Semantic Code Refactoring for Abstract Data Types

SHANKARA PAILOOR, University of Texas, USA
YUEPENG WANG, Simon Fraser University, Canada
IŞIL DILLIG, University of Texas, USA

Modifications to the data representation of an abstract data type (ADT) can require significant semantic refactoring of the code. Motivated by this observation, this paper presents a new method to automate semantic code refactoring tasks. Our method takes as input the original ADT implementation, a new data representation, and a so-called *relational representation invariant* (relating the old and new data representations), and automatically generates a new ADT implementation that is semantically equivalent to the original version. Our method is based on counterexample-guided inductive synthesis (CEGIS) but leverages three key ideas that allow it to handle real-world refactoring tasks. First, our approach reduces the underlying relational synthesis problem to a set of (simpler) programming-by-example problems, one for each method in the ADT. Second, it leverages symbolic reasoning techniques, based on logical abduction, to deduce code snippets that should occur in the refactored version. Finally, it utilizes a notion of *partial equivalence* to make inductive synthesis much more effective in this setting. We have implemented the proposed approach in a new tool called Revamp for automatically refactoring Java classes and evaluated it on 30 Java class mined from Github. Our evaluation shows that Revamp can correctly refactor the entire ADT in 97% of the cases and that it can successfully re-implement 144 out of the 146 methods that require modifications.

CCS Concepts: • **Software and its engineering → Abstract data types**.

Additional Key Words and Phrases: Program Synthesis, Abstract Data Types, Refactoring

ACM Reference Format:
Shankara Pailoor, Yuepeng Wang, and Işıl Dillig. 2024. Semantic Code Refactoring for Abstract Data Types. *Proc. ACM Program. Lang.* 8, POPL, Article 28 (January 2024), 32 pages. https://doi.org/10.1145/3632870

1 INTRODUCTION

Abstract data types (ADTs) separate the software interface from its underlying data representation, allowing code modifications that are hidden from clients. However, even small changes to the data representation can require substantial modifications to the underlying *implementation* of the ADT. As an example, consider the code shown on the left-side of Figure 1, which is taken from the BitmapTracker ADT in Glide [gli 2023], a popular image loading and caching library for Android. The original implementation of this ADT uses two data structures: a set (`pending`) to keep track of bitmaps that are pending deletion (represented by their hash code), along with a separate data structure (`cntr`) to keep track of bitmaps and the number of times they have been acquired. Rather than maintaining this information across two data structures, the developers decide to consolidate them as a single hash map also named `cntr`. The new data structure associates each bitmap with a newly defined `InnerTracker` object which has a field (`pending`) to keep track of whether the bitmap is pending deletion and a field (`refs`) to store the number of times the bitmap has been acquired. As is evident from the "diff" in Figure 1, making this change also necessitates substantial

Authors' addresses: Shankara Pailoor, spailoor@cs.utexas.edu, University of Texas, Austin, USA; Yuepeng Wang, yuepeng@sfu.ca, Simon Fraser University, Vancouver, Canada; Işıl Dillig, isil@cs.utexas.edu, University of Texas, Austin, USA.

ACM 2475-1421/2024/1-ART28
https://doi.org/10.1145/3632870

```
public class BitmapTrackerOrig {
    public Set<Integer> pending;
    public BitmapReferenceCounter cntr;

    public static class BitMapReferenceCounter() {
        public  Map<Integer, Integer>  cntrs = new HashMap<>();
        ....
    }

    public BitmapTrackerOrig() {
        this.cntr = new BitmapReferenceCounter();
        this.pending = new HashSet<>();
    }

    public void acquireBitmap(Bitmap bitmap) {
        int hashCode = bitmap.hashCode();
        pending.remove(hashCode);
        cntr.inc(hashCode);
    }

    public void releaseBitmap(Bitmap bitmap) {
        int hashCode = bitmap.hashCode();
        if (cntr.dec(hashCode) == 0
            &&!pending.contains(hashCode)) {
            cntr.rem(hashCode);
        }
    }

    public void rejectBitmap(Bitmap bitmap) {
        int hashCode = bitmap.hashCode();
        pending.remove(hashCode);
        if (cntr.get(hashCode) == 0) {
            cntr.rem(hashCode);
        }
    }

    public void markPending(Bitmap bitmap) {
        int hashCode = bitmap.hashCode();
        if (!cntr.cntrs.containsKey(hashCode))
            return;
        pending.add(hashCode);
    }
```

```
public class BitmapTrackerNew {
    public Map<Integer, InnerTracker> cntr;

    public static class InnerTracker {
        public int refs = 0;

        public boolean pending = false;
        ...
    }

    public BitmapTrackerNew() {
        this.cntr = new HashMap<>();
    }

    public void acquireBitmap(Bitmap bitmap) {
        int hashCode = bitmap.hashCode();
        InnerTracker tracker = cntr.get(hashCode);
        if (tracker == null) {
            tracker = new InnerTracker();
        }
        tracker.acquire();
        cntr.put(bitmap.hashCode(), tracker);
    }

    public void releaseBitmap(Bitmap bitmap) {
        int hashCode = bitmap.hashCode();
        InnerTracker tracker = cntr.get(hashCode);
        if (tracker != null) {
            if (tracker.release()) {
                cntr.remove(hashCode);
            }
        }
    }

    public void rejectBitmap(Bitmap bitmap) {
        int hashCode = bitmap.hashCode();
        InnerTracker tracker = cntr.get(hashCode);
        if (tracker != null) {
            if (tracker.reject()) {
                cntr.remove(hashCode);
            }
        }
    }

    public void markPending(Bitmap bitmap) {
        int hashCode = bitmap.hashCode();
        InnerTracker tracker = cntr.get(hashCode);
        if (tracker != null) {
            tracker.markPending();
        }
    }
```

Fig. 1. Example ADT refactoring to motivating our technique

modifications to the code of the BitmapTracker ADT. More generally, such code refactorings can be quite involved and sometimes even introduce subtle bugs and security vulnerabilities [cve 2003, 2005; ref 2009, 2022].

```
1 public boolean check(BitmapTrackerOrig o, BitmapTrackerNew n) {
2     if (!o.cntr.cntrs.keySet().equals(n.cntr.keySet())
3         return false;
4     for(Entry c : o.cntr.cntrs) {
5         InnerTracker inner = n.cntr.get(c.getKey());
6         if (inner.refs != c.getValue()) return false;
7         if (inner.pending != pending.contains(c.getKey()))
8             return false;}
9     return true;
10 }
```

Fig. 2. Relational representation invariant.

Motivated by this problem, the goal of this paper is to automate this *semantic code refactoring* task for abstract data types. Given the original ADT implementation and a *relational specification* relating the old and new data representations, our method automatically synthesizes the new ADT implementation from its original version. Because such relational specifications can be easily expressed as a simple boolean function (e.g., see Figure 2), our method can greatly simplify the ADT refactoring task compared to manually changing the implementation. Furthermore, this automated refactoring approach can eliminate subtle bugs that may be introduced during the manual refactoring process.

To gain some intuition about the relational specifications required by our method, consider the boolean `check` procedure shown in Figure 2. At a high level, this method describes the refactoring task for the `BitmapTracker` ADT by providing a *relational representation invariant (RRI)*, which is similar to the standard notion of *representation (rep) invariant* [Delaware et al. 2015; Guttag et al. 1978; Miltner et al. 2020]. Just as a rep invariant checks whether an ADT instance obeys key data integrity constraints, an RRI checks key data integrity constraints *between* two alternative representations of an ADT. For example, going back to our running `BitmapTracker` example, the `check` function in Figure 2 states the following relationship between the original fields and new one:

(1) The map (`cntr.cntrs`) in the original implementation and map (`counter`) in the new version must have the same keys;
(2) For every (id, count) entry in the original map (`cntr.cntrs`), there should exist an entry (id, tracker) in the new map (`cntr`) such that count = tracker.refs, and tracker.pending is true if and only if `pending` contains id.

Given such a relational representation (expressed as a boolean function), the goal of our method is to automatically generate the code shown on the right hand side of Figure 1 from its original version on the left.

The key contribution of this paper is a novel program synthesis technique for solving this problem. Despite being an instantiation of the popular counterexample-guided inductive synthesis (CEGIS) paradigm at a high level, our synthesis approach utilizes three novel insights that allow automating real-world ADT refactoring tasks:

- **Idea #1: Specification strengthening:** When the verifier fails to prove equivalence between the original implementation and a candidate synthesis result, it can provide a counterexample in the form of a disequality $f(I) \neq O$, meaning that the implementation of function f should not produce ADT instance O when executed on input I. However, because this feedback is very weak, the CEGIS loop can take many iterations to converge. Our approach addresses this problem by utilizing the semantics of the RRI to strengthen the specification into equalities (i.e., input-output examples) rather than disequalities.
- **Idea #2: Mining code snippets via symbolic reasoning:** Our approach leverages symbolic reasoning techniques, based on *logical abduction*, to identify key building blocks that are likely to

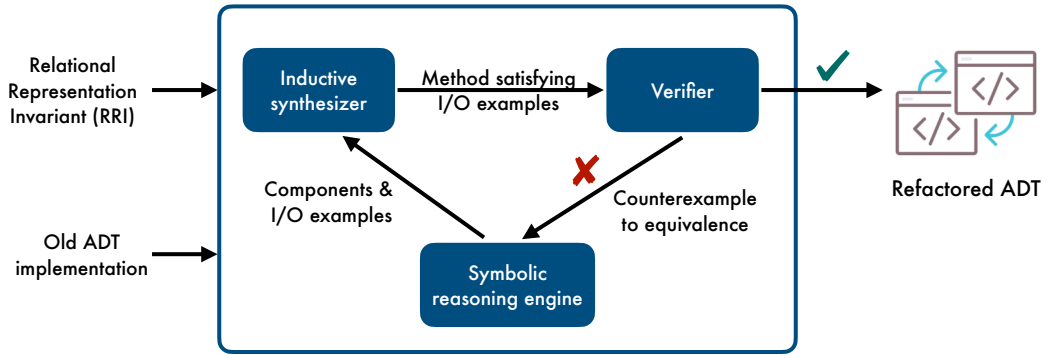

Fig. 3. Overview of our approach

be used in the refactored implementation. Because the identified code snippets can be complex statements or expressions, symbolic reasoning can dramatically reduce the search space that the synthesizer needs to explore.

- **Idea #3: Exploiting partial equivalence:** An inductive synthesizer typically enumerates many incorrect programs before it finds the target implementation. In our context, these enumerated programs are often not *completely* equivalent to the original implementation, but they are *partially equivalent* with respect to a *subset* of the ADT fields. Our other key insight is to leverage this notion of *partial equivalence* to progressively build up larger programs from smaller ones that are correct with respect to *some* ADT fields.

Figure 3 gives a schematic overview of our high-level approach. As shown in this figure, our technical approach is an instance of CEGIS and incorporates an inductive synthesizer and a verifier. In this context, the inductive synthesizer takes as input (1) a set of input-output examples E (for each method m in the ADT) and (2) a grammar defining its search space, and it outputs a new method implementation m' satisfying all examples in E. The verifier is then tasked with checking whether m and m' are equivalent *modulo* the user-specified RRI. If verification succeeds, m' is added to the refactored implementation of the ADT and the algorithm moves on to the next method. On the other hand, if verification fails, the verifier outputs a *counterexample to equivalence*, which is a pair of inputs I, I' for m and m' such that I, I' satisfy the RRI but $m(I)$ and $m'(I')$ do *not* satisfy it. This counterexample is then provided as input to the *symbolic reasoning engine*, which is one of the key novelties of our technique. In particular, the symbolic reasoning engine performs two functions: First, it converts the counterexample to equivalence produced by the verifier to a concrete input-output example for the target function by utilizing the semantics of the RRI (i.e., **Idea #1**). This specification strengthening idea essentially allows converting a weak disequality of the form $m'(I') \neq O_1$ to an equality $m'(I') = O_2$, thereby allowing the CEGIS loop to make much faster progress. Second, it uses the counterexample, together with the RRI and the implementation of m, to infer code snippets that are *likely* to be used in the refactored implementation (i.e., **Idea #2**). As stated earlier, these inferred snippets are useful because they allow the inductive synthesizer to leverage complex expressions as components rather than having to search for them from scratch.

The other novel aspect of our ADT refactoring algorithm is the inductive synthesis engine depicted in Figure 4. Similar to many other inductive synthesizers, our method is based on enumerative search; however, it utilizes the notion of *partial equivalence* (i.e., **Idea #3**) to perform bi-directional search [Alur et al. 2015; Lee 2021; Shi et al. 2019]. In more detail, the search engine performs top-down enumeration by maintaining a worklist of *partial programs* that are gradually expanded to form complete programs. The key difference, however, is that, rather than *discarding*

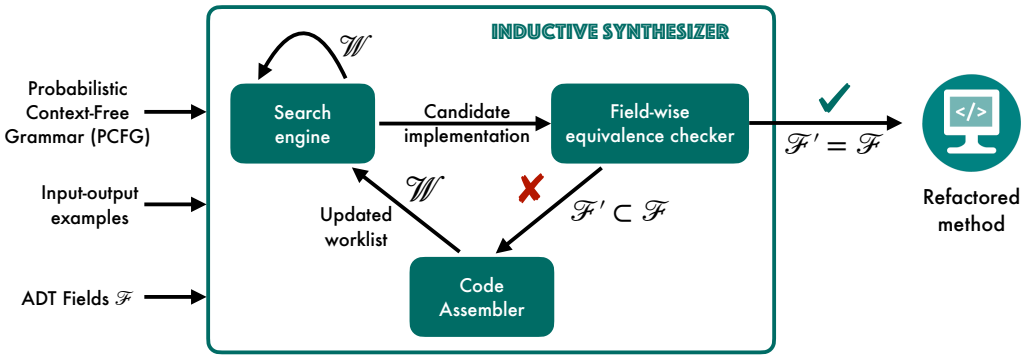

Fig. 4. Overview of our inductive synthesizer

complete programs that are inconsistent with the input-output examples, our approach *retains* those programs that produce the intended output with respect to a *subset* of the ADT fields. To do so, it injects these "partially equivalent" programs into the frontier of the search engine by combining them with existing programs in the worklist (i.e., Code Assembler component in Figure 4). Hence, this strategy combines top-down enumeration with bottom-up search and extends the idea of *partial satisfaction* used in prior work [Lee 2021; Shi et al. 2019] to the program equivalence setting.

We have implemented our proposed approach in a tool called Revamp targeting Java programs, and we use Revamp to perform semantic ADT refactoring tasks mined from GitHub commits. Our benchmarks span 30 different ADTs and require re-implementing a total of 146 methods. Our experiments show that Revamp can successfully refactor 29 of the 30 (97%) ADTs and reimplement 144 of these methods (99%). We also compare Revamp against other synthesis tools and show that it significantly outperforms these baselines, both in terms of the number of tasks it can solve as well as average synthesis time.

To summarize, this paper makes the following key contributions:

- We introduce the semantic ADT refactoring problem as the task of synthesizing the new ADT implementation from its original version and a *relational representation invariant*.
- We propose a novel technique that combines symbolic reasoning (based on logical abduction) with counterexample-guided inductive synthesis to derive the refactored ADT implementation.
- We show how to reduce the ADT refactoring problem to a set of *programming-by-example (PBE)* tasks (one for each method of the ADT), and we present an effective inductive synthesis approach that leverages the notion of *partial equivalence*.
- We perform an empirical evaluation of our approach on 30 real-world ADTs (spanning 146 refactored methods) mined from GitHub. Our results show that our tool, Revamp, can correctly refactor 99% of the methods and successfully generate the entire ADT in 97% of the cases.

2 OVERVIEW

In this section, we give an overview of our technique through an illustrative example.

Refactoring task. Figure 5 presents the refactoring of a simple 2D Rectangle ADT. This abstract data type exposes several methods that allow users to create, modify, and query information about the rectangle such as scaling and flipping it or getting the minimum x coordinate of any point on the rectangle. The class `RectOrig` (upper left box) is the original implementation that represents the rectangle using three fields, namely the rectangle's lower left corner (`lc`), height (`height`), and width (`width`). Now, suppose that the developer wishes to change the data representation to instead

Original Implementation

```
1 public class RectOrig {
2   Point2D lc; // left corner
3   double height, width;
4
5   public double getMinX() {..}
6   public double getMinY() {..}
7   ..
8   public RectOrig(Point2D lc, double height, width){..}
9
10  public void scaleAndFlip(double h, double w,
11                           bool flipX, bool flipY)
12  {
13    if (height - 2*h < 0 || width - 2*w < 0)
14      return;
15    lc.x = lc.x + w;
16    lc.y = lc.y + h;
17    height = height - 2*h;
18    width = width - 2*w;
19    if (flipX)
20      lc.x = -(lc.x + width);
21    if (flipY)
22      lc.y = -(lc.y + height);
23  }
23}
```

Relational Representation Invariant

```
1 bool check(RectOrig o, RectNew r) {
2   if (o.height <= 0 || o.width <= 0)
3     return false;
4   return o.width = r.maxX - r.minX
5            && o.height = r.maxY - r.minY
6            && o.lc.x = r.minX
7            && o.lc.y = r.minY;
9 }
```

New Implementation

```
1 public class RectNew {
2   double minX, maxX;
3   double minY, maxY;
4
5   public double getMinX() {..}
6   public double getMinY() {..}
7   ..
8   public RectNew(Point2D lc, double height, width){..}
9
10  public void scaleAndFlip(double h, double w,
11                           bool flipX, bool flipY)
12  {
13    if (maxY - minY - 2*h < 0
14              || maxX - minX - 2*w < 0)
15      return;
16    maxX = maxX - w;
17    minX = minX + w;
18    maxY = maxY - h;
19    minY = minY + h;
20    if (flipX) {
21      double tmp = maxX;
22      maxX = -minX;
23      minX = -tmp;
24    }
25    if (flipY) {
26      double tmp = maxY;
27      maxY = -minY;
28      minY = -tmp;
29    }
30  }
31}
```

Fig. 5. Motivating example showing the original implementation (upper left), the RRI (the lower left), and the refactored implementation (right).

use four fields that store the minimum and maximum x and y coordinates of the rectangle, as shown on the right side of Figure 5. As we can see by comparing the implementation of `RectOrig` and `RectNew`, changing the data representation induces significant modifications to the code.

Using Revamp. Our tool, Revamp, is designed to automatically perform this refactoring task given a relational representation invariant (RRI) provided by the user. The boolean function `check` (bottom left of Figure 5) corresponds to exactly such an RRI and states the following relationship between the fields of `RectOrig` and `RectNew`:

(1) The width (resp. height) of the rectangle should be equal to the difference between the maximum and minimum x (resp. y) coordinates.
(2) The x (resp. y) coordinate of the left corner should be equal to the `minX` (resp. `minY`) coordinate.

Given such an RRI, Revamp synthesizes new implementations for all the methods of the rectangle ADT. For example, Figure 5 shows the new implementation of the `scaleAndFlip` method that is automatically synthesized by Revamp.

Our approach. We now give an overview of the salient aspects of our approach. Our solution independently synthesizes each method of the ADT using the well-known CEGIS paradigm [Jha et al. 2010; Solar-Lezama et al. 2005, 2006], but it leverages three key observations that we illustrate through the running example of Figure 5.

> ***Observation #1:*** *Given a suitable RRI, we can reduce the ADT refactoring task to a set of* programming-by-example *problems, one for each method of the ADT.*

Suppose that, for a given method m of the original ADT, we attempt to synthesize a method m' of the new ADT but find that m' is not equivalent to m. In the CEGIS paradigm, we would ask the verifier to provide a counterexample. In this case, a counterexample is a pair of input ADT instances I, I' for m and m' respectively. Since the candidate implementation is wrong, we know the following two facts:

(1) I and I' must satisfy the RRI; otherwise, the verifier output is not a valid counterexample;
(2) The output ADTs, $m(I)$ and $m'(I')$, do not satisfy the RRI.

Now, what can we learn from such a counterexample? Naively, we can simply learn that the refactored implementation should not produce output O' when executed on I'. In other words, we can learn $m'(I') \neq O'$. However, this is a very weak signal for the inductive synthesizer — it simply states that the output must differ from O', but, since an ADT typically has *many* fields, this specification only rules out a tiny fraction of the behaviors that should be exhibited on input I'.

One of our key observations is that, under certain realistic assumptions about the RRI, we can use the verifier's output to learn not only what the target implementation should *not* produce on a given input, but rather what it *should* produce. That is, rather than learning a disequality, we can learn an *equality*, which corresponds to an input-output example for the target method and serves as a much stronger signal for the inductive synthesizer.

To understand why we can do this, suppose that we have synthesized a wrong implementation of `scaleAndFlip` and suppose that the synthesized code differs from the original version for the following input ADT I (for the original version):

$$I = \{lc.x = -1, lc.y = -1, width = 2, height = 2\} \tag{1}$$

and input ADT I' for the refactored version:

$$I' = \{minX = -1, minY = -1, maxX = 1, minX = 1\} \tag{2}$$

as well as the following function arguments:

$$V = \{h = 0.5, w = 0.5, flipX = false, flipY = false\} \tag{3}$$

Given this input ADT I and arguments V, the original `scaleAndFlip` implementation produces the output ADT O:

$$O = \{lc.x = -0.5, lc.y = -0.5, width = 1, height = 1\} \tag{4}$$

Now, by considering this output O *in conjunction* with the semantics of the RRI, we can determine precisely what the refactored method should return. Specifically, because the RRI specifies that `minX` and `minY` of the new rectangle must equal `lc.x` and `lc.y` of the original rectangle respectively, we know the refactored implementation must produce an ADT where `minX` and `minY` are both equal to -0.5. Similarly, because `maxX` and `maxY` in the new implementation are completely determined by variables `lc`, `height`, and `width` in the original implementation, we can infer (using an SMT solver) that $maxY = 0.5$ and $maxX = 0.5$. This observation allows us to obtain the following output O' for the refactored version of `scaleAndFlip`:

$$O' = \{minX = -0.5, minY = -0.5, maxX = 0.5, maxY = 0.5\} \tag{5}$$

> ***Observation #2:*** *We can use symbolic reasoning to learn useful code snippets that are likely to be used in the refactored implementation.*

Our second observation is that the semantics of the RRI is *not only* useful for specification strengthening *but also* for learning code snippets that the inductive synthesizer should use. To gain intuition, let us examine the execution path taken by the original `scaleAndFlip` method given the input ADT I and argument values V from Equations 1 and 3. This input exercises the following path P in the original implementation:

```
assume(height - 2*h >= 0);
assume(width - 2*w >= 0);
lc.x := lc.x + w;
```

```
lc.y := lc.y + h;
height := height - 2h;
width := width - 2w;
assume(!flipX);
assume(!flipY);
```

Now, we know from the RRI that `lc.x` is equal to `minX` and that `lc.y` is equal to `minY`. Hence, it is fairly easy to deduce that there must be a corresponding execution path of the refactored implementation that contains the statements `minX := minX + w` as well as `minY := minY + h`.

But what can we deduce about `maxX` and `maxY` of the refactored ADT? Unlike `minX` and `minY`, there is no obvious mapping from `maxX` and `maxY` to one of the variables in the original implementation. Nonetheless, we can use symbolic reasoning to infer how `maxX` and `maxY` should be updated in the corresponding execution path P' of the refactoring. To see how, observe that P and P' must satisfy the following Hoare triple:

$$\{height = maxY - minY\} \;\; P; P' \;\; \{height = maxY - minY\}$$

because the RRI stipulates that *height* is equal to $maxY - minY$. Since we have already established that P' must contain the statement `minY := minY + h` per the discussion above, it becomes clear that P' should also contain the statement `maxY := maxY+ h` in order for the above Hoare triple to be valid. Using similar reasoning, we can infer that P' should also update `maxX` using the statement `maxX := maxX + w`.

As illustrated through this example, we can symbolically deduce code snippets that the refactored implementation should contain by considering the RRI in conjunction with the original implementation. In Section 4.4, we show how this kind of deduction can be performed using a combination of symbolic execution and logical abduction.

> ***Observation #3:*** *We can leverage* partial equivalence *with respect to ADT fields to make inductive synthesis more effective.*

Consider the refactored implementation of `scaleAndFlip`, shown on the right side of Figure 5, which performs multiple distinct updates to each of the four fields of the new ADT. Our key observation here is that field updates are oftentimes independent of each other, meaning that the new value of a field *often* does not depend on the values of several other fields. For example, in the refactored code, the update on `maxX` is completely independent of the update to `maxY`. However, the refactored implementation is only correct when the updates on all fields are correct. As a result, a search-based synthesizer might discard candidate synthesis results even when it produces the correct implementation when considering a *subset* of the fields.

To gain more intuition about this idea, consider a variant of the `scaleAndFlip` implementation on the right side of Figure 5 with all updates to `maxY` and `minY` deleted. While this implementation is *not* equivalent to the original version, it *is* equivalent when we only consider the values of fields `maxX` and `minX`. We refer to this weaker notion of equivalence with respect to a subset of the fields as *partial equivalence*, and we leverage this concept to make inductive synthesis more efficient by combining top-down search with bottom-up synthesis.

For example, suppose we encounter a code snippet $maxX := maxX - w$ that correctly updates the $maxX$ field. Later, when we enumerate the partially equivalent program $maxY := maxY - h$, we can combine it with $maxX := maxX - w$ to obtain a larger code snippet that correctly updates both fields. Because there are several reasonable ways to combine code snippets, our approach considers several combinations, such as:

```
1  maxX := maxX - w;
2  maxY := maxY - h;
```

as well as

```
1  if (??) {
2     maxX := maxX - w;
3  } else {
4     maxY := maxY - h;
5  }
```

By identifying and combining code snippets that are partially equivalent to the original code, we can often quickly build up the correct refactored implementation.

3 PROBLEM STATEMENT

We represent an abstract data type (ADT) $\mathcal{A}$ as a set $\Sigma_{\mathcal{A}}$ of operations (methods) that can be invoked on instances of that type. For example, a `Stack` ADT is represented using the signature $\Sigma = \{$`push`, `pop`, `top`, `empty`, `constructor`$\}$. Every element $m \in \Sigma$ has its own signature $\mathcal{A} \times T \rightarrow \mathcal{A} \times T'$. That is, every method takes as input an argument of type T and instance of the ADT and returns an output of type T', along with a (possibly modified) instance of that ADT.

An *implementation* $I_{\mathcal{A}}$ of an ADT $\mathcal{A}$ is a tuple (F, M) where F is a set of fields (the *data representation*) and M is a mapping from every element $m \in \Sigma_{\mathcal{A}}$ to its concrete implementation m_I. We write $o \in I_{\mathcal{A}}$ to denote instances of $I_{\mathcal{A}}$. Also, we use the notation $\mathcal{A}_F$ to denote any implementation of $\mathcal{A}$ with data representation F, and we write $o \in \mathcal{A}_F$ to denote that o is an instance of some $I_{\mathcal{A}} = (F, \_)$. Since different implementations of an ADT can use different data representations, we next introduce the notion of *relational representation invariant (RRI)*:

*Definition 3.1 (***Relational Representation Invariant (RRI)***).* Let F and F' be two different data representations of the same ADT $\mathcal{A}$ and let $\sim \subseteq \mathcal{A}_F \times \mathcal{A}_{F'}$ be a binary relation. We say that $\sim$ is a *relational representation invariant* between $\mathcal{A}_F$ and $\mathcal{A}_{F'}$ if it has the following properties:

$$\forall o \in \mathcal{A}_F. \forall o_1, o_2 \in \mathcal{A}_{F'}.\ o \sim o_1 \wedge o \sim o_2 \Rightarrow o_1 = o_2 \tag{6}$$

$$\forall o \in \mathcal{A}_F. \exists o' \in \mathcal{A}_{F'}.\ o \sim o' \tag{7}$$

Intuitively, Equations 6 and 7 (henceforth jointly referred to as the **RRI property**) state that the binary relation should be precise enough so that, for any instance o of $\mathcal{A}_F$, we can reconstruct a unique instance o' of $\mathcal{A}'_F$ satisfying $o \sim o'$. Note that this RRI property is a basic requirement for being able to automate the refactoring task. Without such a property, there may be multiple implementations of the new ADT that will satisfy the RRI but yield semantically different ADT instances. In such a case, it is unclear which refactoring the programmer actually intended, so we require the RRI to satisfy Equations 6 and 7.

Theorem 3.2. *Let F and F' be two different data representations of the same ADT $\mathcal{A}$ and let $\sim$ be a relational representation invariant between $\mathcal{A}_F$ and $\mathcal{A}_{F'}$. Then for any instance $o \in \mathcal{A}_F$ there is a unique $o' \in \mathcal{A}_{F'}$ such that $o \sim o'$.*

Proof. From Equation 7, it follows there exists o' such that $o \sim o'$. To show uniqueness, consider an o'' such that $o \sim o'$ and $o \sim o''$. Then from equation 6, we have that $o' = o''$. □

Next, we define equivalence modulo an RRI:

*Definition 3.3 (***Input/output equivalence modulo RRI***).* Let o, o' be a pair of ADT instances over data representations F, F' respectively, and let v, v' be the arguments or return value of a

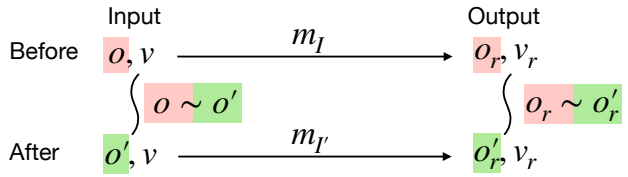

Fig. 6. Illustration of correctness of ADT refactoring. Objects in pink denote the version before refactoring, and objects in green denote the version after refactoring.

function. We say that (o, v) is equivalent to (o', v') modulo an RRI $(\sim)$, denoted $(o, v) \simeq (o', v')$ iff $o \sim o'$ and $v = v'$.

*Definition 3.4 (***Method equivalence modulo RRI***).* Let $\mathcal{A}$ be an ADT containing method signature $m : \mathcal{A} \times T \rightarrow \mathcal{A} \times T'$, and let $I = (F, M)$ and $I' = (F', M')$ be two implementations of $\mathcal{A}$. We say that m_I is equivalent to $m_{I'}$ modulo RRI $\sim$, denoted $m_I \simeq m_{I'}$ if:

$$\forall \mathcal{I} \in (I \times T).\ \forall \mathcal{I}' \in (I' \times T).\ \mathcal{I} \simeq \mathcal{I}' \Rightarrow m_I(\mathcal{I}) \simeq m_{I'}(\mathcal{I}')$$

In other words, two implementations of the same method are equivalent if they produce equivalent outputs when executed on equivalent inputs (modulo the RRI).

*Definition 3.5 (***Correctness of ADT Refactoring***).* Let $I = (F, M)$ and $I' = (F', M')$ be two different implementations of the same ADT $\mathcal{A}$, and let $\sim$ be an RRI between $\mathcal{A}_F$ and $\mathcal{A}_{F'}$. Then, we say that I' is a correct refactoring of I with respect to $\sim$, denoted $I \simeq I'$, if, for every $m \in \Sigma_{\mathcal{A}}$, we have $m_I \simeq m_{I'}$.

Intuitively, I' is a correct refactoring of I with respect to $\sim$ if invoking corresponding methods of I, I' on equivalent inputs (modulo $\sim$) results in outputs that are equivalent modulo the RRI. This is illustrated schematically in Figure 6. Based on this notion of correctness, we can now formally state our problem definition:

*Definition 3.6 (***ADT Refactoring Problem***).* Let $I = (F, M)$ be the original implementation of an ADT $\mathcal{A}$. Then, given a new data representation F' of $\mathcal{A}$ and an RRI $\sim$, the ADT refactoring problem is to synthesize a new $I' = (F', M')$ such that $I \simeq I'$.

4 ADT REFACTORING ALGORITHM

In this section, we describe our algorithm for solving the ADT refactoring problem stated in Definition 3.6. Our top-level algorithm is modular and constructs the refactored ADT by synthesizing one method at a time. In what follows, we first introduce some useful definitions and then present our method-level refactoring algorithm.

4.1 Preliminaries

Recall that each method of the ADT takes as input an ADT instance o and arguments v and returns an ADT instance o' and return values v'. We use the notation $\mathcal{I} = (o, v)$ to denote the inputs to an ADT method, and $\mathcal{O} = (o_r, v_r)$ to denote its outputs. Given an input or output $V = (o, v)$, we also write $V.adt$ and $V.val$ to denote o and v respectively.

*Definition 4.1 (***IO Example***).* An input-output (IO) example E for a method m is a pair $(\mathcal{I}, \mathcal{O})$ where $\mathcal{I}$ denotes m's inputs and $\mathcal{O}$ denotes its outputs.

Given an IO example E (or set of IO examples), we write $E.in$ to denote the inputs and $E.out$ to denote the outputs. Next, we define the notion of *counterexample to equivalence*, which plays a central role in our refactoring procedure:

procedure REFACTOR(m_I, F', $\sim$, $\mathcal{G}_{\mathcal{M}}$)
input: A method implementation m_I for the original ADT implementation $I = (F, M)$
input: The data representation F' for the new ADT implementation
input: A relational representation invariant $\sim$ as a boolean function
input: The meta-grammar $\mathcal{G}_{\mathcal{M}}$ presented in Figure 7
output: A refactored ADT implementation $m_{I'}$ that is equivalent to m_I modulo $\sim$
  $\mathcal{G} \leftarrow \mathsf{InitPCFG}(\mathcal{G}_{\mathcal{M}}, m_I, F')$
  $R \leftarrow \emptyset$ ▷ Relational IO Examples
  while True **do**
    $m_{I'} \leftarrow \textsc{Synthesize}(R.new, \mathcal{G})$ ▷ Inductive synthesis from IO examples
    cex, verified $\leftarrow \mathsf{Verify}(m_I, m_{I'}, \sim)$ ▷ Get counterexample to equivalence
    if verified **then**
      return $m_{I'}$
    $r \leftarrow \underline{\textsc{StrengthenSpec}}(cex, m_I, m_{I'}, \sim)$ ▷ Obtain IO example for refactoring
    $R \leftarrow R \cup \{r\}$
    $\mathcal{G} \leftarrow \underline{\textsc{InferSnippets}}(r, m_I, \sim, \mathcal{G})$ ▷ Learn useful code snippets and update grammar
  return $\perp$

Algorithm 1. Method refactoring procedure. The underlined statements are the main deviations from a standard CEGIS loop.

$$
\begin{array}{rcl}
\text{Stmt } S & \rightarrow & A \mid S_1; S_2 \mid \texttt{if}(B)\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}(B)\ S \mid \texttt{return}\ \vec{v} \\
 & & \mid \texttt{assert}(B) \mid \texttt{for}\ (v \in E)\{S\} \\
\text{Atom } A & \rightarrow & L \leftarrow E \mid E.m(E_1, \ldots, E_n) \mid \mathsf{new}\ C \\
\text{Expr } E & \rightarrow & v \mid E.f \mid E[E] \mid E_1 \otimes E_1,\ \otimes \in \{+, -, \times, \div, \mathsf{mod}\} \\
\text{LHS } L & \rightarrow & v \mid L[E] \mid L.f \\
\text{Pred } B & \rightarrow & E \mid \neg B \mid\ \ B_1 \oplus B_2,\ \oplus \in \{\geq, <, =\}
\end{array}
$$

Fig. 7. Java-like meta-grammar for method implementations. C is the name of an ADT implementation (Java class).

*Definition 4.2 (***Counterexample to equivalence***).* Let m_I and $m_{I'}$ be the corresponding methods for two different implementations of the same ADT of $\mathcal{A}$, and let $\sim$ be an RRI relating two different data representations of $\mathcal{A}$. A counterexample to equivalence is a pair of inputs $\mathcal{I}, \mathcal{I}'$ such that (a) $\mathcal{I} \simeq \mathcal{I}'$ and (b) $m_I(\mathcal{I}) \not\simeq m_{I'}(\mathcal{I}')$.

Intuitively, a counterexample to equivalence establishes that two method implementations are not equivalent modulo the given RRI.

*Definition 4.3 (***Relational IO Example***).* A relational IO example for a method m and RRI $\sim$ is a pair of IO examples (E_1, E_2) such that E_1 is a set of IO examples for m's original implementation, E_2 is a set of IO examples for m's refactored implementation. Furthermore, we have $E_1.in \simeq E_2.in$ and $E_1.out \simeq E_2.out$.

Intuitively, a relational IO example for a method has pairs of inputs and outputs that are equivalent modulo the RRI. Given a (set of) relational IO examples R, we use the notation $R.old$ to denote the IO examples for the original implementation and $R.new$ to denote the IO examples for the new (refactored) version.

4.2 Top-Level Procedure

In this section, we present our top-level algorithm, summarized in Algorithm 1, for refactoring an ADT method. This algorithm takes as input (1) the original implementation m_I, (2) the new data

1: **procedure** StrengthenSpec(cex, m_I, $m_{I'}$, RRI)
input: A counterexample to equivalence $cex = (\mathcal{I}, \mathcal{I}')$
input: The original ADT implementation m_I
input: The incorrect method implementation $m_{I'}$
input: RRI, the relational representation invariant expressed as a boolean function
output: A relational IO example r
2: $(o_r, v_r) \leftarrow m_I(\mathcal{I})$
3: $RRI^{\partial} \leftarrow (RRI[o_r/arg_1]) \downarrow (ret = \top)$ ▷ Restrict RRI's first input to o_r and return value to true
4: $o_r' \leftarrow \text{Model}([\![RRI^{\partial}]\!])$ ▷ Invoke SMT solver to find concrete output ADT for refactoring
5: $E_1 \leftarrow (\mathcal{I}, (o_r, v_r))$
6: $E_2 \leftarrow (\mathcal{I}', (o_r', v_r))$ ▷ Construct IO example for refactoring
7: **return** (E_1, E_2) ▷ Return relational IO example

Algorithm 2. Strengthens a counterexample to a relational IO example. We use the notation $f \downarrow (ret = v)$ to only consider executions where f returns value v.

representation F', (3) the user-specified RRI $\sim$, and (4) the meta-grammar $\mathcal{G}_{\mathcal{M}}$ presented in Figure 7 corresponding to a core subset of Java since our implementation targets Java. Refactor starts by calling InitPCFG to instantiate the meta-grammar into a PCFG by adding productions specific to the method being refactored and the new ADT. Internally, InitPCFG statically analyzes the method being refactored and adds the following terminals: (1) all function calls (including constructors) that are accessible from the new ADT, (2) all variables accessible from the method being refactored, (3) all fields accessible from the ADT being refactored. It then assigns uniform probabilities for all productions sharing the same non-terminal, but, as we discuss shortly, these probabilities and productions are updated after learning code snippets via symbolic reasoning.

The main loop of the Refactor procedure (lines 4–11) is an instantiation of the CEGIS framework, with key differences underlined in Algorithm 1. The algorithm internally maintains a set of *relational* IO examples R and, in each iteration, it attempts to find a candidate refactoring $m_{I'}$ that is consistent with IO examples $R.new$ by calling the Synthesize procedure (line 4). It then calls Verify to check whether the candidate refactoring $m_{I'}$ is equivalent (modulo the RRI) to the original implementation m_I and obtains a counterexample to equivalence, cex, otherwise. The novel part of the synthesis procedure corresponds to lines 9–11 and involves two auxiliary procedures:

- **StrengthenSpec:** Given a counterexample to equivalence, the StrengthenSpec procedure is used to obtain the corresponding relational IO example.
- **InferSnippets:** Given a new relational counterexample r, InferSnippets (a) identifies a set of useful code snippets, (b) adds these snippets as productions to the grammar, and (c) updates the probabilities of each production in the PCFG.

In the subsequent sections, we explain the StrengthenSpec, InferSnippets, and Synthesize procedures in more detail.

4.3 Specification Strengthening

Our method for specification strengthening is presented in Algorithm 2. This procedure takes as input a counterexample to equivalence ($cex = (\mathcal{I}, \mathcal{I}')$) between m_I and $m_{I'}$ and infers an input-output example for the target refactoring by utilizing the semantics of the RRI. In more detail, the algorithm works as follows:

- First, it executes the reference implementation m_I on input $\mathcal{I}$ to obtain the new ADT instance o_r and return value v_r (line 2).

- Next, at line 3, it obtains a partially evaluated version RRI^{∂} of the RRI by substituting its first argument with ADT instance o_r and restricting its return value to $true$.[1] Note that, because of the RRI properties from Equations 6 and 7, RRI^{∂} can always be simplified to straight-line code.
- Then, the algorithm converts RRI^{∂} to a logical formula, $[\![RRI^{\partial}]\!]$, that encodes its semantics. Note that there is only one free variable in this formula, which corresponds to the refactored version of the ADT. Furthermore, because of the RRI property, we can show that the formula $[\![RRI^{\partial}]\!]$ has a unique model o'_r (line 4).
- Finally, since the output of the refactored method is given by (o'_r, v_r), StrengthenSpec constructs an IO example for the refactored version and returns the corresponding relational IO example.

The remainder of this subsection states and proves the claims made in this discussion.

Theorem 4.4. *Given a relational representation invariant RRI, expressed as a deterministic boolean function, and a concrete instance o for the original ADT, there is a unique path p_o in RRI such that for any instance o' for the new ADT, if RRI returns true given o and o' as input, then RRI executes p_o.*

Proof. From Theorem 3.2, we know that for any instance o of the original ADT, there is a unique instance o' in the new ADT such that $RRI(o, o')$ returns true. Since RRI is deterministic, there is exactly one path, p_o, that can be executed when evaluating RRI on inputs o and o'. □

We refer to p_o as the satisfying path for o, and it follows from this theorem that RRI^{∂} (at line 3 of Algorithm 2) can be expressed as a straight-line program P_o by converting path p_o to its code representation in the standard way [Dijkstra 1975]. Next, we define the logical encoding function ($[\![\cdot]\!]$) used at line 4 of Algorithm 2.

*Definition 4.5 (***Logical encoding***).* Let m be a method that takes inputs x and returns outputs y. We say that a formula $[\![m]\!]$ is a *logical encoding* of m iff, for any interpretation $\mathcal{M}$ of $[\![m]\!]$, we have:

$$\mathcal{M} \models [\![m]\!] \text{ iff } m(\mathcal{M}(x)) = \mathcal{M}(y)$$

In other words, $[\![m]\!]$ is a logical encoding of method m if the models of $[\![m]\!]$ correspond precisely to the input-output behavior of m. Note that $[\![m]\!]$ can always be computed precisely for loop-free code using standard techniques [Dijkstra 1975].

Example 4.6. Consider the following simple RRI expressed as a boolean function between ADT implementations O, N where O has an integer field x and N has an integer field y:

```
boolean rri(O o, N n) {
  if (o.x > 0) return o.x == n.y + 1;
  else return o.x == n.y - 1;
}
```

Its logical encoding is the following formula (where ret denotes the return value of `rri`):

$$(o.x > 0 \rightarrow (\text{ret} = \top \leftrightarrow (o.x = n.y + 1)))$$
$$\wedge (o.x \leq 0 \rightarrow (\text{ret} = \top \leftrightarrow (o.x = n.y - 1)))$$

We can show that the formula $[\![RRI^{\partial}]\!]$ constructed at line 4 of Algorithm 2 has a unique model:

Theorem 4.7. *Let RRI^{∂} be the partially evaluated function from line 3 of Algorithm 2. Then, there is a unique model satisfying $[\![RRI^{\partial}]\!]$.*

[1] Given RRI with body S followed by return statement `return ret`, we use the notation $RRI \downarrow (ret = \top)$ to indicate the program S; `assert`$(ret = \top)$; `return` ret.

1: **procedure** InferSnippets(r, m_I, $\sim$, $\mathcal{G}$)
input: A Relational IO Example $r = (E_1, E_2)$
input: Original method implementation m_I over input V_o and output V_{o_r}
input: An RRI $\sim$ over inputs V_o, V_n
input: A PCFG $\mathcal{G}$
output: An updated PCFG $\mathcal{G}'$
2: $\mathcal{G}' \leftarrow \mathcal{G}$
3: $p_\sim \leftarrow$ InducedPath($\sim, E_1.in, E_2.in$)
4: $p'_\sim \leftarrow$ InducedPath($\sim, E_1.out, E_2.out$)
5: $p_m \leftarrow$ InducedPath($m_I, E_1.in$)
6: $\phi \leftarrow [\![p_\sim]\!] \wedge V_o.val = V_n.val$
7: $\psi \leftarrow [\![p'_\sim]\!] \wedge V_{o_r}.val = V_{n_r}.val$
8: $\chi \leftarrow$ Abduce($\phi \wedge [\![p_m]\!] \wedge ? \models \psi, V_n \cup V_{n_r}$)
9: $\mathcal{S} \leftarrow \emptyset$
10: **for all** $l \in$ Literals(χ) **do**
11: $\mathcal{S} \leftarrow \mathcal{S} \cup$ Literal2Snippet(l, Literals(χ))
12: **return** AddToGrammar($\mathcal{G}', \mathcal{S}$)

Algorithm 3. Infers new code snippets that can help the synthesizer solve the IO example c.

1: **procedure** AddToGrammar($\mathcal{G}, \mathcal{S}$)
input: A PCFG $\mathcal{G} = (\mathcal{G}_c, p)$
output: A new PCFG $\mathcal{G}' = (\mathcal{G}'_c, p')$ which includes the snippets $\mathcal{S}$
2: $\mathcal{G}'_c \leftarrow \mathcal{G}_c$; $p' \leftarrow p$
3: **for all** $S \in \mathcal{S}$ **do**
4: $N_s \leftarrow$ GetNonterminal(S)
5: $\mathcal{G}'_c$.addProduction($(N_s \rightarrow S)$)
6: **for all** $\mathcal{N} \in$ nonTerminals($\mathcal{G}_c$) **do**
7: $R_\mathcal{N} \leftarrow$ getProductions($\mathcal{G}_c, \mathcal{N}$)
8: $R_\mathcal{S} \leftarrow$ getAddedProductions($\mathcal{G}_c, \mathcal{N}$)
9: $p' \leftarrow p'[r_S \rightarrow \frac{(1-\epsilon_\mathcal{N})}{|R_S|},\ \forall r_S \in R_S]$
10: $p' \leftarrow p'[r_d \rightarrow \frac{\epsilon_\mathcal{N}}{|R_\mathcal{N}|-|R_S|},\ \forall r_d \in R \setminus R_S]$
11: **return** $(\mathcal{G}'_c, p')$

Algorithm 4. Adds code snippets $\mathcal{S}$ to PCFG $\mathcal{G}$ with base grammar $\mathcal{G}_c$ and probability function p and updates the probabilities. $\epsilon_\mathcal{N}$ is a real-valued parameter between 0 and 1 that is associated with nonterminal $\mathcal{N}$ in the grammar.

Proof. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be models of $[\![RRI^\partial]\!]$ which map input variable arg_2 to a concrete value. Since they are both models of $[\![RRI^\partial]\!]$, we have that $RRI^\partial(\mathcal{M}_1(arg_2)) = RRI^\partial(\mathcal{M}_2(arg_2)) = \text{True}$. From the definition of RRI^∂, we have that $RRI^\partial(\mathcal{M}_1(arg_2)) = RRI(o_r, \mathcal{M}_1(arg_2))$ for a concrete instance o_r. Likewise, we have $RRI^\partial(\mathcal{M}_2(arg_2)) = RRI(o_r, \mathcal{M}_2(arg_2))$. Since $RRI(o_r, \mathcal{M}_1(arg_2)) = \text{True}$ and $RRI(o_r, \mathcal{M}_2(arg_2)) = \text{True}$, and RRI expresses a relational representation invariant, we can infer from Equation 6 that $\mathcal{M}_1(arg_2) = \mathcal{M}_2(arg_2)$. Thus, $\mathcal{M}_1 = \mathcal{M}_2$. □

Example 4.8. Consider the RRI from Example 4.6 and input instance $o = \{x \rightarrow 1\}$, Then, $[\![RRI^\partial]\!]$ is the formula $n.y = 0$ and the model returned for n is $\{y \rightarrow 0\}$.

4.4 Inferring Code Snippets

In this section, we describe the InferSnippets procedure that is used for identifying code snippets that are likely to be useful for inductive synthesis. This procedure, presented in Algorithm 3, takes as input a relational IO example $r = (E_1, E_2)$, the original method implementation m_I, the RRI $\sim$ (over old variables V_o and new variables V_n) and PCFG $\mathcal{G}$ and returns a new PCFG $\mathcal{G}'$ with additional code snippets added as productions. At a high level, InferSnippets works as follows:

- First, it executes the RRI on the input examples $E_1.in$ and $E_2.in$ and obtains a straight-line program $p_\sim$ corresponding to the path taken when running the RRI on these inputs (line 3).
- Next, it does the same but for output examples $E_1.out$ and $E_2.out$ to obtain another straight-line program $p'_\sim$ that corresponds to the execution path of the RRI on $E_1.out$ and $E_2.out$ (line 4).
- At line 5, it obtains a straight-line program p_m corresponding to the execution of the original method m on input $E_1.in$.
- Lines 6–8 set up an abduction problem, with the goal of inferring a logical specification of the refactored procedure over variables V_n, V_{n_r}. Recall that logical abduction is the problem of inferring a missing hypothesis: Specifically, given a premise ϕ, desired conclusion ψ, and a set of variables V, logical abduction infers a missing (and consistent) hypothesis χ over variables V

such that: $\phi \wedge \chi \models \psi$. To see what abduction has to do with our problem, recall that we would like to find an implementation of $m_{I'}$ satisfying the following Hoare triple:

$$\{V_o \simeq V_n\} \qquad V_{o_r} := m_I(V_o); \quad V_{n_r} := m_{I'}(V_n) \qquad \{V_{o_r} \simeq V_{n_r}\}$$

Essentially, formula ϕ at line 6 of Algorithm 3 corresponds to the precondition of the Hoare triple, ψ from line 7 corresponds to the post-condition, and $[\![p_m]\!]$ is the logical encoding of an execution path for function m_I. Thus, to infer how $m_{I'}$ should behave for the corresponding execution path, we need to find a formula χ over variables V_n, V_{n_r} such that the following entailment holds:

$$\phi \wedge [\![p_m]\!] \wedge \chi \models \psi$$

Finding such a χ is precisely an abduction problem; hence, formula χ at line 8 of the algorithm provides a sufficient condition for correctness of $m_{I'}$ for a specific execution path. Since there are standard techniques for performing logical abduction [Albarghouthi et al. 2016; Dillig and Dillig 2013; Dillig et al. 2012, 2013] based on quantifier elimination, we do not discuss the Abduce procedure in detail here.

- Next, lines 9–11 of Algorithm 3 mine useful statements and expressions $\mathcal{S}$ from the logical specification χ of $m_{I'}$. The basic idea is to translate literals of χ to suitable expressions/statements in the source language via the call to Literal2Snippet at line 11. Since this Literal2Snippet procedure is based on simple syntax-directed translation, we do not discuss it in detail and provide more implementation details in Section 5.
- Finally, the algorithm adds the mined components $\mathcal{S}$ to the grammar and updates the probabilities of the productions accordingly (line 12).

We now illustrate the InferSnippets procedure using a concrete example:

Example 4.9. Consider again the running example in Section 2. Suppose that the input to InferSnippets is $E_1 := ((I, V), O)$, $E_2 := ((I', V), O')$ [2] where I, I', V, O, and O' are given in Equations 1-5. Then, InferSnippets constructs the following logical encodings of $p_\sim$, $p'_\sim$ and p_m:

$$\begin{array}{rcl}
[\![p_\sim]\!] & := & height \geq 0 \wedge width \geq 0 \wedge lc.x = minX \wedge lc.y = minY \\
& & \wedge height = maxY - minY \wedge width = maxX - minX \\
[\![p'_\sim]\!] & := & height' \geq 0 \wedge width' \geq 0 \wedge lc.x' = minX' \wedge lc.y' = minY' \\
& & \wedge height' = maxY' - minY' \wedge width' = maxX' - minX' \\
[\![p_m]\!] & := & width - 2w \geq 0 \wedge height - 2h \geq 0 \wedge \neg flipX \\
& & \wedge \neg flipY \wedge lc.x' = lc.x + w \wedge lc.y' = lc.y + h \\
& & \wedge height' = height - 2h \wedge width' = width - 2w
\end{array}$$

Since ScaleAndFlip does not return any additional values, InferSnippets sets $\phi = [\![p_\sim]\!]$ and $\psi = [\![p'_\sim]\!]$. Next it calls Abduce with $V_n = \{V, I'\}$ and $V_{n_r} = \{O'\}$ which returns the following:

$$\chi = \left\{ \begin{array}{lll}
maxX - minX - 2w \geq 0 \wedge & maxY - minY - 2h \geq 0 & \\
\wedge \ \neg flipX \wedge \neg flipY & \wedge \ minX' = minX + w & \wedge \ maxX' = maxX - w \\
\wedge \ minY' = minY + h & \wedge \ maxY' = maxY - h & \wedge \ maxX - minX \geq 0 \\
\wedge \ maxY - minY \geq 0 & &
\end{array} \right.$$

Finally, InferSnippets extracts the literals from χ and converts them to snippets via syntax-directed translation. In particular, it will add the following snippets to the grammar:

(1) (Atomic (A)): $minX := minX + w$, $minY := minY + h$, $maxX := maxX - w$, and $maxY := maxY - h$

(2) (Boolean (B)): $maxX - minX - 2w \geq 0$, $maxY - minY - 2h > 0$, $\neg flipX$, $\neg flipY$, $maxX - minX \geq 0$, $maxY - minY \geq 0$.

[2] The return values are omitted for E_1 and E_2 because ScaleAndFlip does not return a value.

1: **procedure** Synthesize($\mathcal{E}$, F', $\mathcal{G}$)
input: A set of IO examples $\mathcal{E}$
input: Fields F' for new ADT implementation
input: A PCFG $\mathcal{G}$
output: A refactored ADT implementation $m_{I'}$
2: $\mathcal{W} \leftarrow \{?\}$
3: $\mathcal{M} \leftarrow \emptyset$
4: **while** $\mathcal{W} \neq \emptyset$ **do**
5: $P \leftarrow$ SelectBest($\mathcal{W}$)
6: **if** IsComplete(P) **then**
7: **if** IsConsistent($P, \mathcal{E}$) **then**
8: **return** P
9: $F'_{\equiv} \leftarrow$ GetEqFields($P, \mathcal{E}$)
10: **if** $|F'_{\equiv}| \geq \gamma \times |F|$ **then**
11: $\mathcal{W} \leftarrow \mathcal{W} \cup$ Combine($P, \mathcal{M}$)
12: $\mathcal{M}[P] \leftarrow F'_{\equiv}$
13: **continue**
14: **else**
15: $\mathcal{P} \leftarrow$ expand($P, \mathcal{G}$)
16: $\mathcal{W}$.addAll($\{P' \mid P' \in \mathcal{P}\}$)
17: **return** $\perp$

Algorithm 5. Inductive Synthesis procedure.

1: **procedure** Combine(P, $\mathcal{M}$, $F'_{\equiv}$)
input: A program P that is partially correct
input: A component map $\mathcal{M}$ of programs that partially satisfy the IO examples
input: A set of fields $F'_{\equiv}$ that P got correct across the IO examples
output: A set $\mathcal{P}$ of partial programs derived by combining P with another program in $\mathcal{M}$
2: $\mathcal{P} \leftarrow \emptyset$
3: **for all** $P' \in$ Domain($\mathcal{M}$) **do**
4: $F_{\equiv_{P'}} \leftarrow \mathcal{M}[P']$
5: **if** $F_{\equiv_{P'}} \not\subset F'_{\equiv} \wedge F'_{\equiv} \not\subset F_{\equiv_{P'}}$ **then**
6: $\mathcal{P} \leftarrow \mathcal{P} \cup$ Merge(P, P')
7: **return** $\mathcal{P}$

Algorithm 6. Bottom-up search procedure. Combine generates new partial programs by combining P with existing components in $\mathcal{M}$. The Merge procedure used at line 6 is presented as inference rules in Figure 8.

Next, we turn our attention to the AddToGrammar procedure invoked at line 12 of Algorithm 3 and summarized in Algorithm 4. This algorithm takes as input the current PCFG $\mathcal{G}$ along with the generated code snippets $\mathcal{S}$ and produces a new PCFG $\mathcal{G}'$ that includes $\mathcal{S}$. For each code snippet S, it identifies the corresponding nonterminal $\mathcal{N}$ in $\mathcal{G}$ such that $\mathcal{N} \Rightarrow^* S$ in the base grammar (line 4) and adds the new production $(\mathcal{N} \rightarrow S)$ to $\mathcal{G}$ (line 5) and also recomputes the probabilities of the productions in $\mathcal{G}'_c$ (lines 6 - 10). In particular, for each nonterminal $\mathcal{N}$, it obtains all snippets $R_{\mathcal{S}}$ added for $\mathcal{N}$ (line 8), and for each snippet r_S, it sets its probability to be $(1 - \epsilon_{\mathcal{N}})/|R_{\mathcal{S}}|$ (line 9). Hence, all snippets added to the grammar for $\mathcal{N}$ have the same probability. Likewise, for all productions in the base grammar, the algorithm sets their probability to be $\epsilon_{\mathcal{N}}/(|R_{\mathcal{N}}| - |R_{\mathcal{S}}|)$. The parameter $\epsilon_{\mathcal{N}}$ is a real value in the interval $[0, 1]$ associated with nonterminal $\mathcal{N}$. Intuitively, a small value of ϵ will bias the search towards programs that use the added snippets.

4.5 Inductive Synthesis Algorithm

In this section, we present our inductive synthesis algorithm for finding a program that satisfies a given set of input-output examples. As mentioned earlier, this algorithm leverages the notion of *partial equivalence*:

*Definition 4.10 (***Partial equivalence***).* Let m_1 and m_2 be two different implementations of the same ADT method for a data representation F'. We say that m_1 and m_2 are partially equivalent modulo fields $F \subseteq F'$, denoted $m_1 \equiv_F m_2$, iff:

$$\forall \mathcal{I}. \left(m_1(\mathcal{I}) = (o_r, v) \wedge m_2(\mathcal{I}) = (o'_r, v) \Longrightarrow \bigwedge_{f \in F} o_r.f = o'_r.f \right) \tag{8}$$

In other words, two implementations are equivalent modulo fields F if they produce the correct values for *only* those fields. In general, while we could attempt to verify partial equivalence for all possible inputs, the goal of inductive synthesis is to find a program that satisfies a given *finite* set of examples. Hence, when testing partial equivalence as part of the inductive synthesis procedure, we restrict the domain of $\mathcal{I}$ in Equation 8 to only the provided input examples.

Our inductive synthesis algorithm is summarized in Algorithm 5. At a high level, it performs top-down search over programs in $\mathcal{G}$ using the notion of partial equivalence to incorporate bottom-up synthesis. As is standard in top-down synthesis [Feser et al. 2015; Gulwani et al. 2017], the algorithm maintains a worklist $\mathcal{W}$ of partial programs P where each partial program can be viewed as a sequence of grammar symbols (both terminals and non-terminals) in $\mathcal{G}$. In each iteration, the algorithm chooses the "best" partial program P in the work list (line 5), where SelectBest is a heuristic ranking function that scores programs according to the probability of that partial program according to the PCFG as well as other factors like size.[3] If the dequeued program contains a non-terminal $\mathcal{N}$ (meaning that the call to IsComplete at line 6 returns false), the algorithm expands $\mathcal{N}$ by replacing $\mathcal{N}$ with the right-hand-side of all grammar productions of the form $\mathcal{N} \rightarrow \alpha$ and adds the resulting partial programs to the worklist (lines 14–15). On the other hand, if P is complete (meaning it contains only terminal symbols), the algorithm checks whether P is consistent with all examples $\mathcal{E}$ (line 7). If so, P is returned as a solution.

The novel part of our inductive synthesis algorithm corresponds to lines 8–12 in Algorithm 5. As mentioned earlier, this part of the algorithm combines top-down search with bottom-up synthesis by leveraging the notion of partial equivalence. In particular, line 8 of the algorithm checks whether P satisfies the input-output examples for some subset of the fields $F'_{\equiv} \subset F'$. Specifically, for each field $f \in F'_{\equiv}$, we have $P(\mathcal{I}).f = \mathcal{O}.f$ for each $(\mathcal{I}, \mathcal{O}) \in \mathcal{E}$. Intuitively, if the fraction of such fields is above a certain threshold γ (line 9), this program is considered a useful building block and added to a map $\mathcal{M}$ (line 11). Additionally, P is combined with existing building blocks in $\mathcal{M}$ via the Combine procedure (line 10), and all of the resulting programs are added to the worklist.

The Combine procedure is presented in Algorithm 6: given a complete program P and previously discovered components $\mathcal{M}$, it generates new programs by combining P with each $P' \in \mathcal{M}$ via the Merge procedure. Note that the algorithm only merges P and P' if one of them is not strictly better than the other one (check at line 5 of Algorithm 6).

Finally, Figure 8 formalizes the Merge procedures using inference rules that derive judgments of the form $P_1, P_2 \vdash \theta$, where θ is a set of new partial programs. According to the Seq rule, two code snippets can be combined sequentially to obtain a larger snippet. The first conditional rule, If-1 combines two snippets P_1 and P_2 by introducing a conditional and yields the partial program `if (??)` P_1 `else` P_2. The second conditional rule If-2 combines two if statements that share the same predicate e. In particular, it generates a set of new if statements (with the same predicate e) but where the true and false branches are obtained by recursively merging the corresponding branches. The final For rule is similar to If-2 but for loops instead of conditionals.

4.6 Properties of Our Refactoring Technique

Assuming a sound Verify procedure, the soundness of our algorithm follows straightforwardly from the check performed on line 6 of Refactor. Thus, we conclude this section by proving the completeness of our end-to-end algorithm.

Theorem 4.11 (Completeness). *If there is an implementation $m_{I'}$ such that $m_{I'}$ and m_I are equivalent modulo RRI $\sim$, then Refactor$(m_I, F', \sim, \mathcal{G})$ returns an $m'_{I'}$ such that $m'_{I'}$ is equivalent to m_I modulo $\sim$.*

[3] More implementation details about SelectBest are provided in Section 5.

$$\frac{}{P_1, P_2 \vdash \{P_1; P_2\}}\ \text{Seq} \qquad \frac{}{P_1, P_2 \vdash \{\texttt{if(??)}\ P_1\ \texttt{else}\ P_2\}}\ \text{If-1}$$

$$\frac{\forall i \in [1,2].\ P_i := \texttt{if}(e)\ P_i'\ \texttt{else}\ P_i'' \quad P_1', P_2' \vdash \theta_1 \quad P_1'', P_2'' \vdash \theta_2}{P_1, P_2 \vdash \{\texttt{if}(e)\ S_1\ \texttt{else}\ S_2 \mid (S_1, S_2) \in \theta_1 \times \theta_2\}}\ \text{If-2}$$

$$\frac{\forall i \in [1,2].\ P_i := \texttt{for(v}\ \in\ e)\{\ S_i\ \} \quad S_1, S_2 \vdash \theta}{P_1, P_2 \vdash \{\texttt{for(v}\ \in\ e)\{\ S\ \} \mid S \in \theta\}}\ \text{For}$$

Fig. 8. Representative rules describing how we combine code snippets

5 IMPLEMENTATION

We have implemented the ideas presented in this paper as a new tool called Revamp, which takes three inputs: (1) the original ADT, expressed as a Java class, (2) declaration of the new ADT, and (3) an RRI expressed as a boolean Java function. Revamp additionally takes a time limit t indicating the maximum time for refactoring a method. Revamp is written in Java and internally uses JBMC [Cordeiro et al. 2018] for verification and counterexample generation and the Z3 solver [de Moura and Bjørner 2008] for determining logical satisfiability. Revamp also uses the Soot framework[Lam et al. 2011] for identifying methods and variables that are in scope. In the rest of this section, we describe important optimizations used by Revamp along with other relevant implementation details omitted from Section 4.

Specifying RRIs As mentioned above, Revamp expects the user to express the intended RRI as a boolean Java function taking two inputs: an instance of the original ADT and an instance of the new ADT. In theory, the RRI should relate all the fields of the original to the fields of the new to satisfy equations 6 and 7; however, in practice it can be cumbersome for users to write a complete RRI since the original ADT may contain several fields many of which should remain unchanged by the intended refactoring.

To help users write concise RRIs, Revamp allows them to specify an RRI over the subset of the new and original ADT fields relevant to the refactoring task. Before synthesizing the refactored implementation, Revamp statically analyzes the RRI to identify the relevant fields from the original and new implementation. It then attempts to infer a one-to-one correspondence between the unspecified fields in the original and new ADTs. Specifically, for every unspecified field in the original code, it expects to find a corresponding field in the new ADT with the same name and type (and vice-versa). If no such correspondence can be found, it returns an error describing which unspecified fields could not be matched.

After finding such a correspondence, Revamp then constructs an updated RRI asserting that the values of the unspecified fields should be equal. The updated RRI looks like the code snippet below where fields f_1 through f_n are the unspecified ones and $\equiv$ is shorthand for deep equality checks:

```
static boolean updatedRRI(O o, N n) {
    b := o.f1 ≡ n.f1 && ... && o.fn ≡ n.fn;
    return b && origRRI(o, n);
}
```

Validating RRIs Recall that the correctness of our refactoring technique relies on the fact that the RRI satisfies equations 6 and 7. To help users check that their RRIs satisfy these equations, Revamp includes two utilities. First, to check whether equation 6 holds for an RRI *rri*, Revamp encodes Equation 6 as the following code snippet with assertions:

```
static void rri_check1() {
    o := nondetOrig();
    n1 := nondetNew(); n2 := nondetNew();
    Verifier.assume(rri(o, n1) && rri(o, n2));
    assert(n1 ≡ n2);
}
```

Here, `nondetOrig` and `nondetNew` are Revamp-generated functions that construct arbitrary instances of the new and original ADT. The third line in the function body introduces assume statements that encode the antecedent of Equation 6. The final line checks that the consequent of Equation 6 holds by asserting that the two instances must be equal. Given such a code snippet, Revamp utilizes a verifier (specifically, JBMC [Cordeiro et al. 2018]) to check the assertion. If the verifier finds a counterexample, Revamp presents this counterexample to the user to help them fix the RRI.

Revamp also provides a utility to help users check whether their RRI satisfies Equation 7. In principle, verifying Equation 7 requires solving a formula with quantifier alternation ($\forall\exists$) which SMT solvers struggle with. To bypass this problem, Revamp instantiates the universal quantifier with several concrete instances $r_1, \ldots, r_n$ of the old ADT and constructs the following code snippet:

```
static void rri_check2(r1, ..., rn) {
    o := r1; n := nondetNew();
    assert(¬rri(o, n));
    ...
    o := rn; n := nondetNew();
    assert(¬rri(o, n));
}
```

If the RRI satisfies Equation 7, then *every* assertion in the above code snippet *should be* violated. Thus, Revamp uses a model checker to try to construct a counterexample for each of the assertions.

Computing Logical Encodings. Recall from Section 4.3 that Revamp needs to encode loop-free functions as logical formulas. Given a method m with arguments $\bar{V}$, Revamp derives the logical encoding of this method by computing the weakest precondition of the following code snippet with respect to True:

$$r = O.m(\bar{V})$$
$$\texttt{assert}(O' \equiv O \wedge \text{ret} \equiv r);$$

Here, O' is a variable representing the ADT after calling m and ret is a variable denoting m's return value. Thus, the weakest precondition of this code snippet describes the updated ADT instance O' and return value ret in terms of fields of the input ADT.

When generating the weakest preconditions, we model most Java constructs in the standard way: for example, we model references using an array-based encoding that has been popularized by ESC-Java [Flanagan et al. 2002]. Specifically, we introduce an array for each field and model loads and stores using select and update functions in the theory of arrays. However, some language constructs like bit manipulation and non-linear arithmetic are not solver-friendly, especially for performing quantifier-elimination, and so we encode those operations using uninterpreted functions.

Implementation of Verify from Algorithm 1. Revamp implements the Verify procedure from Algorithm 1 by encoding the correctness check as a code snippet with assertions/assumptions and calling a model checker (JBMC). In particular, given a method m from the original ADT, a candidate refactored implementation m' of m, and RRI rri, Revamp constructs the following code snippet:

```
static void harness_m() {
    o := nondetOrig(); n := nondetNew();
```

```
        Verifier.assume(rri(o, n));
        v1 := nondet[T1](); ...; vn := nondet[Tn]()
        r1 := o.m(v1,...,vn);
        r2 := n.m(v1,...,vn);
        assert(rri(o, n) && r1 ≡ r2);
    }
```

This code snippet first constructs original and new ADT instances o and n that satisfy rri. It then calls the original and refactored implementation with the same (arbitrary) arguments. Finally, it asserts that the rri holds on o and n after the method calls and that the return values are the same.

Logical Abduction. Revamp solves the logical abduction problem presented in Section 4.4 by performing quantifier elimination, similar to previous work in solving logical abduction problems [Albarghouthi et al. 2016; Dillig et al. 2012, 2013]. One detail worth noting is that the formulas in our setting are in the combined theory of uninterpreted functions, arrays, and integers, and this theory does not admit quantifier elimination (i.e., there may not always be an equivalent quantifier-free formula). However, we can still approximate quantifier elimination using the cover [Gulwani and Musuvathi 2008] operation, which is a generalization of quantifier elimination. In particular, given a formula $\exists V.\phi(V, X)$ where $X \cap V = \emptyset$, the cover of ϕ is a quantifier free formula $\psi(X)$ such that

(1) $\exists V.\phi \models \psi$
(2) For any $\varphi(X)$ such that $\exists V.\phi \models \varphi$ we have $\psi \models \varphi$.

These conditions are sufficient for solving the logical abduction problem.

Translating Literals into Code Snippets. Recall from Section 4.4 that InferSnippets generates code snippets by first solving a logical abduction problem and then translating each literal in the solution into code snippets. In the vast majority of cases, the translation is straightforward. In particular, Revamp translates literals of the form $I_k \oplus I_j$ where $\oplus$ is a comparison operator and I_k and I_j are logical expressions over input variables into their corresponding comparison code snippets $e_k \oplus e_j$ where e_k and e_j are translations of I_k and I_j respectively. Literals of the form $O = I$ where O is an expression solely over output variables is translated into assignments $e_O := e_I$ where e_O and e_I are the translations of O and I. Other literals such as $O_i \oplus O_j$ are translated into assertions like $\texttt{assert}(e_{O_i} \oplus e_{O_j})$. There are two cases handled separately:

- Literals of the form $O.f = I$ where f is not accessible from the new implementation cannot be translated into assignments; these require the introduction of function calls. In this case, Revamp first attempts to recursively translate O and I into expressions E_O and E_I. It then uses the SPARK pointer analysis [Lhoták and Hendren 2003] within Soot to identify all accessible methods m that may *write* to f. For each m, Revamp constructs a code snippet of the form $m(??_1, .., ??_n)$ where $??_i$ indicates an unknown argument.
- Expressions of the form $I.f$ where f is not accessible cannot be directly translated into field dereferences and must be accessed via a method call. In this case, Revamp translates I into snippets E_I and then identifies all accessible methods m that may *read* f and generates function expressions with unknown arguments (as in the previous case).

Example 5.1. Consider the following literal that occurs in a formula:

$$o'_r.\text{map.keys}[0] = \text{getComponent}(\text{arg1})$$

where map is a field in the new ADT of type Map and keys is a private field in the map implementation. Intuitively, this literal encodes an update to map's keys. Revamp will recursively generate the code snippets this.map and arg1.getComponent() and then identify the methods in Map which can write to keys. Revamp finds that the only methods that can update the keys are put, remove,

and clear, so it returns the snippets $??_1$.put($??_2$, $??_3$), $??_1$.remove($??_2$) and $??_1$.clear() along with this.map and arg1.getComponent().

SelectBest. Recall that the inductive synthesis algorithm utilizes a function called SelectBest for choosing the most promising element in the worklist. Our implementation of SelectBest prioritizes programs that use the inferred code snippets as well as those that have been obtained by combining partially equivalent implementations. To prioritize programs that use the inferred snippets, we set $\epsilon_{\mathcal{N}}$ to be 0.01 for all nonterminals $\mathcal{N}$; this gives all productions that were not added by InferSnippet a low probability. To prioritize combined partially equivalent programs, Revamp associates each program P with an integer N_P which indicates the number of fields we expect the program to update correctly across the IO examples. Whenever Revamp combines programs P_1 and P_2 that satisfy fields F_1' and F_2', it associates every resulting combination P with $N_P = |F_1' \cup F_2'|$. Revamp then assigns each program the following score:

$$Score(P) := Min(M, N_P) \cdot Pr_{\mathcal{G}}(P)$$

and SelectBest selects the program with the highest score. The extra parameter M is effectively an upper bound on N_P that prevents the synthesizer from generating large programs that overfit the I/O examples. We set M to be 10 in our experimental evaluations.

6 EVALUATION

We evaluated Revamp with the goal of answering the following research questions:

- **(RQ1)** *Usefulness*: Can Revamp automate real-world ADT refactoring tasks?
- **(RQ2)** *Comparison to Existing Tools*: How does Revamp compare against state-of-the-art Java synthesizers and large language models such as ChatGPT?
- **(RQ3)** *Ablation*: How impactful are Revamp's key ideas?

Benchmarks. To evaluate Revamp on real-world ADT refactoring tasks, we wrote a GitHub crawler that looks for candidate Java projects where the data representation has been changed in between commits. Among the results returned by the crawler, we manually inspected the results in order of popularity and retained the first 30 classes that (a) indeed change the underlying data representation, and (b) require modifying more than 10 lines of code. Nearly all of our benchmarks come from widely-used, large projects like Netty [net 2022], Elessandra [ele 2022], Cassandra [cas 2022]. Moreover, the refactorings require changing an average of 50 lines of code and modifying 34% of the ADT methods.

Writing RRIs. To evaluate Revamp on these benchmarks, we had to manually write relational representation invariants for each benchmark. Before doing so, we first tried to understand the behavior of the benchmark as well as the developer's intentions when refactoring the data representation. For the most part, we were able to get a correct understanding of the code base (both original and new implementations) by examining the diff from the git commit as well as the source code of the original implementation; however we sometimes (roughly half of the time) also executed some methods of the original and new ADTs on inputs (1-2) we crafted to confirm the methods returned the result (and modified the new ADT fields) we expected.

After getting a comprehensive mental model of the code base and the developer's intentions for the refactoring task, we then wrote the RRI for the benchmark; overall, writing the RRIs took less than 5 minutes on average. If we had executed the original and new ADT methods during our examination of the benchmark, we would then utilize those examples (inputs and observed outputs) to confirm that the RRI we had written was consistent with them. We also used Revamp's RRI checker to validate that properties (6) and (7) held for the RRIs.

We recall two cases where we made mistakes writing the RRI and in both cases we ended up catching the mistakes using the examples and Revamp's checker. In particular, using examples helped catch one simple mistake where, in our first attempt to write the RRI for the StreamCacheService benchmark, we confused two subfields of the same type and used one in place of the other in the RRI. But when we sanity checked the RRI against our examples, the RRI did not hold. Next, Revamp's checker found that the RRI we had written for the Zookeeper benchmark violated property (6). In that refactoring task, the original ADT had a HashMap m1 and the new ADT had a HashMap m2 such that every entry in m1 had a corresponding entry in m2 and vice versa. Our original RRI overlooked the bidirectional relationship and only asserted that every entry in m1 had a corresponding entry in m2. The checker returned a counterexample where, in one case, the keys of m2 were a strict superset of the keys in m1, indicating we had missed the reverse check in the RRI necessary to satisfy (6).

Overall, we estimate that the lead author, the one who wrote the RRIs for the benchmarks, spent 30 minutes per benchmark (15 hours total) including writing and validating the correctness of the RRI (with less than 5 minutes on average writing the RRI). They had 8.5 years of programming experience with 5 years of experience writing Java code and 1 year of professional experience in Java at the time. We further estimate that 80-90% of the author's time was spent understanding the behavior of the original ADT and developer-intended refactoring with the remaining time spent writing and validating the RRIs.

Experimental Setup. All experiments involving Revamp and other Java synthesizers are conducted on a Google Cloud [gce 2013] e2-standard-8 machine with a Debian 11 OS, 64 GB of RAM, and 128GB of hard disk space. For all experiments, we use a time limit of 15 minutes for refactoring each ADT method.

6.1 Main Results

To answer our first research question, we report the number of ADT methods that Revamp is able to successfully refactor as well as the time to perform each refactoring. The result of this experiment is presented in Table 1. The first two columns report the GitHub project the benchmark belongs to and the Java class being refactored. The columns **# Old Fields** and **# New Fields** describe the number of fields in the original and new data representations that are *relevant* to the refactoring task. The next two columns together describe the average size of the initial PCFG across the functions being refactored and the size of the grammar in the final iteration of the CEGIS loop. The difference between the two indicates the number of components Revamp added during synthesis. The next four columns describe the number of functions in the ADT, the number which required refactoring, the size of the RRI in LOC, and the overall size of the refactoring measured as a diff between the original and new ADT implementations. Finally, the last column states the time (in seconds) taken by Revamp to refactor the entire ADT.

The key takeaway from this experiment is that Revamp is able to successfully refactor 29 out of 30 benchmarks and 144 out of 146 functions across all the ADTs. For each of the refactored methods, we also manually inspected the result and verified that the resulting code is equivalent to the manually refactored version in all but one case, where the programmer written code is actually buggy (explained in more detail below).

There is one benchmark, namely GeocodingLookupService, that Revamp fails to solve. This ADT contains three methods that require refactoring, but Revamp fails to synthesize two of these three methods within the 15 minute time limit.

Table 1. Main experimental results. ⊥ indicates time out (> 15 minutes). The Cassandra benchmark is written as Stream[In/Out]Session to indicate that the original refactoring spanned two very similar classes. Rather than consider them as separate benchmarks, we consider it as one larger task. The columns labeled # Init. Prods and # Final Prods list the average number of productions in the grammar at the start and by the end of synthesis respectively. The column labeled "Total Funcs" shows the total number of methods in the class, and the column labeled "Rel. Funcs" shows the number of relevant methods that require modifications. The Diff column shows the number of lines in the diff produced by DiffChecker [DiffChecker 2021] and the RRI size column shows the size of the RRI in LOC. Lastly, the Time column indicates the time taken to refactor the entire ADT (ignoring verification time).

Project	Class	# Old Fields	# New Fields	# Init. Prods	# Final Prods	Total Funcs	Rel. Funcs	RRI Size	Diff	Time
bisq-networks	MathUtils	1	2	244.3	443.3	7	3	1	11	8.1
cassandra	Stream[In/Out]Session	4	4	282.4	371.5	24	12	8.5	66	158.4
elessandra	FieldData	1	2	204.8	355.2	6	5	9	39	30.2
elessandra	FieldMapper	1	3	266	344.8	10	6	14	55	153.2
elessandra	MemoryTranslog	1	3	167.8	305.5	9	5	9	41	43.2
elessandra	WeightFunction	2	1	226.5	262.2	2	2	1	24	88.4
falcon	CLIParser	1	3	325.2	552.2	8	5	15	55	148.2
falcon	EntityProxyUtil	1	2	285.5	388.1	15	6	18	75	234.9
game-of-life	EndlessGrid	1	2	215	319.5	14	2	11	7	13.9
glide	BitmapTracker	2	1	225.5	335.1	8	6	8	88	283.7
graylog2-server	StreamCacheService	1	3	316.4	552.2	9	5	10	45	101.7
guava	SingletonImmutableMap	1	2	134.7	197.2	11	6	3	63	181.3
guice	StackTraceElements	1	2	324.3	442.1	5	4	15	33	116
hadoop	IncrementalBlockReportManager	1	2	337.1	544.2	12	3	13	38	464.1
hbase	DataBlockEncoding	2	2	383	499	13	3	14	64	609.9
javaparser	Node	1	1	133	168	42	2	3	15	14.5
jdbi	Bindings	2	3	253.8	410.7	9	5	14	38	138.8
jenkins-ci	GitLabConnectionConfig	1	1	352	552.2	11	5	9	43	101
jmist	Box2	3	4	197.1	233.3	12	12	5	104	257
junit	BlockJunit4ClassRunner	1	2	233.8	462.9	8	5	17	50	868.2
netty	DefaultChannelPipeline	3	4	586.3	733.3	64	6	16	45	86.6
netty	SpdySession	1	3	270.2	366.2	24	5	9	63	192.6
osmand	GeocodingLookupService	1	2	391.3	662.9	14	3	11	33	⊥
pixeldungeon	Level	1	9	397.8	588.8	12	3	17	85	988.8
pravega	StreamSegmentContainerMetadata	1	3	532.5	882.2	13	6	13	54	118
wicket	AsynchronousPageStore	1	2	319.5	488.3	7	4	9	22	275.7
wicket	RequestAdapter	1	2	188	255.2	9	3	8	21	77.8
wicket	TagIdentifier	1	1	260.7	387	10	3	7	72	117
Xodus	PersistentSequentialDictionary	1	2	263.3	355.2	13	7	13	68	875.5
Zookeeper	NettyServerCnxnFactory	1	1	171.3	299.3	25	4	9	37	146
Averages	-	**1.4**	**2.5**	**282.9**	**425.3**	**14.2**	**4.9**	**9.8**	**48.8**	**237.7**

Case Study: AsynchronousPageStore. One of our benchmarks, namely the AsynchronousPageStore class from the Wicket project[wic 2023], is an interesting case study because the manually-performed refactoring introduces a subtle bug. The original version of this class maintains a queue of `PendingAdd` tasks where each task has a unique identifier called a `key`. The new data representation includes an additional hash map called `map` which tracks each task in the queue by its `key`. Hence, whenever a task is added or removed from the queue, `map` also needs to be suitably updated. However, the manual re-implementation of the `run` method fails to correctly update `map` in an edge case where an auxiliary procedure called by `run` throws an exception. This refactoring created a memory leak because some entries from the HashMap would never get removed in the exception cases. On the other hand, the new implementation synthesized by Revamp handles this edge case correctly and fixes the memory leak in the manual refactoring.

Result 1: Revamp is able to refactor the entire ADT for 97% of the classes and synthesize 99% of all method implementations. Furthermore, the automatic refactoring performed by Revamp does not contain a subtle bug introduced when manually refactoring one of the benchmarks.

Table 2. Comparison between Revamp and baseline tools including JSketch, Volt, and ChatGPT. The column **Synth Time** describes the average time taken by the tool in seconds when it succesfully refactors a method.

Tool	# Bench. Solved	# Funcs Refactored	Synth Time (s)
JSketch	4/30 (13%)	44/146 (30%)	168.3
Volt	8/30 (27%)	83/146 (57%)	132.4
ChatGPT	7/30 (23%)	92/146 (63%)	-
Revamp	29/30 (97%)	144/146 (99%)	42.83

6.2 Comparison against Baseline Tools

To answer our second research question, we compared Revamp against two relevant baselines. While there is no existing technique that directly addresses our problem, we adapted three tools to our setting:

- **JSketch:** Our first baseline is JSketch [Jeon et al. 2015], a generic synthesis framework that is an adaptation of Sketch [Solar-Lezama et al. 2006] to synthesis tasks in Java. To use JSketch, we first manually created harnesses for all 146 methods that required refactoring. Specifically, for every method Revamp was able to refactor, we supplied the I/O examples generated by Revamp as test cases for JSketch. For methods that Revamp failed to refactor, we supplied test cases manually. Subsequently, we used the initial grammar from Revamp and constructed a corresponding JSketch generator for each nonterminal in our grammar (statement generator, expression generator, etc.). Moreover, for each method in the original ADT requiring refactoring, we wrote the corresponding declaration in the new ADT and populated the method's body with a call to the statement generator. Finally, once JSketch produced a program consistent with all examples, we used JBMC to verify its correctness.
- **Volt:** Our second baseline is Volt [Pailoor et al. 2021], a state-of-the-art CEGIS-based synthesizer for Java. While Volt primarily targets data structure refinements, its underlying synthesis algorithm is fairly general and performs enumerative search with SMT-based pruning. As such, we replaced our method refactoring procedure with Volt's synthesis algorithm.
- **ChatGPT:** Our third baseline is ChatGPT [OpenAI 2021][4], a state-of-the-art large language model (LLM), which has shown proficiency at many coding tasks including code synthesis. To use ChatGPT, we provided it the original ADT implementation, the declaration of the new ADT, the RRI and asked it the following: "For every method in [the original ADT] generate an equivalent method in [the new ADT] that preserves [RRI function]". We then compiled the generated code, and for cases where compilation was successful, we also attempted to verify equivalence using JBMC.

Results. The results of this comparison are summarized in Table 2. Here, the column labeled "# Bench. Solved" shows how many of the 30 ADTs were correctly refactored by each tool. Note that we consider a benchmark to be "solved" if the tool is able to correctly refactor *all* ADT methods that require refactoring. To give a more fine-grained view of the results, the next column labeled "# Funcs Refactored" shows the number of ADT methods that were correctly refactored. Finally, the last column labeled "Synth Time" provides the average running time of each tool across all successfully refactored methods in seconds.

The first observation about Table 2 is that the other baselines solve less than a third of the benchmarks solved by Revamp. In particular, JSketch can only solve 4 out of 30 benchmarks and

[4]We evaluated on the May 26, 2023 version of the web application using GPT 3.5

Table 3. Synthetic tasks to evaluate how well ChatGPT performs on unseen benchmarks

Name	# Funcs	# Solved (ChatGPT)	# Solved (Revamp)
Counters	2	1	2
Mercury Timekeeping	2	0	2
Point Summation	2	0	2
Piecewise Function	2	1	2
Incremental Average	2	0	2

correctly refactors only 44 out of the 146 methods. Volt solves 8 of the benchmarks and correctly refactors 83 methods, more than twice the number of methods as JSketch but 60 less than Revamp. ChatGPT, on the other hand, is able to completely solve 7 benchmarks and refactors 92 of the functions, the most among the three baselines. However, upon closer inspection, we found that, in many cases, the ChatGPT result matches the human-refactored version on GitHub *verbatim*, including the same helper functions and local variables names. When we attempted to change the variable and field names slightly, ChatGPT ignored these changes and simply regenerated the (incorrect) version found on GitHub in some cases. These results indicate that there is cross-contamination between training and test data, as ChatGPT has likely been trained on the same GitHub benchmarks.

ChatGPT's Performance on Unseen Benchmarks. To alleviate these concerns about ChatGPT, we performed another experiment on five manually-crafted ADT refactoring tasks described below:

- **Counters:** The original ADT consists of a single field called `map` which maps integers to integers. It consists of two methods `add` and `remove` which inserts a tuple and removes an entry from the map respectively. The refactored ADT has two additional fields `evens` and `odds` which track the number of odd and even keys in `map`. The refactored implementation needs to change `add` and `remove` to update `evens` and `odds` correctly.
- **Mercury Timekeeping:** The original ADT consists of a single field `ts` which tracks the number of seconds elapsed. It consists of two methods `set` and `add_s` which sets and increments the timer. The refactored ADT expresses the time elapsed in terms of years, days, hours, minutes and seconds; however the years and days units are in terms of Mercury years and days.
- **Point Sum:** The original ADT consists of two integer fields `x` and `y` representing a 2D point and two methods `moveY` and `moveX`, both of which take an integer argument `v` and increase `x` and `y` by `v` respectively. The new ADT consists of two integer fields `sum` and `diff` and the RRI specifies that `sum` = `y` + `x` and `diff` = `y` − `x`.
- **Piecewise:** The original ADT implementation consists of an integer field `x` and the new implementation has an integer field `y` and the RRI specifies that `y` = f(`x`) where f is a piecewise linear function. The original implementation consists of two methods: `piecewise` and `add`. The former expresses another piecewise linear function $g(x)$ and the latter takes as input an integer parameter `v` and increments `x` by `v`.
- **Incremental Average:** The original ADT consists of a `List` of integers called `vs` and contains two methods `set` and `add` which appends a list of elements to `vs` and adds an element `v` to `vs`. The new implementation contains an integer field called `avg`, and the RRI specifies that `avg` should equal the (rounded-down) average of the elements in `vs`.

We provide the code of these benchmarks along with our interaction with ChatGPT in the supplementary material.

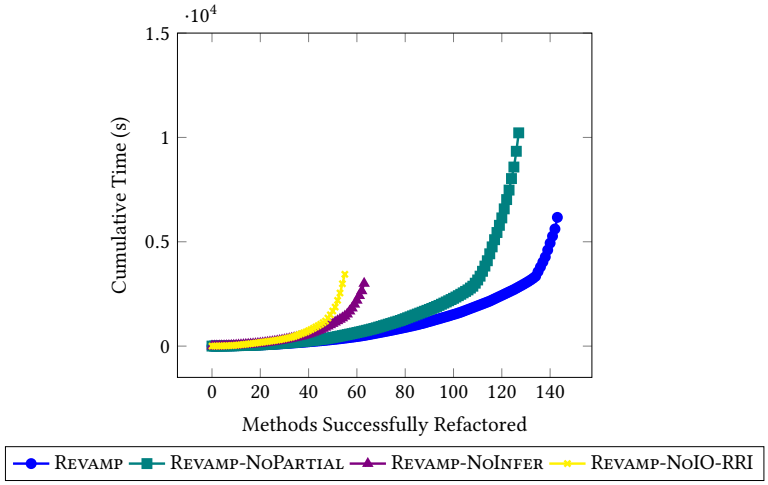

Fig. 9. Ablation results.

The results of this experiment are presented in Table 3. For this benchmark set, ChatGPT was not able to solve any of the benchmarks and could correctly refactor only 2 out of the 10 methods. In contrast, Revamp is able to correctly refactor all benchmarks. Upon closer inspection of these results, we observe that ChatGPT struggles to reason about edge cases. For example, in the Counters task, it correctly increments the counters in the `remove` method but incorrectly increments them in `add` because of an edge case (namely, when the key is already in the map).

Result 2: Revamp is able to solve more than 3x the number of the benchmarks and refactor 62 more functions than the next closest baselines.

6.3 Ablation Study

In this section, we present the results of an ablation study that is designed to assess the relative impact of our key ideas. In particular, we consider three variants of Revamp:

- Revamp-NoIO-RRI is a variant of Revamp that does not infer relational IO examples given a counterexample to equivalence. It instead requires the output of the inductive synthesizer to satisfy the RRI. This variant also cannot infer code snippets using abductive reasoning, as that procedure relies on input-output examples but will combine programs as long as they satisfy the RRI on a subset of the inputs.
- Revamp-NoInfer is a variant that is identical to Revamp except it does not infer additional code snippets using abductive reasoning. It still constructs IO examples and uses partial equivalence to combine programs during synthesis.
- Revamp-NoPartial only differs from Revamp in that it does not use partial equivalence to determine if code snippets should be combined. Instead, it only combines code snippets that completely satisfy an IO example.

The results of this ablation study are presented in Figure 9. Compared to Revamp-NoIO-RRI, and Revamp-NoInfer, Revamp is able to successfully refactor nearly 80 more methods. Finally, Revamp is able to refactor 16 more methods than Revamp-NoInfer, and among the benchmarks that Revamp-NoPartial can solve, Revamp does so nearly 4.2× faster.

Result 3: Each of our three key ideas outlined in Section 2 have a significant impact on Revamp's performance.

7 RELATED WORK

Data structure verification and synthesis. There is a line of work on verifying, repairing, and synthesizing data structure implementations. Demsky and Rinard [Demsky and Rinard 2003a,b, 2005] study runtime error detection and repair of data structures based on boolean constraints, and Lam et al. [Lam et al. 2005a,b] perform static analysis to verify data structure consistency. More recently, there has been a line of work to synthesize data structure methods via deductive synthesis [Delaware et al. 2015; Hawkins et al. 2011, 2012a,b; Itzhaky et al. 2021; Qiu and Solar-Lezama 2017]. Fiat [Delaware et al. 2015] performs deductive synthesis to generate SQL-like query and insertion operations through steps of refinement. RelC [Hawkins et al. 2011, 2012a,b] views a data representation as a set of primitive data structures (e.g., List, Set, Map) and synthesizes operations from a relational algebra description and functional dependencies. Similarly, Cozy [Loncaric et al. 2018, 2016] synthesizes efficient implementations of complex collection data structures from high-level specifications. By contrast, Revamp does not restrict ADT implementations to only use primitive data structures. Furthermore, Revamp focuses on refactoring existing ADT implementations instead of synthesizing an implementation from a high-level description. Among this line of work, Volt [Pailoor et al. 2021] is the most closely related to Revamp. In particular, given a data structure, a new set of auxiliary fields, and an integrity constraint, Volt can automatically refine the data structure in a way that satisfies the specified integrity constraint. However, Revamp studies a more general semantic code refactoring problem for ADTs, where the relationship between two versions of ADTs are specified using relational representation invariants. In particular, Volt is designed to only synthesize *updates* to existing code whereas Revamp handles a broader class of refactorings which can include *updates*, *reads*, *additions*, and *deletions* of an ADT implementation.

Quantifier elimination in synthesis. Recall that Revamp uses quantifier elimination to solve the logical abduction problem introduced in Section 4.4. However, we note that Revamp is not the first work to use quantifier elimination in the context of program synthesis. Comfusy [Kuncak et al. 2010a,b, 2012] and AE-VAL [Fedyukovich et al. 2019] apply quantifier-elimination within a deductive synthesizer to incrementally rewrite a logical specification over integer and rational arithmetic into straight-line code. Unlike Revamp, both procedures are fully deductive and operate on a restricted language expressing loop-free arithmetic code. As such, these techniques are not directly applicable in our setting. Unlike these approaches, Revamp uses quantifier elimination within a CEGIS-loop to learn new snippets that will be useful for synthesis from previous failed attempts.

Constraint-based program synthesis. Constraint-based program synthesis has been studied extensively and applied to many scenarios, such as writing bit-manipulating programs [Gulwani and Venkatesan 2009; Jha et al. 2010] and generating Datalog programs [Albarghouthi et al. 2017] for program analysis. Several frameworks are developed for general constraint-based program synthesis, including Sketch [Solar-Lezama et al. 2008, 2006], JSketch [Jeon et al. 2015], Rosette [Torlak and Bodík 2014], and CVC5 [Barbosa et al. 2022]. Revamp does not reduce the ADT code refactoring problem into a constraint-solving problem directly. Instead, Revamp learns new code snippets to be used in synthesis by solving a logical abduction problem.

Relational program synthesis. Revamp is also related to a line of work [Hu and D'Antoni 2017; Miltner et al. 2018, 2019; Srivastava et al. 2011; Wang et al. 2018] on relational program synthesis, where the goal is to synthesize programs based on relational specifications that relate

multiple programs or multiple runs of a program. For example, Relish [Wang et al. 2018] leverages hierarchical finite tree automata to synthesize comparators, string encoders and decoders. Genic [Hu and D'Antoni 2017] and PINS [Srivastava et al. 2011] study the program inversion problem [Dijkstra 1978] using symbolic extended finite transducers and path-based inductive synthesis, respectively. At a high level, Revamp can be viewed as solving a new relational program synthesis problem specified by the relational representation invariant between two ADTs. As discussed in Section 1, the synthesis problem is challenging for existing techniques due to the complexity of the new data representation and the large search space of the new ADT implementation. Revamp makes the synthesis feasible by using symbolic reasoning to identifying code snippets likely to be used by the refactored implementation and a notion of partial equivalence to quickly combine code snippets into larger programs.

Strengthening specifications in synthesis. There is a line of related work which also strengthens specifications during synthesis by generating I/O examples. In particular, Toshokan [Huang and Qiu 2022] is a synthesis framework for Java that allows users to synthesize code involving library functions without providing models or axioms for the libraries. Instead, Toshokan iteratively builds a model of the library on-the-fly by running the actual library function on the counterexample inputs generated in each iteration of its CEGIS loop to recover specific input-output examples. Like Toshokan, JDial [Hu et al. 2019] also iteratively strengthens models of external functions by executing the concrete library function on unseen inputs during synthesis. However, both of these approaches are specific to strengthening the models of external functions and so cannot be used to strengthen the overall synthesis specification in our setting.

Automatic program refactoring. Another line of work related to Revamp is automatic program refactoring [Altidor and Smaragdakis 2014; Ge et al. 2012; Kataoka et al. 2001; Tip et al. 2011; Wang et al. 2020; Yaghmazadeh et al. 2018]. Given that the refactoring process can be tedious, sub-optimal, and error-prone, researchers have studied automatic refactoring approaches in various scenarios, such as optimizing database applications [Cheung et al. 2013], evolving database schemas [Wang et al. 2019, 2020; Yaghmazadeh et al. 2018], and improving gas efficiency of smart contracts [Chen et al. 2022]. To facilitate automatic refactoring, prior work has leveraged different kinds of specifications, including invariants [Kataoka et al. 2001], type constraints [Tip et al. 2011], inner-class equivalence predicates [Samak et al. 2019], and integrity constraints [Pailoor et al. 2021]. Unlike prior work, Revamp introduces a new way to specify ADT refactorings using relational representation invariants and uses the RRIs to construct I/O examples for each method.

8 CONCLUSION

We introduced the semantic ADT refactoring problem, which requires generating a new implementation of an ADT for a new data representation. We also introduced a novel technique, based on inductive synthesis, for solving this problem. Our algorithm takes as input the old ADT implementation, a new data representation, and a relational representation invariant and automatically synthesizes the new implementation of each ADT method.

We have implemented our ideas as a new tool called Revamp for refactoring Java classes given a suitable relational representation invariant expressed as a boolean function. We evaluated Revamp on 30 ADT refactoring tasks that collectively require refactoring over 140 methods. Revamp is able to successfully refactor 97% of the benchmarks (i.e., classes) and 99% of the method implementations. Furthermore, while the manual refactoring introduces a subtle bug in one of the benchmarks, the Revamp-synthesized implementation does not suffer from this problem. We also compared Revamp against several baselines (JSketch, Volt, and ChatGPT) and showed that they are far inferior to Revamp in terms of the percentage of classes/methods they can correctly refactor. Finally, we

also presented several ablation studies and demonstrated that the three key ideas underlying our approach are all important for its real-world practicality.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Xiaokang Qiu for their thoughtful and constructive feedback. This material is based upon work partially supported by National Science Foundation under Grant Nos. CCF-1762299 and CCF-1918889. 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.

REFERENCES

2003. bind8 negative cache poison attack. https://vulners.com/freebsd/F04CC5CB-2D0B-11D8-BEAF-000A95C4D922.

2005. CVE-2005-0034. https://nvd.nist.gov/vuln/detail/CVE-2005-0034.

2009. Linux devs exterminate security bugs from kernel. https://www.theregister.com/2009/12/11/linux_kernel_bugs_patched/.

2013. Google Cloud Platform (GCP). https://cloud.google.com/.

2022. Cassandra. https://github.com/apache/cassandra.

2022. Elessandra. https://github.com/strapdata/elassandra.

2022. How refactoring code in Safari's WebKit resurrected 'zombie' security bug. https://www.theregister.com/2022/06/21/apple-safari-zombie-exploit/.

2022. Netty. https://github.com/netty/netty.

2023. Glide. https://github.com/bumptech/glide.

2023. Wicket. https://github.com/apache/wicket.

Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal Specification Synthesis. *SIGPLAN Not.* 51, 1 (jan 2016), 789–801. https://doi.org/10.1145/2914770.2837628

Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. 2017. Constraint-Based Synthesis of Datalog Programs. In *Principles and Practice of Constraint Programming - 23rd International Conference (CP) (Lecture Notes in Computer Science, Vol. 10416)*. Springer, 689–706. https://doi.org/10.1007/978-3-319-66158-2_44

John Altidor and Yannis Smaragdakis. 2014. Refactoring Java generics by inferring wildcards, in practice. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. ACM, 271–290. https://doi.org/10.1145/2660193.2660203

Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. 2015. Synthesis through Unification. *CoRR* abs/1505.05868 (2015). arXiv:1505.05868 http://arxiv.org/abs/1505.05868

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference (TACAS) (Lecture Notes in Computer Science, Vol. 13243)*. Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Yanju Chen, Yuepeng Wang, Maruth Goyal, James Dong, Yu Feng, and Isil Dillig. 2022. Synthesis-Powered Optimization of Smart Contracts via Data Type Refactoring. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 145:1–145:29.

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*. ACM, 3–14. https://doi.org/10.1145/2491956.2462180

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.

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *The 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

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 (POPL)*. ACM, 689–700. https://doi.org/10.1145/2676726.2677006

Brian Demsky and Martin C. Rinard. 2003a. Automatic detection and repair of errors in data structures. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 78–95. https://doi.org/10.1145/949305.949314

Brian Demsky and Martin C. Rinard. 2003b. Static Specification Analysis for Termination of Specification-Based Data Structure Repair. In *14th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 71–84. https://doi.org/10.1109/ISSRE.2003.1251032

Brian Demsky and Martin C. Rinard. 2005. Data structure repair using goal-directed reasoning. In *27th International Conference on Software Engineering (ICSE)*. ACM, 176–185. https://doi.org/10.1145/1062455.1062499

DiffChecker. 2021. *DiffChecker*. https://www.diffchecker.com/

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (aug 1975), 453–457. https://doi.org/10.1145/360933.360975

Edsger W. Dijkstra. 1978. Program Inversion. In *Program Construction, International Summer School, July 26 - August 6, 1978, Marktoberdorf, Germany (Lecture Notes in Computer Science, Vol. 69)*. Springer, 54–57. https://doi.org/10.1007/BFb0014657

Isil Dillig and Thomas Dillig. 2013. Explain: A Tool for Performing Abductive Inference. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 684–689.

Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated error diagnosis using abductive inference. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 181–192. https://doi.org/10.1145/2254064.2254087

Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. ACM, 443–456. https://doi.org/10.1145/2509136.2509511

Grigory Fedyukovich, Arie Gurfinkel, and Aarti Gupta. 2019. Lazy but Effective Functional Synthesis. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11388)*, Constantin Enea and Ruzica Piskac (Eds.). Springer, 92–113. https://doi.org/10.1007/978-3-030-11245-5_5

John Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. *ACM SIGPLAN Notices* 50 (06 2015), 229–239. https://doi.org/10.1145/2813885.2737977

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 234–245. https://doi.org/10.1145/512529.512558

Xi Ge, Quinton L. DuBose, and Emerson R. Murphy-Hill. 2012. Reconciling manual and automatic refactoring. In *34th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 211–221. https://doi.org/10.1109/ICSE.2012.6227192

Sumit Gulwani and Madan Musuvathi. 2008. Cover Algorithms and Their Combination. In *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems* (Budapest, Hungary) *(ESOP'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 193–207.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. *Enumerative Search*. Now Publishers Inc., 57–64.

Sumit Gulwani and Ramarathnam Venkatesan. 2009. *Component Based Synthesis Applied to Bitvector Circuits*. Technical Report MSR-TR-2010-12. https://www.microsoft.com/en-us/research/publication/component-based-synthesis-applied-to-bitvector-circuits/

John V. Guttag, Ellis Horowitz, and David R. Musser. 1978. Abstract Data Types and Software Validation. *Commun. ACM* 21, 12 (dec 1978), 1048–1064. https://doi.org/10.1145/359657.359666

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. 2011. Data representation synthesis. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 38–49. https://doi.org/10.1145/1993498.1993504

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. 2012a. Concurrent data representation synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*. ACM, 417–428. https://doi.org/10.1145/2254064.2254114

Peter Hawkins, Martin C. Rinard, Alex Aiken, Mooly Sagiv, and Kathleen Fisher. 2012b. An introduction to data representation synthesis. *Commun. ACM* 55, 12 (2012), 91–99. https://doi.org/10.1145/2380656.2380677

Qinheping Hu and Loris D'Antoni. 2017. Automatic program inversion using symbolic transducers. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 376–389. https://doi.org/10.1145/3062341.3062345

Qinheping Hu, Roopsha Samanta, Rishabh Singh, and Loris D'Antoni. 2019. Direct Manipulation for Imperative Programs. In *Static Analysis: 26th International Symposium, SAS 2019, Porto, Portugal, October 8–11, 2019, Proceedings* (Porto, Portugal). Springer-Verlag, Berlin, Heidelberg, 347–367. https://doi.org/10.1007/978-3-030-32304-2_17

Kangjing Huang and Xiaokang Qiu. 2022. Bootstrapping Library-Based Synthesis. In *Static Analysis: 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5–7, 2022, Proceedings* (Auckland, New Zealand). Springer-Verlag, Berlin, Heidelberg, 272–298. https://doi.org/10.1007/978-3-031-22308-2_13

Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021. Cyclic Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, 944–959. https://doi.org/10.1145/3453483.3454087

Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. 2015. JSketch: sketching for Java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 934–937. https://doi.org/10.1145/2786805.2803189

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 215–224. https://doi.org/10.1145/1806799.1806833

Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. 2001. Automated Support for Program Refactoring Using Invariants. In *International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 736–743. https://doi.org/10.1109/ICSM.2001.972794

Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2010a. Comfusy: A Tool for Complete Functional Synthesis. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 430–433.

Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2010b. Complete Functional Synthesis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 316–329. https://doi.org/10.1145/1806596.1806632

Viktor Kuncak, Mikael Mayer, Ruzica Piskac, and Philippe Suter. 2012. Software Synthesis Procedures. *Commun. ACM* 55 (02 2012), 103–111. https://doi.org/10.1145/2076450.2076472

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

Patrick Lam, Viktor Kuncak, and Martin C. Rinard. 2005a. Generalized Typestate Checking for Data Structure Consistency. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference (VMCAI) (Lecture Notes in Computer Science, Vol. 3385)*. Springer, 430–447. https://doi.org/10.1007/978-3-540-30579-8_28

Patrick Lam, Viktor Kuncak, and Martin C. Rinard. 2005b. Hob: A Tool for Verifying Data Structure Consistency. In *Prcoeedings of the 14th International Conference on Compiler Construction (CC) (Lecture Notes in Computer Science, Vol. 3443)*. Springer, 237–241. https://doi.org/10.1007/978-3-540-31985-6_16

Woosuk Lee. 2021. Combining the Top-down Propagation and Bottom-up Enumeration for Inductive Program Synthesis. *Proc. ACM Program. Lang.* 5, POPL, Article 54 (jan 2021), 28 pages. https://doi.org/10.1145/3434335

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

Calvin Loncaric, Michael D. Ernst, and Emina Torlak. 2018. Generalized data structure synthesis. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 958–968. https://doi.org/10.1145/3180155.3180211

Calvin Loncaric, Emina Torlak, and Michael D. Ernst. 2016. Fast synthesis of fast collections. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 355–368. https://doi.org/10.1145/2908080.2908122

Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing bijective lenses. *Proc. ACM Program. Lang.* 2, POPL (2018), 1:1–1:30. https://doi.org/10.1145/3158089

Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2019. Synthesizing symmetric lenses. *Proc. ACM Program. Lang.* 3, ICFP (2019), 95:1–95:28. https://doi.org/10.1145/3341699

Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. 2020. Data-Driven Inference of Representation Invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3385412.3385967

OpenAI. 2021. *ChatGPT*. https://openai.com

Shankara Pailoor, Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2021. *Synthesizing Data Structure Refinements from Integrity Constraints*. Association for Computing Machinery, New York, NY, USA, 574–587. https://doi.org/10.1145/3453483.3454063

Xiaokang Qiu and Armando Solar-Lezama. 2017. Natural Synthesis of Provably-Correct Data-Structure Manipulations. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 65 (oct 2017), 28 pages. https://doi.org/10.1145/3133889

Malavika Samak, Deokhwan Kim, and Martin C. Rinard. 2019. Synthesizing Replacement Classes. *Proc. ACM Program. Lang.* 4, POPL, Article 52 (dec 2019), 33 pages. https://doi.org/10.1145/3371120

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. https://doi.org/10.1145/3290386

Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. 2008. Sketching concurrent data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*. ACM, 136–148. https://doi.org/10.1145/1375581.1375599

Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. 2005. Programming by sketching for bit-streaming programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 281–294. https://doi.org/10.1145/1065010.1065045

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 404–415. https://doi.org/10.1145/1168857.1168907

Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. 2011. Path-based inductive synthesis for program inversion. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 492–503. https://doi.org/10.1145/1993498.1993557

Frank Tip, Robert M. Fuhrer, Adam Kiezun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. 2011. Refactoring using type constraints. *ACM Trans. Program. Lang. Syst.* 33, 3 (2011), 9:1–9:47. https://doi.org/10.1145/1961204.1961205

Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 530–541. https://doi.org/10.1145/2594291.2594340

Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 286–300. https://doi.org/10.1145/3314221.3314588

Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data Migration Using Datalog Program Synthesis. *Proc. VLDB Endow.* 13, 7 (mar 2020), 1006–1019. https://doi.org/10.14778/3384345.3384350

Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018. Relational program synthesis. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 155:1–155:27. https://doi.org/10.1145/3276525

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

Received 2023-07-11; accepted 2023-11-07