Type-Directed Bounding of Collections in Reactive Programs

Tianhan Lu(✉), Pavol Černý, Bor-Yuh Evan Chang, and Ashutosh Trivedi

University of Colorado Boulder, Boulder, USA
{tianhan.lu,pavol.cerny,bec,ashutosh.trivedi}@colorado.edu

Abstract. Our aim is to statically verify that in a given reactive program, the length of collection variables does not grow beyond a given bound. We propose a scalable type-based technique that checks that each collection variable has a given refinement type that specifies constraints about its length. A novel feature of our refinement types is that the refinements can refer to *AST counters* that track how many times an AST node has been executed. This feature enables type refinements to track limited flow-sensitive information. We generate verification conditions that ensure that the AST counters are used consistently, and that the types imply the given bound. The verification conditions are discharged by an off-the-shelf SMT solver. Experimental results demonstrate that our technique is scalable, and effective at verifying reactive programs with respect to requirements on length of collections.

1 Introduction

Collections are widely used abstract data types in programs. Collections, by providing a layer of abstraction, allow a programmer to flexibly choose different implementations leading to better modularity essential for developing good quality software. Since collections are extensively used, related performance issues have attracted considerable attention [20,29,30]. Besides performance issues, improper usage of collections may lead to security vulnerabilities such as denial-of-service (DoS) attacks. The performance and security issues are more pronounced in reactive programs such as service threads in operating systems or web applications. An important category of DoS vulnerabilities is out-of-memory error caused by collections with excessively large lengths.

Problem. The goal of this paper is to verify bounds on collection lengths using a scalable type-directed approach. Given constraints on inputs, our technique statically verifies at any point of execution total length of collection variables is less than a given bound. Verifying bound on collection lengths for reactive programs brings the following *challenges*:

Non-termination. Reactive programs do not terminate. The most common method for resource bound analysis is based on finding loop bounds [8,14,15,17,24,31]. This method therefore does not directly apply to reactive programs.

© Springer Nature Switzerland AG 2019
C. Enea and R. Piskac (Eds.): VMCAI 2019, LNCS 11388, pp. 275–296, 2019.
https://doi.org/10.1007/978-3-030-11245-5_13

Scalability. We need a scalable and modular solution, because real world reactive programs such as web servers are large (e.g. up to 30*kloc*).

Non-inductiveness of invariants. The necessary safety invariants might be *non-inductive*. For instance, collection lengths of a program may be bounded, but this is at first glance not provable by checking each statement in isolation, because a particular statement might simply add an element to a collection, thus breaking an invariant that is naively constructed to help verifying boundedness.

Approach. We now describe our approach, with a focus on how the three challenges are addressed. We develop a refinement type system, where the user is able to specify bounds on collection lengths, as well as an overall guarantee on the total length of all collections. These bounds might be symbolic, referring to for instance to bounds on lengths of input collections. Our tool QUANTM then type checks the program, and proves (or refutes) the overall guarantee.

First, to address the challenges of non-termination, our system relies purely on *safety properties*, never requiring a liveness property such as termination. We also do not require finding loop bounds.

Second, to address the challenge of scalability, we use type-based reasoning only. This entails checking at most one invariant per collection, as opposed to one invariant per each code location (as the approaches based on abstract interpretation [15,17] might need).

Third, to address the challenge of non-inductiveness of invariants, we allow the refinement refer to *AST counters* that count how many times an Abstract Syntax Tree (AST) node has been executed. For instance, consider the fragment:

```
while (true) { if (*) { C: s.add(r1);...;D: t.add(r2); }  }
```

and suppose we are interested in the invariant $|\texttt{len}(s) - \texttt{len}(t)| \leq 1$, that is, the difference between lengths of the two collections `s` and `t` is at most 1. The invariant is not inductive, the statement `s.add(r)` breaks it. However, let `C` be a counter associated with the AST node of `s.add(r1)`, and `D` with `t.add(r2)`. The invariant $\texttt{len}(s) + D = \texttt{len}(t) + C$ holds. We can then add a counter axiom $(D + 1 \equiv C) \vee (C \equiv D)$ as the two statements are inside a same basic block. Counter axioms are the place where the limited amount of flow-sensitive information that our system uses is captured. The inductive invariant and the axiom together imply the property we are interested in: $|\ \texttt{len}(s) - \texttt{len}(t)\ | \leq 1$.

Contributions. The main contributions of this paper are

- **Refinement types for collection lengths.** We propose to encode the total length of collection variables as safety properties of all reachable program states, as opposed to relying on analyzing time bounds. We develop a refinement type system where the refinements allow reasoning about collection lengths.
- **AST counters for inductive invariants.** A novel feature of our refinement types is that the refinements can refer to *AST counters* that track how many times an AST node has been executed. This feature enables type refinements to track limited flow-sensitive information.

– **Empirical evaluation.** Experimental results show that our approach scales to programs up to 30*kloc* (180*kloc* in total), within 52 s of analysis time per benchmark. Moreover, we discovered a Denial-of-Service vulnerability in one of our benchmarks because of correctly not being able to verify boundedness.

2 Overview

We demonstrate our approach for verifying the total collection lengths for reactive programs on a motivating example in Fig. 1.

2.1 Using Quantm

Overall, a user interacts with our tool QUANTM as follows. First, they write a driver that encodes a particular usage pattern that they are interested in. Then they specify invariants as type annotations. After these two steps, our type system will take care of the rest by automatically checking if the invariant relations are valid. If the invariants are indeed valid, QUANTM will automatically discharge a query to an off-the-shelf SMT solver, returning the result "verified" or "not verified". The "verified" answer is conclusive, as our method is sound. The "not verified" is inconclusive: either the bound does not hold, or the user has not provided sufficient invariants to answer the verification problem.

Example (Blogging Server). We simplified code from a Java web server based on the Spring framework that allows users to upload a blog post, delete a blog post and render a list of posts as an html page. Callback methods `postNewBlog`, `deleteBlog`, and `showBlogs` implement these functionalities. Method `driver` encodes an infinite input sequence that a user of our tool is interested in: it first reads a blog from input and appends it to the database, then renders the blog as an HTML page, and finally removes the blog from database. Our goal is to verify the boundedness of total collection lengths in every method separately, when input variables satisfy given constraints (e.g., inputs can have upper bounds on their length). In particular, callback methods `postNewBlog` and `deleteBlog` do not declare collection-typed variables and therefore they are vacuously bounded. More interestingly, we would like to verify the following bounding predicates denoted by `@Guarantee` in Fig. 1.

– The total length of collection variables in method `driver` is less than 2, i.e. $\texttt{len}(blogDB) < 2$
– Total length of collection variables in method `showBlogs` is less than or equal to length of input variable `blogDB`, i.e. $\texttt{len}(toShow) \leq \texttt{len}(blogDB) + 2$

We emphasize that our approach is able to verify above bounds when there exist neither time bounds nor input bounds, because input variables `input` and `blogDB` have no constraint at all, i.e. a `true` constraint.

The notation `@Inv` in Fig. 1 denotes a refinement type. The content inside the brackets following `@Inv` is the refinement of that particular type. For example, $\texttt{len}(blogDB) = \text{c8} - \text{c10}$ is a type refinement on variable `blogDB`.

```
1  void driver(@Inv("true") List<String> input) {
2      @Guarantee("len(blogDB)<2")
3      @Inv("len(blogDB)=c8-c10") List<String> blogDB = new List<String>();
4      @Inv("iterOf(input)") Iterator<String> it = input.iterator();
5      String blog;
6      while(*) {
7          blog = it.next();
8          c8:     postNewBlog(blog, blogDB);
9          c9:     showBlog(blogDB);
10         c10:    deleteBlog(blogDB);
11 } }
12 @Summary{"len(blogDB')=len(blogDB)+1"}
13 void postNewBlog(String blog, List<String> blogDB) {//callback: add post
14     blogDB.add(blog);
15 }
16 @Summary{"len(blogDB')=len(blogDB)-1"}
17 void deleteBlog(List<String> blogDB) { //callback: delete last post
18     blogDB.remove();
19 }
20 @Summary{"len(blogDB')=len(blogDB)"}
21 void showBlogs(@Inv("true") List<String> blogDB) {
22     @Guarantee("len(toShow)<=len(blogDB)+2")
23     //callback: display blog contents
24     @Inv("len(toShow)-idx(it)=c28+c30+c33-c32") List<String> toShow = new
        List<String>();
25     @Inv("iterOf(blogDB)")Iterator<String> it = blogDB.iterator();
26     String blog;
27     blog = "Welcome!\n";
28     c28:  toShow.add(b);
29     blog = "Blog begins:\n";
30     c30:  toShow.add(b);
31     while(*) {
32         c32:    blog = it.next();
33         c33:    toShow.add(blog);
34     }
35 // render toShow as an HTML page
36 }
```

Fig. 1. Motivating example: a simplified version of a blogging server.

Specifying Invariants with AST Counters. We now explain the role of the AST counters in the invariant. For example, if we look at the inner loop at line 31–34 in Fig. 1, the property we most likely need for list `toShow` is $\texttt{len}(toShow) \leq \texttt{idx}(it)+2$, where $\texttt{idx}(it)$ represents the number of elements that has been visited using iterator it. However, this property is actually not inductive because it breaks after line 28(as well as line 30), as $\texttt{len}(toShow)$ is incremented by 1 but nothing else is updated in the invariant. However, we can add AST counters to the invariant, and obtain $\texttt{len}(toShow) - \texttt{idx}(it) = \text{c28} + \text{c30} + \text{c33} - \text{c32}$. We thus obtain an inductive invariant that is then used as the type of `toShow`.

The purpose of these counters is to enable writing expressive invariants. The interesting invariants usually do not depend on the value of the counters (the value grows without bound for nonterminating programs), just on relations between counters of different AST nodes. These could be seen on the example in the previous section.

As another example, consider how we reason about the non-terminating loop at line 6–11, we first summarize the effects of callback `postNewBlog` and `deleteBlog` on any collection variable passed in as argument(s), which is to add 1 element to or remove 1 element from list `blogDB`. Method summaries are

automatically applied at invocation sites. Next, since we have AST counters, we are now able to easily define the length of variable `blogDB` as an inductive invariant $\texttt{len}(blogDB) = \text{c8} - \text{c10}$ (shown at line 2) that hold at before and after every execution step. Note that this invariant serves as a safety property of all program states under the existence of non-terminating executing traces, which is the root cause of the mainstream approach in resource bound analysis to fail under the scenario of reactive programs.

2.2 Inside Quantm

Typechecking. Our type system is based on Liquid types [22], where the refinements can express facts about collections and AST counters. Our type checking rules are standard, with added rules that capture the semantics of collections (lists) and counters.

Constraints on AST Counters. Constraints on AST counters are generated from the Abstract Syntax Tree structure of the program. For instance, AST counter `c32` is always either greater than (by 1) AST counter `c33` (after executing line 32) or equal to it (after executing line 33) at any time during an execution. We formalize this and other relations on counters in a set of axioms.

Verification Condition Generation. We generate verification conditions that ensure that the AST counters are used consistently, and that the types imply the given bound. For instance, now that we have invariants describing lengths of list `blogDB` and `toShow` in the method `showBlogs`, we can plug in counter axioms and check the required implications. For instance, the type of `toShow` is $\texttt{len}(toShow) - \texttt{idx}(it) = \text{c28} + \text{c30} + \text{c33} - \text{c32}$. From the counter axioms, we have that $\text{c28} \leq 1 \wedge \text{c30} \leq 1$ (as the corresponding statements are executed once at most) and $(\text{c32} \equiv \text{c33} + 1) \vee (\text{c32} \equiv \text{c33})$ (as the corresponding statements are sequentially executed). We then use an off-the-shelf SMT solver to check that the inductive invariant and the counter axioms imply the guarantee that the user specified: $\texttt{len}(toShow) \leq \texttt{len}(blogDB) + 2$.

3 Quantm Type System

In this section, we present the core calculus of our target program along with the types and refinements, and operational semantics. As usual, we write $\mathbb{B}$ and $\mathbb{Z}$ for the Boolean and integer domains. We write $\overline{v}$ to denote a list of syntactic elements separated either by comma or semicolon: $v_1, v_2, \ldots, v_k$ or $v_1; v_2; \ldots; v_k$. We also write $(\overline{v} :: v_{k+1})$ for the list value $(v_1, v_2, \ldots, v_k, v_{k+1})$. We model other types of collection data types (such as sets and maps) as lists because of being only interested in sizes of collection-typed variables.

3.1 Syntax and Refinement Types

Core Calculus. Our core calculus focuses on methods manipulating collections as shown in Fig. 2a. A method M is composed of a sequence of input-variable declarations $\overline{\tau\ u}$, a sequence of initialized local-variables declarations $\overline{\tau\ x = e}$, and a

$$
\begin{array}{lll}
\text{Method definition} & M & ::= \overline{\tau\ u}\ \overline{\tau\ x = e}\ s \\
\text{Compound statements} & s & ::= s_{\mathrm{B}} \mid \{\overline{s}\} \mid \texttt{if}(e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}(e)\ s \\
\text{Basic statements} & s_{\mathrm{B}} & ::= x = e \mid x = z.\texttt{next}() \mid y.\texttt{rmv}() \mid y.\texttt{add}(x) \mid \texttt{skip} \\
\text{Expressions} & e & ::= x \in X \mid u \in U \mid n \in \mathbb{Z} \mid b \in \mathbb{B} \mid y.\texttt{iter}() \\
 & & \quad \mid \texttt{new List}[\tau_{\mathrm{B}}] \mid e_1 \oplus e_2 \mid e_1 \bowtie e_2 \mid e_1 \vee e_2 \mid \neg e \\
\text{Variables} & u, x, y, z & ::= x \in X \mid u \in U \mid y, z \in X \cup U
\end{array}
$$

(a) The core calculus.

$$
\begin{array}{lll}
\text{Base types} & \tau_{\mathrm{B}} & ::= \texttt{Int} \mid \texttt{Bool} \mid \texttt{Iter}[\tau_{\mathrm{B}}] \mid \texttt{List}[\tau_{\mathrm{B}}] \\
\text{Refinement types} & \tau & ::= [\![\tau_{\mathrm{B}} \lfloor r]\!] \\
\text{Refinements} & r & ::= b \in \mathbb{B} \mid x_{\mathrm{bool}} \mid \texttt{iterOf}(x_{\mathrm{list}}) \mid e_1^{\tau} \bowtie e_2^{\tau} \mid r_1 \vee r_2 \mid \neg r \\
\text{Refinement expressions} & e^{\tau} & ::= n \in \mathbb{Z} \mid \nu_{\mathrm{int}} \mid x_{\mathrm{int}} \mid \texttt{len}(e^{\tau}_{\mathrm{list}}) \mid \texttt{idx}(e^{\tau}_{\mathrm{iter}}) \mid e_1^{\tau} \oplus e_2^{\tau} \mid c \in C \\
\text{List expressions} & e^{\tau}_{\mathrm{list}} & ::= \nu_{\mathrm{list}} \mid x_{\mathrm{list}} \\
\text{Iterator expressions} & e^{\tau}_{\mathrm{iter}} & ::= \nu_{\mathrm{iter}} \mid x_{\mathrm{iter}} \\
\text{Typing context} & \Gamma & ::= \cdot \mid \Gamma, x : \tau
\end{array}
$$

(b) Types and refinements.

Fig. 2. (a) The core calculus for methods manipulating collections. The operator $\oplus$ stands for arithmetic operators, while $\bowtie$ stands for comparison operators. (b) The types and corresponding refinements. The subscripts in variables $x_{\mathrm{bool}}, x_{\mathrm{int}}, x_{\mathrm{list}}, x_{\mathrm{iter}} \in X \cup U$ are used to emphasize their types, $\oplus$ is arithmetic operator restricted to linear arithmetic, and $\bowtie$ is a comparison operator.

method body s that is composed of basic and compound statements. We denote the set of input variables and local variables by U and X, respectively. The basic statements $x = z.\texttt{next}()$, $y.\texttt{rmv}()$, and $y.\texttt{add}(x)$ provide standard operations on iterator variable z and collection variable y. In addition, we have standard assignment statement $x = e$, where e is an expression without side effects.

Refinement Type System. Our type system, shown in Fig. 2b, permits type refinements over base types integer `Int`, boolean `Bool`, iterator `Iter` and list `List`. A refinement type $[\![\tau_{\mathrm{B}} \lfloor r]\!]$ further qualifies variables by providing an assertion over the values of the variable using a predicate r. A unique feature of our refinement predicates is that, the predicates can refer to AST counters $c \in C$ to track limited flow-sensitive information. Moreover, predicate can refer to the variable on which the refinement is expressed using the self-reference variable ν. A refinement can be expressed as an arbitrary Boolean combination of Boolean values b, Boolean-typed program variables x_{bool}, predicates $\texttt{iterOf}(x_{\mathrm{list}})$ (expressing that the variable is an iterator of a list variable x_{list}), and comparisons between *refinement expressions*. A refinement expression e^{τ} is integer-typed and can be composed of integer values n, integer-typed variable x_{int}, length expressions $\texttt{len}(e^{\tau}_{\mathrm{list}})$ (representing the length of list expression e^{τ}_{list}), index expressions $\texttt{idx}(e^{\tau}_{\mathrm{iter}})$ (representing the current index of an iterator expression e^{τ}_{iter}), *AST counter variables*, and arithmetic operations over other refinement expressions. An AST counter variable $c \in C$ is associated with an AST node. Intuitively, it counts the number of times an AST node has been executed. List expressions e^{τ}_{list} could be ν_{list} (which refers to the refined variable itself) or a list-typed

program variable x_{list}. Explanation for the iterator expression e^{τ}_{iter} is analogous. Typing context Γ is a mapping from variables to their types. Overall, our refinement language is in a decidable logic fragment EUFLIA ($\overline{\text{E}}$quality, $\overline{\text{U}}$ninterpreted $\overline{\text{F}}$unctions and $\overline{\text{LI}}$near $\overline{\text{A}}$rithmetic) where $\texttt{len}(e^{\tau}_{\text{list}})$, $\texttt{idx}(e^{\tau}_{\text{iter}})$ and $\texttt{iterOf}(e^{\tau}_{\text{iter}})$ are treated as uninterpreted functions.

3.2 Operational Semantics

We define small-step operational semantics of our core calculus as well as semantics of type refinements in Figs. 3 and 4. An environment (or equivalently, a state) η is a mapping from program variables to values, which intuitively serves as a stack activation record. The domain of variable values include integers, booleans, iterators, and list values. The calculus also supports lists of lists. We denote the initial environment as η_{init}. Environment η_{init} initializes counters as zero, input variables as concrete input values, and local variables as their initial values specified in the method.

Figure 3 defines the small-step operational semantics for our core calculus. We use the following three judgment forms:

1. Judgment form $\langle \eta, e \rangle \rightsquigarrow e'$ states that expression e is evaluated to expression e' in one evaluation step under environment η,
2. Judgment form $\langle \eta, s \rangle \rightsquigarrow \langle \eta', s' \rangle$ states that after one evaluation step of executing statement s under environment η, the environment changes to η' and the next statement to be evaluated is s', and
3. Judgment form $\langle \eta, s \rangle \twoheadrightarrow \eta'$ expresses the AST counter state transitions by modifying η to increment the counter value associated with statement s.

Compared with standard operational semantics (IMP language [28]), there are two main differences. The first difference is that we introduce collections into our core calculus. The semantics of collection operations is straightforward as shown in Fig. 3. The other significant difference is due to the use of AST counters in refinement types. Most of the differences from non-standard semantics is related to handling of these counters. The function $\kappa(s, M)$ returns the unique counter c associated with the statement s in the method M. Notice that the intermediate derivations of the rules may produce auxiliary statements that are not present in the original program. Since the refinement types may not refer to these counters, we ignore counter values for these auxiliary statements by associating them with a same special counter $\perp$, whose value we do not care about. E.g., The conclusion of the rule E-IfExpr introduces a new $^{\perp}\texttt{if}$ statement along with original statements s_1 and s_2, associating this new if-else statement with counter $\perp$. Rules E-Counter and E-Counter-Aux are mainly concerned with AST counter bookkeeping. The explanation of other rules is straightforward.

Types and Refinements. Figure 4 defines semantics of types and refinements. Judgment form $v \vDash_{\eta} \tau$ states that the value v conforms to a type τ under environment η. The semantics of the base-types $\eta[x] \vDash_{\eta} \tau_{\text{B}}$ is straightforward and hence omitted. The judgment form $\vDash_{\eta} x : r$ states that variable x to which

$$\text{Environment } \eta ::= \cdot \mid \eta[x \to v] \mid \eta[u \to v] \mid \eta[c \hookrightarrow n]$$
$$\text{Values} \quad v ::= n \in \mathbb{Z} \mid b \in \mathbb{B} \mid \texttt{Iter}(n \in \mathbb{N}, x \in X \cup U) \mid (v_1, v_2, \ldots, v_n)$$

(a) Environment and Values

$$\textsc{E-Var}\ \frac{}{\langle \eta, x\rangle \rightsquigarrow \eta[x]} \qquad \textsc{E-ArithL}\ \frac{\langle \eta, e_1\rangle \rightsquigarrow e_1'}{\langle \eta, e_1 \oplus e_2\rangle \rightsquigarrow e_1' \oplus e_2} \qquad \textsc{E-AritHR}\ \frac{\langle \eta, e_2\rangle \rightsquigarrow e_2'}{\langle \eta, v \oplus e_2\rangle \rightsquigarrow v \oplus e_2'} \qquad \textsc{E-Neg}\ \frac{\langle \eta, e\rangle \rightsquigarrow e'}{\langle \eta, \neg e\rangle \rightsquigarrow \neg e'}$$

$$\textsc{E-CompL}\ \frac{\langle \eta, e_1\rangle \rightsquigarrow e_1'}{\langle \eta, e_1 \bowtie e_2\rangle \rightsquigarrow e_1' \bowtie e_2} \qquad \textsc{E-CompR}\ \frac{\langle \eta, e_2\rangle \rightsquigarrow e_2'}{\langle \eta, v \bowtie e_2\rangle \rightsquigarrow v \bowtie e_2'} \qquad \textsc{E-OrL}\ \frac{\langle \eta, e_1\rangle \rightsquigarrow e_1'}{\langle \eta, e_1 \vee e_2\rangle \rightsquigarrow e_1' \text{ or } e_2} \qquad \textsc{E-OrR}\ \frac{\langle \eta, e_2\rangle \rightsquigarrow e_2'}{\langle \eta, v \vee e_2\rangle \rightsquigarrow v \vee e_2'}$$

$$\textsc{E-Counter}\ \frac{\kappa(s, M) = c}{\langle \eta, s\rangle \twoheadrightarrow \eta[c \hookrightarrow \eta[c] + 1]} \qquad \textsc{E-Counter-Aux}\ \frac{\kappa(s, M) = \bot}{\langle \eta, s\rangle \twoheadrightarrow \eta}$$

$$\textsc{E-Iterator}\ \frac{}{\langle \eta, y.\texttt{iter}()\rangle \rightsquigarrow \texttt{Iter}(0, y)} \qquad \textsc{E-NewList}\ \frac{}{\langle \eta, \texttt{new List}[\tau_{\mathrm{B}}]\rangle \rightsquigarrow ()} \qquad \textsc{E-Assign}\ \frac{\langle \eta, x = e\rangle \twoheadrightarrow \eta' \qquad \langle \eta, e\rangle \rightsquigarrow^* v}{\langle \eta, x = e\rangle \rightsquigarrow \langle \eta'[x \to v], \texttt{skip}\rangle}$$

$$\textsc{E-Next}\ \frac{\langle \eta, x = z.\texttt{next}()\rangle \twoheadrightarrow \eta' \qquad \eta[z] = \texttt{Iter}(i, y) \qquad \eta[y] = (v_1, \ldots, v_n) \qquad i < n{-}1}{\langle \eta, x = z.\texttt{next}()\rangle \rightsquigarrow \langle \eta'[z \to \texttt{Iter}(i+1, y)][x \to v_{i+1}], \texttt{skip}\rangle}$$

$$\textsc{E-Add}\ \frac{\langle \eta, y.\texttt{add}(x)\rangle \twoheadrightarrow \eta' \qquad \eta[y] = (\overline{v}) \qquad \langle \eta, x\rangle \rightsquigarrow v}{\langle \eta, y.\texttt{add}(x)\rangle \rightsquigarrow \langle \eta'[y \to (\overline{v} :: v)], \texttt{skip}\rangle} \qquad \textsc{E-Remove}\ \frac{\langle \eta, y.\texttt{rmv}()\rangle \twoheadrightarrow \eta' \qquad \eta[y] = (\overline{v} :: v)}{\langle \eta, y.\texttt{rmv}()\rangle \rightsquigarrow \langle \eta'[y \to (\overline{v})], \texttt{skip}\rangle}$$

$$\textsc{E-IfExpr}\ \frac{\langle \eta, \texttt{if}(e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2\rangle \twoheadrightarrow \eta' \qquad \langle \eta, e\rangle \rightsquigarrow e'}{\langle \eta, \texttt{if}(e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2\rangle \rightsquigarrow \langle \eta', {}^{\bot}\texttt{if}(e')\ \texttt{then}\ s_1\ \texttt{else}\ s_2\rangle}$$

$$\textsc{E-IfTrue}\ \frac{\langle \eta, \texttt{if(true)}\ \texttt{then}\ s_1\ \texttt{else}\ s_2\rangle \twoheadrightarrow \eta'}{\langle \eta, \texttt{if(true)}\ \texttt{then}\ s_1\ \texttt{else}\ s_2\rangle \rightsquigarrow \langle \eta', s_1\rangle} \qquad \textsc{E-IfFalse}\ \frac{\langle \eta, \texttt{if(false)}\ \texttt{then}\ s_1\ \texttt{else}\ s_2\rangle \twoheadrightarrow \eta'}{\langle \eta, \texttt{if(false)}\ \texttt{then}\ s_1\ \texttt{else}\ s_2\rangle \rightsquigarrow \langle \eta', s_2\rangle}$$

$$\textsc{E-While}\ \frac{\langle \eta, \texttt{while}(e)\ s\rangle \twoheadrightarrow \eta'}{\langle \eta, \texttt{while}(e)\ s\rangle \rightsquigarrow \langle \eta', {}^{\bot}\texttt{if}(e)\ \texttt{then}\ {}^{\bot}\{s; {}^{\bot}\texttt{while}(e)\ s\}\ \texttt{else}\ \texttt{skip}\rangle}$$

$$\textsc{E-Block}\ \frac{\langle \eta, \{\overline{s}\}\rangle \twoheadrightarrow \eta' \qquad \overline{s} = s_1; s_2; \ldots; s_n \qquad \langle \eta', s_1\rangle \rightsquigarrow \langle \eta'', s_1'\rangle \qquad \overline{s'} = {}^{\bot}s_1'; s_2; \ldots; s_n}{\langle \eta, \{\overline{s}\}\rangle \rightsquigarrow \langle \eta'', {}^{\bot}\{\overline{s'}\}\rangle} \qquad \textsc{E-BlockSkip}\ \frac{\overline{s} = \texttt{skip}; s_2; \ldots; s_n \qquad \overline{s'} = s_2; \ldots; s_n}{\langle \eta, \{\overline{s}\}\rangle \rightsquigarrow \langle \eta, {}^{\bot}\{\overline{s'}\}\rangle}$$

(b) Operational semantics

Fig. 3. Environment, values, and small-step operational semantics.

$$
\begin{array}{lll}
\eta[x] \vDash_\eta [\![\tau_{\mathrm{B}} \lfloor r]\!] & \text{iff} & \eta[x] \vDash_\eta \tau_{\mathrm{B}} \text{ and } \vDash_\eta x : r \\
\vDash_\eta b & \text{iff} & b \equiv \texttt{true} \\
\vDash_\eta x : y_{\mathrm{bool}} & \text{iff} & \vDash_\eta \eta[y_{\mathrm{bool}}] \\
\vDash_\eta x : \neg r & \text{iff} & (\vDash_\eta x : r) \neq \texttt{true} \\
\vDash_\eta x : r_1 \vee r_2 & \text{iff} & \vDash_\eta x : r_1 \text{ or } \vDash_\eta x : r_2 \\
\vDash_\eta x : \texttt{iterOf}(y) & \text{iff} & \text{for some } i \geq 0 \text{ we have } \eta[x] = \texttt{Iter}(i, y) \\
\vDash_\eta x : e_1^\tau \bowtie e_2^\tau & \text{iff} & \texttt{eval}(x : e_1^\tau)_\eta \bowtie \texttt{eval}(x : e_2^\tau)_\eta \\
\texttt{eval}(x : e^\tau)_\eta & = & v, \text{ where } e = \texttt{subst}(x : e^\tau)_\eta \wedge \langle \eta, e \rangle \rightsquigarrow^* v \\
\texttt{subst}(x : e^\tau)_\eta & = & (e^\tau[x/\nu])[n_i/\texttt{len}(x^i_{\mathrm{list}}), k_j/\texttt{idx}(y^j_{\mathrm{iter}})]_{\,\forall x^i_{\mathrm{list}},\, y^j_{\mathrm{iter}}} \\
 & & \text{where } \eta[x^i_{\mathrm{list}}] = (v_1, \ldots, v_{n_i}) \text{ and } \eta[y^j_{\mathrm{iter}}] = \texttt{Iter}(k_j, *)
\end{array}
$$

Fig. 4. Refinement semantics.

expression ν in refinement r refers, conforms to the refinement under the environment η. We exploit helper functions $\texttt{eval}(x : e^\tau)_\eta$ and $\texttt{subst}(x : e^\tau)_\eta$ in refinement semantics defined in the following fashion:

- Function $\texttt{eval}(x : e^\tau)_\eta$ takes a refinement expression e^τ, a variable x (to which self-reference ν in e^τ refers), and an environment η as inputs and then returns the evaluation of refinement expression.
- Function $\texttt{subst}(x : e^\tau)_\eta$ takes as inputs refinement expression e^τ, variable x (to which expression ν refers), and environment η, and returns an expression that is the result of first substituting self-reference ν with variable x and then substituting every $\texttt{len}(x_{\mathrm{list}})$ in e^τ with length of list-typed variable x_{list}, as well as every $\texttt{idx}(y_{\mathrm{iter}})$ with index value of iterator-typed variable y_{iter}.

We write $\rightsquigarrow^*$ for the transitive closure of $\rightsquigarrow$. Most of the refinement semantics are straightforward. In particular, the semantics of $\texttt{iterOf}(y)$ is that variable x, to which ν refers, is an iterator for list-typed variable y.

3.3 Well-Typed Methods

We say that an environment η is *reachable* in a method M if $\langle \eta_{\mathrm{init}}, M \rangle \rightsquigarrow^* \langle \eta, s \rangle$. We write $\mathsf{ReachEnv}(M)$ for the set of all reachable environments of M. We say that an environment η is *well-typed* in M if all of the variables conform to their types, i.e. for all $x \in X \cup U$ with type $[\![\tau_{\mathrm{B}} \lfloor r]\!]$, we have that $\eta[x] \vDash_\eta [\![\tau_{\mathrm{B}} \lfloor r]\!]$. We write $\mathsf{WellTyped}(M)$ for the set of all well-typed environments in M. We say that a method M is *well-typed* if all of the reachable states of M are well-typed, i.e. $\mathsf{ReachEnv}(M) \subseteq \mathsf{WellTyped}(M)$.

4 Collection Bound Verification Problem

Given a method M, our goal is to verify that if the inputs to the method satisfy a given assumption $\phi_{\mathcal{A}}$, then the method M guarantees that the collection lengths remain bounded. The guarantee requirements $\phi_{\mathcal{G}}$ can be expressed as a predicate constructed using the refinement language introduced in Fig. 2b Observe that,

since this verification condition is not attached to any particular variable, it is free from predicates $\texttt{iterOf}(x_{\text{list}})$ and self-reference ν. We further assume that the assumptions on the input variables are expressed using type refinements on the input variables. Formally, we are interested in the following problem:

Definition 1 (Collection Bound Verification Problem). *Given a method M along with its input variables with types and refinements $u_i : \tau_i$, and a guarantee requirement $\phi_{\mathcal{G}}$, verify that every reachable environment satisfies $\phi_{\mathcal{G}}$, i.e. for all $\eta \in \mathsf{ReachEnv}(M)$ we have that $\vDash_{\eta} \phi_{\mathcal{G}}$.*

We present a type-directed approach to solve this problem. We first propose type-checking rules to verify if the method is well-typed. Then, we discuss how to automatically derive AST counter relation axioms in Sect. 4.2. Finally, we reduce solving the verification problem into issuing SMT queries, in Sect. 4.3, using as constraints the type refinements verified in Sect. 4.1 as well as AST counter relation axioms extracted from Sect. 4.2.

4.1 Type Checking

Our key analysis algorithm is encoded into refinement type checking rules shown in Fig. 5. Subtyping between two refinement types is defined as the implication relation between two refinements using the following rule:

$$\frac{\tau_{\text{B}1} <: \tau_{\text{B}2} \qquad r_1 \implies r_2}{[\![\tau_{\text{B}1} \lfloor r_1]\!] <: [\![\tau_{\text{B}2} \lfloor r_2]\!]} \; <:\text{-RefinementTyp}$$

Figure 5 defines type-checking rules for refinement types, while the rules for base types are standard and thus presented in companion paper [27]. Notation $\tau[e^{\tau\prime}/e^{\tau}]$ denotes substituting expression e^{τ} with $e^{\tau\prime}$ in the refinement of type τ.

The Judgment form $\Gamma \vdash s$ states that the statement s is successfully type checked under typing context Γ if premises are satisfied. We case split on the right hand side of assignment statement $x = e$ into: Rule T-AssignIter, T-Assign, T-AssignList, and T-AssignNewList. Intuitively, type checking rules check that after applying each corresponding evaluation rule, type refinements should still be valid. More specifically, in each type checking rule we check for all refinements, if its validity before applying a corresponding evaluation rule implies its validity afterwards. For example, after applying Rule E-Add, the environment has the following updates: length of collection variable y is incremented by 1 and the associated AST counter's value is incremented by 1. Therefore Rule T-Add checks the implication of validity between a type $\tau_w[w/\nu]$ and the result after applying to it a substitution $(\tau_w[w/\nu])[(\texttt{len}(y)+1)/\texttt{len}(y), (c+1)/c]$, which precisely expresses the actual value of type $\tau_w[w/\nu]$ after applying Rule E-Add in terms of its value beforehand. Rule T-Remove is dual to Rule T-Add. In Rule T-AssignIter, in addition to subtyping checking, we also check for variable z if its refinement will still

hold true after substituting $\texttt{iterOf}(*)$ with $\texttt{iterOf}(y)$. The intuition behind is that after evaluating statement $z = y.\texttt{iter}()$, variable z will become an iterator for variable y, no matter what list it was an iterato for. For a reader interested in why we must treat refinement $\texttt{iterOf}(x)$ differently, the root cause here is that unlike $\texttt{idx}(z)$ specifying a property of **one** variable, $\texttt{iterOf}(x)$ actually specifies a relation between **two** variables. Rule T-Assign checks if refinements will still hold true when x becomes e, no matter if variable x is integer-typed, boolean-typed or iterator-typed (where $\texttt{idx}(x)$ becomes $\texttt{idx}(e)$). Rule T-AssignList and Rule T-Assign are similar, except that in Rule T-AssignList we check if refinements will still hold true when $\texttt{len}(x)$ becomes $\texttt{len}(e)$. We split Rule T-AssignList from Rule T-Assign, avoiding simply checking if x becoming e will break any refinement, because assignment $x = e$ does not make refinement $\texttt{iterOf}(x)$ become $\texttt{iterOf}(e)$. In Rule T-Next, besides checking the validity of implication, we also check if every type refinement is logically equivalent to itself being existentially quantified by variable x. Intuitively, this ensures soundly that the assignment in statement $x = z.\texttt{next}()$ will not break any refinement, since there is no constraint on list elements retrieved from list variable z by invoking $z.\texttt{next}()$. Just like Rule E-Counter interleaves with every evaluation rule in Fig. 3, Rule T-Counter serves as a premise for every type checking rule of compound statements. For every type checking rule of basic statements, Rule T-Counter is embedded into subtyping checking. Rule T-Decl checks that all local variables' type refinements are valid, given their initial values. We also define a helper function $\langle\!\langle s_1 \rangle\!\rangle \subseteq \langle\!\langle s_2 \rangle\!\rangle$ that is used in Rule T-Decl, which describes AST sub-node relations between AST node s_2 and its sub-node s_1.

$$\textsc{SubNode-Block}\quad \frac{\overline{s} = s_1; \ldots; s_n}{\langle\!\langle s_i \rangle\!\rangle \subseteq \langle\!\langle \{\overline{s}\} \rangle\!\rangle,\ \text{for all } i \in \{1, \cdots, n\}} \qquad \textsc{SubNode-While}\quad \frac{}{\langle\!\langle s \rangle\!\rangle \subseteq \langle\!\langle \texttt{while}(e)\ s \rangle\!\rangle}$$

$$\textsc{SubNode-If}\quad \frac{}{\langle\!\langle s_i \rangle\!\rangle \subseteq \langle\!\langle \texttt{if}(e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \rangle\!\rangle,\ \text{for } i \in \{1, 2\}}$$

4.2 AST Counter Axioms

We next present the AST counter relation axioms. The goal of deriving counter relation axioms is to improve verification precision by having additional constraints when encoding the problem statement into SMT queries. We let counter relations precisely correspond to abstract syntax tree structure of a program. Respecting semantics of counters, these counters keep record of the number of times a particular AST node has been executed at runtime.

The function $\Delta(s)$ takes as input a statement s and statically outputs a predicate about the relations on all AST sub nodes of statement s, as well as counter relation axioms derived from all AST sub nodes themselves. For example, Rule R-Block extracts counter relations from a block of statements $\{\overline{s}\}$.

For $1 \leq j \leq n-1$, in the constraint d_j the counter c_i associated with statement s_i is either: (a) equal to counter c_{i+1} associated with statement s_{i+1}, when statement s_i and s_{i+1} have both been executed; or (b) the counter c_i is equal to $c_{i+1}+1$, when statement s_i has been executed, but not statement s_{i+1}. Intuitively, constraint d_j describes a set of valid counter relations at one program state, which is immediately after executing statement s_j but before executing statement s_{j+1}. Constraint d_n denotes the counter relations right after finishing executing block statement $\{\overline{s}\}$. Additionally, the value of counter c_0 (associated with block statement $\{\overline{s}\}$ itself) is always equivalent to the value of counter c_1 (associated with the first statement s_1 in the block), respecting operational semantics of $\{\overline{s}\}$ defined in Rule E-Block of Fig. 3. Furthermore, the constraints C_i, for $1 \leq i \leq n$, are recursively generated from every statement s_i. Intuitively, these relations describes counter relations when flow-sensitively executing the code block $\{\overline{s}\}$ (Fig. 6).

As another example, Rule R-While extracts counter relations from a while loop. Note that we cannot conclude any relations between counter c_{b} (associated with loop body s) and counter c_0 (associated with loop $\texttt{while}(e)\ s$), because although loop body s may be executed for a positive number times or may not be executed, loop $\texttt{while}(e)\ s$ will always be executed for one more time whenever executing this AST node, according to Rule E-While in Fig. 3. Other rules are straightforward. Proof of soundness for above counter relations is straightforward and hence omitted.

4.3 Collection Bound Verification

We formalize our approach that solves the collection bound verification problem for method M by constructing an SMT query. We first obtain constraints from type refinements and AST counter axioms, and then generate the following SMT query that searches for counterexamples for the guarantee $\phi_{\mathcal{G}}$:

$$\Psi_{\text{ast}} \wedge \bigwedge_{\langle\!\langle \tau\ u \rangle\!\rangle \subseteq \langle\!\langle M \rangle\!\rangle} \Phi(u:\tau) \wedge \bigwedge_{\langle\!\langle \tau\ x \rangle\!\rangle \subseteq \langle\!\langle M \rangle\!\rangle} \Phi(x:\tau) \wedge \neg\phi_{\mathcal{G}},$$

where Ψ_{ast} are the constraints generated from functions $\Delta(s)$ defined in Sect. 4.2. The helper function $\Phi(x:\tau)$, defined in Fig. 7 takes as input a variable x together with its type τ, and returns refinement constraints from type τ. Intuitively, constraint Ψ_{ast} soundly constrains the possible values that AST counters could take when flow-sensitively executing a program. Constraints $\Phi(u:\tau)$ encode assumptions on the inputs to the method, and constraints $\Phi(x:\tau)$ soundly constrain the values that local variables could take. Together they constitute a constraint on all reachable program states (which is proven in Sect. 5). In other words, the conjunction of constraints defines a set of program states that is a sound over-approximation of every actual reachable program states of method M. Therefore, the answer to the query provides a sound solution to the collection bound verification problem.

T-Add

$$\frac{\kappa(y.\texttt{add}(x), M) = c \qquad \forall(w : \tau_w) \in \Gamma.(\tau_w[w/\nu])\texttt{<:}(\tau_w[w/\nu])[(\texttt{len}(y)+1)/\texttt{len}(y), (c+1)/c]}{\Gamma \vdash y.\texttt{add}(x)}$$

T-Remove

$$\frac{\kappa(y.\texttt{rmv}(), M) = c \qquad \forall(w : \tau_w) \in \Gamma.(\tau_w[w/\nu])\texttt{<:}(\tau_w[w/\nu])[(\texttt{len}(y)-1)/\texttt{len}(y), (c+1)/c]}{\Gamma \vdash y.\texttt{rmv}()}$$

T-AssignIter

$$\frac{\kappa(z = y.\texttt{iter}(), M) = c \qquad \forall(w : \tau_w) \in \Gamma.(\tau_w[w/\nu])\texttt{<:}(\tau_w[w/\nu])[0/\texttt{idx}(z), (c+1)/c] \qquad \Gamma \vdash z : \tau_z \qquad (\tau_z[z/\nu])\texttt{<:}(\tau_z[z/\nu])[0/\texttt{idx}(z), (c+1)/c, \texttt{iterOf}(y)/\texttt{iterOf}(*)]}{\Gamma \vdash z = y.\texttt{iter}()}$$

T-Assign

$$\frac{\kappa(x = e, M) = c \qquad \Gamma \vdash x : [\![\tau_{\mathrm{B}} \lfloor r_x]\!] \qquad \tau_{\mathrm{B}} \text{ is not a list type} \qquad \forall(w : \tau_w) \in \Gamma.(\tau_w[w/\nu])\texttt{<:}(\tau_w[w/\nu])[e/x, (c+1)/c]}{\Gamma \vdash x = e}$$

T-AssignList

$$\frac{\kappa(x = e, M) = c \qquad \Gamma \vdash x : [\![\tau_{\mathrm{B}} \lfloor r_x]\!] \qquad \tau_{\mathrm{B}} \text{ is a list type} \qquad \forall(w : \tau_w) \in \Gamma.(\tau_w[w/\nu])\texttt{<:}(\tau_w[w/\nu])[\texttt{len}(e)/\texttt{len}(x), (c+1)/c]}{\Gamma \vdash x = e}$$

T-AssignNewList

$$\frac{\kappa(x = \texttt{new List}[\tau_{\mathrm{B}}], M) = c \qquad \forall(w : \tau_w) \in \Gamma.(\tau_w[w/\nu])\texttt{<:}(\tau_w[w/\nu])[0/\texttt{len}(x), (c+1)/c]}{\Gamma \vdash x = \texttt{new List}[\tau_{\mathrm{B}}]}$$

T-Counter

$$\frac{s \neq s_B \qquad \kappa(s, M) = c \qquad \forall(w : \tau_w) \in \Gamma.(\tau_w[w/\nu])\texttt{<:}(\tau_w[w/\nu])[(c+1)/c]}{\Gamma \vdash_c s}$$

T-Next

$$\frac{\kappa(x = z.\texttt{next}(), M) = c \qquad \forall(w : \tau_w) \in \Gamma.(\tau_w[w/\nu])\texttt{<:}(\tau_w[w/\nu])[(\texttt{idx}(z)+1)/\texttt{idx}(z), (c+1)/c] \qquad \forall(w : [\![\tau_w \lfloor r_w]\!]) \in \Gamma.r_w \iff \exists x.r_w}{\Gamma \vdash x = z.\texttt{next}()}$$

T-Decl

$$\frac{\langle \eta_{\text{init}}, e\rangle \rightsquigarrow v \qquad \Gamma \vdash s \qquad \forall \langle\!\langle \tau\ x = e \rangle\!\rangle \subseteq \langle\!\langle M \rangle\!\rangle.v \vDash_{\eta_{\text{init}}} \tau}{\Gamma \vdash \overline{\tau\ u}\ \overline{\tau\ x = e}\ s}$$

T-Block

$$\frac{\overline{s} = s_1; \ldots; s_n \qquad \Gamma \vdash_c \{\overline{s}\} \qquad \Gamma \vdash s_i, \text{ for all } i \in \{1, \cdots, n\}}{\Gamma \vdash \{\overline{s}\}}$$

T-While

$$\frac{\Gamma \vdash_c \texttt{while}(e)\ s \qquad \Gamma \vdash s}{\Gamma \vdash \texttt{while}(e)\ s}$$

T-Skip

$$\frac{}{\Gamma \vdash \texttt{skip}}$$

T-If

$$\frac{\Gamma \vdash_c \texttt{if}(e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \texttt{if}(e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2}$$

Fig. 5. Type checking rules

$$\textsc{R-Block}\quad \frac{\overline{s} = s_1; \ldots; s_n \qquad \Delta(s_i) = C_i \text{ and } \kappa(s_i, M) = c_i, \text{ for all } i \in \{1, \cdots, n\} \qquad d_j = (\bigwedge_{i=1}^{j} c_i \equiv c_{i-1} \wedge c_{j+1}{+}1 \equiv c_j), \text{ for all } j \in \{1, \cdots, n{-}1\} \qquad \kappa(\{\overline{s}\}, M) = c_0 \qquad d_n = (c_0 \equiv \cdots \equiv c_n)}{\Delta(\{\overline{s}\}) = \bigwedge_{i=1}^{n} C_i \wedge \bigvee_{i=1}^{n} d_i}$$

$$\textsc{R-While}\quad \frac{\Delta(s) = C \qquad \kappa(\texttt{while}(e)\ s, M) = c_0 \qquad \kappa(s, M) = c_{\mathrm{b}}}{\Delta(\texttt{while}(e)\ s) = C} \qquad \textsc{R-Basic}\quad \frac{}{\Delta(s_{\mathrm{B}}) = \texttt{true}}$$

$$\textsc{R-If}\quad \frac{\Delta(s_i) = C_i \text{ and } \kappa(s_i, M) = c_i, \text{ for } i \in \{1, 2\} \qquad \kappa(\texttt{if}(e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2, M) = c_0}{\Delta(\texttt{if}(e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2) = (c_0 \equiv c_1 + c_2) \wedge C_1 \wedge C_2}$$

Fig. 6. AST counter axioms

$$\begin{aligned}
&\Phi(x : \tau_{\mathrm{B}}) && \stackrel{\text{def}}{=\!=} \texttt{true}\\
&\Phi(x : [\![\tau_{\mathrm{B}} \lfloor\, x_{\mathrm{bool}}]\!]) && \stackrel{\text{def}}{=\!=} x_{\mathrm{bool}}[x/\nu]\\
&\Phi(x : [\![\tau_{\mathrm{B}} \lfloor\, e_1^{\tau} \bowtie e_2^{\tau}]\!]) && \stackrel{\text{def}}{=\!=} e_1^{\tau}[x/\nu] \bowtie e_2^{\tau}[x/\nu]\\
&\Phi(x : [\![\tau_{\mathrm{B}} \lfloor\, r_1 \vee r_2]\!]) && \stackrel{\text{def}}{=\!=} r_1[x/\nu] \vee r_2[x/\nu]\\
&\Phi(x : [\![\tau_{\mathrm{B}} \lfloor\, \neg r]\!]) && \stackrel{\text{def}}{=\!=} \neg r[x/\nu]\\
&\Phi(x : [\![\tau_{\mathrm{B}} \lfloor\, \texttt{iterOf}(y)]\!]) && \stackrel{\text{def}}{=\!=} 0 \le \texttt{idx}(x) \le \texttt{len}(y)
\end{aligned}$$

Fig. 7. The helper function Φ for extracting refinement constraints.

5 Soundness

In this section, we present theorems on refinement preservation and refinement progress. Intuitively, refinement preservation guarantees that if a program passes refinement type checking (Sect. 4.1), then it will always end up in a well-typed environment (Sect. 3.3), under which we perform bound verification (Sect. 4.3). Refinement progress states that a program that passes type checking will not get stuck. Refinement preservation is the core theorem, we prove it below.

Theorem 1 (Refinement preservation). *If we have that $\eta \vDash \Gamma$, $\Gamma \vdash s$, and $\langle \eta, s\rangle \rightsquigarrow \langle \eta', s'\rangle$, then $\eta' \vDash \Gamma$ and $\Gamma \vdash s'$.*

Proof. Given $\eta \vDash \Gamma$ and $\Gamma \vdash s$ and $\langle \eta, s\rangle \rightsquigarrow \langle \eta', s'\rangle$, we focus on proving $\eta' \vDash \Gamma$, because the validity of $\Gamma \vdash s'$ is directly implied from the premises in Fig. 5. The goal is to prove for every variable x_i with type τ_i in $\texttt{Dom}(\eta)$, we have $\eta'[x_i] \vDash_{\eta'} \tau_i[x_i/\nu]$.

– **Rule E-Add**: We need to prove that if $\eta \vDash \Gamma$ and $\langle \eta, y.\texttt{add}(x)\rangle \rightsquigarrow \langle \eta', \texttt{skip}\rangle$, then $\eta' \vDash \Gamma$. From the Rule T-Add, we have

(Fact 1): $\eta \vDash \Gamma$ implies that $\eta \vDash \Gamma[(\texttt{len}(y)+1)/\texttt{len}(y), (c+1)/c]$, where we define $\Gamma[(\texttt{len}(y)+1)/\texttt{len}(y), (c+1)/c]$ as performing substitution $[(\texttt{len}(y)+1)/\texttt{len}(y), (c+1)/c]$ for all types in typing context Γ.
From the Rule E-ADD, we can infer that if $\langle \eta, y.\texttt{add}(x)\rangle \rightsquigarrow \langle \eta', \texttt{skip}\rangle$, then $\eta'(z) = \eta(z)$ for variables other than c and y. Furthermore, $\eta'[c] = \eta[c] + 1$ an $\texttt{len}(\eta'[y]) = \texttt{len}(\eta[y]) + 1$. Based on these properties of η', we prove by a simple induction on the structure of refinements that (Fact 2): if $\eta \vDash \Gamma[(\texttt{len}(y)+1)/\texttt{len}(y), (c+1)/c]$ then $\eta' \vdash \Gamma$.
By chaining Fact 1 and Fact 2, we can conclude the proof.

The other cases are similar or simpler, and can be found in the companion paper [27]. □

Corollary 1 states that all reachable program states are well-typed (Sect. 3.3). The proof immediately follows from Theorem 1.

Corollary 1. *If $\Gamma \vdash M$ and $\eta \in \mathit{ReachEnv}(M)$ then $\eta \vDash \Gamma$.*

Theorem 2 (Refinement progress). *If $\eta \vDash \Gamma$ and $\Gamma \vdash s$, then either statement s is `skip`, or there exist η' and s' such that $\langle \eta, s\rangle \rightsquigarrow \langle \eta', s'\rangle$*

The proof of Theorem 2 is standard and hence omitted.

6 Experiment

We implemented our tool QUANTM in Scala using the Checker Framework [12, 21], Microsoft Z3 [11] and Scala SMT-LIB [2]. QUANTM is implemented as a Java annotation processor, relying on the Checker Framework to extract type annotations and perform type checking. Microsoft Z3 served as an off-the-shelf SMT solver. We also used Scala SMT-LIB for parsing string-typed annotations. We chose several web applications as benchmarks (180k lines of code in total), each of which supports various functionalities. Benchmarks were collected from different sources, including GitHub (jforum3 with 218 stars and SpringPetClinic with 2325 stars), Google Code Archive (jRecruiter[1]), and DARPA STAC project [1] (TextCrunchr, Braidit, WithMi, Calculator, Battleboats, Image_processor, Smartmail, Powerbroker, and Snapbuddy). To set up the experiments, we created drivers invoking callback methods in patterns that imitate standard usage. To support the Object-Oriented feature (which is orthogonal to the problem and approach in the paper), we not only annotate collection-typed local variables, but also annotate collection-typed object fields that are reachable from local variables. Then we gave bounds to each method as tight as possible and used Microsoft Z3 to verify the bounds.

6.1 Research Questions

We evaluated our technique by answering the following research questions

RQ1. **Bound verification.** How effective is *AST Counter Instrumentation*? That is, what percentage of methods and collection variables were verified w.r.t. their specifications.

RQ2. **Analysis speed.** How fast/scalable is our verification technique?

[1] https://code.google.com/archive/p/jrecruiter/.

Table 1. Benchmark results. "Lines of code" counts the total lines of code in projects. "Verified methods" gives number of verified methods and unverified methods. Number of verified methods is split into *non-vacuously* and *vacuously* verified, where being vacuously verified means not declaring any local collection variables. "Verified/Unverified collections" gives the number of collection variables that are verified versus unverified. "Callbacks" gives the number of invoked callbacks in drivers. "Method summaries" gives the number of method summaries supporting verifying collection variables that are inter-procedurally mutated. "Analysis time" indicates the speed of our analysis on each benchmark. Experiments were conducted on a 4-core 2.9 GHz Intel Core i7 MacBook with 16 GB of RAM running OS X 10.13.6.

Benchmarks	Lines of code	Verified/Unverified methods	Verified/Unverified collections	Call-backs	Summaries	Analysis time (s)
TextCrunchr	2,150	(13 + 190)/5	23/5	4	9	14.7
Braidit	20,835	(8 + 2114)/0	50/0	8	8	84.2
jforum3	22,813	(35 + 1675)/8	54/10	24	27	69.8
jRecruiter	13,936	(29 + 933)/5	40/4	10	7	45.1
SpringPetClinic	1,429	(6 + 98)/0	11/1	9	12	15.8
WithMi	24,927	(30 + 2515)/5	35/2	5	4	82.0
Calculator	5,378	(20 + 316)/2	25/6	5	3	18.2
Battleboats	21,525	(8 + 2171)/6	12/2	5	2	75.6
Image_processor	1,365	(4 + 110)/0	5/0	0	0	7.8
Smartmail	1,977	(7 + 137)/4	11/3	0	0	10.9
Powerbroker	29,374	(22 + 3015)/8	27/3	3	8	91.6
Snapbuddy	34,797	(57 + 2940)/8	88/9	5	2	107.0
Total	180,506	(239 + 16214)/51	381/45	78	82	622.6

RQ1: Bound Verification. We verified 16453 methods in total, 239 of which are non-vacuously verified (who declares at least one collection variable) and the rest are vacuously verified (who declares no collection variable). If not considering vacuously verified methods, then we verified 239 out of 290 (82.4%) methods. In order to verify method boundedness, we also wrote and verified global invariants on 381 collection variables out of a total of 426 (89.4%), as well as provided 82 method summaries. We invoked 78 callbacks from drivers. We believe this result demonstrates that our technique is effective at verifying method and variable specifications. Our technique works very well when there is no statement reading a list-typed element from a collection, which if it happens, constitutes the vast majority of the causes of the 51 unverified methods and 45 unverified collections, because to ensure soundness we had to enforce no constraint on these list-typed variables read from collections, leading to unboundedness. We currently do not support this feature in the type system and we will leave it for future work. Note that in the table we did not include unverified methods and variables caused by orthogonal problems such as Java features (e.g. dynamic dispatch) discussed in Sect. 6.2.

We attribute the effectiveness of *AST Counter Instrumentation*to the scalable type checking approach and our AST counter-base approach. Also note that without *AST Counter Instrumentation*, it would have not been possible to **flow-insensitively** verify the desired properties. The detailed results from each benchmark are in Table 1.

Alias. Note that the operational semantics defined in Fig. 3 does not support aliasing among collection-typed variables. This is because aliasing is orthogonal to the problem and approach in the paper. To demonstrate that this is indeed the case, we extend our framework with aliasing, which we present in the companion paper [27]. The implementation uses the framework from the appendix.

Case Studies. We present an interesting loop that we found and simplified from the jforum3 benchmark. In the while loop at line 4–17, line 5 reads in a String with `readLine` and line 14 adds an element to list `comments`. Although the while loop may not terminate, the inductive invariant $\texttt{len}(comments) - \texttt{idx}(reader) = c5\text{–}c14$ is preserved before and every execution step. Here we consider variable `reader` as an iterator, respecting the semantics of the `readLine` API.

```
1  @Inv("len(comments)-idx(reader)=c5-c14") List<String> comments = new
       ArrayList<>();
2  // ...
3  String s;
4  while (true) {
5  c5: s = reader.readLine();
6      if (s != null) {
7          s = s.trim();
8      }
9      if (s == null || s.length() < 1) {
10         continue;
11     }
12     if (s.charAt(0) == '#') { // comment
13     if (collectComments && s.length() > 1) {
14         c14: comments.add(s.substring(1));
15     }
16     continue;
17 }
```

We also discovered a Denial-of-Service bug from benchmark TextCrunchr that is caused by a collection variable with an excessively large length. TextCrunchr is a text analysis program that is able to perform some useful analysis (word frequency, word length, etc.), as well as process plain text files and compressed files, which it will uncompress to analyze the contents. The vulnerability is in the decompressing functionality where it uses a collection variable `queue` to store files to be decompressed. Our tool QUANTM correctly did not verify the boundedness of variable `queue` and we believe this leads to TextCrunchr's being vulnerable to a Zip bomb attack, because variable `queue` may store an exponential number of files that is caused by a carefully crafted zip file, which contains other carefully crafted zip files inside, thus leading to an exponential number of files to be stored in variable `queue` and to be decompressed.

RQ2: Analysis Speed. On average, it takes 51.9 s to analyze a 15k lines of code benchmark program (including vacuously verified methods) with QUANTM. The detailed results from each benchmark are in Table 1. Given the lines of code of our benchmarks, we believe this result indicates that the speed of our analysis benefits from being type-based and flow-insensitive, exhibiting the potential of scaling to even larger programs.

6.2 Limitations, Future Work and Discussion

In experiments, we encountered collection variables that could not be annotated, leaving QUANTM unable to verify boundedness of methods that declare them. We next categorize the reasons and discuss future work for improvements:

- To ensure soundness, we enforce no constraint on a collection-typed variable's length (i.e. allow it to be infinitely long), when it is the result of reading a list-typed element from a collection. The reason is that the type system does not yet support annotating lengths of inner lists. This extension of our type system is left for future work.
- Not discovering the right global invariants. In the future we plan to automate the invariant discovery process with abstract interpretation, which will hopefully help uncover more invariants.
- Imprecision and "soundiness" caused by Java features such as aliasing, dynamic dispatch, inner class, class inheritance and multi-threading. We regard these as orthogonal problems to our problem statement and we could extend our type system to support them.

Integration with Building Tools. To evaluate how user-friendly QUANTM is for a developer, we also evaluated how QUANTM integrates with open source repositories (i.e. jforum3, jRecruiter and SpringPetClinic) that use popular building tools (e.g. Maven). We discovered that the configuration is reasonably easy: Developers only need to add several Maven dependencies (including QUANTM, Checker framework, Scala library, Scala SMT-LIB and Microsoft Z3's Java bindings) into `pom.xml` and specify QUANTM as an additional annotation processor. After that, a developer could immediately start using our tool!

Annotation Workflow. The typical annotation workflow of a user is to first configure QUANTM as an annotation processor, and then compile the target code/project without any annotations. Note that errors and warnings are expected if QUANTM cannot prove boundedness of a procedure, which is intrinsically caused by insufficient annotations (i.e. type refinements). In the end, a user will fix the errors and warnings by annotating collection variables. In the case of a method returning a locally allocated collection variable, we inlined the method into its caller to ensure soundness. Additionally, to perform interprocedural analysis, we introduce method summaries to describe changes in lengths of collection-typed variables caused by method invocation. Method summaries are expressed in the refinement language defined in Fig. 2b, together with variables in their primed version, which denotes the values after method invocation. Method summaries are automatically applied when type checking a method invocation statement. The annotation burden for method summaries was light (6.8 methods on average) in the experiments.

7 Related Works

Type Systems for Resource Analysis. Type-based approaches have been proposed for resource analysis [9,25,26]. These works verify size relations

between input and output list variables as a function specification. Additionally, there is a line of works that combines a type-based approach with the idea of amortized analysis [18,19] to analyze resource usage. These approaches are not able to analyze programs with mutation and it is also unclear how to adapt them into a setting with mutation. The reason is the need for the analysis to be flow-sensitive in the presence of mutation, because mutation causes program variables' sizes to change. In contrast, we emphasize that it is our novelty to introduce *AST Counter Instrumentation*, making it possible to write flow-insensitive types in the presence of mutation, and thus enjoy the benefits of a type-based approach—being compositional and scalable. We put back the limited flow-sensitive information (in the form of counter axioms) only after the type checking phase.

Resource Analysis by Loop-Bound Analysis. Bound analysis techniques [8, 14,15,17,24,31] emphasize that time bounds (especially loop bounds) are necessary for resource bound analysis and therefore they focus on obtaining loop bounds. However, time boundedness is actually only a sufficient condition for resource boundedness. In contrast, our approach verifies resource bounds even when time bounds are not available. The other difference is that, Gulwani et al. and Zuleger et al.'s works [15,17,31] generate invariants at different program locations, as opposed to our approach of using same invariants at all program locations. In more detail, Carbonneaux et al. [8] use a Hoare logic style inter-procedural reasoning to derive constraints on unknown coefficients of loop bounds, who are in the form of pre-defined templates consisting of multivariate intervals. Gulwani et al. [15] introduce a technique to first transform multi-path loops into loop paths who interleave in an explicit way, and then generate different invariants at different program locations. In another work, Gulwani et al. [17], compute the transitive closure of inner loops, which are invariants only hold true at the beginning of loops. It also utilizes several common loop patterns to obtain ranking functions, which are eventually used to compute loop bounds. Sinn et al. [24] flatten multi-path loops into sets of mutual independent loop paths and uses global lexicographic ranking functions to derive loop bounds. Similarly, Giesl et al. [14] use a standard ranking function approach to obtain loop bounds for its bound analysis, which is a component of its interleaving of size analysis and bound analysis. Zuleger et al. [31] txake size-change abstraction approach from termination analysis domain into bound analysis. Size-change abstraction relates values of variables before and after a loop iteration at the beginning of a loop, which are eventually used to obtain loop bounds. Additionally, although Gulwani et al. [16] also adopt a counter approach by instrumenting loops with counters, the functionalities of counters are different. In our *AST Counter Instrumentation* approach, counters enable writing flow-insensitive global invariants under the scenario of mutation. In contrast, the functionality of counters in Gulwani et al.'s work [16] is to make it explicit if one loop is semantically (instead of syntactically) nested in another loop: each loop is associated with a counter and this work encodes loop nest relations as counter dependencies.

Resource Analysis by Cost-Recurrence Relations. A classical approach to cost analysis [3–7,13] is to derive a set of cost recurrence relations from the original program, whose closed-form solutions will serve as an over-approximation of the cost usage. As pointed out by Alonso et al. [7], one of the limitations in recurrence relation approach is that it poorly supports mutation, because of ignoring the side effects of a callee that may have on its caller. Since collection APIs typically have side effects, recurrence relation may not be the best approach to reason about collection variables' lengths. Additionally, we believe our approach applies to a wider class of programs, because it is more difficult to find closed-form solutions than our approach of checking if a set of constraints implies the desired property.

Refinement Types. Our type system is inspired by Rondon et al. [22]. The subsequent work by Rondon et al. [23] propose a flow-sensitive refinement type system to reason about programs with mutation. Similarly, Coughlin et al.'s work [10] handles mutation via a flow-sensitive approach, allowing type refinements to temporarily break and then get re-established later at some other control locations. It adopts flow-sensitive analysis between control locations who break and re-establish the invariant, respectively. Compared with Rondon et al. [23] and Coughlin et al. [10], our work is different because it separates types and refinements from counter relation axioms, where types and refinements are flow-insensitive but counter relation axioms are flow-sensitive. The advantage of our approach over Coughlin et al.'s work is that, to verify a given property, Coughlin et al.'s work is more expensive because it is sensitive to the distance (in terms of lines of code) between any two relevant control locations (i.e., where the first location breaks an invariant and the second location potentially restores the invariant). More specifically, Coughlin et al.'s work has to perform flow-sensitive and path-sensitive symbolic execution between any two relevant control locations. In comparison, our approach is insensitive to the distance between any two relevant control locations.

8 Conclusion

We proposed a technique that statically verifies the boundedness of total length of collection variables when given constraint(s) on input(s). Our technique is able to verify the above property for non-terminating reactive programs. To ensure scalability, we take a type-based approach and enforce using global inductive invariants, as opposed to different invariants at different program locations. To design global invariants for programs supporting mutation, we introduce AST counters, which track how many times an AST node was executed. We then add axioms on relations of the counter variables. Experimental results demonstrate that our technique is scalable, and effective at verifying bounds.

We plan to build on this work in at least the following two directions: (i) extending from collection lengths to general memory usage, (ii) generalizing the AST counter technique and applying it in different contexts.

References

1. DARPA space-time analysis for cybersecurity program (STAC). http://www.darpa.mil/program/space-time-analysis-for-cybersecurity
2. Scala library for parsing and printing the SMT-LIB format. https://github.com/regb/scala-smtlib
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of Java bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_12
4. Albert, E., Genaim, S., Gomez-Zamalloa, M.: Heap space analysis for Java bytecode. In: Proceedings of the 6th International Symposium on Memory Management, pp. 105–116. ACM (2007)
5. Albert, E., Genaim, S., Gómez-Zamalloa Gil, M.: Live heap space analysis for languages with garbage collection. In: Proceedings of the 2009 International Symposium on Memory Management, pp. 129–138. ACM (2009)
6. Albert, E., Genaim, S., Masud, A.N.: More precise yet widely applicable cost analysis. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 38–53. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_5
7. Alonso-Blas, D.E., Genaim, S.: On the limits of the classical approach to cost analysis. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 405–421. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33125-1_27
8. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. ACM SIGPLAN Not. **50**(6), 467–478 (2015)
9. Chin, W.N., Khoo, S.C.: Calculating sized types. High.-Order Symb. Comput. **14**(2–3), 261–300 (2001)
10. Coughlin, D., Chang, B.Y.E.: Fissile type analysis: modular checking of almost everywhere invariants. ACM SIGPLAN Not. **49**, 73–85 (2014)
11. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
12. Dietl, W., Dietzel, S., Ernst, M.D., Muşlu, K., Schiller, T.W.: Building and using pluggable type-checkers. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 681–690. ACM (2011)
13. Flores-Montoya, A., Hähnle, R.: Resource analysis of complex programs with cost equations. In: Garrigue, J. (ed.) APLAS 2014. LNCS, vol. 8858, pp. 275–295. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12736-1_15
14. Giesl, J., et al.: Proving termination of programs automatically with AProVE. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 184–191. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_13
15. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. ACM SIGPLAN Not. **44**, 375–385 (2009)
16. Gulwani, S., Mehra, K.K., Chilimbi, T.: Speed: precise and efficient static estimation of program computational complexity. ACM SIGPLAN Not. **44**, 127–139 (2009)
17. Gulwani, S., Zuleger, F.: The reachability-bound problem. ACM SIGPLAN Not. **45**, 292–304 (2010)
18. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. ACM SIGPLAN Not. **46**, 357–370 (2011)

19. Hoffmann, J., Das, A., Weng, S.C.: Towards automatic resource bound analysis for OCaml. ACM SIGPLAN Not. **52**, 359–373 (2017)
20. Olivo, O., Dillig, I., Lin, C.: Static detection of asymptotic performance bugs in collection traversals. ACM SIGPLAN Not. **50**, 369–378 (2015)
21. Papi, M.M., Ali, M., Correa Jr., T.L., Perkins, J.H., Ernst, M.D.: Practical pluggable types for Java. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis, pp. 201–212. ACM (2008)
22. Rondon, P.M., Kawaguci, M., Jhala, R.: Liquid types. ACM SIGPLAN Not. **43**, 159–169 (2008)
23. Rondon, P.M., Kawaguchi, M., Jhala, R.: Low-level liquid types. ACM SIGPLAN Not. **45**, 131–144 (2010)
24. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable approach to bound analysis and amortized complexity analysis. In: Computer Aided Verification-26th International Conference (CAV 14), pp. 743–759 (2014)
25. Vasconcelos, P.B.: Space cost analysis using sized types. Ph.D. thesis, University of St Andrews (2008)
26. Vasconcelos, P.B., Hammond, K.: Inferring cost equations for recursive, polymorphic and higher-order functional programs. In: Trinder, P., Michaelson, G.J., Peña, R. (eds.) IFL 2003. LNCS, vol. 3145, pp. 86–101. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27861-0_6
27. Lu, T., Cerný, P., Chang, B.-Y.E., Trivedi, A.: Type-directed bounding of collections in reactive programs. CoRR abs/1810.10443 (2018)
28. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge (1993)
29. Xu, G., Rountev, A.: Precise memory leak detection for Java software using container profiling. In: Proceedings of the 30th International Conference on Software Engineering, pp. 151–160. ACM (2008)
30. Xu, G., Rountev, A.: Detecting inefficiently-used containers to avoid bloat. ACM SIGPLAN Not. **45**, 160–173 (2010)
31. Zuleger, F., Gulwani, S., Sinn, M., Veith, H.: Bound analysis of imperative programs with the size-change abstraction. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 280–297. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_22