# Complexity-Guided Container Replacement Synthesis

CHENGPENG WANG, The Hong Kong University of Science and Technology, China
PEISEN YAO, The Hong Kong University of Science and Technology, China
WENSHENG TANG, The Hong Kong University of Science and Technology, China
QINGKAI SHI, Ant Group, China
CHARLES ZHANG, The Hong Kong University of Science and Technology, China

Containers, such as lists and maps, are fundamental data structures in modern programming languages. However, improper choice of container types may lead to significant performance issues. This paper presents Cres, an approach that automatically synthesizes container replacements to improve runtime performance. The synthesis algorithm works with static analysis techniques to identify how containers are utilized in the program, and attempts to select a method with lower time complexity for each container method call. Our approach can preserve program behavior and seize the opportunity of reducing execution time effectively for general inputs. We implement Cres and evaluate it on 12 real-world Java projects. It is shown that Cres synthesizes container replacements for the projects with 384.2 KLoC in 14 minutes and discovers six categories of container replacements, which can achieve an average performance improvement of 8.1%.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Software performance**; **Data types and structures**.

Additional Key Words and Phrases: program synthesis, program optimization, data structure specification.

**ACM Reference Format:**
Chengpeng Wang, Peisen Yao, Wensheng Tang, Qingkai Shi, and Charles Zhang. 2022. Complexity-Guided Container Replacement Synthesis. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 68 (April 2022), 31 pages. https://doi.org/10.1145/3527312



## 1 INTRODUCTION

General-purposed programming languages, including Java and C++, support a variety of containers, which creates great convenience of developing software systems. Unfortunately, performance issues often emerge because of inefficient usage of container types. Programmers are often unaware of more efficient container types under their development context and tend to choose the container types that they are most familiar with. For example, in the program shown in Figure 1, the use of the container type ArrayList introduces unnecessary time overhead because the method, ArrayList.contains, performs linear searching. The same functionality can be supported efficiently by the class HashSet. It is quite surprising to find that 16% of execution time of the 3D design software, Raytrace, is introduced by inefficient container types [Jung et al. 2011], affecting the performance of ray tracing greatly. Moreover, there is abundant evidence that inefficient containers also increase

Authors' addresses: Chengpeng Wang, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, China, cwangch@ust.cse.hk; Peisen Yao, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, China, pyao@cse.ust.hk; Wensheng Tang, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, China, wtangae@cse.ust.hk; Qingkai Shi, Ant Group, China, qingkai.sqk@antgroup.com; Charles Zhang, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, China, charlesz@cse.ust.hk.


2475-1421/2022/4-ART68
https://doi.org/10.1145/3527312

```
public List load(String[] a, int n) {
  List<String> u = new ArrayList<>();
  for (int i = 0; i < n; i++)
    if (!u.contains(a[i])) u.add(a[i]);
  return u;
}
```

```
public void check(String[] a, String s) {
  List v = load(a, a.length);
  if (v.contains(s))
    return true;
  return false;
}
```

Fig. 1. An efficient usage of ArrayList in the project iotdb

other resource consumption, including memory [Basios et al. 2018; Jung et al. 2011], energy [Hasan et al. 2016; Manotas et al. 2014; Oliveira et al. 2021], and CPU usage [Basios et al. 2018].

**Goal and Challenge.** Given a set of container types, our goal is to synthesize alternative container types and the associated methods at the container allocation sites and the container method call sites, respectively, such that the program after replacements preserves the original semantics and executes more efficiently for large inputs. We also expect our synthesis algorithm to be general enough, supporting the program optimization to decrease other kinds of resource consumptions, such as the memory and CPU usage.

However, it is far from trivial to achieve the goal. First, we can not determine the container types to replace the original ones without violating the behavioral equivalence [Nicola 2011] if we do not know how container objects are manipulated. Although several container types are interchangeable, e.g., ArrayList and LinkedList, the replacement patterns might be quite restrictive. Second, it is far from practical to derive a tight bound of the time complexity for a general container-manipulating program [Gulwani et al. 2009a,c; Wilhelm et al. 2008], so we can not explicitly compare the complexity of the program before and after replacements to guide the synthesis.

**Existing Effort.** The existing works attempt to tackle the problem from two perspectives. One line of the existing approaches attempts to find optimal container usage by minimizing the resource consumption upon a given test suite [Basios et al. 2018; Manotas et al. 2014]. They mutate container types and evaluate the resource consumption via dynamic profiling until the minimal consumption is reached. The other line of the works selects better container types by performing a prediction task [Jung et al. 2011; Kennedy and Ziarek 2015; Shacham et al. 2009]. Based on the heap information and container usage patterns in a specific execution, they predict optimal container types by utilizing a prediction model, which is specified manually or obtained in a training process. Unfortunately, the existing works suffer from three drawbacks:

- *Huge time overhead.* They rely on the execution of the test suite, making the whole process quite time consuming [Jung et al. 2011; Manotas et al. 2014]. Particularly, the first line of the works executes the test suite iteratively to find the optimal selection and suffer the huge time overhead. For example, [Basios et al. 2018] takes 3.1 hours optimizing a project on average, which poses an enormous obstacle to large-scale adoption.
- *Unsoundness.* They can not guarantee the semantic equivalence of the program, as they can not discover how each container object is manipulated and determine the equivalent container types soundly. Although several approaches assume several container types are interchangeable [Basios et al. 2018; Jung et al. 2011], the assumptions do not hold in certain cases, such as transforming LinkedHashMap to HashMap in the presence of map traversal.
- *Overfitting.* The effectiveness of the optimization can be degraded when the test suite or the training data does not provide general inputs. The program after replacements can execute slower when the inputs exercise the program along previously uncovered paths [Xu 2013].

**Insight and Solution.** We observe that container method calls reveal the intention of the programmers for which they use the containers. Specifically, programmers concern with specific

*container properties*, such as the size, index or value-ownership, and index-value correlation. Container methods allow programmers to manipulate a container object by querying and modifying container properties. Our insight is that we can optimize a container-manipulating program if the intention can be achieved by other container types and methods with lower time complexity. In Figure 1, for example, the ArrayList object allocated in the method load is only manipulated by the methods ArrayList.add and ArrayList.contains. The programmers only wish to know whether an element is stored in the list, i.e., the value-ownership property of the ArrayList object. Thus, we can replace ArrayList with HashSet to avoid linear searching caused by ArrayList.contains, thereby improving program efficiency.

Based on the insight, we present Cres, a container replacement synthesizer to improve program efficiency. Cres synthesizes container replacements preserving program behavior and achieves the optimization for general inputs.

- To assure the behavioral equivalence, we propose the notion of *container behavioral equivalence* to determine the method candidates. Specifically, Cres analyzes container method calls to determine the concerned container properties. A method is a candidate of a container method call if it queries and modifies the concerned container properties in the same way as the original one.
- To achieve the optimization, we introduce the concept of *container complexity superiority* to constrain the complexity of container methods in the replacements. Specifically, Cres selects methods with low complexity from candidates so that the total complexity of the container method calls manipulating the object is lower than the one in the original program.

With the benefit of our insight, Cres can find the opportunity of achieving input-agnostic optimization and improve program efficiency significantly. To the best of our knowledge, Cres is the first work to guarantee the behavior equivalence without any assumption on interchangeable container types. Moreover, Cres escapes from the burden of huge overhead because it does rely on program execution and performs efficient static reasoning.

We evaluate Cres upon 12 real-world Java projects with intensive usage of containers in Java Collections Framework (JCF), of which the sizes range from 18.6 KLoC to 384.2 KLoC. Cres synthesizes 107 replacements covering six categories [Cres 2021], such as replacing ArrayList with HashSet, replacing TreeMap with HashMap, etc. Particularly, 71 replacements in six projects have been confirmed by the developers. The time consumption of each project is decreased by 8.1% on average after replacements. Moreover, Cres finishes analyzing any project in 14 minutes, which distinguishes it from existing approaches suffering from the heavy time burden [Basios et al. 2018; Manotas et al. 2014]. We also prove its soundness theoretically to guarantee the behavioral equivalence. Cres has been integrated into the static analysis platform in the Ant Group, an international IT company providing the financial service for over 1 billion global users. In summary, we make the following main contributions:

- We propose a novel abstraction of containers and introduce two principled notions, namely *container behavioral equivalence* and *container complexity superiority*, to guide the synthesis.
- We establish an abstract domain and propose the *container property analysis* to guarantee the behavioral equivalence of the programs.
- We implement a synthesis framework Cres and evaluate it on real-world Java applications, showing that it synthesizes replacements efficiently and significantly improves program efficiency.

## 2 CRES IN A NUTSHELL

In this section, we present a motivating example to state the importance of replacing inefficient containers for program efficiency improvement (§ 2.1), and illustrate the key idea of our approach to solving the problem of complexity-guided container replacement synthesis (§ 2.2).

```
1  public List getResources(String dir) {
2    List r = new ArrayList<File>(); //o2
3    for (int i = 0; i < RES_NUM; i++) {
4      File s = getFile(dir, i);
5      if (!r.contains(s))
6        r.add(s);
7    }
8    return r;
9  }
10 public List getPrivate(String dir) {
11   List p = new ArrayList<File>(); //o11
12   for (int i = 0; i < N_PRIVATE; i++)
13     p.add(getPrivateFile(dir, i));
14   return p;
15 }
16 public List getProtected(String dir) {
17   List q = new ArrayList<File>(); //o17
18   for (int i = 0; i < N_PROTECTED; i++)
19     q.add(getProtectedFile(dir, i));
20   return q;
21 }
22 public boolean invisible(ArrayList l) {
23   return l.contains(INVISIBLE_FILE);
24 }
25 public List getAllFiles(String dir) {
26   List f = new ArrayList<File>(); //o26
27   for (int i = 0; i < N_FILE; i++)
28     f.add(getFile(dir, i));
29   return f;
30 }
31 public void access(String dir, int token) {
32   List f = getAllFiles(dir);
33   List r = getResources("/OOPSLA");
34   List p = getPrivate("/Data");
35   List q = getProtected("/Data");
36   for (File file : f) {
37     if (!invisible(p)
38             && p.contains(file))
38       continue;
39     if (!invisible(q)
40             && q.contains(file)
41             && q.indexOf(file) > token)
42       continue;
43     if (r.contains(file))
44       System.out.println("Access");
45   }
46 }
```

Fig. 2. A motivating program accessing the available and visible files in a specific directory

## 2.1 Motivating Example

Figure 2 presents the example extracted and simplified from the projects iotdb and google-http-java-client. The container objects $o_2$, $o_{11}$, $o_{17}$, and $o_{26}$ are allocated by the allocation statements at lines 2, 11, 17, and 26, respectively. The methods getAllFiles and getResources collect the files in the directories named *dir* and OOPSLA, and store them in $o_{26}$ and $o_2$, respectively. The methods getPrivate and getProtected collect the files demanding two different access privileges and store them in two ArrayList objects $o_{11}$ and $o_{17}$, respectively. We can obtain three observations as follows.

- The ArrayList object $o_2$ is manipulated by the methods ArrayList.add and ArrayList.contains, so the programmers only wish to know whether an element is stored in $o_2$, i.e., the value-ownership of $o_2$. Notice that a HashSet object also supports the value-ownership checking and returns the same result. Besides, the method HashSet.contains works with amortized constant time. Therefore, the program will be more efficient if we replace ArrayList with HashSet at line 2.
- The ArrayList object $o_{26}$ is manipulated by the insertions and traversal. The methods of LinkedList also support the same functionalities. Besides, the method LinkedList.add runs in constant time complexity while the method ArrayList.add works with amortized constant time because of memory reallocation. Thus, we can replace ArrayList with LinkedList to reduce time consumption.
- The ArrayList object $o_{11}$ is created for the value-ownership checking. However, $o_{11}$ and $o_{17}$ are provided as the parameters of the method invisible. The programmers are concerned about the index-value correlation of $o_{17}$ in the invocation of the method ArrayList.indexOf. If we replace the type of $o_{11}$ with HashSet, the code cleanliness can be degraded, as the method invisible should be inlined at two call sites. Thus, we only leverage LinkedList to avoid memory reallocation.

The replacements can bring a significant improvement in program efficiency. For example, the total execution time of the corresponding test cases in google-http-java-client can be reduced by 27.1% if we replace an ArrayList object with a LinkedList object. A large body of literature also reveals that inefficient container types can introduce unnecessary time consumption and even increase time complexity [Basios et al. 2018; Jung et al. 2011; Shacham et al. 2009]. Thus, it is meaningful to synthesize container replacements to improve program efficiency.

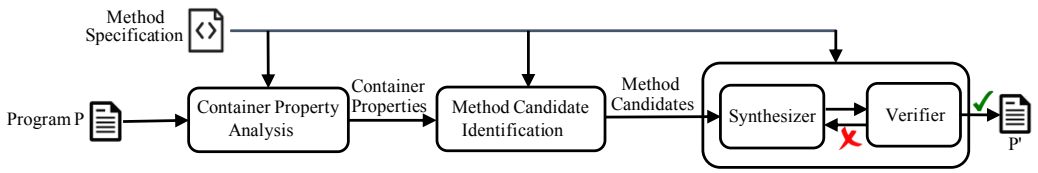


Fig. 3. Schematic overview of our approach

## 2.2 Synthesizing Replacement

The synthesized container replacements should preserve the program behavior and achieve the optimization for large inputs. Unfortunately, it is non-trivial to find the replacements satisfying the two constraints. First, we can not determine which container types can guarantee the behavioral equivalence after replacements if we do not know how container objects are manipulated. Even if several container types are interchangeable in any usage context, such as LinkedList and ArrayList, more general replacements, such as transforming the type of $o_2$ to HashSet, can not be discovered [Basios et al. 2018; Oliveira et al. 2021; Shacham et al. 2009]. Second, it is far from practical to derive a tight bound of the time complexity for a real-world program [Gulwani et al. 2009a,c; Wilhelm et al. 2008]. We need an effective and computable measure to guide the synthesis such that the synthesis can achieve the input-agnostic optimization for large inputs.

The key idea of our approach comes from the observation about the intention of container usage. We realize that the purpose of programmers is to utilize specific facts about containers, which we call *container properties*. Programmers can query and update the container properties by invoking container methods. In Figure 2, for example, the programmers are concerned about the value-ownership of $o_2$, i.e., the fact that whether an object is stored in $o_2$. The methods ArrayList.contains and ArrayList.add query and update the value-ownership, respectively. When the concerned properties can be updated and queried by more efficient methods in the same way, we can replace the types and methods to improve program efficiency. Specifically, we propose two concepts to address the challenges:

- We introduce *container behavioral equivalence* to determine the methods that query and update the concerned properties in the same way as the original ones. For instance, only the value-ownership of $o_2$ is concerned in Figure 2, and the methods of HashSet guarantee the container behavioral equivalence. Thus, replacing it with a HashSet object preserves the behavioral equivalence.
- We propose *container complexity superiority* to measure whether the methods manipulating a container object are more efficient after replacements. In Figure 2, HashSet.contains has much lower time complexity than ArrayList.contains, and HashSet.add and ArrayList.add do not have significant difference in complexity. After transforming the type from ArrayList to HashSet, the new program has container complexity superiority.

Based on our insight, we can improve the program efficiency for general inputs if the replacements guarantee the container behavioral equivalence and the container complexity superiority simultaneously. Figure 3 shows the workflow of our approach, which consists of three stages.

- In the first stage, the container property analysis identifies which container properties are queried and how they are updated upon each container object in the program. For example, it can discover that only the value-ownership is queried upon $o_2$ and $o_{11}$ in Figure 2.
- In the second stage, the methods are identified as the method candidates if they the preserve container behavioral equivalence. For instance, the methods HashSet.contains and HashSet.add are the candidates of the container method calls at line 5, 6, and 43, as they query and update the value-ownership in the same way as the methods ArrayList.contains and ArrayList.add.

$$
\begin{aligned}
&\textit{Program}\ \ P := F+\\
&\quad\textit{Container method}\ \ F_C := f_C(v_1, \ldots, v_m)\\
&\textit{Function}\ \ F := f(v_1, \ldots, v_n)\{S;\ \ \textbf{return}\ e\}\\
&\textit{Statement}\ \ S := v = \textbf{new}\ \ \tau \mid v = e \mid S_1; S_2 \mid \textbf{return}\ e\\
&\qquad\qquad \mid \textbf{if}\ (e)\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \mid \textbf{while}\ \ (e)\ \ \textbf{do}\ \ S\ \ \textbf{od}\\
&\qquad\qquad \mid v = c.f_C(v_1, \ldots, v_m) \mid v = f(v_1, \ldots, v_n)\\
&\textit{Expression}\ \ e := a \mid v \mid u_1 \oplus u_2 \mid\ \otimes u\\
&\textit{Variable}\ \ v := c \mid u\\
&\textit{Operator}\ \ \oplus := \wedge \mid\ \vee\ \mid\ +\ \mid\ -\ \mid\ =\ \mid \cdots \qquad \otimes := \neg \mid\ -\ \mid \cdots
\end{aligned}
$$

Fig. 4. The syntax of the language

- In the third stage, we instantiate a CEGIS paradigm [Alur et al. 2013; Gulwani et al. 2011; Solar-Lezama et al. 2008] to synthesize container types and methods. A synthesizer selects efficient method candidates and resolves the counterexamples in the consequent rounds if type checking fails in the verification. For instance, the actual parameters of the method invisible are inconsistent at lines 37 and 39 if we replace $o_{11}$ with a HashSet object, so the synthesizer refines the type of $o_{11}$ by selecting another type LinkedList in a consequent round.

Specifically, the generation and selection of method candidates both rely on sound reasoning about the queried container properties and how they are updated in the program. Technically, we establish an abstract domain to abstract the container-property queries in the program, which guide the generation of method candidates and further guarantee the container behavioral equivalence.

With the benefits of our insight, our approach stands out due to the following three perspectives.

- *The low overhead introduced by the synthesis.* The synthesis algorithm does not rely on any input and execution of the program, and static reasoning of container properties is sufficient to identify candidates with quite low overhead.
- *Sound and various replacements.* The algorithm analyzes the concerned container properties to guide method candidate identification, which not only guarantees the behavioral equivalence but also discovers the replacements uncovered by existing approaches, such as replacing ArrayList with HashSet, and replacing LinkedHashMap with HashMap.
- *Input-agnostic optimization.* The algorithm utilizes the complexity specification of container methods to guide the synthesis so that the replacements are insensitive to the program inputs, and the time complexity of the program is more likely to be decreased.

## 3 PROBLEM FORMULATION

In this section, we first present the language used in this paper and its concrete state (§ 3.1). We then define the behavioral equivalence (§ 3.2) to constrain the program behavior after container replacements. Finally, we formalize the problem of complexity-guided container replacement synthesis (§ 3.3).

### 3.1 Program Syntax and Concrete State

Let $\mathcal{C}$ and $\mathcal{M}$ denote the family of the container types and their methods, respectively. Also, we let method denote the function mapping a container type $\tau$ to the set of container methods supported by $\tau$. Figure 4 shows the syntax of the language. The expressions include literals, variable expressions, and unary/binary expressions. A statement can be an allocation statement, an

```
public boolean foo1(String a) {
  List<String> l = new ArrayList<>();
  l.put("PL");  l.put("SE");
  boolean b1 = l.contains(a);
  return b1;
}
```

```
public boolean foo2(String a) {
  Set<String> s = new HashSet<>();
  s.add("PL");  s.add("SE");
  boolean b2 = s.contains(a);
  return b2;
}
```

Fig. 5. Two behaviorally equivalent programs

assignment, a sequencing, a branch, a loop, a function call, or a return statement. Particularly, a function call is either an invocation of a user-defined function $f$ or a container method $f_C$ with the receiver container $c$. A program has a unique function as its entry, which has a unique return statement.

We denote the sets of program variables and values by Var and Val := Addr ∪ OVal, respectively. Specifically, Addr is a set of addresses of objects, OVal is a set of non-address values, and Idx ⊆ Val includes the index values of the containers. Formally, we can define the **concrete state** as follows.

*Definition 3.1.* (Concrete State) A concrete state $s \in \mathsf{State}$ is a 3-tuple $(\varepsilon, \mu, \beta)$, where

- An *environment* $\varepsilon \in \mathsf{Env} := \mathsf{Var} \to \mathsf{Val}$ maps a set of variables Var to a set of values Val.
- A *memory* $\mu \in \mathsf{Mem} := (\mathsf{Addr}, \mathsf{Idx}) \to \mathsf{Val}$ maps a pair of an address and an index to a value, which is the value stored at the index of a container object.
- A *base* $\beta \in \mathsf{Base} := \mathsf{Addr} \to \mathsf{U}$, where $\mathsf{U} := \{((A, \preceq_1), \ldots, (A, \preceq_k)) \mid A \subseteq \mathsf{Idx}\}$, maps an address to a $k$-tuple, of which the entry is a partially ordered set. Each partial order $\preceq_i$ determines a specific order of the indexes of the container object stored at the address.

The concrete state supports the semantics of container methods with various features. In JCF, for instance, TreeMap supports accessing the value associated to the largest key, and LinkedHashMap supports iterating according to the insertion order. These advanced features can be expressed by specific partial orders in the base.

*Example 3.2.* Consider the function foo1 in Figure 5. At the exit of the function, we have $\varepsilon(l) = a_l$, $\mu(a_l, 0) =$ "PL" and $\mu(a_l, 1) =$ "SE", where $a_l$ is the address where the ArrayList object is allocated. Particularly, we enforce $\beta(a_l)$ equal to $\emptyset$, as its semantics does not rely on any order of the indexes.

### 3.2 Behavioral Equivalence

The program after the replacements should preserve the semantic equivalence. Based on the concrete state, we define the **behavioral equivalence** [Nicola 2011] for two programs to constrain the input-output relationship, which is a specific form of program behavior.

*Definition 3.3.* (Behavioral Equivalence) $P$ is behaviorally equivalent to $P'$, denoted by $P \simeq P'$, if and only if for any input, the expressions $e$ and $e'$ in the return statements of the entry functions evaluate to the same value, i.e., $[\![e]\!](s) = [\![e']\!](s')$. $[\![e]\!]$ is the function mapping a concrete state to the value of the expression. $s$ and $s'$ are the concrete states of $P$ and $P'$ at the exits, respectively.

*Example 3.4.* Suppose that $s_1 = (\varepsilon_1, \mu_1, \beta_1)$ and $s_2 = (\varepsilon_2, \mu_2, \beta_2)$ are the concrete states at the exits of the functions foo1 and foo2 in Figure 5, respectively. We have $\varepsilon_1(b1) = T$ and $\varepsilon_2(b2) = T$ iff $a$ is equal to "PL" or "SE", i.e., $[\![b1]\!](s_1) = [\![b2]\!](s_2)$, indicating that they are behaviorally equivalent.

Behavioral equivalence defines an equivalence relation between two programs based on the input-output relationship, which is a program behavior concerned in many scenarios [Nicola 2011]. The program after replacements should be behaviorally equivalent to the original program, preserving the semantics we are concern about.

### 3.3 Problem Statement

To synthesize container replacements, we identify the allocation statements of container objects and container method calls as the skeleton of synthesis. Given a program $P$, we denote the set of the two kinds of statements by $\mathcal{S}$. In what follows, we let $\mathcal{S}_a \subseteq \mathcal{S}$ and $\mathcal{S}_c \subseteq \mathcal{S}$ denote the sets of container allocation statements and container method calls, respectively. We state the problem of complexity-guided container replacement synthesis as follows.

*Definition 3.5.* (Complexity-Guided Container Replacement) Given a program $P$, we aim to synthesize the replacement mappings $\psi_c : \mathcal{S}_c \to \mathcal{M}$ and $\psi_a : \mathcal{S}_a \to \mathcal{C}$. For $st_c \in \mathcal{S}_c$ and $st_a \in \mathcal{S}_a$, we replace the container method $f_C$ invoked by $st_c$ with $\psi_c(st_c)$, and replace the container type $\tau$ used in $st_a$ with $\psi_a(st_a)$, which should satisfy: (1) Behavioral equivalence: $P'$ and $P$ are behavioral equivalent; (2) Complexity superiority: $P'$ consumes no more time than $P$ for any large input.

Intuitively, the behavioral equivalence and the complexity superiority formulate our expectations on the new program after replacements. To solve the problem, we establish the abstraction for containers in § 4 and design the synthesis algorithm to synthesize the container replacement mappings in § 5.

**Remark.** We only concentrate on the statement-wise replacements synthesis in our problem. A major advantage of performing such a form of replacements is that the program structure is not affected by the replacements. If we conduct more aggressive changes to the code, e.g., defining two functions to replace the invocation of the function invisible at lines 37 and 39 in Figure 2, the program after replacements can have a big difference from the original one, which degrades the code cleanliness and increases the difficulty of the maintenance.

## 4 PROGRAM ABSTRACTION

In this section, we first introduce the notion of the container-property query and establish the abstract states (§ 4.1). We then propose the method semantic specification and define the concept of the container behavioral equivalence to guarantee the behavioral equivalence (§ 4.2). Finally, we define the notion of the container complexity superiority as the heuristic guidance to synthesize replacements satisfying the complexity superiority (§ 4.3).

### 4.1 Container Property Abstraction

As explained in § 2.2, we can represent the intention of utilizing containers by the concerned container properties, which are specific forms of facts about containers. To show the intention of container usage, we define the container-property query (§ 4.1.1) and establish an abstract domain to abstract the concerned container properties (§ 4.1.2).

*4.1.1 Container-Property Query.* We introduce the concept of the *container property* to indicate the intention of container usage. A container property is essentially a numeric quantity or a predicate upon the indexes and the values of a container object. Intuitively, it is a specific form of facts about a container object. Table 1 shows the typical container properties of commonly-used container types in JCF, which depict the following facts.

- size shows the size of a container object, i.e., the number of the values in the container.
- $\mathsf{isIdx}(\lambda)$ and $\mathsf{isVal}(v)$ indicate the index-ownership and value-ownership, respectively. $\lambda$ or $v$ is an index or a value of the container if and only if $\mathsf{isIdx}(\lambda) = T$ or $\mathsf{isVal}(v) = T$.
- $\mathsf{isCor}(\lambda, v)$ indicates the index-value correlation. The index $\lambda$ is paired with the value $v$ if and only if $\mathsf{isCor}(\lambda, v) = T$.
- $\mathsf{InsOrd}(\lambda_1, \lambda_2)$ indicates the insertion order of indexes. $\lambda_1$ is inserted before $\lambda_2$ if and only if $\mathsf{InsOrd}(\lambda_1, \lambda_2) = T$.

Table 1. Examples of container properties

| | size | isIdx($\lambda$) | isVal($v$) | isCor($\lambda, v$) | InsOrd($\lambda_1, \lambda_2$) | KeyOrd($\lambda_1, \lambda_2$) |
|---|---|---|---|---|---|---|
| ArrayList | ✓ | | ✓ | ✓ | | |
| LinkedList | ✓ | | ✓ | ✓ | | |
| HashSet | ✓ | ✓ | ✓ | | | |
| TreeSet | ✓ | ✓ | ✓ | | | ✓ |
| LinkedHashSet | ✓ | ✓ | ✓ | | ✓ | |
| HashMap | ✓ | ✓ | ✓ | ✓ | | |
| TreeMap | ✓ | ✓ | ✓ | ✓ | | ✓ |
| LinkedHashMap | ✓ | ✓ | ✓ | ✓ | ✓ | |

- KeyOrd($\lambda_1, \lambda_2$) indicates the order of keys. $\lambda_1$ is larger than $\lambda_2$ if and only if KeyOrd($\lambda_1, \lambda_2$) = $T$.

Let Property denote the family of the container properties. Based on the concept of the container property, we define the **container-property query** to formalize which container property is utilized by a container method call.

*Definition 4.1.* (Container-Property Query) A container-property query is a function $q$ mapping a pair of a concrete state and a container variable to a container property $p$, i.e.,

$$q : \text{State} \times \text{Var} \longrightarrow \text{Property}$$
$$(s, c) \mapsto p$$

Furthermore, we can construct a family of container-property queries $\mathcal{Q}$ to represent all the possible container-property queries induced by container methods.

*Example 4.2.* Consider the container type LinkedHashMap as an example. The methods LinkedHashMap.containsKey and LinkedHashMap.containsValue induce the queries of the container properties isIdx($\lambda$) and isVal($v$), respectively. Besides, the method LinkedHashMap.get queries the container property isCor($\lambda, v$). Its iterator also queries the container property InsOrd($\lambda_1, \lambda_2$), as its semantics relies on the insertion order.

*4.1.2 Abstract State.* Based on container-property queries, we can establish an abstraction of concrete states in § 3.1. To assure the boundedness of the abstract domain, we adopt the allocation site-based memory abstraction [Kanvar and Khedker 2016], and introduce an abstract object to summarize the memory objects allocated by the same allocation statement. Formally, we define the **abstract state** for container-manipulating programs as follows.

*Definition 4.3.* (Abstract State) $\mathcal{V}_c$ is the set of container variables and $\mathcal{O}_c$ is the set of abstract container objects. An abstract state is $\widetilde{s} = (\widetilde{\varepsilon}, \widetilde{\rho})$, where

- $\widetilde{\varepsilon} : \mathcal{V}_c \longrightarrow 2^{\mathcal{O}_c}$ indicates the points-to fact of container variables. For each container variable $c \in \mathcal{V}_c$, $\widetilde{\varepsilon}(c)$ is the set of abstract container objects which $c$ may point to.
- $\widetilde{\rho} : \mathcal{O}_c \longrightarrow 2^{\mathcal{Q}}$ indicates the property-query fact of container objects. For each container object $o \in \mathcal{O}_c$, $\widetilde{\rho}(o)$ contains the container-property queries occurring upon the object $o$.

*Example 4.4.* Consider the function foo1 in Figure 5. We have $\mathcal{O}_c = \{o_2\}$, where $o_2$ is the ArrayList object allocated at line 2. The container object is only created for the value-ownership checking. Before the return statement, the abstract state is $(\widetilde{\varepsilon}, \widetilde{\rho})$, where $\widetilde{\varepsilon}(l) = \{o_2\}$, $\widetilde{\rho}(o_2) = \{q\}$, and $q(s, c)$ is isVal($v$), indicating that only the value-ownership of $o_2$ is concerned in the program.

Intuitively, an abstract state abstracts away the facts irrelevant to container variables and objects. Based on the abstract state, we can determine how a container object is utilized in the program by identifying (1) which container objects are manipulated and (2) which container properties are

concerned. The abstract state provides sufficient information about the intention of container usage and enables us to examine the behavioral equivalence.

### 4.2 Behavior Constraint

Based on our insight, the behavioral equivalence must hold if the container properties are queried and updated in the same way as the original program. To formulate the criteria explicitly, we first introduce the notion of the container-property modifier and provide a novel representation of method semantic specification (§ 4.2.1). We then propose the container behavioral equivalence to specify the constraints that guarantee the behavioral equivalence (§ 4.2.2).

*4.2.1 Method Semantic Specification.* To maintain the container content for further queries in the program, each container method updates the memory $\mu$ and base $\beta$ in the concrete state and modifies the container properties. To depict the effect of a container method, we define the **container-property modifier** formally as follows.

*Definition 4.5.* (Container-Property Modifier) A container-property modifier is a function $t$ mapping a 4-tuple, which consists of a container variable, a tuple of parameter variables, a concrete state, and a container property, to a container property, i.e.,

$$t : \mathsf{Var} \times \mathsf{Var}^* \times \mathsf{State} \times \mathsf{Property} \rightarrow \mathsf{Property}$$

$$(c, args, s, p) \mapsto p'$$

where $p$ and $p'$ indicate the container properties before and after applying the modifier, respectively.

Similar to container-property queries, we can construct a family of container-property modifiers $\mathcal{T}$ to enumerate all the possible effects of container methods.

*Example 4.6.* Suppose that the language only supports the usage of LinkedHashMap. A container-property modifier can be one of the following forms: (1) Increasing or decreasing the size by at most one; (2) Inserting or removing an index or a value; (3) Inserting or removing a pair; (4) Inserting or removing an element from the partially ordered set, which preserves the insertion order.

Notice that a container-property modifier can affect several container properties. To show the effect explicitly, we establish a function $\omega_{\mathcal{T}} : \mathcal{T} \rightarrow 2^{Q}$ mapping a container-property modifier to the set of the container-property queries affected by it. For instance, when $t$ inserts or removes an element in the $i$-th partially ordered set of the base, it can affect the query of $i$-th partial order.

Given a container method, its semantics is essentially a combination of two orthogonal parts, namely specific container-property queries and container-property modifiers. Formally, we can define the **method semantic specification** as follows.

*Definition 4.7.* (Method Semantic Specification) The method semantic specification is a function $\alpha_{\mathcal{M}} : \mathcal{M} \rightarrow 2^{Q} \times 2^{\mathcal{T}}$. For a given $f_C \in \mathcal{M}$, $(Q, T) := \alpha_{\mathcal{M}}(f_C)$ indicates the container-property queries and the container-property modifiers induced by $f_C$, respectively.

*Example 4.8.* The method semantic specification maps the method LinkedHashMap.get to $(\{q\}, \varnothing)$, where $q(s, c) = \mathsf{isCor}(\lambda, v)$. Similarly, the method LinkedHashMap.put is mapped to $(\varnothing, \{t_1,\ t_2,\ t_3,\ t_4,\ t_5\})$, where $t_1$ increases the size by at most one, and $t_i$ $(2 \leq i \leq 5)$ insert an element or a pair to update $\mathsf{isIdx}(\lambda)$, $\mathsf{isVal}(v)$, $\mathsf{isCor}(\lambda, v)$, and $\mathsf{InsOrd}(\lambda_1, \lambda_2)$, respectively.

The method semantic specification describes the semantics of container methods in a compact way, blurring the details of how the memory is updated and container property is computed. Using the abstraction, we propose the container property analysis in § 5.1 to compute the abstract states, which maintain the concerned container properties of each container object. The properties provide sufficient guidance to guarantee the behavioral equivalence in the container replacements.

*4.2.2 Container Behavioral Equivalence.* To guarantee the behavioral equivalence in the container replacements, the container methods in the new program $P'$ should query and modify the concerned container properties in the same way as the ones in the program $P$. Besides, we need to constrain the types of container objects manipulating by the same container method call, which should be equal to assure that $P'$ is well-typed. To provide the criteria of the behavioral equivalence for our problem explicitly, we define the **container behavioral equivalence** formally as follows.

*Definition 4.9.* (Container Behavioral Equivalence) Given two programs $P$ and $P'$, where $P'$ is obtained by applying $\psi_a$ and $\psi_c$ to $P$ to perform container replacements. $P$ and $P'$ have the container behavioral equivalence relation, denoted by $P \simeq_C P'$, if and only if for any $st_c \in \mathcal{S}_c$ in the form of $v = c.f_C(v_1, \ldots, v_m)$, $\psi_a$ and $\psi_c$ satisfy

$$Q = Q' \tag{1}$$

$$\forall o \in \widetilde{\varepsilon}_{st_c}(c)\ \forall q \in \widetilde{\rho}_e(o),\ \eta(T, q) = \eta(T', q) \tag{2}$$

$$\forall o_1 \in \widetilde{\varepsilon}_{st_c}(c)\ \forall o_2 \in \widetilde{\varepsilon}_{st_c}(c),\ \mathrm{alloc}(o_1, st_{a1}) \wedge \mathrm{alloc}(o_2, st_{a2}) \rightarrow \psi_a(st_{a1}) = \psi_a(st_{a2}) \tag{3}$$

where $(Q, T) := \alpha_{\mathcal{M}}(f_C)$ and $(Q', T') := \alpha_{\mathcal{M}}(\psi_c(st_c))$. $(\widetilde{\rho}_e, \widetilde{\varepsilon}_e)$ and $(\widetilde{\rho}_{st_c}, \widetilde{\varepsilon}_{st_c})$ are the program states at the exit of $P$ and before $st_c$, respectively. The predicate $\mathrm{alloc}(o, st_a)$ indicates the relation that $o$ is allocated by $st_a$. $\eta$ is defined as follows:

$$\eta(T, q) = \{t \mid q \in \omega_{\mathcal{T}}(t), t \in T\} \tag{4}$$

The intuition behind Definition 4.9 is straightforward. The constraints in Equations 1 and 2 assure that the returned value of a container method call in $P'$ is always the same as the one in $P$, as the methods in $P'$ query and modify the concerned container properties in the same way as the ones in $P$. Meanwhile, Equation 3 constrains the types of container objects manipulated by the same container method call, assuring the program $P'$ is well-typed.

Obviously, we can explicitly examine the equations based on the abstract states to determine the methods and types in the replacements. Specifically, we utilize Equations 1 and 2 to identify possible methods for container method replacements (§ 5.2), and leverage Equation 3 to refine the replacements in the synthesis (§ 5.3). Formally, we state Theorem 4.10 to justify that container replacements assuring container behavioral equivalence finally guarantee behavioral equivalence.

Theorem 4.10. *Container behavioral equivalence relation is a behavioral equivalence relation, i.e.,*

$$P \simeq_C P' \Rightarrow P \simeq P'$$

Proof. According to Definition 4.9, $P'$ only differs from $P$ in terms of the container allocation statements and container method calls. Therefore, we only need to prove that for each container method call $v = c.f_C(v_1, \ldots, v_m)$ in $P$ and the corresponding call $v' = c.f'_C(v_1, \ldots, v_m)$ in $P'$, we have the equality relation $[\![v]\!](s) = [\![v']\!](s')$, where $f'_C := \psi_c(st_c)$. $s$ and $s'$ are the program states after the container method calls in $P$ and $P'$, respectively.

If not, we can find a control flow path $l$ in $P$ and $l'$ in $P'$ containing $st_c := v = c.f_C(v_1, \ldots, v_m)$ and $st'_c := v' = c.f'_C(v_1, \ldots, v_m)$, respectively, which are the first container method calls in $l$ and $l'$ violating the equality relation. According to Equation 1, we have $Q_1 = Q'_1$, where $(Q_1, T_1) := \alpha_{\mathcal{M}}(f_C)$ and $(Q'_1, T'_1) := \alpha_{\mathcal{M}}(\psi_c(st_c))$. Obviously, there exists a pair of container method calls located not after $st_c$ and $st'_c$ in $l$ and $l'$, respectively, which induce different modifiers upon a container property in $Q$. Denote the two method calls by $st_p$ and $st'_p$, which invoke $g_C$ and $g'_C$, respectively. Assume that $(Q_2, T_2) := \alpha_{\mathcal{M}}(g_C)$ and $(Q'_2, T'_2) := \alpha_{\mathcal{M}}(g'_C)$. Then, we have

$$\exists q^* \in Q,\ \eta(T_2, q^*) \neq \eta(T'_2, q^*)$$

The method calls manipulate the container object $o$, of which the properties in $Q$ are queried by $st_c$ and $st_c'$ afterwards. According to the definition of $\widetilde{\rho}_e$, we have $Q \subseteq \widetilde{\rho}_e(o)$, thus we have

$$\exists q^* \in \widetilde{\rho}_e(o),\ \eta(T_2, q^*) \neq \eta(T_2', q^*)$$

This contradicts with Equation 2. The theorem is proved. □

Theorem 4.10 enables us to guarantee the behavioral equivalence by examining the container behavioral equivalence. For each container method call, we can identify its method candidates which satisfy the constraints in Definition 4.9. Finally, we can select efficient container candidates so that the replacements are likely to satisfy the complexity superiority.

### 4.3 Complexity Guidance

To achieve the optimization, we expect the new program to satisfy the complexity superiority. Specifically, the synthesis algorithm should be aware of the time complexity of each container method. To this end, we propose the method complexity specification (§ 4.3.1) and then define container complexity superiority to provide the effective guidance for the synthesis (§ 4.3.2).

*4.3.1 Method Complexity Specification.* To depict time complexity of a container method in a fine-grained manner, we define a family of time complexity functions $\mathcal{TC}$ to represent different time complexities, which include (amortized) constant time complexity, (amortized) linear time complexity, etc. The functions of amortized time complexity are introduced as symbols to distinguish them from constant time complexity, linear time complexity, etc. Meanwhile, there exists an order between several container methods even if they have the same time complexity function. For example, the method LinkedHashMap.put has to maintain the indexes in a linked list to preserve the insertion order, and consumes more time than the method HashMap.put. Based on $\mathcal{TC}$, we can formalize the **method complexity specification** as follows.

*Definition 4.11.* (Method Complexity Specification) The method complexity specification is a function $CS$ mapping a container method $f_C$ to its complexity score $\theta \cdot tc(n)$, indicating its time complexity and a constant factor.

*Example 4.12.* The methods HashMap.put and LinkedHashMap.put are mapped to $\theta_1 \cdot tc(n)$ and $\theta_2 \cdot tc(n)$, respectively, where $\theta_1 < \theta_2$. $tc(n)$ is the function of amortized constant time complexity. $\theta_1 < \theta_2$ indicates that the method LinkedHashMap.put consumes more time than the method HashMap.put.

Based on Definition 4.11, we can measure the total time complexity score of container method calls by a function of $n$. Then we can naturally compare the order of the complexity scores of container method calls in two programs by comparing the coefficients of $tc_i(n)$, where $tc_i(n)$ is the time complexity function occurring in the complexity scores.

*4.3.2 Container Complexity Superiority.* Although the method complexity specification provides an abstraction of the efficiency of each container method, we are still unaware of the frequency of container method calls, and estimating the time complexity for a general program is far from practical. To establish effective guidance for synthesis, we define the **container complexity superiority** formally as the heuristic criteria of the complexity superiority.

*Definition 4.13.* (Container Complexity Superiority) Let $P'$ be the program obtained by applying $\psi_a$ and $\psi_c$ to $P$. $P'$ has the container complexity superiority over $P$ if and only if for any $st_a \in \mathcal{S}_a$ and $o$ allocated by $st_a$, we have

$$\sum_{st_c \in \mathcal{S}_c(o)} CS(\psi_c(st_c)) \leq \sum_{st_c \in \mathcal{S}_c(o)} CS(f_C) \tag{5}$$

where $f_C$ is the container method in $st_c$, and $\mathcal{S}_c(o)$ contains the container method calls that manipulate $o$ in an execution of $P$, i.e.,

$$\mathcal{S}_c(o) = \{st_c \mid st_c := v = c.f_C(v_1, \ldots, v_m) \in \mathcal{S}_c, o \in \widetilde{\varepsilon}_{st_c}(c)\} \tag{6}$$

$\widetilde{\varepsilon}_{st_c}$ indicates the points-to facts before the statement $st_c$.

*Example 4.14.* Assume that we have specified the method complexity specifications as follows:

$$CS(\text{ArrayList.add}) = tc(n) \qquad CS(\text{ArrayList.contains}) = n$$
$$CS(\text{HashSet.add}) = 2 \cdot tc(n) \qquad CS(\text{HashSet.contains}) = 1$$

$tc(n)$ is the time complexity function of amortized constant complexity. In Figure 2, the total time complexity score of the container method calls manipulating $o_2$ is $2n + tc(n)$, After replacing it with a HashSet object, the score is $2 + 2 \cdot tc(n) < 2n + 2 \cdot tc(n)$, showing the container complexity superiority of the program after the replacements.

Checking the complexity superiority requires precise reasoning of program complexity. However, deriving a tight bound of program complexity is stunningly difficult [Wilhelm et al. 2008] and far from practical for a real-world program [Gulwani et al. 2009a; Xie et al. 2016], especially for programs involving sophisticated manipulations of data structures [Fiedor et al. 2018; Gulwani et al. 2009c; Lu et al. 2021; Srikanth et al. 2017]. Although the container complexity superiority does not imply the complexity superiority, it provides the effective guidance to find the opportunity of synthesizing the replacements to improve program efficiency, as evidenced by our evaluation in § 7.

## 5 SYNTHESIS ALGORITHM

This section presents our synthesis algorithm that achieves the goals described in § 4.2 and § 4.3. It takes as inputs the source code of a program $P$ and the container method specifications. The algorithm finally computes the container replacement mappings $\psi_a$ and $\psi_c$, based on which we can obtain a new program $P'$. As shown in § 4.2 and § 4.3, the container behavioral equivalence and the container complexity superiority pose sophisticated constraints for the container replacement mappings $\psi_a$ and $\psi_c$. To satisfy all the constraints, our synthesis algorithm works with three stages as follows:

- To understand the intention of container usage, we present the container property analysis to determine which container-property queries occur upon a container object (§ 5.1).
- To assure $P'$ queries and modifies container properties in the same way as $P$, we identify the method candidates for a container method call based on Equations 1 and 2 (§ 5.2).
- A synthesizer selects the methods from the method candidates with lowest time complexity to guarantee the container complexity superiority. A verifier performs type checking by examining whether Equation 3 holds to assure the container behavioral equivalence. If the type checking fails, the synthesizer refines the synthesized types and methods in the consequent rounds (§ 5.3).

We also state the soundness and complexity of the synthesis theoretically (§ 5.4). The soundness theorem guarantees that the new program $P'$ must be behaviorally equivalent to $P$. For clarity, we use the program in Figure 2 to explain each stage of our approach throughout this section.

### 5.1 Container Property Analysis

According to Definition 4.1, we can compute container-property queries to reveal the intention of container usage. Suppose we have obtained points-to fact $\widetilde{\varepsilon}$ at each program location based on an off-the-shelf points-to analysis. Using the method semantic specification, we can easily compute the property-query fact $\widetilde{\rho}$ at each program location. Finally, we obtain the property-query fact $\widetilde{\rho}_e$ at the exit of the program, indicating all the container-property queries occurring in the program.

$$\frac{\widetilde{\rho} \vdash S_1 \rightsquigarrow \widetilde{\rho}_1 \quad \widetilde{\rho}_1 \vdash S_2 \rightsquigarrow \widetilde{\rho}'}{\widetilde{\rho} \vdash S_1; S_2 \rightsquigarrow \widetilde{\rho}'} \text{(Sequencing)}$$

$$\frac{\widetilde{\rho} \vdash S \rightsquigarrow \widetilde{\rho}' \quad \widetilde{\rho} = \widetilde{\rho}'}{\widetilde{\rho} \vdash fix(S) \rightsquigarrow \widetilde{\rho}} \text{(Fix-I)}$$

$$\frac{\begin{array}{c}\widetilde{\rho} \vdash S_1 \rightsquigarrow \widetilde{\rho}_1 \quad \widetilde{\rho} \vdash S_2 \rightsquigarrow \widetilde{\rho}_2 \\ \widetilde{\rho}' = \widetilde{\rho}_1[o \mapsto \widetilde{\rho}_1(o) \cup \widetilde{\rho}_2(o) \mid o \in O_c]\end{array}}{\widetilde{\rho} \vdash \textbf{if } (e) \textbf{ then } S_1 \textbf{ else } S_2 \rightsquigarrow \widetilde{\rho}'} \text{(Branch)}$$

$$\frac{\begin{array}{c}\widetilde{\rho} \vdash S \rightsquigarrow \widetilde{\rho}_1 \quad \widetilde{\rho} \neq \widetilde{\rho}_1 \\ \widetilde{\rho}_2 = \widetilde{\rho}_1[o \mapsto \widetilde{\rho}(o) \cup \widetilde{\rho}_1(o) \mid o \in O_c] \\ \widetilde{\rho}_2 \vdash fix(S) \rightsquigarrow \widetilde{\rho}'\end{array}}{\widetilde{\rho} \vdash fix(S) \rightsquigarrow \widetilde{\rho}'} \text{(Fix-II)}$$

$$\frac{\begin{array}{c}(Q, T) = \alpha_{\mathcal{M}}(f_C) \\ \widetilde{\rho}' = \widetilde{\rho}\,[o \mapsto \widetilde{\rho}(o) \cup Q \mid o \in \widetilde{\varepsilon}(c)]\end{array}}{\widetilde{\rho} \vdash v = c.f_C(v_1, \ldots, v_m) \rightsquigarrow \widetilde{\rho}'} \text{(ContainerCall)}$$

$$\frac{\widetilde{\rho} \vdash fix(S) \rightsquigarrow \widetilde{\rho}'}{\widetilde{\rho} \vdash \textbf{while } (e) \textbf{ do } S \textbf{ od} \rightsquigarrow \widetilde{\rho}'} \text{(Loop)}$$

Fig. 6. Abstract transformers in the container property analysis

Figure 6 defines the abstract transformers of program statements. Specifically, we should handle four program constructs, including a sequencing, a branch, a container method call, and a loop.

- The rule of sequencing is simple, in which the transformer is exactly the composition of the transformers of its parts.
- For a branch, the transformer merges the container-property queries occurring upon a container object along two paths.
- The rule ContainerCall relies on the points-to fact $\widetilde{\varepsilon}$ before the statement to identify the container object $o$ manipulated by the container method call. $Q$ indicates the container-property queries induced by $f_C$. To update $\widetilde{\rho}$, we merge $\widetilde{\rho}(o)$ with $Q$ directly to show that the container-property queries in $Q$ occur upon $o$.
- To compute the container-property queries in a loop, we need to calculate the fixed point by applying the transformer of the loop body iteratively. Due to the finite sizes of $Q$ and $O_c$, the fixed point must be reached after applying the rule Fix-II finite times.

*Example 5.1.* Consider the ArrayList object $o_{11}$ in Figure 2. According to the method semantic specification, we have $\widetilde{\varepsilon}(r) = \{o_{11}\}$ and $\widetilde{\rho}(o_{11}) = \varnothing$ before line 23. The method ArrayList.contains queries the container property isVal($v$), i.e., the value-ownership of $o_{11}$. By applying the rule ContainerCall, we have $\widetilde{\rho}'(o_{11}) = \{q\}$, where $q(s, c) = \text{isVal}(v)$, indicating that the value-ownership query has occurred upon $o_{11}$ after line 23. Similarly, $o_{11}$ is also manipulated by the container method call at line 38, and we can obtain $\widetilde{\rho}_e(o_{11}) = \{q\}$ at the exit of the program, which means that only the value-ownership query occurs upon $o_{11}$.

Our container property analysis reasons how container objects are utilized in a flow-sensitive manner. Crucially, $\widetilde{\rho}_e$ over-approximates the container-property queries occurring upon container objects, and provides the sufficient guidance for method candidate identification to guarantee the container behavioral equivalence. It is worth noting that pointer analysis affects the precision of the container property analysis. When the points-to facts are imprecise, the container property analysis can discover that a container object $o$ is manipulated by a container method call, while $o$ is not pointed by $c$ in any concrete execution. Therefore, $\widetilde{\rho}_e$ can contain the container-property queries which do not occur in any execution. We will quantify the effect of pointer analysis in the evaluation to show that its imprecision degrades the effectiveness of the replacements.

**Algorithm 1:** Identifying method candidates.

**Input:** $P$: A container-manipulating program; $\alpha_{\mathcal{M}}$: Method semantic specification;
**Output:** $\widehat{\psi}_c$: Method candidate mapping;

```
1 S_a, S_c ← getSkeleton(P);
2 ρ̃_e ← getQueryFact(P);
3 ψ̂_c ← [st_c ↦ ∅ | st_c ∈ S_c];
4 foreach st_c := v = c.f_C(v_1, ..., v_m) ∈ S_c do
5     ε̃_{st_c} ← getPTFact(P, st_c);
6     foreach f'_C ∈ M do
7         if isEquivalent (f_C, f'_C, ρ̃_e, ε̃_{st_c}, α_M) then
8             ψ̂_c(st_c) ← ψ̂_c(st_c) ∪ f'_C;
9 return ψ̂_c;
```

### 5.2 Method Candidate Identification

To guarantee the behavioral equivalence, we have to determine the container methods preserving the container behavioral equivalence. Specifically, the constraints in Equations 1 and 2 should be satisfied so that the concerned container properties can be queried and modified int the same way as the original program. Formally, we define the **method candidate** as follows.

*Definition 5.2.* (Method Candidate) Given a container method call $st_c \in \mathcal{S}_c$, a container method $f'_C \in \mathcal{M}$ is a method candidate of $st_c$ if and only if it satisfies Equations 1 and 2.

Essentially, we should compute the method candidate mapping $\widehat{\psi}_c : \mathcal{S}_c \to 2^{\mathcal{M}}$ to indicate the method candidates of a container method call. At a high level, we can leverage the method semantic specification $\alpha_{\mathcal{M}}$ and the property-query fact $\widetilde{\rho}_e$ at the exit to identify the method candidates.

Algorithm 1 shows the procedure of identifying method candidates. It first utilizes the points-to fact $\widetilde{\varepsilon}_{st_c}$ to identify the container objects manipulated by the container method call $st_c$. getQueryFact returns the property-query fact $\widetilde{\rho}_e$ at the exit of $P$, and isEquivalent checks whether Equations 1 and 2 hold for a container method call $st_c$. Utilizing the method semantic specification $\alpha_{\mathcal{M}}$, isEquivalent enumerates each container object $o$ manipulated by $st_c$ and checks whether the method $f'_C$ queries and modifies the concerned container properties of $o$ in the same way as the original method $f_C$ in $P$. Finally, Algorithm 1 collects the method candidates for each container method call.

*Example 5.3.* Assume that $C = \{\mathsf{ArrayList}, \mathsf{LinkedList}, \mathsf{HashSet}\}$. Consider the object $o_{11}$ in Figure 2. Utilizing the container property analysis, we obtain $\widetilde{\rho}_e(o_{11}) = q$, where $q(s, c) = \mathsf{isVal}(v)$. The container method calls $st_c@l_{23}$ and $st_c@l_{38}$ manipulate $o_{11}$ at lines 23 and 38, respectively. The methods HashSet.contains and ArrayList.contains both induce the value-ownership query and do not induce any container-property modifier, so Equations 1 and 2 both hold. Similarly, we have

$$\widehat{\psi}_c(st_c@l_{23}) = \widehat{\psi}_c(st_c@l_{38}) = \{\mathsf{ArrayList.contains}, \mathsf{LinkedList.contains}, \mathsf{HashSet.contains}\}$$

Recall that Theorem 4.10 states that container behavioral equivalence implies the behavioral equivalence. According to Algorithm 1, Equations 1 and 2 must hold if we select the method for a container method call $st_c$ from its method candidate set $\widehat{\psi}_c(st_c)$. Next, we can assure the container behavioral equivalence as long as the replacements satisfy Equation 3, i.e., the new program $P'$ is well-typed. Therefore, we can obtain a well-typed program $P'$ after the container replacements, which is behaviorally equivalent to $P$.

### 5.3 Container Replacement Synthesis

To improve program efficiency, we should select the methods from $\widehat{\psi}_c(st_c)$ for each container method call $st_c$ to satisfy the container complexity superiority in Definition 4.13. Besides, we have to conduct type checking by examining Equation 3 to assure the container behavioral equivalence.

To meet the two requirements, we instantiate a counterexample-guided inductive synthesis (CEGIS) paradigm [Cheung et al. 2013; Gulwani et al. 2011; Solar-Lezama et al. 2008; Yaghmazadeh et al. 2017]. Algorithm 2 shows the procedure of container replacement synthesis. At a high level, it processes a container allocation statement $st_a$ in a round and finally synthesizes the container replacement mappings $\psi_a$ and $\psi_c$. Specifically, each round contains the following two steps:

- *Guess replacements*: The synthesizer selects the most efficient methods from the method candidates for the container method calls manipulating $o$, where $o$ is allocated by $st_a$.
- *Type checking*: The verifier performs type checking by examining Equation 3. The container allocation statements are reprocessed in the consequent rounds if they violate Equation 3.

Initially, Algorithm 2 sets the types and methods to $\perp$ in $\psi_a$ and $\psi_c$ to indicate undefined types and methods, respectively. Besides, it introduces the mapping $\widehat{\psi}_a$ to maintain feasible types for container allocation statements, and all the types are regarded as feasible initially. Each round of Algorithm 2 synthesizes the replacements for the container object $o$ allocated by $st_a$. For clarity, we introduce the function callSites to obtain the container method calls manipulating $o$.

Next, to illustrate each step, we use the object $o_{11}$ in Figure 2 as an example. Suppose that $st_a@l_{17}$ has been processed before $st_a@l_{11}$ in the CEGIS loop, where $st_a@l_{17}$ and $st_a@l_{11}$ allocate $o_{17}$ and $o_{11}$, respectively. At the beginning of the round, we have $\psi_a(st_a@l_{17})$ = LinkedList and HashSet $\notin \widehat{\psi}_a(st_a@l_{17})$, as the value-ownership of $o_{17}$ is necessary in the program, and LinkedList supports more efficient insertions than ArrayList by avoiding memory reallocation.

**Guess Replacements.** The synthesizer enumerates each feasible container type $\tau' \in \widehat{\psi}_a(st_a)$ and utilizes getMinCS to find the method candidate supported by $\tau'$ with the lowest complexity (lines 11-14). If $\tau'$ does not support any method candidate, getMinCS returns a symbolic method $\top$ with *MAX_CS* as its time complexity score, and $\tau'$ is removed from $\widehat{\psi}_a(st_a)$, indicating that $\tau'$ is not the feasible type of $st_a$. Finally, the synthesizer selects the container type with the smallest sum of time complexity scores (lines 15-19).

*Example 5.4.* The synthesizer selects the methods HashSet.contains and HashSet.add to manipulate $o_{11}$. Because the sum of their complexity scores is smaller than that of any other selection, the synthesizer enforces $\psi_a(st_a@l_{11})$ = HashSet.

**Type Checking.** The verifier performs type checking by examining whether Equation 3 holds (lines 25-26). If type checking fails, it adds the allocation statements to the set of counterexamples *CE*, which are refined by being reprocessed in the consequent rounds (lines 27-31). Moreover, we constrain that the counterexamples have the same set of feasible container types (line 29), pruning off the type selections causing the failure of type checking in the consequent rounds.

*Example 5.5.* Before type checking, we have $\psi_a(st_a@l_{17})$ = LinkedList and $\psi_a(st_a@l_{11})$ = HashSet. The container objects $o_{11}$ and $o_{17}$ are both manipulated by the container method call at line 23, violating the constraint in Equation 3, so they are refined and reprocessed in the consequent rounds. At the end of this round, we have HashSet $\notin \widehat{\psi}_a(st_a@l_{11})$, as HashSet is not the feasible container type for $st_a@l_{17}$. Furthermore, their feasible type LinkedList is selected in the consequent rounds, finally passing type checking.

Particularly, the verifier updates a mapping $\sigma$ to show the relation between $st_a$ and $st'_a$ that the two allocated objects can be manipulated by the same container method call (lines 22-24).

**Algorithm 2:** Container replacement synthesis.

**Input:** $P$: A program; $\widehat{\psi_c}$: Method candidate mapping; $CS$: Method complexity specification;
**Output:** $\psi_a, \psi_c$: Container replacement mappings;

1 $\mathcal{S}_a, \mathcal{S}_c \leftarrow \texttt{getSkeleton}(P)$;
2 $\varphi_a, \varphi_c \leftarrow \texttt{getOriginalUsage}(P)$;
3 $\psi_a \leftarrow [st_a \mapsto \bot \mid st_a \in \mathcal{S}_a]$; $\psi_c \leftarrow [st_c \mapsto \bot \mid st_c \in \mathcal{S}_c]$;
4 $\sigma \leftarrow [st_a \mapsto \varnothing \mid st_a \in \mathcal{S}_a]$; $\widehat{\psi_a} \leftarrow [st_a \mapsto C \mid st_a \in \mathcal{S}_a]$;
5 **foreach** $st_a \in \mathcal{S}_a$ **do**
6  /* Synthesizer: Guess replacements */
7  $min \leftarrow MAX\_CS$;
8  $\mathcal{S}_a \leftarrow \mathcal{S}_a \setminus \{st_a\}$;
9  **foreach** $\tau' \in \widehat{\psi_a}(st_a)$ **do**
10  $\psi_c' \leftarrow \psi_c$;
11  **foreach** $st_c \in \texttt{callSites}(st_a)$ **do**
12  $\psi_c'(st_c) \leftarrow \texttt{getMinCS}(\widehat{\psi_c}(st_c) \cap \texttt{method}(\tau'), CS)$;
13  **if** $\psi_c'(st_c) = \top$ **then**
14  $\widehat{\psi_a}(st_a) \leftarrow \widehat{\psi_a}(st_a) \setminus \{\tau'\}$;
15  $cur \leftarrow \texttt{getCSSum}(\texttt{callSites}(st_a), \psi_c', CS)$;
16  **if** $cur < min$ **then**
17  $min \leftarrow cur$;
18  $\psi_c \leftarrow \psi_c'$;
19  $\psi_a(st_a) \leftarrow \tau'$;
20
21  /* Verifier: Type checking */
22  **foreach** $st_a' \in \mathcal{S}_a$ **do**
23  **if** $\texttt{callSites}(st_a) \cap \texttt{callSites}(st_a') \neq \varnothing$ **then**
24  $\sigma(st_a') \leftarrow \sigma(st_a') \cup \{st_a\}$;
25  $CE \leftarrow \{st_a' \mid st_a' \in \sigma(st_a),\ \psi_a(st_a') \neq \psi_a(st_a),\ \psi_a(st_a') \neq \bot\} \cup \{st_a\}$;
26  **if** $|CE| > 1$ **then**
27  **foreach** $st_a' \in CE$ **do**
28  $\mathcal{S}_a \leftarrow \mathcal{S}_a \cup \{st_a'\}$;
29  $\widehat{\psi_a}(st_a') \leftarrow \bigcap_{st_a'' \in CE} \widehat{\psi_a}(st_a'')$;
30  $\psi_a(st_a') \leftarrow \bot$;
31  $\psi_c \leftarrow [st_c \mapsto \bot \mid st_c \in \texttt{callSites}(st_a')]$;
32 **return** $\psi_a, \psi_c$;

Intuitively, $\sigma$ maintains the constraints for type checking, which are refined and utilized inductively in each round. To improve the efficiency of the synthesis, we use several data structures to cache the relationships frequently utilized in the synthesis. For example, we memorize the set of container method calls manipulating the container object allocated by a specific allocation statement so that we can get the value of callSites at lines 15, 23, and 31 without unnecessary recomputation.

Algorithm 2 synthesizes the container replacements inductively to guarantee the container behavioral equivalence and the container complexity superiority. Specifically, the counterexample-guided refinement assures that container behavioral equivalence must hold in the synthesis. Besides, the selected candidates have the lowest complexity among the method candidates, which assures the container complexity superiority. Even if type checking fails, the trivial selection, i.e., setting all the types and methods to the original ones, is still permissive in the consequent rounds, so the sum of the time complexity scores can not be increased. Obviously, the method complexity specifications determine the complexity guidance and further affect the synthesized replacements. We will configure different specifications to quantify the influence and demonstrate the advantages of the form of our method complexity specifications in Definition 4.11.

### 5.4 Summary

Based on the sound points-to facts, our approach synthesizes the container replacements efficiently, which do not change the program semantics. We formulate two theorems to state the soundness and the complexity of Algorithm 2.

Theorem 5.6. *(Soundness Theorem) $\psi_a$ and $\psi_c$ provide sound container replacements, i.e., the program $P'$ obtained by applying $\psi_a$ and $\psi_c$ for replacements has behavioral equivalence relation with the original program $P$.*

Proof. Based on Theorem 4.10, we only need to prove that for any $st_c \in \mathcal{S}_c$ and $f'_C \in \widehat{\psi}_c(st_c)$, $f'_C$ and $f_C$ satisfy the three equations in Definition 4.9, where $f_C$ is the container method invoked in $st_c$. In Algorithm 1, isEquivalent checks whether Equations 1 and 2 are satisfied. In Algorithm 2, the verifier performs type checking and examines whether Equation 3 holds. Given sound points-to facts, Equation 3 must hold for the synthesized container replacement mappings. Thus, the soundness of the synthesis totally relies on the soundness of the container property analysis.

The off-the-shelf points-to analysis provides a sound result $\widetilde{\varepsilon}$ for the abstract transformers in Figure 6. Consider an arbitrary container method call $v = c.f_C(v_1, \ldots, v_m)$. For any concrete execution of the program, the container object manipulated by the method call can be abstracted by an abstract container object $o \in \widetilde{\varepsilon}(c)$. We use a set $Q$ to denote the set of the container-property queries induced by the call, i.e., $(Q, T) := \alpha_{\mathcal{M}}(f_C)$.

The rule ContainerCall adds all the container-property queries in $Q$ to $\widetilde{\rho}(o)$, which is a subset of $\widetilde{\rho}_e(o)$. Thus, the container-property queries occurring on the concrete container object must be included by $\widetilde{\rho}_e(o)$, which means the rule ContainerCall defines a sound abstract transformer for container method calls. Similarly, we can prove the other three rules, i.e., the rules Sequencing, Branch and Loop, define sound abstract transformers. Finally, the soundness of container property analysis assures the soundness of container replacements. □

Theorem 5.7. *(Complexity of Synthesis) Assume $|\mathcal{S}_a| < |\mathcal{S}_c|$. The time complexity of Algorithm 2 is $O(|C|^2 \cdot |\mathcal{M}| \cdot |\mathcal{S}_a| \cdot |\mathcal{S}_c|)$.*

Proof. First, consider the guessing process, which corresponds to the steps from line 9 to line 19. The upper bound of the iteration count from line 9 to line 19 is

$$sup_{st_a \in \mathcal{S}_a} |\widehat{\psi}_a(st_a)| = O(|C|)$$

Similarly, the upper bound of the iteration count from line 11 to line 14 is $O(|\mathcal{S}_c|)$, as $\mathsf{callSites}(st_a) \subseteq \mathcal{S}_c$. Notice that the function getMinCS has to find the minimal value from at most $|\mathcal{M}|$ unordered elements, so it runs in $O(|\mathcal{M}|)$. The function getCSSum at line 15 also runs in $O(|\mathcal{S}_c|)$. In each round, the synthesizer guesses the replacements in

$$O(|C| \cdot (|\mathcal{S}_c| \cdot |\mathcal{M}| + |\mathcal{S}_c|)) = O(|C| \cdot |\mathcal{S}_c| \cdot |\mathcal{M}|)$$

Second, consider the counterexample generation in the type checking, which corresponds to the steps from line 22 to line 25. The upper bound of the iteration count is $|\mathcal{S}_a|$. Meanwhile, the disjointness checking at line 23 can be preprocessed in $O(|\mathcal{S}_a| \cdot |\mathcal{S}_c|)$ before the synthesis. By looking up the memorization, the step at line 23 can be achieved in constant time. Also, the construction of $CE$ at line 25 runs in $O(|\mathcal{S}_a|)$. Therefore, the counterexamples are generated in $O(|\mathcal{S}_a|)$.

Third, consider the second loop in the type checking, which correspond to the steps from line 27 to line 31. The upper bound of the iteration count is $|CE| = O(|\mathcal{S}_a|)$. The computation at line 29 can be hoisted out of the loop, which runs in $O(|\mathcal{S}_a|)$. The result can be cached and reused in each iteration in $O(1)$. Therefore, the loop runs in $O(|\mathcal{S}_a|)$.

According to the above results, we can conclude that each round of the synthesis runs in

$$O(|C| \cdot |\mathcal{M}| \cdot |\mathcal{S}_c| + |\mathcal{S}_a| + |\mathcal{S}_a|) = O(|C| \cdot |\mathcal{M}| \cdot |\mathcal{S}_c| + 2|\mathcal{S}_a|)$$

Finally, consider the upper bound of the number of the rounds in the synthesis. According to the step at line 29, $|\widehat{\psi}_a(st'_a)|$ must decrease by at least one, where $st'_a$ will be resolved in the consequent round. On the one hand, $|\widehat{\psi}_a(st'_a)|$ is bounded by $|C|$, as $\widehat{\psi}_a(st'_a) \subseteq C$. On the other hand, $|\widehat{\psi}_a(st'_a)|$ must be larger than 0 at the end of the synthesis. Therefore, the number of the rounds is bounded by $|\mathcal{S}_a| \cdot |C|$. Assume $|\mathcal{S}_a| < |\mathcal{S}_c|$. The time complexity of Algorithm 2 is

$$\begin{aligned}
&O(|C| \cdot |\mathcal{M}| \cdot |\mathcal{S}_c| + 2|\mathcal{S}_a|) \cdot O(|\mathcal{S}_a| \cdot |C|) + O(|\mathcal{S}_a| \cdot |\mathcal{S}_c|) \\
=&O(|C|^2 \cdot |\mathcal{M}| \cdot |\mathcal{S}_a| \cdot |\mathcal{S}_c| + 2|C| \cdot |\mathcal{S}_a|^2 + |\mathcal{S}_a| \cdot |\mathcal{S}_c|) \\
=&O(|C|^2 \cdot |\mathcal{M}| \cdot |\mathcal{S}_a| \cdot |\mathcal{S}_c|)
\end{aligned}$$

Particularly, the assumption is introduced to simplify the estimated complexity of Algorithm 2. In general, the container object allocated by a container allocation statement is often manipulated by more than one container method call, and a container method call often only manipulates a single container object. Thus, the assumption holds in almost all the programs. □

It is worth mentioning that Theorem 5.7 does not provide a tight upper bound of the complexity. Actually, the overhead depends on the multiple aspects of the container usage. For example, the way of manipulating container objects affects the result of the container property analysis, and further determines the size of callSites($st_a$). Besides, it is more likely to trigger the refinements if a large number of container objects are manipulated by the same container method calls. In practice, the synthesis performs with almost linear scalability, which is evidenced by our experiments.

## 6 IMPLEMENTATION

We have implemented our approach as a tool named Cres. The inputs of Cres are the source code of a Java application and the method specifications, including the method semantic specification and the method complexity specification. Cres itself is implemented based on Pinpoint [Shi et al. 2018, 2021], the static analysis platform in the Ant Group. When analyzing a Java program, the frontend of Pinpoint transforms the class files to LLVM IR [Lattner and Adve 2004], and then Cres identifies the container allocation statements and container method calls to obtain the skeleton. In what follows, we discuss more key configurations and designs for the synthesis algorithm.

**Method Specifications.** In the implementation, we concentrate on the containers in JCF. We specify the method semantic specification in the configuration file by assigning a pair of container-property queries and modifiers to each container method. Particularly, we adopt the insertion order and the key order as the partial orders to describe the container properties of LinkedHashMap and TreeMap, respectively. Besides, we provide the method complexity specification in a fine-grained manner. Specifically, we specify the constant factor $\theta$ along with time complexity to show

the difference between the methods with the same time complexity. For example, the factor $\theta$ of LinkedList.add is smaller than that of LinkedHashSet.add, indicating that the latter consumes more time than the former due to the extra maintenance of a linked list, although they both run in amortized constant time.

**On-demand Points-to Analysis.** To support our container property analysis, we utilize Pinpoint to perform flow and context-sensitive pointer analysis. We are only concerned about the points-to facts of container variables, as the points-to facts of other variables do not affect the result of the container property analysis. Therefore, we query the points-to facts on demand to avoid unnecessary overhead in the container property analysis. Besides, the points-to facts at each program location are the prerequisite of examining the constraints to achieve the container behavioral equivalence and the container complexity superiority. To avoid redundant points-to query, we memorize the points-to facts and obtain the grace performance of the synthesis. Moreover, the points-to facts can also be obtained from other off-the-shelf pointer analyses [Li et al. 2011; Späth et al. 2016; Zhang et al. 2013], which means that Cres can be implemented easily based on other static analysis platforms [Arzt et al. 2014; Shi et al. 2018; Sui and Xue 2016].

## 7 EVALUATION

We evaluate the effectiveness and efficiency of Cres by investigating the research questions:

- **RQ1:** What is the improvement Cres achieves for real-world programs?
- **RQ2:** Which kinds of container replacements does Cres synthesize?
- **RQ3:** What is the time and space overhead of Cres?

**Result Highlights**. In summary, Cres is unusually effective and efficient.

- *Significant efficiency improvement of experimental subjects*: The execution time is reduced by 8.1% on average. Particularly, Cres reduces the time consumption of the project google-http-java-client by 27.1%.
- *Various replacement patterns and many confirmations*: Cres discovers 107 replacements in six patterns, and 71 replacements have been confirmed by the developers. Several patterns, such as replacing ArrayList with HashSet, are uncovered by previous works.
- *Ability to scale to large-scale programs*: Cres finishes analyzing the project iotdb with 384.2 KLoC in 14 minutes within 10 GB peek memory. The memory and time overhead is almost linear with the size of the project.

We also design a group of ablation studies to quantify the influence of the method complexity specifications and the precision of the pointer analysis. At the end of the evaluation, we discuss the quality of the replacements, the limitations of Cres, and several future directions.

### 7.1 Experimental Setup

**Subjects**. We evaluate Cres on 12 real-world Java projects, which are shown in Table 2. The projects are actively maintained and widely used in both academia and industry, covering different sizes (ranging from 18.6 KLoC to 384.2 KLoC) and diverse categories (such as microservice platforms, RPC frameworks, data management systems, etc.) Besides, the projects contain intensive usage of containers with various types, which provides more opportunities for Cres to find different patterns of container replacements.

**Experimental Setting**. For each project, we perform a whole-program analysis to obtain the points-to facts of container variables. Because we can not obtain the inputs for the projects in the real-world scenario, we follow the existing works and utilize the test suites of the projects to measure their time consumption [Basios et al. 2018]. To make the measurement more convincing,

Table 2. The medium ratio of reduced and original execution time and 95% confidence interval of the ratio

| Project | Description | Size (KLoC) | Medium (%) | 95% CI (%) |
|---|---|---|---|---|
| bootique | Microservice platform | 18.6 | 4.5 | [4.4, 4.6] |
| mapper | Server application | 22.4 | 7.3 | [7.0, 7.6] |
| incubator-eventmesh | Eventing infrastructure | 24.9 | 4.1 | [3.9, 4.3] |
| google-http-java-client | Web client | 25.2 | 27.1 | [25.9, 28.3] |
| light-4j | Microservice platform | 44.3 | 5.2 | [5.0, 5.4] |
| roller | Server application | 54.4 | 9.5 | [9.2, 9.8] |
| IginX | Data management system | 68.1 | 3.5 | [3.4, 3.6] |
| sofa-rpc | RPC framework | 76.4 | 3.7 | [3.4, 4.0] |
| Glowstone | Server application | 85.6 | 13.1 | [12.9, 13.3] |
| dolphinscheduler | Eventing infrastructure | 89.5 | 5.3 | [5.1, 5.5] |
| dubbo | RPC framework | 196.5 | 7.5 | [7.2, 7.8] |
| iotdb | Data management system | 384.2 | 6.3 | [6.2, 6.4] |
| | | | 8.1 | [7.8, 8.4] |

we repeat the execution of the test suite of each project 100 times and perform Mann-Whitney U test to examine whether the improvement is statistically significant [Arcuri and Briand 2011; Fay and Proschan 2010]. We conduct all the experiments on a 64-bit machine with 40 Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz and 512GB of physical memory.

### 7.2 Answers to Research Questions

Cres aims to reduce the execution time of all the evaluated subjects. We quantify the effectiveness and efficiency of Cres by answering the three research questions.

*7.2.1 Study of RQ1.* We utilize the test suite of each project to measure the execution time of the test tasks affected by the replacements. Specifically, we compute the medium value and 95% confidence interval of the ratio between the reduced time consumption and the original one.

Table 2 shows the ratio of reduced time consumption and the original one for each project. The lower bound of 95% confidence interval is positive in each project, which means that Cres can improve the efficiency of all the projects statistically significantly. On average, the medium of reduced time cost ratio reaches 8.1%, and the 95% confidence interval is [7.8, 8.4] . This demonstrates the effectiveness of Cres in improving the efficiency of real-world projects.

Particularly, the medium of reduced time ratio reaches 27.1% for the project google-http-java-client, and its 95% confidence interval is [25.9, 28.3]. The project is the HTTP client library for Java, supporting the access of the resource on the web via HTTP. Any project which depends on the library can benefit from the improvement of efficiency, which shows the significant impact of Cres. Another example is that the medium of the ratio reaches 13.1% for the project Glowstone. It is a customizable server for the game Minecraft, and its efficiency improvement promotes the performance of the service, shortening the response time of the interactions in the game. Generally, the improvement can benefit the applications depending on these projects, showing the great impact of Cres on the performance optimization of real-world programs.

**Answer to RQ1:** Cres improves the efficiency of all the subjects significantly, and the medium of reduced time ratio reaches 8.1% on average.

Table 3. The counts of different replacements

| Project | #Conf/#Total | #R1 | #R2 | #R3 | #R4 | #R5 | #R6 |
|---|---|---|---|---|---|---|---|
| bootique | 0/4 | | | 4 | | | |
| mapper | 0/6 | | 5 | | 1 | | |
| incubator-eventmesh | 19/19 | 1 | 16 | 2 | | | |
| google-http-java-client | 0/4 | | 4 | | | | |
| light-4j | 0/5 | | 2 | 3 | | | |
| roller | 0/6 | | | 5 | 1 | | |
| IginX | 11/11 | | 9 | | 1 | | 1 |
| sofa-rpc | 12/12 | | 5 | | | 2 | 5 |
| Glowstone | 0/11 | | 6 | 3 | 1 | | 1 |
| dolphinscheduler | 7/7 | | 6 | 1 | | | |
| dubbo | 12/12 | 1 | 3 | 1 | | 2 | 5 |
| iotdb | 10/10 | 2 | 1 | 6 | | | 1 |
| | 71/107 | 4 | 57 | 25 | 4 | 4 | 13 |

R1: LinkedList⇒ArrayList
R2: ArrayList⇒LinkedList
R3: ArrayList⇒HashSet
R4: TreeMap⇒HashMap
R5: LinkedHashMap⇒HashMap
R6: LinkedHashSet⇒HashSet

```
1 public boolean isExcluded(String s) {
2   List exclusions = new ArrayList<String>();
3   if (EXCLUSIONS != null)
4     exclusions = EXCLUSIONS;
5   return MANAGEMENT.equals(s)
6     || SCALABLE_CONFIG.equals(s)
7     || exclusionList.contains(s);
8 }
```

(a) An inefficient usage of ArrayList in light-4j

```
1 public T getPath(T p, T q, T r) {
2   List v = new ArrayList<>();
3   for (T c = p; c != q; c = c.pre())
4     v.add(c.pre().post().indexOf(c));
5   for (Integer i : v)
6     r= r.post().get(i);
7   return r;
8 }
```

(b) An inefficient usage of ArrayList in mapper

Fig. 7. Examples of inefficient usage of containers

*7.2.2 Study of RQ2.* Table 3 displays the patterns of container type replacements synthesized by Cres. There are 107 replacements covering six different categories, and 71 replacements in six projects have been confirmed by the developers. If Cres replaces the container type of a container object, the methods in the container method calls manipulating the object are also replaced, e.g., the invoked container method at line 5 in Figure 2 is the method HashSet.contains in the replacements. However, the names of the methods are the same before and after the replacements in almost all the cases, so we do not discuss how the methods are replaced in detail.

**Case Studies**. Particularly, we show two examples of typical replacement patterns as follows.

*Transform ArrayList to HashSet.* Figure 7a shows the container replacement in the project light-4j. Based on the result of the container property analysis, we find that the ArrayList object pointed by EXCLUSIONS and exclusions is only created for querying the value-ownership, as the method calls manipulating the object are the insertions or checking whether an object is stored in the list. Cres synthesizes the replacement in which the types of EXCLUSIONS and exclusions are changed to HashSet safely, and the time complexity of querying the value-ownership can be reduced from linear complexity to amortized constant complexity.

*Transform ArrayList to LinkedList.* Figure 7b shows the transformation from ArrayList to LinkedList in the project mapper. The ArrayList object allocated at line 2 is manipulated by the method ArrayList.add and its iterator in the iteration. According to the documentation, we find that

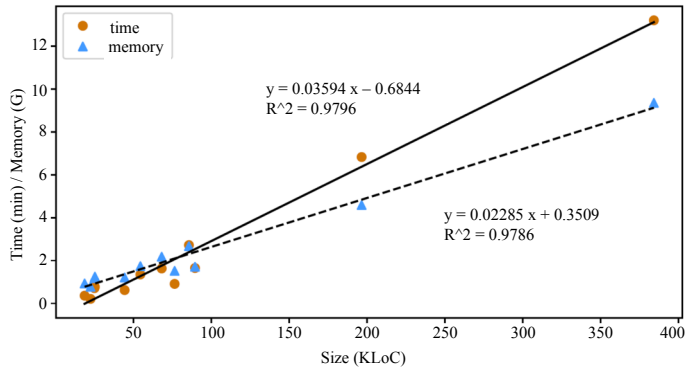


Fig. 8. Time and memory overheads of Cres

the method ArrayList.add has amortized constant time complexity due to the memory reallocation, while the method LinkedList.add has constant time complexity, and no memory reallocation occurs in the insertions. Meanwhile, there is no difference in time complexity between the iterations over these two types of containers. Therefore, Cres synthesizes the replacement in which ArrayList is replaced with LinkedList to reduce the time cost of insertions.

Compared with the existing approaches [Basios et al. 2018; Oliveira et al. 2019; Shacham et al. 2009], Cres can discover more general optimization patterns. As shown by the example in Figure 2, it can replace an ArrayList object with a HashSet object if only the method ArrayList.contains is invoked after its insertions. Meanwhile, we remark that Cres only aims to discover the replacements that can make the difference in time complexity and does not consider the environmental factors, such as the microarchitecture, which can also affect the execution time. Therefore, Cres might miss the opportunity of performing environment-dependent optimizations. However, Table 2 has shown that Cres is effective enough to improve the efficiency.

**Answer to RQ2:** Cres synthesizes 107 container replacements in various patterns without changing the program behavior, 71 of which are confirmed by the developers.

*7.2.3 Study of RQ3.* We investigate the efficiency of Cres by measuring its time and memory overhead in the synthesis. The overhead of each project is shown Figure 8. Overall, Cres finishes the analysis of the program with 384.2 KLoC in 14 minutes with 9.36 GB peak memory consumption. We adapt the regression analysis to study the observed complexity of Cres. The $R$-squared value for time and memory cost are 0.9796 and 0.9786, respectively, which are pretty close to 1. It indicates that the overhead of Cres grows nearly linearly at a gentle rate, permitting Cres to efficiently analyze large-scale programs manipulating containers. Compared with the existing works, Cres features with its efficiency and only needs 2.5 minutes to analyze per project. However, ARTEMIS executes the benchmark iteratively to obtain the optimal solution by genetic algorithm [Basios et al. 2018], and it spends 3.1 hours optimizing a project on average.

**Answer to RQ3:** Cres features linear scalability and finishes the analysis in 14 minutes with 9.36 GB peak memory for the program with 384.2 KLoC.

Table 4. The counts of different replacements synthesized by the ablations

| Project | #R1 | #R2 | #R3 | #R4 | #R5 | #R6 |
|---|---|---|---|---|---|---|
| bootique | | | (4, 4, 3) | | | |
| mapper | | (5, 5, 3) | | (1, 1, 0) | | |
| incubator-eventmesh | (1, 1, 0) | (16, 16, 11) | (2, 2, 0) | | | |
| google-http-java-client | | (4, 4, 1) | | | | |
| light-4j | | (2, 2, 2) | (3, 3, 1) | | | |
| roller | | | (5, 5, 4) | (1, 1, 0) | | |
| IginX | | (9, 9, 7) | | (1, 1, 1) | | (1, 0, 1) |
| sofa-rpc | | (5, 5, 4) | | | (2, 0, 2) | (5, 0, 3) |
| Glowstone | | (6, 6, 6) | (3, 3, 2) | (1, 1, 1) | | (1, 0, 1) |
| dolphinscheduler | | (6, 6, 5) | (1, 1, 1) | | | |
| dubbo | (1, 1, 1) | (3, 3, 2) | (1, 1, 1) | | (2, 0, 1) | (5, 0, 4) |
| iotdb | (2, 2, 1) | (1, 1, 1) | (6, 6, 3) | | | (1, 0, 1) |

### 7.3 Ablation Study

Cres leverages an off-the-shelf pointer analysis to identify the manipulated container objects, so the precision of the pointer analysis can affect the result of the container property analysis and the synthesized container replacements further. Besides, the method complexity specifications are specified manually, and the subjectivity of the specifications might also affect the replacements synthesized by Cres. Therefore, we set up the following ablations to investigate the influence of the pointer analysis and the method complexity specifications.

*7.3.1 Ablation Study Setting.* We propose two groups of the ablation studies to quantify the impact of pointer analysis and the method complexity, respectively.

- We use Cres-NS and Cres-RS to synthesize the replacements with different method complexity specifications. In Cres-NS, the constant factors are all equal to 1. In Cres-RS, the constant factors are randomly generated and conform to the specific order. For example, the constant factor of the method LinkedHashMap.put is larger than the one of the method HashMap.put.
- Cres-P leverages a flow and context-insensitive pointer analysis [Zhang et al. 2013] to perform the container property analysis. Other modules and configurations are not changed.

We evaluate the three ablations upon the projects in Table 2. It is worth mentioning that the complexity function of each container method can be derived from the documentations, not inducing any bias in the manual configuration. Thus, we only quantify the influence of the constant factors in the method complexity specification.

*7.3.2 Ablation Study Result.* Table 4 shows the numbers of the container replacements synthesized by the three ablations [1], based on which we can obtain the following findings:

- Cres-NS can discover all the replacements synthesized by Cres except for the ones belonging to R5 and R6. Without the constant factors, Cres-NS can not distinguish the container methods with the same complexity function, such as the methods LinkedHashMap.put and HashMap.put.
- The replacements synthesized by Cres-RS are the same as the ones synthesized by Cres. The methods of the selected types have lower time complexity than the original ones in the first four patterns. Besides, all the methods of HashMap and HashSet have smaller constant factors than the ones of LinkedHashMap and LinkedHashSet, respectively.

[1] A triple shows the numbers of the replacements synthesized by Cres-NS, Cres-RS, and Cres-P.

```
public Point retrieveValidLastPoint(int n) {
  List<IChunkMetadata> seqDataList = new LinkedList<>();
  for (int i = 0; i < n; i++)
    seqDataList.add(getDataFromDevice());
  for (int i = seqDataList.size() - 1; i >= 0; i--) {
    Point lastPoint = getChunkLastPoint(seqDataList.get(i));
    if (lastPoint.getValue() != null)
      return lastPoint;
  }
  return null;
}
```

Fig. 9. An example in which Cres fails to synthesize the optimal replacements

- Cres-P synthesizes 74 container replacements out of 107 replacements synthesized by Cres. The imprecise points-to information yields the spurious container-property queries of several container objects, which prevents the synthesis algorithm from seizing the opportunity of optimizing the usage of the objects.

As we can see, the effectiveness of Cres does not largely depend on the manual configuration but relies on a precise pointer analysis. It is shown that Cres is easy to be configured by the users, and the documentations of the container methods provide the sufficient knowledge to specify the specifications, which can support the effective replacement synthesis for Cres.

We also apply the replacements synthesized by the ablations for each project and measure the improvement of program efficiency. It is shown that the 95% confidence intervals of the reduce time ratio are [7.0%, 7.6%] and [5.3%, 5.7%] on average for Cres-NS and Cres-P, respectively. We also measure the overhead of the synthesis for the three ablations. Specifically, Cres-NS and Cres-RS take the similar overhead to Cres, as the method complexity specifications only affect the selected method candidates and the types in Algorithm 2. Cres-P is more efficient with the benefit of the light-weighted pointer analysis, e.g., it finishes analyzing the project iotdb in 10.3 minutes with 7.1 GB peek memory. However, Cres is practical enough for the real-world programs, as it does not suffer from the heavy overhead and synthesizes more replacements compared with Cres-P.

### 7.4 Discussion

**Quality of Container Replacement**. As shown by the answer to RQ2, Cres finds 107 replacements covering six categories. Notably, the developers greatly appreciated our efforts. For example, a developer of the project iotdb commented that "*Since we often pay less attention to these details, if a tool can be used to do this work, it will be great!*" A developer of the project sofa-rpc was even inquired about Cres with comments like "*Where can I get the tool?*" Particularly, Cres has been integrated into the CI process in the Ant Group, which is a FinTech company with over 1 billion global users. The synthesized replacements can be forwarded to the developers as suggestions during the development cycle, making the applications deployed and executed economically.

**Limitations**. Although Cres is shown to be effective and efficient, it also comes with several limitations. First, the container complexity superiority formulates the complexity superiority in a heuristic manner. Generally, it is hard to derive a tight bound of the time complexity for a given program [Gulwani et al. 2009a,c; Wilhelm et al. 2008]. Meanwhile, our method complexity specification can not capture dynamic features dependent on the architectures [Jung et al. 2011] and might be imprecise for small inputs. However, it is almost infeasible to quantify the time cost of a container method for any input and execution environment. Fortunately, the experiential results have shown the effectiveness of the guidance provided by method complexity specification and container complexity superiority. Second, Cres can not always find optimal replacements. Consider

the program extracted from the iotdb in Figure 9. Cres can not discover that the random access in the second loop is merely used for traversing the container, so it replaces the LinkedList object with an ArrayList object, while the optimal solution should be replacing the random access in the second loop by an iterator.

**Future Work**. The insight underlying Cres is applicable to reduce other resource consumption and support other kinds of replacements. We only need to provide the method specifications in which the resource consumption of each container method is specified. For example, we can extend the method complexity specification to model the energy cost [Hasan et al. 2016], and then Cres can optimize energy consumption seamlessly. Also, dynamic profiling can be integrated to capture the runtime data [Mudduluru and Ramanathan 2016] so that Cres can benefit from the static complexity model and the runtime overhead model simultaneously. Meanwhile, we can empower Cres with more container properties, such as boundedness. If only a finite number of insertions occur upon a container object, we can safely replace it with an array to avoid the memory bloat [Xu and Rountev 2010; Xu et al. 2010]. We believe that Cres provides a general framework to support the container replacements, reducing different kinds of resource consumption of a program.

## 8 RELATED WORK

There is a large and diverse body of literature touching the topic of this work, which covers the program optimization, synthesis, and verification. We discuss each line of the works as follows.

### 8.1 Container Selection

There is an extensive body of works on selecting efficient container types. Several approaches, e.g., ARTEMIS [Basios et al. 2018] and SEEDS [Manotas et al. 2014], search container types to minimize resource consumption when executing the program with the given test suite. They execute the program thousands of times over the test suite to search the optimal selection, introducing the heavy time burden. Brainy [Jung et al. 2011] and Chameleon [Shacham et al. 2009] utilize dynamic profiling to obtain the heap information and predict the best container types with a prediction model, which is specified by expertise or obtained in the training process. However, the model can be restrictive when the expert knowledge is unavailable or the training data is not general enough. Different from the dynamic profiling based approaches, CT+ [Oliveira et al. 2019, 2021] attempts to reduce resource consumption by analyzing container usage statically, and replace container types based on the class hierarchy diagram. Unfortunately, CT+ can not synthesize more general replacements, such as replacing ArrayList with HashSet, because of the restrictive assumptions on interchangeable container types. In contrast, Cres can discover various replacement patterns with the benefit of our container property analysis.

### 8.2 Performance Analysis

Several recent works leverage program analysis to detect performance issues. Dynamic approaches mutate programs to reach peak resource consumption [Lemieux et al. 2018; Petsios et al. 2017; Wen et al. 2020]. However, they can not evidence the existence of performance bugs or suggest more efficient implementations. Static analysis techniques mainly focus on specific bug types, such as API misuse [Jin et al. 2012], inefficient loops [Song and Lu 2017; Xu et al. 2012], inefficient ORM usage [Yang et al. 2018], etc. Particularly, Clarity [Olivo et al. 2015] detects asymptotic performance bugs in collection traversals, showing great potential in practical use. Our work focuses on a subproblem of performance analysis caused by inefficient container types. Different from many existing works, Cres can fix performance issues by synthesizing container replacements, promoting its practicality of analyzing real-world programs.

### 8.3 Data Structure Synthesis

Data structure synthesis aims to compose data structure designs out of existing data structures. Motivated by various scenarios, data structures are supposed to satisfy different constraints. For example, Volt [Pailoor et al. 2021] is the latest data structure synthesizer, which aims to refine the data structure satisfying integrity constraints when introducing auxiliary fields. Mask [Samak et al. 2020] replaces outdated data structures by synthesizing their methods with the latest ones. The two synthesizers assure functional correctness while do not concern the efficiency. Other works, such as Cozy [Loncaric et al. 2016], RelC [Hawkins et al. 2011, 2012], and Data Calculator [Idreos et al. 2018], combine static cost models with operational semantics and synthesize data structures with an excellent performance to alleviate the human burden in designing data structures. Cres bears similarities to these works in terms of the complexity guidance. However, Cres concentrates more on performance optimization introduced by container replacements, and does not attempt to synthesize new data structures.

### 8.4 Data Structure Specification Inference

Inferring the data structure specifications has been a fundamental problem in the community of programming languages. A variety of the properties, including the points-to information, aliasing, and size, are concerned in a large body of the literature [Chase et al. 1990; Gulwani et al. 2009b; Sagiv et al. 2002]. One typical line of the works is shape analysis [Reps et al. 2007], which aims to infer the linkage properties and numeric properties [Kim and Rinard 2011; Sagiv et al. 2002; Zee et al. 2008]. Although the specifications inferred by shape analysis are sound, the analysis is quite brittle and suffers from the capability problem when analyzing real-world programs [Chang et al. 2020]. Another line of the works synthesizes the data structure specifications via learning techniques [Bastani et al. 2018; Eberhardt et al. 2019]. Particularly, USpec [Eberhardt et al. 2019] does not require the access to the source code of the data structures and only analyzes the usage patterns to obtain the aliasing specifications. However, the extracted specifications are often restrictive, ignoring the numeric properties. Cres relies on the manual annotations to decompose the semantics of container methods into the container-property queries and the container-property modifiers. It would be promising to generate the specifications automatically to support our semantic model.

## 9 CONCLUSION

We have introduced Cres, a novel synthesizer that automatically replaces inefficient container usage. It analyzes the concerned container properties and finds more efficient container methods that preserve the behavioral equivalence to improve program efficiency. Cres is highly effective and efficient in analyzing real-world programs. It synthesizes 107 instances of container replacements covering six categories, which reduce the execution time by 8.1% on average. Cres also stands out with its excellent scalability, and finishes analyzing the project with 384.2 KLoC in 14 minutes. We hope the insight underlying Cres can be extended to reduce other resource consumption, such as memory, energy, and CPU usage.

## ACKNOWLEDGMENT

We thank the anonymous reviewers for valuable feedback on earlier drafts of this paper, which helped improve its presentation. We also appreciate Dr. Xiao Xiao and Dr. Gang Fan for insightful discussions. The authors are supported by the RGC16206517, ITS/440/18FP and PRP/004/21FX grants from the Hong Kong Research Grant Council and the Innovation and Technology Commission, Ant Group, and the donations from Microsoft and Huawei. Peisen Yao is the corresponding author.

## REFERENCES


Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 1–8. https://ieeexplore.ieee.org/document/6679385/

Andrea Arcuri and Lionel C. Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 1–10. https://doi.org/10.1145/1985793.1985795

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269. https://doi.org/10.1145/2594291.2594299

Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T Barr. 2018. Darwinian data structure selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 118–128.

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018. Active learning of points-to specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 678–692. https://doi.org/10.1145/3192366.3192383

Bor-Yuh Evan Chang, Cezara Dragoi, Roman Manevich, Noam Rinetzky, and Xavier Rival. 2020. Shape Analysis. *Found. Trends Program. Lang.* 6, 1-2 (2020), 1–158. https://doi.org/10.1561/2500000037

David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. 1990. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, Bernard N. Fischer (Ed.). ACM, 296–310. https://doi.org/10.1145/93542.93585

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 '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 3–14. https://doi.org/10.1145/2491956.2462180

Cres. 2021. Report of container replacement synthesis. http://list.megatron-report.com

Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin T. Vechev. 2019. Unsupervised learning of API aliasing specifications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 745–759. https://doi.org/10.1145/3314221.3314640

Michael P Fay and Michael A Proschan. 2010. Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics surveys* 4 (2010), 1.

Tomás Fiedor, Lukás Holík, Adam Rogalewicz, Moritz Sinn, Tomás Vojnar, and Florian Zuleger. 2018. From Shapes to Amortized Complexity. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10747)*, Isil Dillig and Jens Palsberg (Eds.). Springer, 205–225. https://doi.org/10.1007/978-3-319-73721-8_10

Sumit Gulwani, Sagar Jain, and Eric Koskinen. 2009a. Control-flow refinement and progress invariants for bound analysis. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 375–385. https://doi.org/10.1145/1542476.1542518

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 62–73. https://doi.org/10.1145/1993498.1993506

Sumit Gulwani, Tal Lev-Ami, and Mooly Sagiv. 2009b. A combination framework for tracking partition sizes. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 239–251. https://doi.org/10.1145/1480881.1480912

Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009c. SPEED: precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 127–139. https://doi.org/10.1145/1480881.1480898

Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. Energy profiles of Java collections classes. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 225–236. https://doi.org/10.1145/2884781.2884869

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 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 38–49. https://doi.org/10.1145/1993498.1993504

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. 2012. Concurrent data representation synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 417–428. https://doi.org/10.1145/2254064.2254114

Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 535–550. https://doi.org/10.1145/3183713.3199671

Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 77–88. https://doi.org/10.1145/2254064.2254075

Changhee Jung, Silvius Rus, Brian P Railing, Nathan Clark, and Santosh Pande. 2011. Brainy: Effective selection of data structures. *ACM SIGPLAN Notices* 46, 6 (2011), 86–97.

Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Comput. Surv.* 49, 2 (2016), 29:1–29:47. https://doi.org/10.1145/2931098

Oliver Kennedy and Lukasz Ziarek. 2015. Just-In-Time Data Structures. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper9.pdf

Deokhwan Kim and Martin C. Rinard. 2011. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 528–541. https://doi.org/10.1145/1993498.1993561

Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 254–265. https://doi.org/10.1145/3213846.3213874

Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 343–353. https://doi.org/10.1145/2025113.2025160

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 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 355–368. https://doi.org/10.1145/2908080.2908122

Tianhan Lu, Bor-Yuh Evan Chang, and Ashutosh Trivedi. 2021. Selectively-Amortized Resource Bounding. In *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12913)*, Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi (Eds.). Springer, 286–307. https://doi.org/10.1007/978-3-030-88806-0_14

Irene Manotas, Lori Pollock, and James Clause. 2014. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*. 503–514.

Rashmi Mudduluru and Murali Krishna Ramanathan. 2016. Efficient flow profiling for detecting performance bugs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 413–424. https://doi.org/10.1145/2931037.2931066

Rocco De Nicola. 2011. Behavioral Equivalences. In *Encyclopedia of Parallel Computing*, David A. Padua (Ed.). Springer, 120–127. https://doi.org/10.1007/978-0-387-09766-4_517

Wellington Oliveira, Renato Oliveira, Fernando Castor, Benito Fernandes, and Gustavo Pinto. 2019. Recommending energy-efficient Java collections. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, 160–170. https://doi.org/10.1109/MSR.2019.00033

Wellington Oliveira, Renato Oliveira, Fernando Castor, Gustavo Pinto, and João Paulo Fernandes. 2021. Improving energy-efficiency by recommending Java collections. *Empir. Softw. Eng.* 26, 3 (2021), 55. https://doi.org/10.1007/s10664-021-09950-y

Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 369–378. https://doi.org/10.1145/2737924.2737966

Shankara Pailoor, Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2021. Synthesizing data structure refinements from integrity constraints. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 574–587. https://doi.org/10.1145/3453483.3454063

Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2155–2168. https://doi.org/10.1145/3133956.3134073

Thomas W. Reps, Mooly Sagiv, and Reinhard Wilhelm. 2007. Shape Analysis and Applications. In *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*, Y. N. Srikant and Priti Shankar (Eds.). CRC Press, 12.

Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24, 3 (2002), 217–298. https://doi.org/10.1145/514188.514190

Malavika Samak, Deokhwan Kim, and Martin C. Rinard. 2020. Synthesizing replacement classes. *Proc. ACM Program. Lang.* 4, POPL (2020), 52:1–52:33. https://doi.org/10.1145/3371120

Ohad Shacham, Martin T. Vechev, and Eran Yahav. 2009. Chameleon: adaptive selection of collections. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 408–418. https://doi.org/10.1145/1542476.1542522

Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 693–706. https://doi.org/10.1145/3192366.3192418

Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2021. Path-sensitive sparse analysis without path conditions. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 930–943. https://doi.org/10.1145/3453483.3454086

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, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 136–148. https://doi.org/10.1145/1375581.1375599

Linhai Song and Shan Lu. 2017. Performance diagnosis for inefficient loops. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 370–380. https://doi.org/10.1109/ICSE.2017.41

Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

Akhilesh Srikanth, Burak Sahin, and William R. Harris. 2017. Complexity verification using guided theorem enumeration. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 639–652. https://doi.org/10.1145/3009837.3009864

Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 265–266. https://doi.org/10.1145/2892208.2892235

Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: memory usage guided fuzzing. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 765–777. https://doi.org/10.1145/3377811.3380396

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.* 7, 3 (2008), 36:1–36:53. https://doi.org/10.1145/1347375.1347389

Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. 2016. Proteus: computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 61–72. https://doi.org/10.1145/2950290.2950340

Guoqing Xu. 2013. CoCo: Sound and Adaptive Replacement of Java Collections. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7920)*, Giuseppe Castagna (Ed.). Springer, 1–26. https://doi.org/10.1007/978-3-642-39038-8_1

Guoqing Xu and Atanas Rountev. 2010. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 160–173. https://doi.org/10.1145/1806596.1806616

Guoqing Xu, Dacong Yan, and Atanas Rountev. 2012. Static Detection of Loop-Invariant Data Structures. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7313)*, James Noble (Ed.). Springer, 738–763. https://doi.org/10.1007/978-3-642-31057-7_32

Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding low-utility data structures. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 174–186. https://doi.org/10.1145/1806596.1806617

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 63:1–63:26. https://doi.org/10.1145/3133887

Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How *not* to structure your database-backed web applications: a study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 800–810. https://doi.org/10.1145/3180155.3180194

Karen Zee, Viktor Kuncak, and Martin C. Rinard. 2008. Full functional verification of linked data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 349–361. https://doi.org/10.1145/1375581.1375624

Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 435–446. https://doi.org/10.1145/2491956.2462159