcal{P}'$ violate the specification for at least one input $C \in C$; hence, $\mathcal{P}'$ can be pruned from the search space without sacrificing completeness. Therefore, we only expand a partial program if $C_{SAT} = C$ (see line 12). On the other hand, if $C_{SAT} = C$ and $\mathcal{P}'$ is a *complete* program, then $\mathcal{P}'$ is a solution to our inductive synthesis problem and is returned at line 6.

Lines 8-10 of the SYNTHESIZE algorithm perform *deduction-guided grammar augmentation*. In particular, if $\mathcal{P}'$ is a sketch that is consistent with a non-empty subset of the counterexamples, it iterates over all the synthesized fragments in $\mathcal{P}'$, and for each synthesized fragment $\mathcal{P}_i$, it invokes the AUGMENTGRAMMAR procedure to add a new production $A \longrightarrow \mathcal{P}_i$ to the grammar. This is based on the observation that program sketches that satisfy some of the counterexamples often tend to occur as sub-components of the final solution. Thus, to avoid re-synthesizing these (potentially large) program sketches from scratch in future iterations, we directly add them as productions to the grammar. In essence, such deduction-guided grammar augmentation allows combining the benefits of top-down and bottom-up search in a goal directed way.

In what follows, we explain the DEDUCE and AUGMENTGRAMMAR procedures in more detail.

Pruning via necessary preconditions. Our technique for checking feasibility of partial programs is presented in Algorithm 4. Given a partial program $\mathcal{P}$, the DEDUCE procedure returns a subset C_{SAT} of C that $\mathcal{P}$ is consistent with. The high level idea is to compute a *necessary precondition* Φ for correctness that any instantiation of $\mathcal{P}$ must satisfy and test whether Φ is satisfied by the counterexamples. If any $C \in C$ violates this necessary precondition, $\mathcal{P}$ is guaranteed to be infeasible.

In more detail, the DEDUCE procedure first inlines the method calls in the $\mathcal{P}$ [1] and calls APPROXIMATE to generate a completion S' of $\mathcal{P}$. Here, S' contains symbolic variables that represent "environment choices", and it is constructed in such a way that, if there exists a completion of $\mathcal{P}$ that satisfies the specification, then it is possible to find values of symbolic variables in S' so that S' satisfies the specification.

Figure 6 presents the APPROXIMATE procedure as inference rules deriving judgments of the form $\vdash \alpha \rightsquigarrow \beta$ where α is a sequence of symbols in the grammar and β is its corresponding replacement (i.e., over-approximation). If α represents an expression or predicate, we obtain its replacement by recursively replacing any non-terminals nested inside it with fresh (unconstrained) variables. For assignments, we replace the assignment $\alpha \leftarrow \beta$ with $\alpha' \leftarrow \beta'$ where α', β' are replacements for α, β respectively and where α does not contain the non-terminal L (ASSIGN-1). On the other hand, if α contains a non-terminal L, we do not know which memory location is being written to; thus, we model it as a non-deterministic assignment to any of the new variables (ASSIGN-2). Similarly, we model an unknown statement S as a write to *all* possible memory locations that may be modified by S (STATEMENT).

Example 4.5. APPROXIMATE produces the following complete program for Listing 2 in Section 2.

```
public void acceptStream(int sId, ...) {
  StreamState newS = new StreamState(...);
  StreamState oldS = actStreams.put(sid, newS);
  AtomicInteger v1, v2; // fresh
  this.als = v1; this.ars = v2; // added
  this.als = new AtomicInteger();
}
```

Given the output S' of APPROXIMATE, the DEDUCE procedure computes the weakest pre-condition of S' with respect to the integrity constraint Φ_c. [2] As stated by the following theorem, the weakest precondition Φ of S' is a *necessary precondition* for S (and therefore $\mathcal{P}$):

Theorem 4.6. *Let S' be an over-approximation of code S. Then, if ϕ is a necessary condition for S' to be correct, then ϕ is also a necessary precondition for S.*

Thus, based on this theorem, if there is a counterexample that is inconsistent with the computed necessary condition Φ, this means that we can prune partial program $\mathcal{P}$ from the search space.

Grammar augmentation. The final piece of our inductive synthesis technique is the AUGMENTGRAMMAR procedure presented in Algorithm 5 for adding new productions to the PCFG. This algorithm takes as input the current PCFG $\mathcal{G}$, a sketch $\mathcal{P}$, the current set of counterexamples C, and it returns an augmented grammar that contains new productions. In particular, the augmented grammar contains a new

[1] We inline method calls here to simplify presentation; our implementation does not perform inlining

[2] Since Φ_c is a boolean function, $WP(S', \Phi_c)$ can be computed as $WP(S'; \Phi_c, \mathit{true})$. The verifier in our implementation is a bounded model checker, so we unroll and compute weakest preconditions in a standard way.

$$\frac{N \in \{E^{\tau}, I, B\}\ v \text{ fresh}}{\vdash N \rightsquigarrow v}\ \textsc{Expr Non-Term} \qquad \frac{\alpha \in \mathcal{T}^*}{\vdash \alpha \rightsquigarrow \alpha}\ \textsc{Terminal} \qquad \frac{\vdash \alpha_i \rightsquigarrow \beta_i \text{ for } 1 \le i \le n}{\vdash op(\alpha_1, \ldots, \alpha_n) \rightsquigarrow op(\beta_1, \ldots, \beta_n)}\ \textsc{Expr Operation}$$

$$\frac{\{f_1, \ldots, f_n\} = \text{NewVars} \quad v_i \text{ fresh}(i \in [1, n])}{\vdash S \rightsquigarrow f_1 \leftarrow v_1; \ldots; f_n \leftarrow v_n}\ \textsc{Statement}$$

$$\frac{L \notin \alpha \quad \vdash \alpha \rightsquigarrow \alpha' \quad \vdash \beta \rightsquigarrow \beta'}{\vdash (\alpha \leftarrow \beta) \rightsquigarrow (\alpha' \leftarrow \beta')}\ \textsc{Assign-1} \qquad \frac{L \in \alpha \quad \beta \rightsquigarrow \beta' \quad \{f_1, \ldots, f_n\} = \text{NewVars}}{\vdash (\alpha \leftarrow \beta) \rightsquigarrow \text{choose}((f_1 \leftarrow \beta'), \ldots, (f_n \leftarrow \beta'))}\ \textsc{Assign-2}$$

$$\frac{\vdash \alpha_1 \rightsquigarrow \beta_1 \quad \vdash \alpha_2 \rightsquigarrow \beta_2 \quad \vdash \alpha_3 \rightarrow \beta_3}{\vdash \texttt{if}(\alpha_1)\ \alpha_2\ \texttt{else}\ \alpha_3 \rightsquigarrow \texttt{if}(\beta_1)\ \beta_2\ \texttt{else}\ \beta_3}\ \textsc{If-Else} \qquad \frac{\vdash \alpha_1 \rightsquigarrow \beta_1 \quad \vdash \alpha_2 \rightsquigarrow \beta_2}{\vdash \texttt{while}(\alpha_1)\ \alpha_2 \rightsquigarrow \texttt{while}(\beta_1)\ \beta_2}\ \textsc{While}$$

$$\frac{\vdash \alpha_1 \rightsquigarrow \beta_1 \quad \vdash \alpha_2 \rightsquigarrow \beta_2}{\vdash \alpha_1; \alpha_2 \rightsquigarrow \beta_1; \beta_2}\ \textsc{Sequence}$$

Figure 6. Approximate inference rules. choose represents a nondeterministic choice between any of its arguments

1: **procedure** AugmentGrammar($\mathcal{G}, \mathcal{P}, C$)
input: PCFG $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathbb{P})$; sketch $\mathcal{P}$
input: Counterexamples C collected so far
output: An updated PCFG $\mathcal{G}'$
2: $\mathcal{T}' \leftarrow \mathcal{T}; \mathbb{P}' \leftarrow \mathbb{P}$;
3: **if** $\mathcal{S} \notin \mathcal{N}$ **then**
4: $\mathcal{N}' \leftarrow \mathcal{N} \cup \{\mathcal{S}\}$
5: $\mathcal{R}' \leftarrow \mathcal{R} \cup \{\mathcal{S} \rightarrow \mathcal{P}\} \cup \{A \rightarrow \mathcal{S}\}$;
6: InitProbabilities($\mathbb{P}, \mathcal{R}'_A$)
7: **else**
8: $\mathcal{N}' \leftarrow \mathcal{N}$; $\mathcal{R}' \leftarrow \mathcal{R} \cup \{\mathcal{S} \rightarrow \mathcal{P}\}$;
9: $Z \leftarrow \sum_{(\mathcal{S} \rightarrow \alpha) \in \mathcal{R}} \text{NumSatisfied}(\alpha, C)$
10: **for all** $r_i \in \{r \mid r \equiv (\mathcal{S} \rightarrow \alpha) \in \mathcal{R}'\}$ **do**
11: $n_i \leftarrow \text{NumSatisfied}(\alpha, C)$
12: $\mathbb{P}'(r_i) \leftarrow n_i / Z$
13: **return** $(\mathcal{N}', \mathcal{T}', \mathcal{R}', \mathbb{P}')$

Algorithm 5. Grammar augmentation procedure. NumSatisfied(α, C) returns the average number of counterexamples in C satisfied by previously explored sketches containing α.

non-terminal $\mathcal{S}$ that represents sketches. It also contains two new productions:

- The production $A \rightarrow \mathcal{S}$ allows using sketches as atomic building blocks when constructing partial programs. (Recall that non-terminal A represents atomic statements.)
- The production $\mathcal{S} \rightarrow \mathcal{P}$ adds sketch $\mathcal{P}$ as a new building block in the grammar.

In more detail, if the input grammar $\mathcal{G}$ does not contain the non-terminal symbol $\mathcal{S}$, we add $\mathcal{S}$ to the set of non-terminals and add $A \rightarrow \mathcal{S}$ to the set of productions. We also call InitProbabilities (discussed in Section 4.4.2) to update the probabilities of all productions starting with non-terminal A. If the grammar already contains productions of the form $\mathcal{S} \rightarrow \alpha$, the probabilities associated with all of these productions also need to be updated. Thus, the loop in lines 10-12 adjusts the probabilities for each production $r_i = (\mathcal{S} \rightarrow \alpha_i)$ to n_i/Z where n_i is the average number of counterexamples satisfied by sketches containing α_i and Z is a normalization term. Here, the intuition is to assign higher probabilities to productions associated with sketches that satisfy more counterexamples.

4.4 PCFG Initialization

As stated earlier, an important observation underlying our solution is that the integrity checker and the surrounding code base contain useful clues that can be used to guide search. Thus, our technique (1) performs static analysis to identify features that are likely to be used in the desired solution, and (2) initializes PCFG probabilities to prioritize programs that use these features.

4.4.1 Mining Features via Static Analysis. Our static analysis extracts three types of code elements (namely, functions, types, and operators) for assigning probabilities to productions. These code elements are extracted by analyzing the integrity checking function and the surrounding code base.

Analysis of integrity checking function. Our method statically analyzes the integrity checker Φ_c to identify (1) types of expressions used in Φ_c, (2) all operators (e.g., bit-shift, addition, etc.) that syntactically appear in Φ_c, and (3) functions that are invoked by Φ_c. Such code elements that syntactically appear in the integrity checker often also tend to appear in the synthesized code; thus, our technique assigns a higher probability to productions involving these code elements. (The mechanism for increasing probabilities is discussed in Section 4.4.2.)

Analysis of existing functions. Recall from Section 2 that existing functions in the code base may be useful for

updating the new fields. Thus, our method statically analyzes all existing functions to identify a subset of methods that return or update values of type τ, where τ is also the type of one of the new fields. In particular, our technique uses an off-the shelf pointer analysis to identify all memory locations of type τ that are updated by some function f. If τ is also the type of a new field, then function f is considered to be a promising candidate and the probability of the corresponding production is increased.

4.4.2 Initializing Probabilities. Given a set of "interesting" productions $\mathcal{R}'$ identified using static analysis, our method initializes PCFG probabilities as follows. First, let $\mathcal{R}_N$ denote the set of productions whose left-hand-side is non-terminal N, and let $\mathcal{R}'_N$ be $\mathcal{R}_N \cap \mathcal{R}'$. To initialize probabilities for productions in $\mathcal{R}_N$, we first define a normalization constant Z as follows:

$$Z = |\mathcal{R}_N \backslash \mathcal{R}'_N| + c_N |\mathcal{R}'_N|$$

where c_N is a constant strictly greater than 1. Then, we assign probabilities to productions $r \in \mathcal{R}_N$ as follows:

$$\mathbb{P}(r) = \begin{cases} 1/Z & \text{if } r \in (\mathcal{R}_N \backslash \mathcal{R}'_N) \\ c_N/Z & \text{if } r \in \mathcal{R}'_N \end{cases}$$

Theorem 4.7. *For each non-terminal N in the grammar, $\mathbb{P}(N)$ defines a valid probability distribution over $\mathcal{R}_N$.*

5 Implementation

We have implemented our proposed approach in a tool called Volt. The inputs to Volt include (1) the source code of a Java data structure D, (2) an integrity constraint implemented as a Java function, and (3) a set of new fields to be added to D. In addition, Volt also takes a time limit t (in seconds) indicating the maximum time it has to synthesize the desired function. Volt itself is implemented in Java and leverages the Z3 SMT solver [10] to check the feasibility of partial programs in the Deduce procedure and the JBMC assertion checker [9] for verification in the Verify procedure. In what follows, we discuss some key optimizations for the synthesis algorithm from Section 4.

Grammar productions. The grammar presented in Figure 3 is simple but unnecessarily permissive. In our implementation, we use a more fine-grained grammar that disallows enumerating obviously useless programs. In particular, our implementation restricts left-hand-side grammar expressions to new fields and fresh (temporary) variables. Second, it restricts method invocations to those that do not have side effects on existing fields. Third, it disallows atomic statements that call pure functions (since they are essentially no-ops). Fourth, it restricts loops to range-based for loops as arbitrary while loops are fairly uncommon compared to range-based counterparts. Finally, it restricts the set of local variables used in right-hand-side expressions to those that are in scope at the relevant program point. Observe that some of these restrictions require source code analysis; therefore our implementation leverages the Soot program analysis infrastructure [26] and the SPARK pointer analysis [31] to perform these optimizations.

Additional pruning strategies. Beyond using a grammar with auxiliary nonterminals, our implementation performs a few other optimizations to reduce search space. First, since multiple updates of a program variable along the same execution path are redundant, Volt disallows enumerating such partial programs. Second, it disallows loops that do not use the iterator in the body. Finally, since writing to temporary variables is only useful if there is a read afterwards, our implementation also avoids enumerating programs that write to, but do not read from, temporary variables.

6 Evaluation

In this section we describe a series of experiments that are designed to answer the following research questions:

1. **(RQ1)** Can Volt be used to refine data structures in real-world Java applications?
2. **(RQ2)** How does Volt compare against other approaches that could be used for solving the same problem?
3. **(RQ3)** How important is each of the three design decisions in Volt (i.e., use of PCFGs, necessary precondition computation, and grammar augmentation)?

Benchmarks. To evaluate Volt on real-world applications, we used a Github crawler to identify popular Java projects that use correlated fields for performance reasons. The crawler looks for commits with messages that match certain relevant keywords such as "performance", "cache", "new fields", etc. We then manually inspected projects returned by the crawler and retained the first 25 classes that indeed have multiple correlated fields. To evaluate Volt on these benchmarks, we manually removed all the declarations and statements involving correlated fields (except for one field), wrote an integrity checking function, and used Volt to automatically derive the original implementation. To determine which fields to remove we used three criteria. First, if fields $f_1, \ldots, f_n$ could be derived from f, we removed each f_i. Second, if the fields could be derived from each other we marked those added in later commits as the auxiliary fields and removed them. Finally if the fields were added in the same commit we used syntactic hints e.g. name of the field to break the tie. For example in our benchmark PersistentSequentialDictionary, one of the correlated fields was called "cache" and the other was called "reverseCache" and we marked the latter as auxiliary.

Setup. In our evaluation, we use a time limit of 1 hour and run all of our experiments on the Google Cloud Engine (GCE) on an 8 vcpu instance with 128GB of memory.

Table 1. Main experimental results. ⊥ indicates the tool timed out (> 1 hour) when solving the benchmark.

Project	Class	LoC	# Productions	# Corr. Fields	Total Funcs.	# Updated Funcs.	Volt
strapdata/elessandra	FieldData	102	1822	2	7	3	32.11
watabou/pixeldungeon	Level	1023	1044	3	35	3	28.24
netty/netty	DefaultChannelPipeline	1049	2150	2	127	6	93.33
netty/netty	SpdySession	361	1230	3	42	5	102.44
apache/wicket	RequestAdapter	171	1832	2	10	2	82.2
bisq-network/bisq	MathUtils	182	1150	2	12	2	125.4
apache/wicket	AsynchronousPageStore	397	2734	3	17	3	222.63
wakaleo/game-of-life	EndlessGrid	156	944	2	14	3	444.3
jenkinsci/gitlab-plugin	GitlabWebhook	350	1322	2	23	2	355.3
pravega/pravega	StreamSegmentContainerMetadata	314	1732	3	27	3	377.2
spring-cloud/spring-cloud-gcp	PartTreeDataStoreQuery	432	1655	2	30	3	822.2
apache/falcon	OozieWorkflowEngine	85	755	2	104	2	83.2
apache/falcon	ConfigurationStore	452	1134	2	30	4	192.2
javaparser/javaparser	LexicalPrinter	554	2215	3	122	3	⊥
jdbi/jdbi	ImmutablePropertiesFactory	1023	3255	2	82	3	663.2
jdbi/jdbi	RowView	197	1683	2	19	3	613.2
strapdata/elessandra	InternalIndexingStats	185	1332	3	11	3	344.8
jacoco/jacoco	MethodAnalyzer	350	1422	2	31	1	35.4
jetbrains/Xodus	PersistentSequentialDictionary	228	1772	3	17	2	143.2
OpenGamma/Strata	FxMatrix	557	1933	3	31	4	79.2
osmandapp/Osmand	GeocodingLookupService	287	1611	3	14	2	⊥
graylog2/graylog2-server	StreamCacheService	96	1033	3	17	2	99.3
spring-projects/spring-framework	DefaultListableBeanFactory	2153	4822	2	116	5	144.4
facebook/buck	DaemonicParserState	755	1933	4	33	6	455.4
junit-team/junit4	BlockJunit4ClassRunner	377	3255	2	33	2	415.3
Average	-	**495.3**	**1830.3**	**2.56**	**40.1**	**2.96**	**264.2**

Table 2. Baseline results. The Avg. Time is the average time over all the solved benchmarks (ignoring verification time) so timeouts do not contribute to the average time.

Tool	# Solved	Avg. Time
Volt	23	264.2
FrAngel	5	160.21
JSketch	2	1033.3

6.1 Main results

To answer our first research question, we evaluated whether Volt is able to automatically refine the benchmarks and how long it takes to do so. The results of this evaluation are summarized in Table 1. Here, the first two columns show the name of the class to be refined and the project it is taken from. The third column indicates the lines of code in the class and the fourth column shows the average number of *initial* productions in our PCFG. The next three columns provide information about the number of new fields to be added, the total number of functions defined in the class, and the number of functions that need to be updated. Finally, the last column shows the running time of Volt.

Overall, Volt is able to automatically refine 23 out of the 25 benchmarks (92%) within the provided time limit, and its average synthesis time is 264.2 seconds. Furthermore, we manually inspected the synthesized code and compared it against the human-written version. In all cases, we confirmed that the synthesized code is correct and matches the human-written code except for minor variations (e.g., `if(b) S1 else S2` vs. `if(!b) S2 else S1`).

Analysis of failed benchmarks. As shown in Table 1, there are two benchmarks that Volt failed to solve within the 1 hour time limit. In particular, Volt is unable to synthesize the desired update for classes GeocodingLookupService and DemonicParserState because the required update logic is very complex for some functions. For instance, a function in DemonicParserState requires adding 15 lines of code with over 150 AST nodes.

6.2 Comparison against baselines

To put these results in context and answer our second research question, we also evaluated Volt against existing tools. While there is no existing technique that addresses exactly our problem, we adapted two program synthesis tools to our problem setting:

- **Frangel:** FrAngel is a component-based synthesis tool that can synthesize code with loops and conditionals [37]. However, since FrAngel only handles input-output examples, we cannot directly use it to solve our problem. Thus, to adapt FrAngel to our setting, we used Volt's modular refinement algorithm (Algorithm 2) but replaced its inductive synthesis engine with FrAngel instead.
- **JSketch:** Our third baseline is JSketch [23], which is another state-of-the-art synthesizer for Java. Since JSketch can also not be used to directly solve our problem, we also perform this comparison by replacing Volt's inductive synthesis engine with JSketch.

Note that both of these baselines are not quite apples-to-apples comparisons as they actually utilize Volt's modular refinement procedure. Nonetheless, they serve as useful baselines for evaluating our proposed inductive synthesis algorithm (Algorithm 3).

As we can see from Table 2, JSketch can only solve 2 of the 25 benchmarks and is more than an order of magnitude

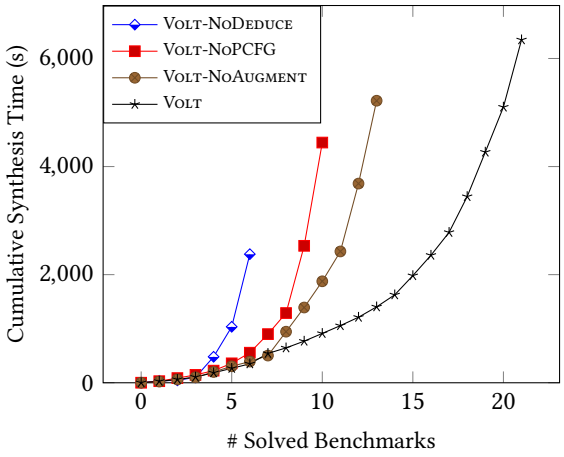

Figure 7. Comparing Volt to baselines.

slower than Volt for these two benchmarks. FrAngel solves 5 of the benchmarks but fails to solve the remaining 20 within the 1 hour time limit. These results indicate that state-of-the-art synthesis tools are not sufficient for solving our problem even when leveraging the modular refinement idea proposed in this paper.

6.3 Ablation Study

In this section, we present the results of an ablation study for answering our third research question. In this evaluation, we consider the following three ablations of Volt:

- Volt-NoPCFG: This is a variant of Volt that uses a CFG (instead of a PCFG). In particular, Volt-NoPCFG uses the integrity constraint to generate an initial CFG without probabilities, and augments it with new productions without probabilities during synthesis. In addition, it still compute preconditions to prune infeasible programs.
- Volt-NoAugment: This is a variant of Volt that does not perform grammar augmentation. That is, Volt-NoAugment starts with the same initial PCFG and performs deduction to prune infeasible programs; however, it does not use deduction to augment the grammar. In other words, the PCFG is not changed throughout.
- Volt-NoDeduce: This is a variant of Volt that does not compute necessary preconditions. As a result, it cannot perform pruning or grammar augmentation, and it only uses the initial PCFG during the entire synthesis process.

The results of this ablation study are summarized in Figure 7. Here, the x-axis shows the number of solved benchmarks (sorted in increasing order of synthesis time), and the y-axis denotes cumulative running time. As we can see, all variants perform significantly worse than Volt; however, the computation of necessary preconditions has the most impact among the three ablations. Overall, these results demonstrate that all three ideas used in our inductive synthesis algorithm are important for making this technique useful in practice.

7 Related Work

Data structure Repair and Verification. There is an extensive body of research on *runtime* detection and repair of data structures from arbitrary boolean constraints [12–15] starting from Demsky and Rinard [13]. Our work is similar to this line of research in that we expect users to provide integrity constraints over the fields of the data structure. However, our problem is fundamentally orthogonal as we want to statically update the data structure so that the integrity constraint holds whereas these approaches mutate the state of the data structure at runtime to satisfy the integrity constraints. There is a parallel line of research [16, 27, 28, 39] on statically verifying properties of data structures. Our implementation of Volt uses a bounded model checker instead of these verifiers since it needs to obtain counterexamples in the CEGIS loop.

Data Invariants. Our work is also related to prior research on maintaining data invariants [30, 36]. The most similar work to ours is Spyder [36]. Like Volt, Spdyer requires the developer to specify an invariant over fields of the data structure and afterwards it automatically synthesizes a new data structure where the invariants are maintained after each basic block. Unlike Volt, Spyder requires the invariants to be *iterator-based* and *alias-free*. In particular, the correlated fields must be iterator based data structures that are structurally similar (i.e. all are the same length) and the contents of the data structures cannot alias each other. The benefit of such restrictions is that it allows Spdyer to perform powerful and efficient transformations such as updating an existing loop header to simultaneously iterate over multiple structures. Volt, on the other hand, allows developers to specify arbitrary invariants so long as they can be encoded as a boolean function. As such, it can handle a much larger class of invariants that Spyder cannot. For example, Spdyer cannot synthesize the desired update in Figure 1 as the correlated fields are not iterator-based. In addition, the technical details of synthesis algorithms are completely different. Spdyer's technique is based on deductive synthesis whereas Volt performs inductive synthesis.

Data representation synthesis. Our work is related to a line of research on so-called *data representation synthesis* [11, 21, 22, 33, 41], where the goal is to synthesize a complete data structure from a specification. For example, Hawkins et al. [21] allow developers to specify data structures as a series of query and update relations, and their synthesis procedure uses rewrite rules along with deduction to generate the concrete implementation. Loncaric et al. [33] follow a similar procedure; however, they use enumerative search to generate the implementation for update relations. Unlike this line of research, our work focuses on cases where developers refine an existing implementation by adding correlated fields. Thus, developers only need to provide a simple integrity

checking function as opposed to a complete implementation in a specification language. Furthermore, our approach can perform an in-place update as opposed to synthesizing the complete data structure from scratch.

Synthesis using probabilistic models. There are several prior techniques that use *probabilistic models* to guide their search [6–8, 25, 29, 38]. Most of these techniques learn a static PCFG through offline training [6, 29] whereas our approach uses static analysis to initialize probabilities and updates them on the fly. Barke et al. [7] also update PCFG probabilities by identifying programs that satisfy some of the input-output examples. However, in contrast to Barke et al. [7], our approach uses deduction to identify promising program sketches and augments the grammar with those productions in addition to updating probabilities. In addition, Concord [8] also combines probabilistic models with deduction; however, it uses deduction to compute a reward for reinforcement-learning guided synthesis, whereas our approach uses deduction to augment the PCFG.

Component-based synthesis. There has been a long line of research on synthesizing programs from a set of components such as library functions [17–20, 24, 34, 37, 40]. Among these component-based synthesis approaches, the most related one is FrAngel [37], which can also synthesize control flow constructs like conditionals and loops. FrAngel is particularly related to our approach in that it learns new components at synthesis time by composing existing components. This is similar to our approach in that we also augment the grammar with new productions. However, a key difference from FrAngel is that Volt uses program analysis and deductive reasoning to learn these productions; furthermore, the new "components" Volt learns are program sketches rather than complete programs. As we show experimentally in Section 6, our proposed inductive synthesis approach significantly outperforms FrAngel in this context.

Acknowledgments

We thank the anonymous reviewers for the helpful feedback. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1908304, CCF-1811865, and CNS-1514435. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the funding sources.

8 Conclusion

We have presented Volt, a tool for refining data structure implementations from integrity constraints. Volt is based on program synthesis and uses a modular instantiation of the CEGIS paradigm powered by a novel inductive synthesizer that incorporates three key ideas: (1) pruning using necessary preconditions, (2) deduction-guided grammar augmentation, and (3) PCFG construction using static analysis. We have evaluated Volt on 25 real-world Java classes with correlated fields and show that Volt can successfully refine 23 out of these 25 (92%) benchmarks. We also compared Volt against other state-of-the-art synthesis tools for Java and showed that our closest competitor can only solve 20% of the benchmarks (despite already incorporating the modular aspect of Volt). We also present several ablations of Volt and show that our main ideas are all crucial for making the proposed approach feasible in practice.

References

[1] [n.d.]. CVE-2005-0034. https://nvd.nist.gov/vuln/detail/CVE-2005-0034.
[2] [n.d.]. CVE-2010-1013. https://nvd.nist.gov/vuln/detail/CVE-2010-1013.
[3] [n.d.]. CVE-2016-5195. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-5195.
[4] [n.d.]. CVE-2017-7308. https://nvd.nist.gov/vuln/detail/CVE-2017-7308.
[5] [n.d.]. Netty. https://github.com/netty/netty.
[6] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. arXiv:1611.01989 [cs.LG]
[7] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-Time Learning for Bottom-Up Enumerative Synthesis.
[8] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. 2020. *Program Synthesis Using Deduction-Guided Reinforcement Learning*. 587–610. https://doi.org/10.1007/978-3-030-53291-8_30
[9] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. 2018. JBMC: A bounded model checking tool for verifying Java bytecode. In *International Conference on Computer Aided Verification*. Springer, 183–190.
[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[11] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proc. of POPL*. 689–700.
[12] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. 2006. Inference and Enforcement of Data Structure Consistency Specifications. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis* (Portland, Maine). 233–244.
[13] Brian Demsky and Martin C. Rinard. 2003. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Anaheim, California). 78–95.
[14] Brian Demsky and Martin C. Rinard. 2003. Static Specification Analysis for Termination of Specification-Based Data Structure Repair. In *Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering* (Denver, Colorado). 71–84.
[15] Brian Demsky and Martin C. Rinard. 2005. Data Structure Repair Using Goal-Directed Reasoning. In *Proceedings of the 2005 International Conference on Software Engineering* (St. Louis, Missouri). 176–185.
[16] Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Precise Reasoning for Programs Using Containers. 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, 187–200. https://doi.org/10.1145/1926385.1926407

[17] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of PLDI*. 420–435.

[18] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of PLDI*. 422–436.

[19] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. In *Proc. of POPL*. 599–612.

[20] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proc. of PLDI*. 229–239.

[21] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2011. Data Representation Synthesis. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 38–49. https://doi.org/10.1145/1993498.1993504

[22] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2012. Concurrent Data Representation Synthesis. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 417–428. https://doi.org/10.1145/2254064.2254114

[23] Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. 2015. JSketch: sketching for Java. In *Proc. of ESEC/FSE*. 934–937.

[24] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proc. of ICSE*. 215–224.

[25] Manos Koukoutos, Mukund Raghothaman, Etienne Kneuss, and Viktor Kuncak. 2017. On Repair with Probabilistic Attribute Grammars. (07 2017).

[26] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective.

[27] Patrick Lam, Viktor Kuncak, and Martin Rinard. 2005. Generalized Typestate Checking for Data Structure Consistency. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation* (Paris, France) *(VMCAI'05)*. Springer-Verlag, Berlin, Heidelberg, 430–447. https://doi.org/10.1007/978-3-540-30579-8_28

[28] Patrick Lam, Viktor Kuncak, and Martin Rinard. 2005. Hob: A Tool for Verifying Data Structure Consistency. https://doi.org/10.1007/978-3-540-31985-6_16

[29] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating Search-Based Program Synthesis Using Learned Probabilistic Models *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 436–449. https://doi.org/10.1145/3192366.3192410

[30] K. Rustan M. Leino and Peter Müller. 2004. Object Invariants in Dynamic Contexts. In *ECOOP 2004 – Object-Oriented Programming*, Martin Odersky (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 491–515.

[31] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using S park. In *International Conference on Compiler Construction*. Springer, 153–169.

[32] Boyang Li, Isil Dillig, Thomas Dillig, K. McMillan, and S. Sagiv. 2013. Synthesis of Circular Compositional Program Proofs via Abduction. In *TACAS*.

[33] Calvin Loncaric, Michael D. Ernst, and Emina Torlak. 2018. Generalized Data Structure Synthesis. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 958–968. https://doi.org/10.1145/3180155.3180211

[34] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. 2019. Trinity: An Extensible Synthesis Framework for Data Science. *PVLDB* 12, 12 (2019), 1914–1917.

[35] Kenneth L. McMillan. 1999. Circular Compositional Reasoning about Liveness. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '99)*. Springer-Verlag, Berlin, Heidelberg, 342–345.

[36] John Sarracino, Shraddha Barke, Nadia Polikarpova, and Sorin Lerner. 2019. Targeted Synthesis for Programming with Data Invariants. *CoRR* abs/1904.13049 (2019). arXiv:1904.13049 http://arxiv.org/abs/1904.13049

[37] Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: component-based synthesis with control structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 73:1–73:29.

[38] Xujie Si, Y. Yang, Hanjun Dai, M. Naik, and L. Song. 2019. Learning a Meta-Solver for Syntax-Guided Program Synthesis. In *ICLR*.

[39] Philippe Suter, Mirco Dotta, and Viktor Kuncak. 2010. Decision Procedures for Algebraic Data Types with Abstractions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 199–210. https://doi.org/10.1145/1706299.1706325

[40] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of PLDI*. 452–466.

[41] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing database programs for schema refactoring. In *Proceedings of PLDI*. 286–300.