Efficient and formal generalized symbolic execution

Xianghua Deng, Jooyong Lee, Robby

Published Version Information

Citation: Deng, X., Lee, J., & Robby. (2012). Efficient and formal generalized symbolic execution. Automated Software Engineering, 19(3), 233-301.

Copyright: © Springer Science+Business Media, LLC 2011

Digital Object Identifier (DOI): doi:10.1007/s10515-011-0089-9

Publisher's Link: http://www.springerlink.com/content/k2wh786791202n43/

Noname manuscript No.
(will be inserted by the editor)

Efficient and Formal Generalized Symbolic Execution

Xianghua Deng · Jooyong Lee · Robby

the date of receipt and acceptance should be inserted later

Abstract Programs that manipulate dynamic heap objects are difficult to analyze due to issues like aliasing. Lazy initialization algorithm enables the classical symbolic execution to handle such programs. Despite its successes, there are two unresolved issues: (1) inefficiency; (2) lack of formal study. For the inefficiency issue, we have proposed two improved algorithms that give significant analysis time reduction over the original lazy initialization algorithm. In this article, we formalize the lazy initialization algorithm and the improved algorithms as operational semantics of a core subset of the Java Virtual Machine (JVM) instructions, and prove that all algorithms are relatively sound and complete with respect to the JVM concrete semantics. Finally, we conduct a set of extensive experiments that compare the three algorithms and demonstrate the efficiency of the improved algorithms.

Keywords Symbolic Execution, Operational Semantics, JVM, Soundness, Completeness

1 Introduction

Programs that manipulate dynamic heap objects (PMDHO) are notoriously difficult to reason due to issues such as aliasing (Ramalingam 1994). In recent years, symbolic execution (King 1976) has regained interests in checking and testing such programs. The lazy initialization algorithm (Khurshid et al 2003) is one of the prominent algorithmic contributions that enable symbolic execution to handle PMDHO. The lazy initialization algorithm essentially maintains a graphical representation of the heap in the program and only materializes the parts of the heap as needed, similarly to TVLA (Lev-Ami and Sagiv 2000). One advantage of the lazy initialization approach is that it can be used to check complex

Xianghua Deng
Pennsylvania State University at Harrisburg (current affiliation: Google Inc.)
E-mail: wdeng@google.com

Jooyong Lee
Korea University
E-mail: jlee@formal.korea.ac.kr

Robby
Kansas State University
E-mail: robby@ksu.edu

assertions of heap structures automatically and without requiring specific abstractions that users have to provide (in contrast to, e.g., instrumentation predicates in TVLA).

Despite its successes in checking properties and generating test cases for PMDHO (Visser et al 2004; Păsăreanu and Visser 2004), there are two important issues regarding the previous work on the lazy initialization: (1) the algorithm is not efficient as it performs exhaustive case splitting on object aliasing cases eagerly; (2) a formal study on the soundness and completeness of the algorithm was not performed.

To address the inefficiency issue, we have proposed two improved lazy initialization algorithms that give orders of magnitude reductions in analysis time (Deng et al 2006, 2007b). In this paper, we give general descriptions of the improved algorithms as well as the formalization of the three lazy initialization algorithms as alternative operational semantics of a core subset of Java Virtual Machine (JVM) instructions. More specifically, the main contributions of this paper are:

- Both informal and formal descriptions of symbolic execution algorithms over heap manipulating programs.[1] We have described the three symbolic execution algorithms which incorporate type variables to address an issue in the original lazy initialization algorithm (Khurshid et al 2003) with respect to subtyping. In addition, for each algorithm, we provide an alternative operational semantics over a core subset of JVM instructions. Based on the formal description, we prove the relative soundness and completeness of the symbolic operational semantics in relation to the JVM concrete operational semantics. (The definition of each relativeness is provided in Section 4.3.) We believe that it is suitable to use the formal description as a reference point when comparing symbolic execution techniques over Java-like programs. We chose to use JVM and its instruction set instead of developing yet another language because (1) it is realistic; (2) the semantics of each instruction is relatively simple; (3) it contains various language features relevant for the algorithms (i.e., dynamic creation of objects, type hierarchy, and subtyping); (4) it serves as a blueprint that guided our implementation that analyzes Java bytecode.
- Kiasan[2] – a next generation implementation of the symbolic operational semantics that is more robust and flexible regarding the choices of bound strategies and underlying decision procedures. Given the close relation between the implementation and the semantics, we believe that Kiasan is robust enough that it can serve as a research vehicle for experimentation and conducting case studies.
- Extensive experimental studies that include checking structural invariants of common data structures and most containers in the `java.util` package, as well as some functional properties of these data structures and algorithms. The experiments compare the performance of the lazy initialization algorithm with the two improved algorithms and provide a benchmark for other analysis techniques.

Organization: The rest of the article is organized as follows. Section 2 describes some background information about basic symbolic execution and the lazy initialization algorithm for handling PMDHO. Section 3 describes three vertically integrated symbolic execution algorithms, namely, lazy, lazier, and lazier# initializations. Section 4 formalizes each of the three algorithms as operational semantics of a core subset of the JVM instructions and proves that these algorithms are relatively sound and complete. Section 5 describes two experimental studies that compare the performance of the three algorithms and provide a

[1] This paper supersedes the preliminary versions shown in our previous work (Deng et al 2006, 2007b).

[2] Kiasan means to reason with analogy or symbolically in Indonesian.

```
1 int abs(int x) {
2   if (x < 0)
3     x = -x;
4   if (x < 0)
5     assert false;
6   return x;
7 }
```

$$\begin{tikzcd}
& & \langle\alpha,T\rangle \arrow[dl, "2{,}T"'] \arrow[drr, "2{,}F"] & & & \\
& \langle\alpha,\alpha<0\rangle \arrow[d, "3"] & & & \langle\alpha,\alpha\geq 0\rangle \arrow[dl, "4{,}T"'] \arrow[dr, "4{,}F"] & \\
& \langle\beta,\alpha<0\wedge\beta=-\alpha\rangle \arrow[dl, "4{,}T"'] \arrow[dr, "4{,}F"] & & \langle\alpha,F\rangle & & \langle\alpha,\alpha\geq 0\rangle \\
\langle\beta,F\rangle & & \langle\beta,\alpha<0\wedge\beta=-\alpha\rangle & & &
\end{tikzcd}$$

Fig. 1 A Symbolic Execution Example

benchmark for other software analysis tools on a standard set of Java examples. Section 6 describes related work, and Section 7 concludes.

2 Background

2.1 Symbolic Execution

King (1976) proposed symbolic execution (SymExe) as a technique for program testing and debugging. One key advantage of SymExe over concrete execution (e.g., traditional testing) is that it can reason about unknown values which are represented as *symbols* (or *symbolic values*) instead of concrete values (e.g., integers).

Figure 1 illustrates the symbolic computation tree of the example method `abs`; each tree node is a symbolic state $\langle \mathtt{x}, \phi \rangle$ consisting of a symbol or a concrete value associated with `x`, and predicate ϕ to constrain the symbol for `x`. When symbolically executing `abs` with no initial information about its argument, the initial state for the `abs` method has a symbol α for `x` and the constraint ϕ set to TRUE (no constraint imposed yet). When executing line 2, the algorithm does not have sufficient information to decide which branch to take because $\phi \wedge (\mathtt{x} < 0)$ (i.e., the condition for the *true* branch) and $\phi \wedge \neg(\mathtt{x} < 0)$ (i.e., the condition for the *false* branch) are all satisfiable under the current symbolic state – thus, both branches are explored.

As each branch is traversed, the constraint is augmented with a predicate corresponding to the logical condition that would have caused the particular branch to be followed. Thus, the constraint ϕ is often referred to as *path condition* because it characterizes the conditions on variables that would be necessary for execution to flow down a particular path. If a path condition becomes FALSE, this means that the symbolic trace is infeasible (i.e., there is no corresponding concrete trace); thus, the trace does not provide useful information for analyzing the behavior of the program. Hence, such paths should be discarded. The symbolic computation tree shows that the *true* branch of line 4 is always infeasible.

Role of decision procedures: Decision procedures are usually employed to determine which branches to follow. As mentioned previously, as soon as a path condition becomes unsatisfiable, it is safe to abandon the corresponding trace. This reduces the number of paths that have to be executed (analyzed) by SymExe, thus reducing the analysis' time cost. In the same spirit, as soon as a path is found to violate some property (e.g., assertion), the error can directly be reported, and SymExe can start exploring one of the remaining paths, if any.

Termination: One of the major issues with SymExe is termination. Since SymExe does not merge state information at program joint points after branches and loops, the analysis may not terminate when the program being checked contains loops or recursions, unless inductive

Algorithm 1: Lazy Initialization Algorithm (an Excerpt of Khurshid et al (2003))

```
1  if f is uninitialized then
2      if f is reference field of type T then
3          nondeterministically initialize f to
4              1. null
5              2. a new object of class T (with uninitialized field values)
6              3. an object created during a prior initialization of a field of type T
7          if method precondition is violated then
8              backtrack()
9      if f is primitive (or string) field then
10         initialize f to a new symbolic value of appropriate type
```

predicates such as loop invariants are provided at these loops and recursive points, as shown by Hantler and King (1976).

Bounding is usually employed to work around this problem. There are a variety of bounding mechanisms that have been used in the literature, such as loop bounding, depth bounding (i.e., limiting the number of execution steps), bounding on the length of method call chains, etc. The use of these bounding mechanisms leads to an underapproximation of program behavior (i.e., unsound in general). Moreover, it does not produce a conclusive analysis report when no bugs are found because it is hard to characterize the program behavior that has been analyzed. SymExe with bounding is, however, still useful because it can easily check sophisticated properties that cannot be checked by unbounded methods, and no false alarm is produced by bounding.

Path explosion: Another major issue with SymExe is the path explosion problem. Since SymExe splits paths on branch points and never merges them back, the number of paths that SymExe explores may grow exponentially. This introduces scalability issues when applying SymExe. Similarly to the termination issue, bounding mechanisms are used to cope with this problem. On the other hand, due to the aggressive path exploration, SymExe often achieves a high level of branch and code coverage.

2.2 Lazy Initialization Algorithm

The basic SymExe algorithm described so far can only analyze programs with scalar data. The lazy initialization algorithm (Khurshid et al 2003) is a graph-based technique that enables SymExe to handle dynamic heap.

Algorithm 1 is an excerpt of Khurshid et al (2003) that illustrates the lazy initialization algorithm. Intuitively, lazy initialization works in the same spirit as with SymExe. That is, it starts with no knowledge of the heap structure, and it discovers the heap structure as it symbolically executes a given program. Unknown object values are represented by special symbols. As the program executes and accesses object fields, it "discovers" (i.e., *materializes*) the field values on an on-demand basis (hence the term "lazy initialization"). When an unmaterialized field is read, if the field's type is a scalar type (Lines 9-10), then a fresh symbol is created for that scalar value. Otherwise, for an unmaterialized reference field (Lines 2-8), the algorithm systematically explores all possible points-to relationships by nondeterministically choosing among the following values for the reference: (a) NULL (Line

Program 1 A Swap Example

```
public class Container<E> {
  private E data;

  //@ ensures data == \old(n.data) && n.data == \old(data);
  public void swap(/*@ non_null @*/ Container<E> n)
  { E e = data; data = n.data; n.data = e; }
}
```

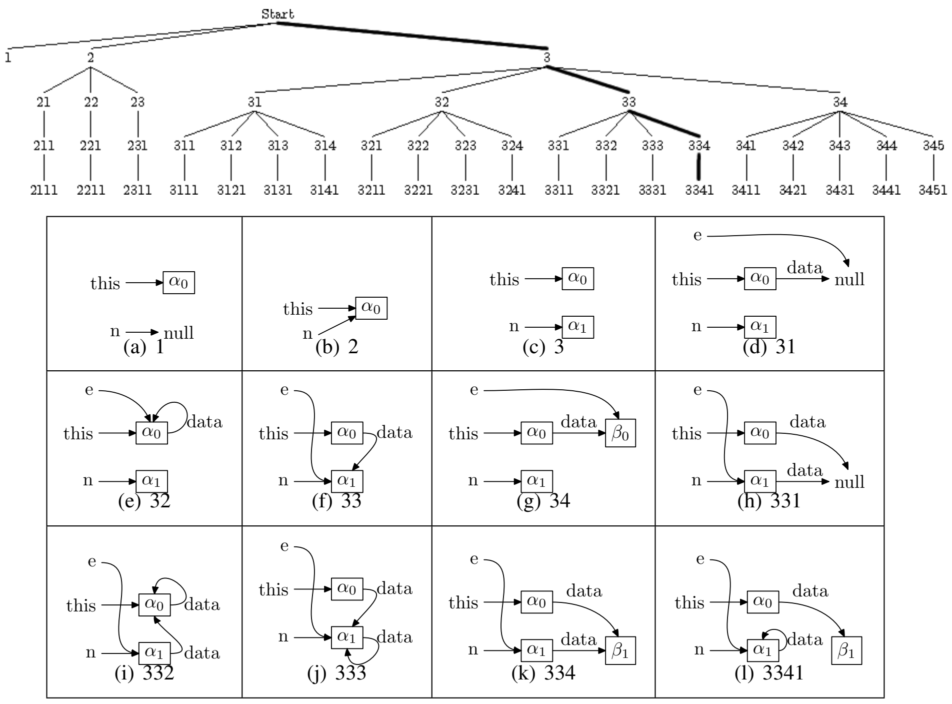

Fig. 2 Lazy Symbolic Computation Tree of the `swap` Example and Heap Configurations of An Example Trace (3-33-334-3341 and Sibling States)

4), (b) a newly materialized object (Line 5), or (c) any materialized object (Line 6). Once objects are materialized, destructive updates are done similarly as in the concrete execution.

To illustrate lazy initialization, consider the `swap` method of class `Container` in Program 1. We use JML contracts (Leavens et al 1998) to express the precondition and postcondition of `swap`. That is, the precondition specifies that `n` is non-NULL, and the postcondition specifies that the `data` values of `this` and `n` have been properly swapped. We use the classical `assume` and `assert` statements to support the checking of pre- and postconditions. Essentially, we `assume` the precondition before the method is executed and `assert` the postcondition after the method. For the example, the `swap` contract is transformed into `assume(n != NULL); swap(n); assert(data==\old(n.data) && n.data==\old(data));`. While in-depth discussion on supporting JML checking (e.g., `\old`) in SymExe is interesting, it is out of the scope of this paper. Instead, we focus on discussing the essence of lazy initialization through the `swap` example.

The top part of Figure 2 illustrates the symbolic computation tree built using lazy initialization. To save space in the display of the tree, we represent each tree node (i.e., system state) by a unique label. The bottom part of Figure 2 shows heap configurations for some of the states in the computation tree.

To generate the computation tree of Figure 2, SymExe begins with a nondeterministic choice of possible aliasing between the method parameter `n` and `this` reference (i.e., states 1, 2, and 3). Note that both the `next` and the `data` fields of `this` and `n` are unknown (i.e., unmaterialized) at these states. Out of the three cases, state 1 does not satisfy the `non_null` precondition for `n`; thus, it is not considered further. Now, consider the subtree starting from state 3. Upon executing `swap`'s first statement (i.e., `E e = data;`), the `this.data` field is now materialized according to the lazy initialization algorithm described earlier; it nondeterministically chooses the value of `this.data` to be: NULL (state 31), equal to `this` i.e., α_0 (state 32), α_1 (state 33), or a fresh symbolic object β_0 (state 34). Let us continue on with state 33. Upon executing `swap`'s second statement (i.e., `data = n.data;`), the algorithm nondeterministically chooses the value of `n.data` to be: NULL (state 331), α_0 (state 332), α_1 (state 333), or a fresh symbolic object β_1 (state 334). Executing `swap`'s last statement (i.e., `n.data = e;`) at state 334 produces state 3341.

We have illustrated how the lazy initialization algorithm symbolically executes the `swap` example following Figure 2's highlighted trace (i.e., trace 3-33-334-3341). Notice that the final state 3341 satisfies the method's postcondition since the old[3] value of `this.data` and `n.data` are α_1 and β_1 (respectively) in states 33 and 334. Similarly, we can show that `swap`'s postcondition is satisfied by all final states of the symbolic computation tree. Since the computation tree characterizes all possible concrete executions of `swap`, we can conclude that the postcondition always holds.

Role of decision procedures: Lazy initialization handles heap object structures directly as graphs, similarly to the heap representations in most explicit state model checkers for object-oriented programs (Brat et al 2000; Robby et al 2003). Thus, decision procedures are not used for heap objects. That is, decision procedures are used only for scalar values as in basic SymExe. From a different point of view, one can consider the lazy initialization algorithm as a decision procedure for object structures with case splitting on possible aliasing scenarios. The algorithm allows direct control over heap objects allowing one to make various observations programmatically, while such API to internal states of decision procedures are usually not provided (e.g., attaching monitors for symbolic heap structures are similar to attaching monitors for concrete heap structures usually used in testing).

Termination: Lazy initialization may not terminate because it can choose to always materialize a new symbolic object; thus, it keeps expanding the heap. As with the basic SymExe, one can use bounding mechanisms to limit the heap expansion. In addition to bounding mechanisms described in the previous section, Khurshid et al (2003) used bounding on the number of objects that can be materialized for each object type. Similar bounding mechanisms were used in Korat (Boyapati et al 2002) and Alloy (Jackson 2002).

Another approach is to summarize the heap structures at join points (Anand et al 2006). While promising, the approach can only work on a predefined set of object structures. In general, it is hard to come up with an automatic and precise heap abstraction mechanism that works for arbitrary kinds of complex heap properties that a user might want to check.

[3] A general mechanism to evaluate `\old` expressions is described in our previous work (Deng et al 2007a).

Path explosion: Lazy initialization potentially contributes to the path explosion problem; in the worst case, the number of paths is exponential with respect to the size of the heap. Bounding can be used to cope with the problem.

3 Symbolic Execution Algorithms in Kiasan

In this section, we will explain the intuition of the three vertically integrated symbolic execution algorithms in Kiasan. The formalization of the three algorithms is presented in Section 4. The first algorithm is essentially a modified version of the lazy initialization algorithm presented by Khurshid et al (2003). The second algorithm, called lazier initialization, significantly improves upon the lazy initialization algorithm. More specifically, it reduces the size of a symbolic computation tree by introducing an object abstraction. The third algorithm, lazier# initialization, improves upon the lazier initialization algorithm by introducing another object abstraction.

3.1 Kiasan's Lazy Initialization Algorithm

Kiasan's lazy initialization algorithm is adapted from the lazy initialization algorithm described by Khurshid et al (2003); it adds support for handling subtyping, i.e., a symbolic object of declared type T can be an object of any subtype of T. To understand the implications of the issue in SymExe and lazy initialization in particular, let us examine Program 2.

Suppose that method `isNext` (Lines 4-7) is being analyzed by the lazy initialization algorithm. At line 5, there are two symbolic objects: one is `this` of type `Node` and the other is `node` of type `ExtendedNode`. At line 6, because `this.next` has not been accessed before (i.e., uninitialized), a lazy initialization will occur. If method `isNext` was called at line 16, `this.next` would point to `en` of type `ExtendedNode`. Therefore, the lazy initialization at line 6 should include `node` of type ExtendedNode in the choosing range. This observation can be generalized as follows. Observation 1: the choosing range of a lazy initialization of a field of type T should include objects of subtypes of T (including T).

Further, suppose that method `isNextObject` in Program 2 is being analyzed by the lazy initialization algorithm. At line 9, there are two symbolic objects, i.e., one for `this` and the other for parameter `node` of type `Object`. At line 10, a lazy initialization takes place for `this.next`. It is easy to observe that `this.next` should be able to point to `node` of type `Object`, given such a call at line 17. This observation can be generalized as follows. Observation 2: the choosing range of the lazy initialization of a field of type T should include symbolic objects of supertypes of T since a symbolic object of supertype of T may represent any concrete object of type T or its subtype.

The two observations are further complicated by run-time type query operations. For example, consider method `nextObjectTypeHierarchy` (Lines 19-22) in Program 2. The lazy initialization of `this.next` at line 21 should not include parameter `node` in the choosing range because it is infeasible by taking into account the type query at line 20 (i.e., the `instanceof`[4] operation). Therefore, the type query operation may limit the choosing range of a lazy initialization by introducing type constraints. Hence, after taken into account the type constraints, the two observations can be summarized as follows: the choosing range of the lazy initialization of a field with type T should include symbolic objects of type T

[4] In Java, `obj instanceof` T returns TRUE if the type of `obj` is T or subtype of T. And the formal semantics of `instanceof` is described in Section 4.

Program 2 An example demonstrating subtyping issue in lazy initialization algorithm

```
1   class Node {
2       Node next;
3       int data;
4       //@requires node != null;
5       public boolean isNext(ExtendedNode node) {
6           return this.next == node;
7       }
8       //@requires node != null;
9       public boolean isNextObject(Object node) {
10          return this.next == node;
11      }
12      void foo() {
13          Node n1 = new Node();
14          ExtendedNode en = new ExtendedNode();
15          n1.next = en;
16          assert (n1.isNext(en));
17          assert (n1.isNextObject(en));
18      }
19      void nextObjectTypeHierarchy(Object node) {
20          if (!(node instanceof Node))
21              assert(this.next != node);
22      }
23  }
24  class ExtendedNode extends Node { }
```

and supertypes of T and subtypes of T without violating the type constraints introduced by operations such as `instanceof`.

The original lazy initialization algorithm can handle the first observation (without considering the type constraints) very well as it initializes an unknown reference type field of type T to any object with type T or its subtype. However, it did not consider the second observation and complication of type query operation (e.g., `instanceof`). [5]

To address this issue, we propose that each symbolic object carries a type variable (i.e., a symbolic type). The type variables are constrained by the rules of the language's type system. For each lazy initialization of a reference type, the modified lazy initialization algorithm checks for compatibility of types in a similar way to the classical SymExe algorithm. That is, the algorithm maintains type constraints as well as constraints on primitive types in the path condition.

To illustrate the type variable approach, let us revisit Program 2. When checking method `isNext` in Program 2 at line 5, there are two symbolic objects: `this` with type τ_1 and `node` with type τ_2 where τ_1 and τ_2 are type variables. The path condition ϕ is $\tau_1 <: \texttt{Node} \wedge \tau_2 <: \texttt{ExtendedNode}$, where $<:$ is the subtype relation. When executing line 6, since `this.next` is not initialized, a lazy initialization is initiated. To check whether `this.next`, whose declared type is `Node`, can point to `node` whose actual type is τ_2, we need to check whether $\tau_2 <: \texttt{Node}$ is compatible with the path condition. And in fact, it is implied by the path condition because of $\tau_2 <: \texttt{ExtendedNode} <: \texttt{Node}$ and transitivity of the subtype relation. Similarly, we can analyze method `isNextObject` correctly. At line 9, there are two same symbolic objects as before, whereas the path condition is different: $\phi = \tau_1 <: \texttt{Node} \wedge \tau_2 <: \texttt{Object}$. At line 10, `this.next` can point to `node` since $\tau_2 <: \texttt{Node}$ is satisfiable under the current path condition. If the path of `this.next` initialized to `node` is chosen, then $\tau_2 <: \texttt{Node}$ is added into the path

[5] It is possible to resolve the issue by performing a nondeterministic choice on the type of the object when a parameter of reference type is first read. However, this solution does not work for any extensible type hierarchy and furthermore is inefficient.

condition. Finally, when analyzing method `nextObjectTypeHierarchy`, at line 21, the path condition contains $\neg(\tau_2 <: \texttt{Node})$ where τ_2 represents the symbolic type of parameter `node` as before. Therefore, the condition for `this.next` to point to `node`, $\tau_2 <: \texttt{Node}$, contradicts the path condition, and this case is excluded as desired.

In summary, the subtyping issue is handled by the introduction of type variables (and constraints over type variables and types) for symbolic objects.

Handling Arrays: Arrays present a unique challenge: the length of an array may be unknown. In addition, arrays can be accessed by a symbolic integer index. To address these issues, we model each array as an accumulator of indexes that have been accessed and their corresponding values. Initially, the accumulator is empty. If the array is accessed with an index *i* whose value can be either a concrete or symbolic integer, then *i* is compared with the already accumulated indexes: if *i* is equal to one of them, its corresponding value is returned as the array-element value; otherwise, a fresh symbolic value is returned after being associated with *i*. Similarly to the lazy initialization of fields, this cuts down the number of paths that have to be explored. More specifically, instead of comparing the index being accessed with all the indexes of the array (which can only be tractably done if the length of the array is bounded), we only compare the index with what have been accessed.

The accumulator approach can be seen as an efficient procedure for implementing the basic array theory (McCarthy 1962). Our goal is to have an algorithm that can handle arrays in symbolic execution similarly to the lazy initialization algorithm which deals with objects in symbolic execution. In addition, the accumulator approach incorporates practical issues such as default values and boundings.

Initial States: Given a method (without loss of generality, we assume it is an instance method), the components of initial states are initialized as follows: (1) each primitive global and parameter is initialized to a fresh primitive symbol; (2) `this` is initialized to a fresh symbol; (3) each nonprimitive global and parameter other than `this` is nondeterministically initialized to NULL, a fresh symbol, or any other symbol of a compatible type. Figure 2 shows three initial states (i.e., states 1, 2, and 3) satisfying the above conditions.

Roles of decision procedures: Since the path condition is enriched with constraints over type variables, decision procedures that can handle type constraints are required. Subtyping relationships can be modeled by an uninterpreted function whose reflexive, antisymmetric, transitive properties are established via axioms with quantifications. Alternatively, a decision procedure for partially ordered sets can be used to solve these constraints.

In Section 4, we establish that the modified lazy initialization algorithm is sound and complete with respect to the concrete execution.

3.2 Lazier Initialization Algorithm

As could be observed in Figure 2, the lazy initialization algorithm may easily produce a rather large computation tree even for a relatively simple method. In our previous paper (Deng et al 2006), we introduced an optimized algorithm dubbed *lazier initialization* based on the observation that when an uninitialized reference type field is first read, it is often unnecessary to know which object the field refers to, and only a notion of object existence is enough. For example, when symbolically executing statement `return o.f == null;`, it is enough to know whether `o.f` is NULL (or not) without having to precisely know which object it refers to.

Algorithm 2: read(State s, Value o, Field f) : Value

```
if o is a symbolic object and f is uninitialized then
    if f is reference type T then
        nondeterministically initialize f to
          1. null
          2. a new symbolic location of type T
    if f is primitive field of type T then
        $o.f \leftarrow$ a new symbolic value of type T
if o is a symbolic location then
    $o' \leftarrow$ nondeterministically choose among
      1. a new symbolic object with all fields undefined
      2. an existing symbolic object with a compatible type
    return read($s[o/o']$, $o'$, $f$)
return $o.f$
```

Based on the observation, we introduce an abstract value which we call a *symbolic (heap) location*. A symbolic location represents a set of all the type-compatible objects in the heap. We denote a symbolic location by a ˆ-accented symbol. Thus, we now have two abstraction levels of objects available for the lazier initialization algorithm, i.e., symbolic objects and symbolic locations.

Algorithm 2 shows how lazier initialization makes use of the new abstraction. When an uninitialized reference type field of a symbolic object is read (Lines 2-5), the field is lazier-ly initialized to either NULL or a fresh symbolic location whose type is set to be the same as the field's type. Primitive type fields are, on the other hand, handled the same way as in the lazy initialization algorithm (Lines 6-7). In the case where the receiver is a symbolic location (Lines 8-12), first, this symbolic location is nondeterministically substituted by either one of existing type-compatible objects or a fresh symbolic object. Then, the algorithm is called recursively to trigger lazy initialization.

The effects of the lazier initialization are: (1) delaying the nondeterministic selection of objects in the lazy initialization algorithm, and (2) object selection may not be needed in some cases. Both effects contribute to smaller sizes of computation trees and therefore provide significant performance gains in practice, as shown by the experimental data in Section 5.

Initial States: Given a method (without loss of generality, we assume it is an instance method), the components of initial states are initialized as follows: (1) each primitive global and parameter is initialized to a fresh primitive symbol; (2) each nonprimitive global and parameter other than `this` is nondeterministically initialized to NULL or a fresh symbolic location; (3) the implicit parameter `this` is initialized to a fresh symbolic location.

Example: To illustrate the lazier initialization algorithm, let us reconsider the `swap` example in Program 1. The top portion of Figure 3 illustrates the symbolic computation tree that can be obtained by the lazier initialization. The highlighted path of the tree corresponds to its counterpart of the lazy computation tree, shown in Figure 2.

Similarly to the lazy initialization algorithm, the lazier initialization algorithm starts with a nondeterministic choice. Note that, however, there is one less state than in the lazy initialization. This is because two initial states, states 2 and 3 of Figure 2, are abstracted into state 2 of Figure 3.

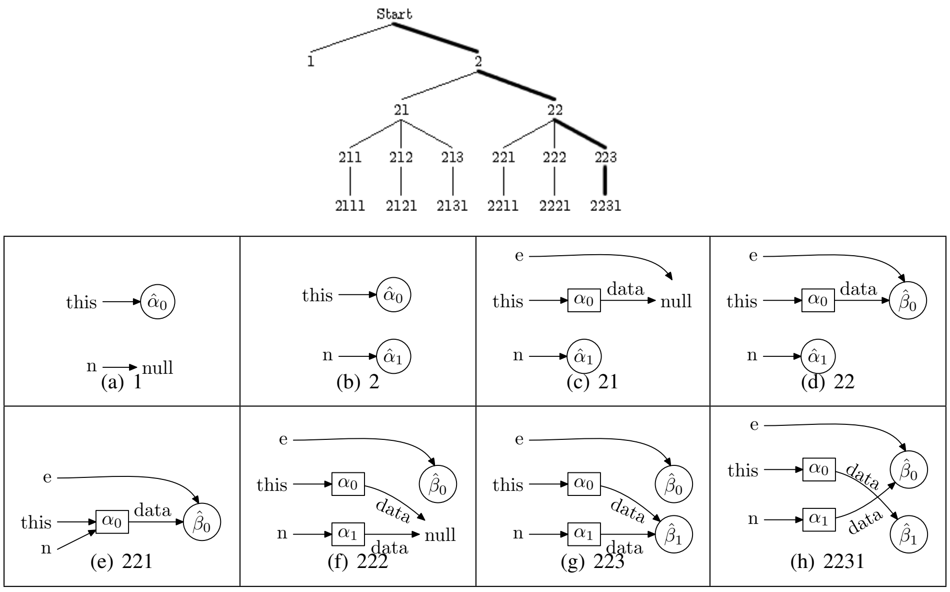

Fig. 3 Lazier Symbolic Computation Tree of the `swap` Example and Heap Configurations of An Example Trace (2-22-223-2231 and Sibling States)

When $\hat{\alpha}_0$'s data field is read at `swap`'s first statement (i.e., `E e = this.data;`), $\hat{\alpha}_0$ is replaced with a fresh symbolic object α_0 because there is no symbolic object yet in the heap. In addition, α_0's `data` field is initialized to either NULL (state 21) or a fresh symbolic location $\hat{\beta}_0$ (state 22). At state 22, there are three possible choices when executing `swap`'s second statement (i.e., `this.data = n.data;`). Symbolic location $\hat{\alpha}_1$, referred to by `n`, can be replaced with either the only existing symbolic object α_0 or a fresh symbolic object α_1. In the former case, the `data` field has already been initialized (state 221). In the latter case, α_1's data field is "lazier-ly" initialized to either NULL (state 222) or a fresh symbolic location $\hat{\beta}_1$ (state 223). Executing `swap`'s last statement (i.e., `n.data = e;`) at state 223 produces state 2231. Notice that state 2231 in Figure 3 safely approximates state 3341 in Figure 2.

As can be observed, the computation tree in Figure 3 is considerably smaller than the one in Figure 2. As mentioned earlier, this is because the nondeterministic object selection is delayed, and the selection sometimes is omitted, as is the case of `this.data` and `n.data`. Moreover, it can also be observed that all the poststates still satisfy `swap`'s postcondition (i.e., `this.data==\old(n.data) && n.data==\old(this.data)`).

In Section 4, we establish that the lazier initialization algorithm is sound and complete with respect to the lazy initialization and the concrete execution.

3.3 Lazier# Initialization Algorithm

We further observed that the size of a lazier computation tree can be reduced in some cases. For example, in the lazier computation tree (Figure 3) of the `swap` example, the distinction between states 21 and 22 is unnecessary because the fact whether or not `this.data` is NULL does not affect the validity of the given postcondition. In general, the lazier initialization algorithm initializes an uninitialized reference type location to either NULL or a fresh

Algorithm 3: read(State s, Value o, Field f) : Value

```
1  if o is a symbolic object and f is uninitialized then
2      if f is reference type T then
3          $o.f \leftarrow$ a fresh symbolic reference of type T
4      if f is primitive field of type T then
5          $o.f \leftarrow$ a fresh symbolic value of type T
6  if o is a symbolic location then
7      $o'' \leftarrow$ nondeterministically choose among
8          1. a fresh symbolic object with all fields undefined
9          2. an existing symbolic object of a compatible type
10     return read($s[o/o'']$, $o''$, $f$)
11 if o is a symbolic reference then
12     $o' \leftarrow$ nondeterministically choose between
13         1. a fresh symbolic location
14         2. NULL
15     return read($s[o/o']$, $o'$, $f$)
16 return $o.f$
```

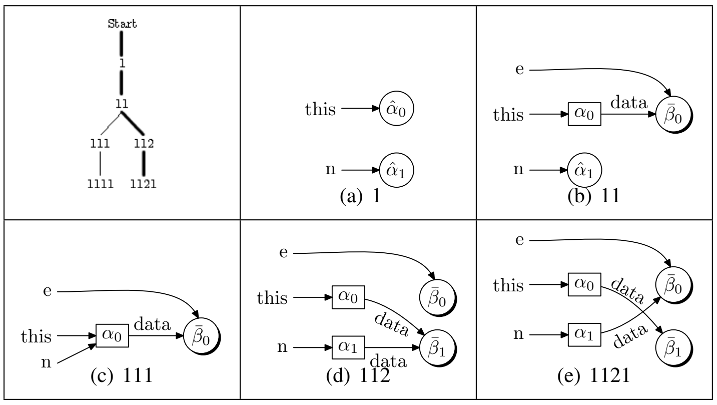

Fig. 4 Lazier# Symbolic Computation Tree of the `swap` Example and Heap Configurations of An Example Trace (1-11-112-1121 and Sibling States)

symbolic location upon access. It is, however, more efficient to defer the nullity decision until this information is necessary (unless non-nullness is assumed by default as in JML). Based on this observation, we developed an even lazier algorithm that we named *lazier# initialization* (Deng et al 2007b).

In the lazier# initialization algorithm, we introduce one more abstract value, *symbolic reference*. A symbolic reference represents a set consisting of NULL and all the type-compatible objects in the heap. Recall that a symbolic location introduced previously does not represent NULL. We denote a symbolic reference by a ¯-accented symbol to distinguish it from a symbolic location denoted by a ˆ-accented symbol. Overall, we use the following three abstraction levels of objects for the lazier# initialization algorithm: (1) symbolic objects as the lowest (finest) level of abstraction, (2) symbolic locations, and (3) symbolic references as the highest (coarsest) level of abstraction.

The lazier# initialization algorithm shown in Algorithm 3 distinguishes three different cases depending on how far the abstraction is progressed. For case 1 (Lines 1-5) where the receiver is a symbolic object and the field f is uninitialized, if the field type is of reference, the field is lazier#-ly initialized to a symbolic reference. Notice that the nondeterministic selection between NULL and a non-NULL value does not take place any more. For case 2 (Lines 6-10) where the receiver is a symbolic location, the receiver is replaced with a symbolic object. As in the case of the lazier initialization, this symbolic object substitute is chosen nondeterministically among a fresh symbolic object and the type-compatible symbolic objects existing in the heap. Then the algorithm proceeds recursively with the substitute. For case 3 (Lines 11-15) where the receiver is a symbolic reference, the receiver is replaced with either a fresh symbolic location or NULL. As case 2, the algorithm proceeds recursively with the substitute. Note that this case can be further optimized by omitting the NULL case if the receiver is known to be non-NULL.

Initial States: Given a method (without loss of generality, we assume it is an instance method), the components of initial states are initialized as follows: (1) each primitive global and parameter is initialized to a fresh primitive symbol; (2) each nonprimitive global and parameter other than `this` is initialized to a fresh symbolic reference or a fresh symbolic location if it is known to be non-NULL; (3) the implicit parameter `this` is initialized to a fresh symbolic location.

Example: To illustrate the lazier# initialization algorithm, let us revisit the `swap` example. The top left corner of Figure 4 illustrates the symbolic computation tree that can be obtained by the lazier# initialization. The highlighted path of the tree corresponds to its counterparts shown earlier in Figures 2 and 3.

The algorithm starts with a single state (i.e., state 1) where `this` and `n` refer to distinct symbolic locations reflecting the fact that `n` is specified as non-NULL in Program 1, and `this` must always be non-NULL.

When $\hat{\alpha}_0$'s data field is read at `swap`'s first statement (i.e., `E e = this.data;`), $\hat{\alpha}_0$ is replaced with a fresh symbolic object α_0, and its `data` field is initialized to a fresh symbolic reference $\bar{\beta}_0$ (state 11). On executing `swap`'s second statement (i.e., `this.data = n.data;`), $\hat{\alpha}_1$ is replaced with either the solely existing symbolic object α_0 or a fresh symbolic object α_1. In the former case, α_0's `data` field has already been initialized (state 111). In the latter case, α_1's `data` field is initialized to a fresh symbolic reference $\bar{\beta}_1$ (state 112). Executing `swap`'s last statement (i.e., `n.data = e;`) at state 112 produces state 1121.

It can be observed in Figure 4 that the states are more abstract compared to the previous two algorithms. It can also be checked that all the post-states still satisfy `swap`'s postcondition (i.e., `this.data==\old(n.data) && n.data==\old(this.data)`).

As with the lazy and lazier initialization algorithms, the lazier# initialization algorithm is also sound and complete with respect to the concrete execution, as described in Section 4.

Optimality: Our experimental data in Section 5 confirms that the lazier# algorithm is significantly faster than the lazier algorithm when analyzing complex data structures. Furthermore, by using the case counting analysis (Deng et al 2010), we demonstrated that the lazier# algorithm is case-optimal with respect to nonisomorphic cases of several complex data structures (Deng et al 2010). That is, it does not generate heap shapes that are overly concrete and overly abstract—the number of the nonisomorphic cases of heap configurations that the algorithm generates matches exactly the number of cases produced by using the case counting analysis technique.

3.4 Bounding Strategies in Kiasan

To address the termination and path explosion issues, we incorporate two bounding techniques to help curb SymExe's complexity. The first technique is k-bounding, which bounds the length of each sequence of lazy/lazier/lazier# initializations originating from each initial symbolic object up to k. In other words, we bound the length of materialization (reference) chains on symbolic objects. For arrays, we additionally bound the number of lazy initializations on distinct array indexes up to k. The second bounding technique is n-bounding, which bounds the number of objects of each (instantiable) type up to n.

These two user-adjustable bounding strategies provide a fair trade-off between analysis cost and behavioral coverage; in contrast to previously discussed bounding strategies, we can quantify the amount of coverage on heap objects for a given k/n bound. That is, when using a bound k/n, the analysis can guarantee the correctness of a program on any heap object configuration (satisfying its contract) with reference chains whose lengths are at most k or the number of objects of each type at most n. In the case where the analysis does not exhaust k/n, a complete behavior coverage is guaranteed (i.e., fully sound).

The two bounding techniques can be combined, for example, the length of reference chains up to 3 and the number of object of `java.lang.Object` up to 4. We can view the combined bounding technique as a pair (k,n) which bounds the lengths of reference chains up to k and the numbers of objects of each type up to n. The k-bound and n-bound can be seen as special cases of the combined bounding technique, more specifically, as $(k,+\infty)$ and $(+\infty,n)$, respectively.

To handle diverging loops, we limit the number of loop iterations (loop-bound) *that do not (lazily) initialize any heap object*, i.e., we prefer exhausting the k or n-bound first before resorting to loop bound to try to guarantee the advertised heap object configuration coverage. Similarly, we also limit the length of method call chain to handle recursions.

3.5 Interprocedural and Modular Analysis in Kiasan

In the case where the program unit N being analyzed invokes a method M, there are two general approaches that are offered in Kiasan: (1) invoke M similarly to a regular concrete program execution (*interprocedural* analysis), and (2) replace M with a model or its contracts (*modular* analysis). These two approaches described next are orthogonal to the central topic of this paper, thus, we only give high-level and intuitive descriptions below.

One challenge in the interprocedural analysis is handling dynamic dispatch of (instance) methods in Java. That is, when N makes a call to M of a class C, it does not necessarily mean that M will actually be called. Instead any method overriding M in the subtypes of C might be called. In a concrete execution, this is not a problem because it knows what the runtime type of the receiver object being used for the method invocation is. However, in Kiasan, the receiver type represents a base type C and all of its subtypes satisfying the path condition. Thus, in general, Kiasan does not know exactly which method to call. One strategy is to consider all possible overriding methods, but this may be too costly if there are many such overriding methods. Another strategy is to use one or a subset of the overriding methods, but this makes the analysis to miss bugs that are caused by other overriding methods not included in the analysis; thus, care must be taken to interpret such analysis result. In both strategies, when one of M's overriding method is used, say from type C', the path condition is enhanced with the new knowledge that the receiver object is now a subtype of C' (and not a subtype of any C'', where $C'' <: C'$ and C'' declares an overriding method, if any).

Regardless of which strategy is picked, the analysis result becomes stale if the class hierarchy is modified or the overriding methods are modified. In such cases, the program has to be re-analyzed. To address this issue, Kiasan offers two modular strategies. The first strategy is allowing the user to replace M with a model method M'. Intuitively, M' should approximate the behavior of M. However, if the approximation is inaccurate, it may produce false alarm (over-approximation), or it may miss bugs (under-approximation). Thus, care must be taken when creating such model method considering the properties that a user intends to check. A more systematic strategy is to replace M with its contracts (e.g., JML). The method contracts are first translated to effective (e.g., by including class invariants) and executable forms. The invocation of M is then substituted by the following: assertion of M's precondition, assigning fresh symbolic values to modified variables, and assumption of M's postcondition. Intuitively, assigning fresh symbolic values to modified variables has the effect of making their values unknown, while they are constrained by assuming M's postcondition.

4 Formalization

In this section, we formalize each of the three SymExe algorithms presented in Section 3 as an alternative operational semantics of a core subset of the JVM instructions, and prove their relative soundness and completeness[6] based on the semantics of JVM concrete execution. Note that the semantics of JVM concrete execution closely follows the standard definition of JVM semantics and serves as a reference basis. The reasons that we formalize the algorithms on the JVM instructions, also known as bytecode, instead of Java source code are:

1. A Java program ultimately runs as a form of bytecode. And bytecode has simpler semantics, and thus it is easier to be formalized;
2. There is no need to be concerned about source-level compilation (e.g., syntactic sugars), and optimizations;
3. The same formalization can be applied to other languages that can be compiled into Java bytecode (e.g., Python/Jython, Ruby/JRuby, and Scala), and also can be easily adapted to similar systems such as .Net (MS 2006).

To simplify the presentation, we put three limits on this formal study. First, we focus on single-threaded programs with method calls abstracted away per discussion in the previous section. Second, we assume that bytecode satisfies all static and structural constraints described in Section 4.8 of the JVM specification (Lindholm and Yellin 1999). Some of the most important constraints are: (1) the operand stack always contains correct numbers and types of operands; (2) for each instruction, the types of the all parameters are correct; (3) field accesses are all legal—private, protected, or public. Third, we provide semantic rules and proofs only for a representative subset of the JVM instructions for the clarity of the presentation (while still conveying the main idea of our approach), since there are more than 200 JVM instructions, and many of them share very similar semantics whose only difference is often operand types (e.g., the JVM has different instructions for adding integers and floating numbers). More precisely, the chosen JVM instructions are `getfield`, `aload`, `astore`, `iadd`, `isub`, `new`, `putfield`, `anewarray`, `iastore`, `iaload`, `instanceof`, `checkcast`, `if_icmplt`, `if_acmpeq`, `ifnull`, and `ifnonnull`.

The rest of this section is organized as follows. Subsections 4.1 and 4.2 present the operational semantics of bytecode for concrete execution, SymExe with lazy initialization

[6] The meaning of relativeness is provided in the beginning of Section 4.3.

List 1 Domains ($\uplus$ denotes disjoint union and $\rightharpoonup$ denotes partial function)

- **PType**, the set of primitive types, consisting of INT, FLOAT, etc.
- **AType**, the set of array types.
- **RType**, the set of record types.
- **SymType**, the set of symbolic types.
- $\mathbf{NPType} = \mathbf{RType} \uplus \mathbf{AType} \uplus \mathbf{SymType}$, the set of nonprimitive types.
- $\tau \in \mathbf{Type} = \mathbf{PType} \uplus \mathbf{NPType}$.
- $c \in \mathbf{Const}$, the set of constants including $\mathbb{N}$, TRUE, FALSE, NULL, etc.
- **ISymbol**, the set of integer symbols.
- **PSymbol**, the set of primitive symbols, including **ISymbol**.
- $l \in \mathbf{Loc}$, the set of locations to model heap addresses.
- $\hat{\alpha} \in \mathbf{SymLoc}$, the set of symbolic locations.
- $\bar{\alpha} \in \mathbf{SymRef}$, the set of symbolic references.
- $f \in \mathbf{Field}$, the set of fields including LEN, DEF, CONC, etc.
- $i \in \mathbf{Index} = \mathbf{Field} \uplus \mathbb{N} \uplus \mathbf{ISymbol}$, the set of indexes.
- $\mathbf{NPSymbol} = \{ \alpha_\tau \mid \alpha_\tau : \mathbf{Index} \rightharpoonup \mathbf{Value} \}$, the set of nonprimitive symbols.
- $\alpha, \beta \in \mathbf{Symbol} = \mathbf{PSymbol} \uplus \mathbf{NPSymbol}$, the set of symbols.
- $\mathbf{Global} = \{ g \mid g : \mathbf{Field} \rightharpoonup \mathbf{Value} \}$, the set of globals.
- $pc \in \mathbf{PC}$, the set of program counters.
- $\mathbf{Local} = \{ \xi \mid \xi : \mathbb{N} \rightharpoonup \mathbf{Value} \}$.
- $\mathbf{Stack} = \{ \omega \mid \omega : \mathrm{Seq}(\mathbf{Value}) \}$ is the set of operand stacks which are modeled by sequences of values.
- $\mathbf{Heap} = \{ h \mid h : \mathbf{Loc} \rightharpoonup \mathbf{NPSymbol} \}$.
- $\phi \in \Phi$, the set of boolean expressions.
- $\sigma \in \mathbf{State} = \mathbf{Global} \times \mathbf{PC} \times \mathbf{Local} \times \mathbf{Stack} \times \mathbf{Heap} \times \Phi$. $\mathbf{State}_c \subseteq \mathbf{State}_s \subseteq \mathbf{State}_a \subseteq \mathbf{State}_b = \mathbf{State}$ represent state domains in concrete JVM, SEL, SELA, and SELB, respectively.
- **Instr**, the set of bytecode instructions with additional `assert` and `assume` instructions.

(SEL), SymExe with lazier initialization (SELA), and SymExe with lazier# initialization (SELB). Subsection 4.3 shows a proof outline of the relative soundness and completeness of SEL, SELA, and SELB using the concrete execution as the basis.

4.1 Semantic Domains

The semantic domains are listed in List 1. Following the Java type system, we distinguish two kinds of types: primitive types (**PType**) and nonprimitive types (**NPType**). **NPType** is further divided into record types (**RType**) that denotes the types of objects, array types (**AType**), and symbolic types (**SymType**) which are used to model the variable types of symbolic objects.

Similarly to types, we distinguish primitive symbols (**PSymbol**) and nonprimitive symbols (**NPSymbol**). **NPSymbol** models objects and arrays as partial functions. In the lazy initialization, an undefined mapping represents an uninitialized field or array index. We use **NPSymbol** for concrete execution as well, and in this case, all the mappings of fields or array indexes are defined. More precisely,

- for each object, we use a function from **Field** to **Value**. Notice that **Field** includes a special field, CONC, which is used to distinguish whether an object is concrete or symbolic. More specifically, CONC is defined for all concrete objects and undefined for symbolic objects. We say an object is concrete if it was created by an allocation instruction (i.e., the `new` instruction); a symbolic object is a parameter or discovered lazily as informally described in the previous section.
- for each array, we use a function from $\mathbb{N} \uplus \mathbf{ISymbol} \uplus \{\mathrm{LEN}, \mathrm{DEF}, \mathrm{CONC}\}$ to **Value**, where each $\alpha \in \mathbf{ISymbol}$ models a symbolic index. LEN and DEF are special fields that

are mapped to the array length and the default value of an array, respectively. Another special field CONC is used in a similar way to the case of an object.

Note that each symbol in **Symbol**, symbolic location, symbolic reference, and field carries its type as a subscript and the type may be omitted when it is not important.

The state components of the JVM are customized as follows. The static fields are modeled as global variables (**Global**). The heap and local variables are modeled as partial functions from locations (**Loc**) to **NPSymbol** (i.e., **Heap**), and from $\mathbb{N}$ to **Value** (i.e., **Local**), respectively. An operand stack is modeled as a sequence of values (i.e., **Stack**). We use a single PC register and a single stack frame because we limit the scope of this formal study to single-threaded programs without method calls. The native method stacks are not modeled since we do not consider native code in this study. We add path condition ϕ to the state to facilitate SymExe. We model a path condition as a conjunctive set of formulas to reflect the fact that constraints are accumulated in each SymExe path.

Based on these customized elements, our formal JVM system maintains a state of the following signature: **Global** $\times$ **PC** $\times$ **Local** $\times$ **Stack** $\times$ **Heap** $\times$ Φ. Notice that we use the same state signature for the concrete execution (**State**$_c$), SEL (**State**$_s$), SELA (**State**$_a$), and SELB (**State**$_b$). The differences between them are mainly made, as explained earlier, by using additional nonprimitive values such as symbolic locations and symbolic references. Although a path condition is not absolutely necessary for concrete execution, we make use of it to indicate whether a certain operation is valid or not by restricting a path condition to either TRUE or FALSE. Appendix A.2 shows an example about how states can be formulated in SEL, SELA, and SELB.

The initial states of concrete execution, SEL, SELA, and SELB are different as well. The most prominent difference lies in the initial heap. Nonprimitive globals and locals, refer to distinct symbolic references in the case of SELB, distinct symbolic locations or NULL in the case of SELA, or nonprimitive symbols or NULL in the case of SEL and the concrete execution. Notice that in SEL and concrete execution, nonprimitive symbols do not have to be distinct (i.e., they can have aliasings). In the case of SEL, we use nonprimitive symbols whose fields and indexes are not initially mapped, whereas all fields and indexes are mapped in concrete execution. If a nonprimitive symbol is referred by a variable (either global or local) in SEL, its type (which is a symbolic type) should be a subtype of the variable type, and this constraint resides in the initial path condition. In other semantics, the initial path conditions are set to TRUE. The rest of the initial state components are similar to each other regardless of the different semantics: primitive variables are initialized to distinct primitive symbols except for the concrete execution; the initial program counter points at the starting point of the method; the initial stack is empty.

4.2 Semantic Rules

We use the following format of operational semantic rules:

$$\frac{\textit{premises}}{\sigma \Rightarrow_{\mathscr{C}/\mathscr{S}/\mathscr{A}/\mathscr{B}} \sigma_1 [\| \; \sigma_2] \mid \text{EXCEPTION}, \sigma_3 [\| \; \sigma_4] | \text{ERROR}, \sigma_5 [\| \; \sigma_6]}$$

The *premises* part contains the current bytecode instruction to be executed, and the consequent part shows how the current state denoted by σ is transformed into the end state(s) by the current bytecode. Either $\mathscr{C}, \mathscr{S}, \mathscr{A}$, or $\mathscr{B}$ is subscripted on $\Rightarrow$ to indicate the rule is

List 2 Auxiliary functions ($\downarrow$ = defined, $\uparrow$ = undefined)

- *default* : $\mathbf{Type} \to \mathbf{Value} = \lambda\tau.v,$ where v is the default value of τ, returns the default value.
- *fields* : $\mathbf{Type} \to \mathscr{P}(\mathbf{Field}) = \lambda\tau.\{f_{\tau'} \mid f_{\tau'}$ is a field in $\tau\}$ returns the fields of a given type.
- $<:$: $\mathbf{NPType} \times \mathbf{NPType} \to Boolean = \lambda(\tau,\tau').(\tau$ is a subtype of $\tau')$ (We use the function as an infix operator, for example, $\tau <: \tau'$).
- *acc-idx* : $\mathbf{NPSymbol} \to \mathscr{P}(\mathbb{N} \cup \mathbf{ISymbol}) = \lambda\alpha.\ \{i \in \mathbb{N} \cup \mathbf{ISymbol} \mid \alpha(i)\downarrow\}$ returns integral indexes of a non-primitive symbol.
- *collect* : $\mathbf{Heap} \to \mathscr{P}(\mathbf{Loc}) = \lambda h.\{l \in \mathrm{dom}\, h \mid h(l)(\textsc{conc})\uparrow\}$ returns locations that map to symbolic objects.
- *symbols* : $\mathbf{State} \to \mathscr{P}(\mathbf{Symbol}) = \lambda\sigma.\{\alpha \mid \alpha$ appears in $\sigma\}$ returns the set of all symbols in a state.
- *sym-locs* : $\mathbf{State} \to \mathscr{P}(\mathbf{SymLoc}) = \lambda\sigma.\{\hat{\alpha} \mid \hat{\alpha}$ appears in $\sigma\}$ returns the set of all symbolic locations in a state.
- *sym-refs* : $\mathbf{State} \to \mathscr{P}(\mathbf{SymRef}) = \lambda\sigma.\{\bar{\alpha} \mid \bar{\alpha}$ appears in $\sigma\}$ returns the set of all symbolic references in a state.
- *new-prim-sym* : $\mathbf{PType} \times \mathscr{P}(\mathbf{Symbol}) \to \mathbf{PSymbol} = \lambda(\tau,S).\alpha_\tau, \alpha_\tau \notin S$ returns a new primitive symbol.
- *new-sym-type* : $\mathscr{P}(\mathbf{Symbol}) \to \mathbf{SymType} = \lambda S.\tau$ s.t. $\tau \in \mathbf{SymType}$ and τ does not appear in S, returns a new symbolic type.
- *array-type* : $\mathbf{Type} \to \mathbf{AType} = \lambda\tau.\tau'$, where τ' is the array type with element type τ, returns a new array type.
- *new-sym* : $\mathscr{P}(\mathbf{Symbol}) \to \mathbf{NPSymbol} = \lambda(S).\alpha_\tau,$ s.t. $\alpha \notin S \wedge \tau = \textit{new-sym-type}(S) \wedge \forall i \in \mathbf{Index}.\alpha(i)\uparrow$, returns a new symbolic record.
- *new-sarr* : $\mathscr{P}(\mathbf{Symbol}) \to \mathbf{NPSymbol} = \lambda(S).\textit{new-sym}(S \cup \{\alpha\})[\textsc{len} \mapsto \alpha]$, where $\alpha = \textit{new-prim-sym}(\textsc{int}, S)$, returns a new symbolic array.
- *new-obj* : $\mathscr{P}(\mathbf{Symbol}) \times \mathbf{RType} \to \mathbf{NPSymbol} = \lambda(S,\tau).\alpha_\tau,$ s.t. $\alpha_\tau \notin S \wedge \forall f_{\tau'} \in \textit{fields}(\tau).\alpha(f_{\tau'}) = \textit{default}(\tau') \wedge \alpha(\textsc{conc})\downarrow$, returns a new concrete object.
- *new-arr* : $\mathscr{P}(\mathbf{Symbol}) \times \mathbf{Type} \times (\mathbb{N} \uplus \mathbf{ISymbol}) \to \mathbf{NPSymbol} = \lambda(S,\tau,v).\alpha_{\tau'},$ s.t. $\alpha_{\tau'} \notin S \wedge \tau' = \textit{array-type}(\tau) \wedge \mathrm{dom}\,\alpha = \{\textsc{def}, \textsc{len}, \textsc{conc}\} \wedge \alpha(\textsc{def}) = \textit{default}(\tau) \wedge \alpha(\textsc{len}) = v$, returns a new concrete array.
- *new-carr* : $\mathscr{P}(\mathbf{Symbol}) \times \mathbf{Type} \times \mathbb{N} \to \mathbf{NPSymbol} = \lambda(S,\tau,m).\alpha_{\tau'},$ s.t. $\alpha_{\tau'} \notin S \wedge \tau' = \textit{array-type}(\tau) \wedge \forall 0 \leq j < m.\alpha_{\tau'}(j) = \textit{default}(\tau) \wedge \alpha_{\tau'}(\textsc{len}) = m$, returns a new concrete array in concrete JVM semantics.
- *code* : $\mathbf{PC} \rightharpoonup \mathbf{Instr}$ which takes a program counter and returns the corresponding instruction that is pointed by the program counter.
- *next* : $\mathbf{PC} \rightharpoonup \mathbf{PC} = \lambda pc.pc'$, where pc' is the address of the instruction that is next to the instruction pointed by pc, returns the next program counter.

in either the concrete execution, SEL, SELA, or SELB, respectively. The end states (separated by $\|$) can be either normal, exceptional, or erroneous. An exceptional or erroneous end state is prefixed with EXCEPTION or ERROR, respectively. Exceptions are handled according to the JVM specification (Lindholm and Yellin 1999) and the execution is stopped if an error occurs. Also, the execution is terminated/ignored when the path condition becomes unsatisfiable. For simplicity, we assume that garbage collection is performed against unreachable concrete objects, i.e., nonprimitive symbols whose CONC field is defined, after every transition.

We name semantic rules in the format of xxxx[#]-C/S/A/B where xxxx corresponds to an instruction name. If there is more than one rule for the same instruction, we distinguish them by number labels. If there is more than one rule whose premises hold true, one of them is applied nondeterministically. The last letter, C, S, A, and B, indicates the semantical rules is first defined for the semantics of concrete execution, SEL, SELA, and SELB, respectively. To facilitate the definition of semantic rules, we define some auxiliary functions, shown in List 2.

GETFIELD1-C
$$\frac{code(pc) = \texttt{getfield}\ f \qquad \omega = l :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} (g, next(pc), \xi, h(l)(f) :: \omega', h, \textsc{True})}$$

GETFIELD2-C
$$\frac{code(pc) = \texttt{getfield}\ f \qquad \omega = \textsc{Null} :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} \text{NullPointerException}, (g, pc, \xi, \omega, h, \textsc{True})}$$

GETFIELD1-S
$$\frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = l :: \omega' \qquad h(l)(f_\tau)\downarrow}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, h(l)(f_\tau) :: \omega', h, \phi)}$$

GETFIELD2-S
$$\frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = l :: \omega' \qquad h(l)(f_\tau)\uparrow \qquad \tau \in \textbf{PType}}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \alpha :: \omega', h[l \mapsto h(l)[f \mapsto \alpha]], \phi) \\ \text{where } \alpha = \textit{new-prim-sym}(\tau, symbols(\sigma))\end{array}}$$

GETFIELD3-S
$$\frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = l :: \omega' \qquad h(l)(f_\tau)\uparrow \qquad \tau \in \textbf{NPType}}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \textsc{Null} :: \omega', h[l \mapsto h(l)[f \mapsto \textsc{Null}]], \phi)}$$

GETFIELD4-S
$$\frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = l :: \omega' \qquad h(l)(f_\tau)\uparrow \qquad \tau \in \textbf{NPType}}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, l' :: \omega', h[l \mapsto h(l)[f \mapsto l']], \phi \cup \{\tau' <: \tau\}) \\ \text{where } l' \in collect(h), \alpha_{\tau'} = h(l')\end{array}}$$

GETFIELD5-S
$$\frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = l :: \omega' \qquad h(l)(f_\tau)\uparrow \qquad \tau \in \textbf{AType}}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, l' :: \omega', h[l \mapsto h(l)[f \mapsto l']][l' \mapsto \alpha_{\tau'}], \\ \phi \cup \{\tau' <: \tau, \alpha(\textsc{len}) \geq 0\}) \\ \text{where } \alpha_{\tau'} = \textit{new-sarr}(symbols(\sigma)), l' \notin \operatorname{dom} h\end{array}}$$

GETFIELD6-S
$$\frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = l :: \omega' \qquad h(l)(f_\tau)\uparrow \qquad \tau \in \textbf{RType}}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, l' :: \omega', h[l \mapsto h(l)[f \mapsto l']][l' \mapsto \alpha_{\tau'}], \phi \cup \{\tau' <: \tau\}) \\ \text{where } \alpha_{\tau'} = \textit{new-sym}(symbols(\sigma)), l' \notin \operatorname{dom} h\end{array}}$$

GETFIELD7-S
$$\frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = \textsc{Null} :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} \text{NullPointerException}, (g, pc, \xi, \omega, h, \phi)}$$

GETFIELD1-A
$$\frac{code(pc) = \texttt{getfield}\ f \qquad \omega = \hat{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{A}} (g, pc, \xi, \omega, h, \phi \cup \{\tau' <: \tau\})[l/\hat{\alpha}] \qquad \text{where } l \in collect(h), h(l) = \alpha_{\tau'}}$$

GETFIELD2-A
$$\frac{code(pc) = \texttt{getfield}\ f \qquad \omega = \hat{\alpha}_\tau :: \omega'}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{A}} (g, pc, \xi, \omega, h[l \mapsto \alpha_{\tau'}], \phi \cup \{\tau' <: \tau\})[l/\hat{\alpha}] \\ \text{where } l \notin \operatorname{dom} h, \alpha_{\tau'} = \textit{new-sym}(symbols(\sigma))\end{array}}$$

GETFIELD3-A
$$\frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = l :: \omega' \qquad h(l)(f_\tau)\uparrow \qquad \tau \in \textbf{NPType}}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{A}} (g, next(pc), \xi, \hat{\alpha}_\tau :: \omega', h[l \mapsto h(l)[f_\tau \mapsto \hat{\alpha}_\tau]], \phi) \\ \text{where } \hat{\alpha} \text{ is fresh}\end{array}}$$

GETFIELD1-B
$$\frac{code(pc) = \texttt{getfield}\ f \qquad \omega = \bar{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{B}} \sigma[\textsc{Null}/\bar{\alpha}]}$$

GETFIELD2-B
$$\frac{code(pc) = \texttt{getfield}\ f \qquad \omega = \bar{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{B}} \sigma[\hat{\alpha}_\tau/\bar{\alpha}_\tau] \qquad \text{where } \hat{\alpha} \text{ is fresh}}$$

GETFIELD3-B
$$\frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = l :: \omega' \qquad h(l)(f_\tau)\uparrow \qquad \tau \in \textbf{NPType}}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{B}} (g, next(pc), \xi, \bar{\alpha}_\tau :: \omega', h[l \mapsto h(l)[f_\tau \mapsto \bar{\alpha}_\tau]], \phi) \\ \text{where } \bar{\alpha} \text{ is fresh}\end{array}}$$

Fig. 5 Rules for the `getfield` Instruction

4.2.1 Semantic Rules for Instruction `getfield`

Instruction `getfield` f reads the f field of the receiver. It is the most interesting instruction because the lazy initialization takes place when an undefined field is read. Figure 5 lists the semantic rules for the `getfield` instruction where we use the binding, $\sigma = (g, pc, \xi, \omega, h, \phi)$. We also use notation $h[f \mapsto v]$ for the update of heap h. More precisely, $h[f \mapsto v](f') = h(f)$ if $f' \neq f$; and v, otherwise. While the concrete execution semantics is defined as usual by GETFIELD1-C and 2-C, it would be more interesting to compare them with SymExe rules.

If the field is undefined, SEL lazily initializes a nonprimitive field to either NULL (GETFIELD3-S), one of type compatible nonprimitive symbols in the heap (GETFIELD4-S), or a fresh nonprimitive symbol representing an array or an object (GETFIELD5-S, GETFIELD6-S). A primitive field is initialized to a fresh primitive symbol (GETFIELD2-S). Otherwise, the existing value is returned if the field was already defined (GETFIELD1-S), and a `NullPointerException` is thrown if the receiver is NULL (GETFIELD7-S).

SELA, on the other hand, lazily initializes a nonprimitive field to a fresh symbolic location (GETFIELD3-A). This rule supersedes three SEL rules about lazy field initializations, i.e., GETFIELD4-S, 5-S, and 6-S. Since it is still possible to initialize an uninitialized field to NULL in SELA, GETFIELD3-S remains to be used. Likewise, GETFIELD1-S, 2-S, and 7-S remain used in SELA. If the receiver is a symbolic location, it is necessary to resolve this symbolic location to either one of type compatible nonprimitive symbols in the heap (GETFIELD1-A) or a fresh nonprimitive symbol (GETFIELD2-A).

Finally, SELB initializes a nonprimitive field to a fresh symbolic reference instead of a symbolic location (GETFIELD3-B). This rule supersedes all the previous rules about lazy initializations of fields, i.e., GETFIELD3-A, GETFIELD3-S, 4-S, 5-S, and 6-S. Notice that GETFIELD3-S is also superseded by GETFIELD3-B, unlike in SELA. This difference results in the nullness branching delay. On the other hand, if the receiver is a symbolic reference, it is necessary to resolve this symbolic reference to either NULL (GETFIELD1-B) or a fresh symbolic location (GETFIELD2-B).

4.2.2 Semantic Rules for Array Accessing Instructions in SEL

Instruction `iaload` is used to access an index of an integer array, and its semantic rules in SEL are shown in Figure 6. As explained in Section 3.1, for the nonexceptional accessing (index within the bounds and the array is not NULL), there are two cases: the accessing index is either equal to one of indexes used before or different from each of those indexes. In the former case, the element value at the chosen index that is assumed to be equal to the current index is returned, and the path condition is updated with the equality relation between that chosen index and the current index (IALOAD2-S). In the latter case, the array is updated with a new mapping from the current index to a fresh value (IALOAD3-S). This fresh value can be either a concrete default value or a symbolic one depending on whether the array is concrete or symbolic. The path condition is also updated to record the fact that the current index is not equal to any of the previously accessed indexes.

If the index is out of bounds or the array is NULL, `IndexOutOfBoundsException` or a `NullPointerException` is thrown, as shown in IALOAD1-S and IALOAD4-S, respectively.

4.2.3 Semantic Rules for Other JVM Instructions

The SEL semantic rules of other JVM instructions together with two additional instructions, `assume` and `assert`, for checking preconditions/postconditions are presented in Figures 7

$$\text{IALOAD1-S}\quad \frac{\mathit{code}(\mathit{pc}) = \texttt{iaload} \qquad \omega = i :: l :: \omega'}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{S}} \text{ArrayIndexOutOfBoundsException},\\ (g, \mathit{pc}, \xi, \omega, h, \phi \cup \{i < 0 \vee i \geq h(l)(\textsc{len})\})\end{array}}$$

$$\text{IALOAD2-S}\quad \frac{\mathit{code}(\mathit{pc}) = \texttt{iaload} \qquad \omega = i :: l :: \omega'}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{S}} (g, \mathit{next}(\mathit{pc}), \xi, \alpha(i') :: \omega', h, \phi \cup \{i = i'\})\\ \text{where } \alpha = h(l), i' \in \mathit{acc\text{-}idx}(\alpha)\end{array}}$$

$$\text{IALOAD3-S}\quad \frac{\mathit{code}(\mathit{pc}) = \texttt{iaload} \qquad \omega = i :: l :: \omega'}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{S}} (g, \mathit{next}(\mathit{pc}), \xi, v :: \omega', h[l \mapsto \alpha[i \mapsto v]], \phi \cup \{\, i \neq i' \mid i' \in I \,\}\\ \cup \{0 \leq i, i < \alpha(\textsc{len}), |I| < \alpha(\textsc{len})\})\\ \text{where } \alpha = h(l), I = \mathit{acc\text{-}idx}(\alpha),\\ v = \begin{cases} \alpha(\textsc{def}) & \text{if } \alpha(\textsc{def})\downarrow \\ \mathit{new\text{-}prim\text{-}sym}(\text{INT}, \mathit{symbols}(\sigma)) & \text{if } \alpha(\textsc{def})\uparrow \end{cases}\end{array}}$$

$$\text{IALOAD4-S}\quad \frac{\mathit{code}(\mathit{pc}) = \texttt{iaload} \qquad \omega = i :: \textsc{Null} :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} \text{NullPointerException}, (g, \mathit{pc}, \xi, \omega, h, \phi)}$$

Fig. 6 Rules for the `iaload` Instruction in SEL

and 8. Note that some of the rules can be optimized if the operands are concrete, for example, the rule IADD-S has no need to create a new symbolic integer if both operands are concrete. We adopt the more generalized treatment since it is mathematically sound and simple.

The listed SEL rules can be easily extended to SELA and SELB rules similarly to the case of `getfield`. In SELA, a symbolic location operand is first substituted by a location. In SELB, a symbolic reference operand is first initialized to a symbolic location or NULL. Interested readers are referred to Appendixes C and D, respectively, for the formal rules. For the concrete execution, the semantic rules are faithful to the JVM specification (Lindholm and Yellin 1999) as presented in Appendix B.

4.3 Relative Soundness and Completeness of Symbolic Execution Rules

In this subsection, we show that the three SymExe algorithms (i.e., SEL, SELA, and SELB) are relatively sound and complete with respect to the concrete execution. We first define the soundness and completeness of a SymExe algorithm as follows: (1) a SymExe algorithm is *sound* if and only if it can find the error when there is an error in the concrete execution, (2) a SymExe algorithm is *complete* if and only if the algorithm does not report false alarms. That is, if an error is found by the algorithm, there must be a corresponding concrete execution trace leading to this error.

We now clarify the sources of relativeness of the soundness and completeness properties of a SymExe algorithm. First, we say that a SymExe algorithm is *relatively sound* when the soundness holds given bounds sufficiently large to find a designated error. It is also assumed that an underlying theorem prover is sound by its usual definition of soundness lest the procedure of pruning infeasible paths unfairly excluding an actually feasible path. (In technical words, if $\vdash \neg pc$, then $\models \neg pc$ for a path condition pc.) Meanwhile, we say that a SymExe algorithm is *relatively complete* when the completeness holds provided that the feasibility of a path condition that can appear during a SymExe session can always be decided by an underlying theorem prover. In other words, an underlying theorem prover is complete by its usual definition of completeness. (In technical words, if $\models \neg pc$, then $\vdash \neg pc$

ALOAD-S
$$\frac{code(pc) = \texttt{aload\_n}}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \xi(n) :: \omega, h, \phi)}$$

ASTORE-S
$$\frac{code(pc) = \texttt{astore\_n} \qquad \omega = v :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi[n \mapsto v], \omega', h, \phi)}$$

IADD-S
$$\frac{code(pc) = \texttt{iadd} \qquad \omega = v_1 :: v_2 :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \alpha :: \omega', h, \phi \cup \{\alpha = v_1 + v_2\})}$$
$$\text{where } \alpha = new\text{-}prim\text{-}sym(\textsc{int}, symbols(\sigma))$$

ISUB-S
$$\frac{code(pc) = \texttt{isub} \qquad \omega = v_1 :: v_2 :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \alpha :: \omega', h, \phi \cup \{\alpha = v_2 - v_1\})}$$
$$\text{where } \alpha = new\text{-}prim\text{-}sym(\textsc{int}, symbols(\sigma))$$

NEW-S
$$\frac{code(pc) = \texttt{new } \tau}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, l :: \omega, h[l \mapsto new\text{-}obj(symbols(\sigma), \tau)], \phi)}$$
$$\text{where } l \notin \operatorname{dom} h$$

PUTFIELD1-S
$$\frac{code(pc) = \texttt{putfield } f \qquad \omega = v :: l :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \omega', h[l \mapsto h(l)[f \mapsto v]], \phi)}$$

PUTFIELD2-S
$$\frac{code(pc) = \texttt{putfield } f \qquad \omega = v :: \textsc{Null} :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} \text{NullPointerException}, (g, pc, \xi, \omega, h, \phi)}$$

ANEWARRAY-S
$$\frac{code(pc) = \texttt{anewarray } \tau \qquad \omega = v :: \omega'}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \omega', h[l \mapsto \alpha], \phi \cup \{v \geq 0\}) \\ \parallel \text{NegativeArraySizeException}, (g, pc, \xi, \omega, h, \phi \cup \{v < 0\})\end{array}}$$
$$\text{where } \alpha = new\text{-}arr(symbols(\sigma), \tau, v), l \notin \operatorname{dom} h$$

IASTORE1-S
$$\frac{code(pc) = \texttt{iastore} \qquad \omega = v :: i :: l :: \omega'}{\begin{array}{l}\sigma \Rightarrow_{\mathscr{S}} \text{ArrayIndexOutOfBoundException}, \\ (g, pc, \xi, \omega, h, \phi \cup \{i < 0 \vee i \geq h(l)(\textsc{len})\})\end{array}}$$

IASTORE2-S
$$\frac{code(pc) = \texttt{iastore} \qquad \omega = v :: i :: l :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \omega', h[l \mapsto \alpha[i' \mapsto v]], \phi \cup \{i = i'\})}$$
$$\text{where } \alpha = h(l), i' \in acc\text{-}idx(\alpha)$$

IASTORE3-S
$$\frac{code(pc) = \texttt{iastore} \qquad \omega = v :: i :: l :: \omega'}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \omega', h[l \mapsto \alpha[i \mapsto v]], \phi \cup \{\, i \neq i' \mid i' \in I \,\} \\ \cup \{0 \leq i, i < \alpha(\textsc{len}), |I| < \alpha(\textsc{len})\}) \qquad \text{where } \alpha = h(l), I = acc\text{-}idx(\alpha)\end{array}}$$

IASTORE4-S
$$\frac{code(pc) = \texttt{iastore} \qquad \omega = v :: i :: \textsc{Null} :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} \text{NullPointerException}, (g, pc, \xi, \omega, h, \phi)}$$

INSTANCEOF1-S
$$\frac{code(pc) = \texttt{instanceof } \tau \qquad \omega = \textsc{Null} :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, 0 :: \omega', h, \phi)}$$

INSTANCEOF2-S
$$\frac{code(pc) = \texttt{instanceof } \tau \qquad \omega = l :: \omega'}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, 1 :: \omega', h, \phi \cup \{\tau' <: \tau\}) \\ \parallel (g, next(pc), \xi, 0 :: \omega', h, \phi \cup \{\tau' \not<: \tau\}) \qquad \text{where } \alpha_{\tau'} = h(l)\end{array}}$$

CHECKCAST1-S
$$\frac{code(pc) = \texttt{checkcast } \tau \qquad \omega = \textsc{Null} :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \omega, h, \phi)}$$

Fig. 7 Other Rules in SEL (1)

CHECKCAST2-S
$$\frac{code(pc) = \texttt{checkcast}\ \tau \qquad \omega = l :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \omega, h, \phi \cup \{\tau' <: \tau\}) \parallel \text{ClassCastException}, (g, pc, \xi, \omega, h, \phi \cup \{\tau' \not<: \tau\}) \quad \text{where } \alpha_{\tau'} = h(l)}$$

IF_ICMPLT-S
$$\frac{code(pc) = \texttt{if\_icmplt}\ pc' \qquad \omega = v_1 :: v_2 :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \omega', h, \phi \cup \{v_2 \geq v_1\}) \parallel (g, pc', \xi, \omega', h, \phi \cup \{v_2 < v_1\})}$$

IF_ACMPEQ1-S
$$\frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = v_2 :: v_1 :: \omega' \qquad v_2 \neq v_1}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \omega', h, \phi)}$$

IF_ACMPEQ2-S
$$\frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = v_2 :: v_1 :: \omega' \qquad v_2 = v_1}{\sigma \Rightarrow_{\mathscr{S}} (g, pc', \xi, \omega', h, \phi)}$$

IFNULL1-S
$$\frac{code(pc) = \texttt{ifnull}\ pc' \qquad \omega = l :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \omega', h, \phi)}$$

IFNULL2-S
$$\frac{code(pc) = \texttt{ifnull}\ pc' \qquad \omega = \textsc{Null} :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} (g, pc', \xi, \omega', h, \phi)}$$

IFNONNULL1-S
$$\frac{code(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = l :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} (g, pc', \xi, \omega', h, \phi)}$$

IFNONNULL2-S
$$\frac{code(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = \textsc{Null} :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \omega', h, \phi)}$$

ASSUME-S
$$\frac{code(pc) = \texttt{assume} \qquad \omega = v :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \omega', h, \phi \cup \{v = 1\})}$$

ASSERT-S
$$\frac{code(pc) = \texttt{assert} \qquad \omega = v :: \omega'}{\sigma \Rightarrow_{\mathscr{S}} (g, next(pc), \xi, \omega', h, \phi \cup \{v = 1\}) \parallel \textsc{Error}, (g, pc, \xi, \omega, h, \phi \cup \{v = 0\})}$$

Fig. 8 Other Rules in SEL(2)

for a path condition *pc*.) Our definition of relative completeness is essentially the same as the one used for the relative completeness of Hoare logic by Cook (1978).

In the rest of this article, we often omit "relative" before soundness and completeness for brevity sake. In order to prove the soundness, we will show that each concrete execution trace has a corresponding trace in each of SEL, SELA, and SELB given sufficient large bounds. Conversely, we will prove completeness by showing that each trace in SEL, SELA, and SELB has a corresponding concrete execution trace.

The rest of this subsection is organized as follows. Subsection 4.3.1 presents concretization (γ) functions which relate each more abstract (coarser) state to a set of less abstract (finer) states in the order of SELB, SELA, SEL, and concrete execution. Subsection 4.3.2 sketches the soundness and completeness proofs based on simulations of Kripke structures.

Limitations of Decision Procedures: It is well known that first-order logic and the Peano arithmetic (natural numbers) are undecidable. So automatic decision procedures (DP) have to either work on some subsets that are decidable or give up the completeness, that is, return unknown for some formulas. Currently, many DPs such as CVC3 (Barrett and Tinelli 2007), Yices (Dutertre and de Moura 2006), Z3 (de Moura and Bjørner 2008) support decidable theories such as Presburger arithmetic and expand to undecidable theories. In general, theories such as rational linear arithmetic, array, bit vector are supported. Language features such as integer overflow and floating point arithmetic are usually not directly supported but can be worked around by using bit vectors. As mentioned earlier, even some theories (e.g., quantifiers and non-linear arithmetic) are supported by a DP, it may still return unknown as results due to their undecidable properties.

Thus, from a practical point of view, the limitations of the decision procedure impose some limitations on any approach that makes use it. On the other hand, as decision procedure techniques are continually being improved, as they have been in recent years, the better the approach is. In light of this, our approach is parameterized on the underlying decision procedure used (as they are orthogonal to reasoning about heap object structures). The formal treatment of our approach presented next (that establishes relative soundness and completeness properties of our approach with respect to the underlying decision procedure) is done to show that our approach does not add unnecessary unsoundness and incompleteness. In other words, relative completeness guarantees a zero false alarm rate when the language expressing path conditions is restricted enough to ensure the complete handling of a theorem prover. Meanwhile, relative soundness guarantees that an error trace can be found if there exists an error in a given program.

4.3.1 Concretization (γ) Functions

1. $\gamma_s : \mathbf{State}_s \to \mathscr{P}(\mathbf{State}_c)$;
2. $\gamma_a : \mathbf{State}_a \to \mathscr{P}(\mathbf{State}_s)$;
3. $\gamma_b : \mathbf{State}_b \to \mathscr{P}(\mathbf{State}_a)$.

First, we take the view that $\mathbf{State}_c \prec \mathbf{State}_s \prec \mathbf{State}_a \prec \mathbf{State}_b$, where $\prec$ means more abstract. Then we can define three concretization (γ) functions that take a more abstract state and return the set of less abstract states that the more abstract state represents. Intuitively,

1. $\sigma_c \in \gamma_s(\sigma_s)$ if and only if σ_s can be transformed to σ_c by the following operations:
   - systematic substitution of primitive symbols with concrete values that satisfy the path condition.
   - a permutation of heap locations.
   - an application of a lazy initialization to an undefined field of a symbolic object/array.
2. $\sigma_s \in \gamma_a(\sigma_a)$ if and only if σ_s is a resulting state of substituting each symbolic location in σ_a with an existing symbolic object of a compatible type in the heap or a fresh symbolic object.
3. $\sigma_a \in \gamma_b(\sigma_b)$ if and only if σ_a is a resulting state of substituting each symbolic reference in σ_b with NULL or a fresh symbolic location.

The formal definitions of the three functions are shown in Appendix E. The γ functions have the following properties:

1. For all $\sigma_s \in \mathbf{State}_s$, if the path condition of σ_s is satisfiable, then $\gamma_s(\sigma_s) \neq \emptyset$.
2. For all $\sigma_a \in \mathbf{State}_a$, if the path condition of σ_a is satisfiable, then $\gamma_a(\sigma_a) \neq \emptyset$.
3. For all $\sigma_b \in \mathbf{State}_b$, if the path condition of σ_b is satisfiable, then $\gamma_a(\sigma_b) \neq \emptyset$.

The properties are readily followed from the definitions.

4.3.2 Soundness and Completeness Proof

Given a method, we model concrete JVM, SEL, SELA, and SELB computation trees using unlabeled Kripke structures (defined in Appendix F): $\mathscr{C} = (\Sigma_{\mathscr{C}}, I_{\mathscr{C}}, \longrightarrow_{\mathscr{C}})$, $\mathscr{S} = (\Sigma_{\mathscr{S}}, I_{\mathscr{S}}, \longrightarrow_{\mathscr{S}}$

$$\begin{tikzcd}
\sigma_c \arrow[d, "\longrightarrow_{\mathscr{C}}"'] & \sigma_s \arrow[l, "\gamma_s"'] \arrow[d, "\longrightarrow_{\mathscr{S}}"'] & \sigma_a \arrow[l, "\gamma_a"'] \arrow[d, "\longrightarrow_{\mathscr{A}}"'] & \sigma_b \arrow[l, "\gamma_b"'] \arrow[d, "\longrightarrow_{\mathscr{B}}"'] \\
\sigma_c' & \sigma_s' \arrow[l, "\gamma_s"'] & \sigma_a' \arrow[l, "\gamma_a"'] & \sigma_b' \arrow[l, "\gamma_b"']
\end{tikzcd}$$

Fig. 9 Simulation Relations

), $\mathscr{A} = (\Sigma_{\mathscr{A}}, I_{\mathscr{A}}, \longrightarrow_{\mathscr{A}})$, and $\mathscr{B} = (\Sigma_{\mathscr{B}}, I_{\mathscr{B}}, \longrightarrow_{\mathscr{B}})$. The components of Kripke structure X where X is $\mathscr{C}$, $\mathscr{S}$, $\mathscr{A}$, and $\mathscr{B}$ are defined as follows:

1. $\Sigma_X = \mathbf{State}_x \cup (\text{EXCEPTION} \times \mathbf{State}_x) \cup (\text{ERROR} \times \mathbf{State}_x)$, where hereafter x denotes c, s, a, and b, respectively, when X is $\mathscr{C}$, $\mathscr{S}$, $\mathscr{A}$, and $\mathscr{B}$. Note that the γ functions are trivially extended to Kripke states.
2. The initial states, I_X, are the same as those defined in Section 4.1.
3. The transition relations are defined as follows: $\sigma_x \longrightarrow_X \sigma_x'$ iff $\sigma_x \Rightarrow_X^n \sigma_x'$ for $n > 0$ such that only the first $n-1$ state transitions must be initializations of symbolic locations or symbolic references, and the path condition of σ_x' is satisfiable.

Notice that the four Kripke structures model the complete computation trees without any bounding so that later proofs will be independent of bounding strategies.

Lemma 1

1. $I_{\mathscr{C}} = \bigcup_{\sigma_s \in I_{\mathscr{S}}} \gamma_s(\sigma_s)$.
2. $I_{\mathscr{S}} = \bigcup_{\sigma_a \in I_{\mathscr{A}}} \gamma_a(\sigma_a)$.
3. $I_{\mathscr{A}} = \bigcup_{\sigma_b \in I_{\mathscr{B}}} \gamma_b(\sigma_b)$.

Proof All three parts can be shown by set inclusions on both directions. For example, for the $\subseteq$ direction of part (1), we show that for all $\sigma_c \in I_{\mathscr{C}}$ there exists a $\sigma_s \in I_{\mathscr{S}}$ such that $\sigma_c \in \gamma_s(\sigma_s)$; for the $\supseteq$ direction of part (1), we show that for all $\sigma_s \in I_{\mathscr{S}}$, $I_{\mathscr{C}} \supseteq \gamma_s(\sigma_s)$. For more detailed proof, readers are referred to Appendix G.

Soundness To prove the soundness, first, we show that there are simulation relations (defined in Appendix F) relating Kripke structures $\mathscr{C}$ to $\mathscr{S}$, $\mathscr{S}$ to $\mathscr{A}$, and $\mathscr{A}$ to $\mathscr{B}$ as illustrated in Figure 9.

Lemma 2 *Given the following relations,*

– $R_{\gamma_s} \subseteq \Sigma_{\mathscr{C}} \times \Sigma_{\mathscr{S}}$ *as* $(\sigma_c, \sigma_s) \in R_{\gamma_s}$ *if and only if* $\sigma_c \in \gamma_s(\sigma_s)$*;*
– $R_{\gamma_a} \subseteq \Sigma_{\mathscr{S}} \times \Sigma_{\mathscr{A}}$ *as* $(\sigma_s, \sigma_a) \in R_{\gamma_a}$ *if and only if* $\sigma_s \in \gamma_a(\sigma_a)$*;*
– $R_{\gamma_b} \subseteq \Sigma_{\mathscr{A}} \times \Sigma_{\mathscr{B}}$ *as* $(\sigma_a, \sigma_b) \in R_{\gamma_b}$ *if and only if* $\sigma_a \in \gamma_b(\sigma_b)$*,*

1. $\mathscr{C} \lhd_{R_{\gamma_s}} \mathscr{S}$*;*
2. $\mathscr{S} \lhd_{R_{\gamma_a}} \mathscr{A}$*;*
3. $\mathscr{A} \lhd_{R_{\gamma_b}} \mathscr{B}$*.*

Proof All three parts can be proved by rule induction. More detailed proof of the simulation relations is presented in a thesis (Deng 2007).

Theorem 1

1. *Given any trace in $\mathscr{C}$: $\sigma_{c_1} \longrightarrow_{\mathscr{C}} \sigma_{c_2} \longrightarrow_{\mathscr{C}} \cdots \longrightarrow_{\mathscr{C}} \sigma_{c_n}$, where $n > 0$ and $\sigma_{c_1} \in I_{\mathscr{C}}$, there exists a trace in $\mathscr{S}$: $\sigma_{s_1} \longrightarrow_{\mathscr{S}} \sigma_{s_2} \longrightarrow_{\mathscr{S}} \cdots \longrightarrow_{\mathscr{S}} \sigma_{s_n}$ such that $\sigma_{c_k} \in \gamma_s(\sigma_{s_k})$ for all $1 \leq k \leq n$.*
2. *Given any trace in $\mathscr{S}$: $\sigma_{s_1} \longrightarrow_{\mathscr{S}} \sigma_{s_2} \longrightarrow_{\mathscr{S}} \cdots \longrightarrow_{\mathscr{S}} \sigma_{s_n}$, where $n > 0$ and $\sigma_{s_1} \in I_{\mathscr{S}}$, there exists a trace in $\mathscr{A}$: $\sigma_{a_1} \longrightarrow_{\mathscr{A}} \sigma_{a_2} \longrightarrow_{\mathscr{A}} \cdots \longrightarrow_{\mathscr{A}} \sigma_{a_n}$ such that $\sigma_{s_k} \in \gamma_a(\sigma_{a_k})$ for all $1 \leq k \leq n$.*
3. *Given any trace in $\mathscr{A}$: $\sigma_{a_1} \longrightarrow_{\mathscr{A}} \sigma_{a_2} \longrightarrow_{\mathscr{A}} \cdots \longrightarrow_{\mathscr{A}} \sigma_{a_n}$, where $n > 0$ and $\sigma_{a_1} \in I_{\mathscr{A}}$, there exists a trace in $\mathscr{B}$: $\sigma_{b_1} \longrightarrow_{\mathscr{B}} \sigma_{b_2} \longrightarrow_{\mathscr{B}} \cdots \longrightarrow_{\mathscr{B}} \sigma_{b_n}$ such that $\sigma_{a_k} \in \gamma_b(\sigma_{b_k})$ for all $1 \leq k \leq n$.*

Proof For each of the three parts, we show there exists a corresponding trace by mathematical induction on the length of the traces. The base cases are direct results of Lemma 1. The induction steps are established by applying Lemma 2.

Corollary 1 *Given any trace in $\mathscr{C}$: $\sigma_{c_1} \longrightarrow_{\mathscr{C}} \sigma_{c_2} \longrightarrow_{\mathscr{C}} \cdots \longrightarrow_{\mathscr{C}} \sigma_{c_n}$, where $n > 0$ and $\sigma_{c_1} \in I_{\mathscr{C}}$, then*

1. *there exists a trace in $\mathscr{S}$: $\sigma_{s_1} \longrightarrow_{\mathscr{S}} \sigma_{s_2} \longrightarrow_{\mathscr{S}} \cdots \longrightarrow_{\mathscr{S}} \sigma_{s_n}$ such that $\sigma_{c_k} \in \gamma_s(\sigma_{s_k})$ for all $1 \leq k \leq n$;*
2. *there exists a trace in $\mathscr{A}$: $\sigma_{a_1} \longrightarrow_{\mathscr{A}} \sigma_{a_2} \longrightarrow_{\mathscr{A}} \cdots \longrightarrow_{\mathscr{A}} \sigma_{a_n}$ such that $\sigma_{c_k} \in \bigcup_{\sigma_s \in \gamma_a(\sigma_{a_k})} \gamma_s(\sigma_s)$ for all $1 \leq k \leq n$;*
3. *there exists a trace in $\mathscr{B}$: $\sigma_{b_1} \longrightarrow_{\mathscr{B}} \sigma_{b_2} \longrightarrow_{\mathscr{B}} \cdots \longrightarrow_{\mathscr{B}} \sigma_{b_n}$ such that $\sigma_{c_k} \in \bigcup_{\sigma_a \in \gamma_b(\sigma_{b_k})} \bigcup_{\sigma_s \in \gamma_a(\sigma_a)} \gamma_s(\sigma_s)$ for all $1 \leq k \leq n$.*

Proof Part (1) is the same as Theorem 1.(1). Part (2) can be shown by combining part (1) and Theorem 1.(2). Similarly, by combining the results of part (2) and Theorem 1.(3), we can get part (3).

With Corollary 1, we are ready to show the (relative) soundness of our SEL, SELA, and SELB with respect to the concrete JVM:

Corollary 2 (Soundness) *If there is a bug in the concrete execution, given sufficiently large bounds, SEL, SELA, and SELB can find the bug.*

Proof Given a bug in the concrete execution, there must be a trace in concrete JVM $\mathscr{C}$ that leads to it. Suppose the trace has n steps in $\mathscr{C}$. By Corollary 1, there exists a corresponding trace with n steps in each of $\mathscr{S}$, $\mathscr{A}$, and $\mathscr{B}$. If the bounds are large enough such that all n step traces in $\mathscr{S}$, $\mathscr{A}$, and $\mathscr{B}$ are explored by SEL, SELA, and SELB respectively, the corresponding symbolic traces are presented in SEL, SELA, and SELB. Therefore, the bug can be found in SEL, SELA, and SELB.

Completeness We will show that every trace in $\mathscr{S}$, $\mathscr{A}$, and $\mathscr{B}$ corresponds to a trace in $\mathscr{C}$.

We first define notion of unlabeled power Kripke structure: given any Kripke structure, $\mathscr{K} = (\Sigma_{\mathscr{K}}, I_{\mathscr{K}}, \longrightarrow_{\mathscr{K}})$, the power Kripke structure of $\mathscr{K}$ is

$$\mathscr{P}(\mathscr{K}) = (\mathscr{P}(\Sigma_{\mathscr{K}}), \mathscr{P}(I_{\mathscr{K}}), \overset{\bullet}{\longrightarrow}_{\mathscr{K}}),$$

satisfying the following condition:for two sets of states $S, S' \subseteq \Sigma_{\mathscr{K}}$, $S \overset{\bullet}{\longrightarrow}_{\mathscr{K}} S'$ only if $\forall \sigma' \in S'. \exists \sigma \in S. \sigma \longrightarrow_{\mathscr{K}} \sigma'$.

Then we introduce power Kripke structures of $\mathscr{C}$, $\mathscr{S}$, $\mathscr{A}$, and $\mathscr{B}$ as $\mathscr{P}(\mathscr{C}) = (\mathscr{P}(\Sigma_{\mathscr{C}}), \mathscr{P}(I_{\mathscr{C}}), \overset{\bullet}{\longrightarrow}_{\mathscr{C}})$, $\mathscr{P}(\mathscr{S}) = (\mathscr{P}(\Sigma_{\mathscr{S}}), \mathscr{P}(I_{\mathscr{S}}), \overset{\bullet}{\longrightarrow}_{\mathscr{S}})$, $\mathscr{P}(\mathscr{A}) = (\mathscr{P}(\Sigma_{\mathscr{A}}), \mathscr{P}(I_{\mathscr{A}}), \overset{\bullet}{\longrightarrow}_{\mathscr{A}})$, and $\mathscr{P}(\mathscr{B}) = (\mathscr{P}(\Sigma_{\mathscr{B}}), \mathscr{P}(I_{\mathscr{B}}), \overset{\bullet}{\longrightarrow}_{\mathscr{B}})$. Next we show there are simulation relations relating $\mathscr{S}$ to $\mathscr{P}(\mathscr{C})$, $\mathscr{A}$ to $\mathscr{P}(\mathscr{S})$, and $\mathscr{B}$ to $\mathscr{P}(\mathscr{A})$.

Lemma 3 *Given the following relations,*

- $R^{\bullet}_{\gamma_s} \subseteq \Sigma_{\mathscr{S}} \times \mathscr{P}(\Sigma_{\mathscr{C}})$ *as* $(\sigma_s, C) \in R^{\bullet}_{\gamma_s}$ *if and only if* $C = \gamma_s(\sigma_s)$*;*
- $R^{\bullet}_{\gamma_a} \subseteq \Sigma_{\mathscr{A}} \times \mathscr{P}(\Sigma_{\mathscr{S}})$ *as* $(\sigma_a, S) \in R^{\bullet}_{\gamma_a}$ *if and only if* $S = \gamma_a(\sigma_a)$*;*
- $R^{\bullet}_{\gamma_b} \subseteq \Sigma_{\mathscr{B}} \times \mathscr{P}(\Sigma_{\mathscr{A}})$ *as* $(\sigma_b, A) \in R^{\bullet}_{\gamma_b}$ *if and only if* $A = \gamma_a(\sigma_b)$*,*

1. $\mathscr{S} \lhd_{R^{\bullet}_{\gamma_s}} \mathscr{P}(\mathscr{C})$*;*
2. $\mathscr{A} \lhd_{R^{\bullet}_{\gamma_a}} \mathscr{P}(\mathscr{S})$*;*
3. $\mathscr{B} \lhd_{R^{\bullet}_{\gamma_b}} \mathscr{P}(\mathscr{A})$*.*

Proof We can also use rule induction to show all three parts. More detailed proof of the simulation relations is presented in thesis (Deng 2007).

Theorem 2

1. *Given any trace in* $\mathscr{S}$*:* $\sigma_{s_1} \longrightarrow_{\mathscr{S}} \sigma_{s_2} \longrightarrow_{\mathscr{S}} \cdots \longrightarrow_{\mathscr{S}} \sigma_{s_n}$ *where* $n > 0$ *and* $\sigma_{s_1} \in I_{\mathscr{S}}$*, there exists a trace in* $\mathscr{C}$*:* $\sigma_{c_1} \longrightarrow_{\mathscr{C}} \sigma_{c_2} \longrightarrow_{\mathscr{C}} \cdots \longrightarrow_{\mathscr{C}} \sigma_{c_n}$ *such that* $\sigma_{c_k} \in \gamma_s(\sigma_{s_k})$ *for all* $1 \leq k \leq n$*.*
2. *Given any trace in* $\mathscr{A}$*:* $\sigma_{a_1} \longrightarrow_{\mathscr{A}} \sigma_{a_2} \longrightarrow_{\mathscr{A}} \cdots \longrightarrow_{\mathscr{A}} \sigma_{a_n}$ *where* $n > 0$ *and* $\sigma_{a_1} \in I_{\mathscr{A}}$*, there exists a trace in* $\mathscr{S}$*:* $\sigma_{s_1} \longrightarrow_{\mathscr{S}} \sigma_{s_2} \longrightarrow_{\mathscr{S}} \cdots \longrightarrow_{\mathscr{S}} \sigma_{s_n}$ *such that* $\sigma_{s_k} \in \gamma_a(\sigma_{a_k})$ *for all* $1 \leq k \leq n$*.*
3. *Given any trace in* $\mathscr{B}$*:* $\sigma_{b_1} \longrightarrow_{\mathscr{B}} \sigma_{b_2} \longrightarrow_{\mathscr{B}} \cdots \longrightarrow_{\mathscr{B}} \sigma_{b_n}$ *where* $n > 0$ *and* $\sigma_{b_1} \in I_{\mathscr{B}}$*, there exists a trace in* $\mathscr{A}$*:* $\sigma_{a_1} \longrightarrow_{\mathscr{A}} \sigma_{a_2} \longrightarrow_{\mathscr{A}} \cdots \longrightarrow_{\mathscr{A}} \sigma_{a_n}$ *such that* $\sigma_{a_k} \in \gamma_b(\sigma_{b_k})$ *for all* $1 \leq k \leq n$*.*

Proof We only prove part (1). Parts (2) and (3) can be shown similarly. By the definition of $\longrightarrow_{\mathscr{S}}$, the path condition of σ_{s_n} must be satisfiable. From the property of γ_s function, $\gamma_s(\sigma_{s_n}) \neq \emptyset$. Define a sequence of states in $\mathscr{P}(\mathscr{C})$ as

$$(C_k = \gamma_s(\sigma_{s_k}))_{1 \leq k \leq n}.$$

Clearly $C_n \neq \emptyset$. After applying mathematical induction with Lemma 3.(1), we get $C_1 \overset{\bullet}{\longrightarrow}_{\mathscr{C}} C_2 \overset{\bullet}{\longrightarrow}_{\mathscr{C}} \cdots \overset{\bullet}{\longrightarrow}_{\mathscr{C}} C_n$. By Lemma 1, $C_1 \subseteq I_{\mathscr{C}}$. Since $C_n \neq \emptyset$, we can pick a $\sigma_{c_n} \in C_n$. From the definition of $\overset{\bullet}{\longrightarrow}_{\mathscr{C}}$, there exists a $\sigma_{c_{n-1}} \in C_{n-1}$ such that $\sigma_{c_{n-1}} \longrightarrow_{\mathscr{C}} \sigma_{c_n}$. After repeating the process $n-1$ times, we get the following trace in $\mathscr{C}$: $\sigma_{c_1} \longrightarrow_{\mathscr{C}} \sigma_{c_2} \longrightarrow_{\mathscr{C}} \cdots \longrightarrow_{\mathscr{C}} \sigma_{c_n}$ where $\sigma_{c_k} \in C_k = \gamma_s(\sigma_{s_k})$ for all $1 \leq k \leq n$.

Corollary 3

1. *Given any trace in* $\mathscr{S}$*:* $\sigma_{s_1} \longrightarrow_{\mathscr{S}} \sigma_{s_2} \longrightarrow_{\mathscr{S}} \cdots \longrightarrow_{\mathscr{S}} \sigma_{s_n}$ *where* $n > 0$ *and* $\sigma_{s_1} \in I_{\mathscr{S}}$*, there exists a trace in* $\mathscr{C}$*:* $\sigma_{c_1} \longrightarrow_{\mathscr{C}} \sigma_{c_2} \longrightarrow_{\mathscr{C}} \cdots \longrightarrow_{\mathscr{C}} \sigma_{c_n}$ *such that* $\sigma_{c_k} \in \gamma_s(\sigma_{s_k})$ *for all* $1 \leq k \leq n$*.*
2. *Given any trace in* $\mathscr{A}$*:* $\sigma_{a_1} \longrightarrow_{\mathscr{A}} \sigma_{a_2} \longrightarrow_{\mathscr{A}} \cdots \longrightarrow_{\mathscr{A}} \sigma_{a_n}$ *where* $n > 0$ *and* $\sigma_{a_1} \in I_{\mathscr{A}}$*, there exists a trace in* $\mathscr{C}$*:* $\sigma_{c_1} \longrightarrow_{\mathscr{C}} \sigma_{c_2} \longrightarrow_{\mathscr{C}} \cdots \longrightarrow_{\mathscr{C}} \sigma_{c_n}$ *such that* $\sigma_{c_k} \in \bigcup_{\sigma_s \in \gamma_a(\sigma_{a_k})} \gamma_s(\sigma_s)$ *for all* $1 \leq k \leq n$*.*
3. *Given any trace in* $\mathscr{B}$*:* $\sigma_{b_1} \longrightarrow_{\mathscr{B}} \sigma_{b_2} \longrightarrow_{\mathscr{B}} \cdots \longrightarrow_{\mathscr{B}} \sigma_{b_n}$ *where* $n > 0$ *and* $\sigma_{b_1} \in I_{\mathscr{B}}$*, there exists a trace in* $\mathscr{C}$*:* $\sigma_{c_1} \longrightarrow_{\mathscr{C}} \sigma_{c_2} \longrightarrow_{\mathscr{C}} \cdots \longrightarrow_{\mathscr{C}} \sigma_{c_n}$ *such that* $\sigma_{c_k} \in \bigcup_{\sigma_a \in \gamma_b(\sigma_{b_k})} \bigcup_{\sigma_s \in \gamma_a(\sigma_a)} \gamma_s(\sigma_s)$ *for all* $1 \leq k \leq n$*.*

Proof Part (1) is directly from Theorem 2.(1). Part (2) can be shown by composing Theorem 2.(2) and part (1). Part (3) can be shown by combining Theorem 2.(3) and part (2).

The (relative) completeness of our SEL, SELA, and SELB with respect to the concrete JVM is the direct consequence of Corollary 3.

Corollary 4 (Completeness) *If SEL, SELA, or SELB finds a bug, it is present in the concrete execution as well.*

Proof Given that SEL, SELA, or SELB finds a bug, there must be a trace of SEL, SELA, or SELB that demonstrates the bug. Therefore, there is a trace of n steps for some n in $\mathscr{S}$, $\mathscr{A}$, or $\mathscr{B}$ that leads to the bug. By Corollary 3, there exists a corresponding concrete trace in $\mathscr{C}$. Hence, the bug exists in concrete execution.

5 Experiments

In this section, we systematically compare the performance of the lazy initialization algorithm with the two improved algorithms (lazier and lazier#) and demonstrate the effectiveness of the lazier# algorithm by experimental studies. We do not compare Symbolic JPF which originated the lazy initialization algorithm against the lazier and lazier# algorithms for three reasons. First, symbolic JPF tool is being reimplemented using custom bytecode interpretation approach thus is not ready for the examples that we consider in this section as of the writing of this document. Second, having corresponded closely with NASA Ames personnel (the developers of the Symbolic JPF), we are confident that our implementation of lazy initialization reflects that strategy implemented in Symbolic JPF. Third, we believe that comparing the three algorithms in Kiasan (our implementation of the algorithms) would present a more controlled experiment than in Kiasan and Symbolic JPF since the goal is to compare algorithms, not tools. In fact, direct comparison of the tools may even obscure inherent differences between the algorithms since there are many engineering differences between the tools, for example, having different implementations and using different theorem provers, etc.

The rest of this section is organized as follows. We will first describe the setup of the experimental studies in Subsection 5.1. Then Subsection 5.2 compares the performance of lazy, lazier, lazier# algorithms and two implementations of the algorithms. Finally, Subsection 5.3 presents a benchmark on common data structures and containers from the JDK library package `java.util`.

5.1 Experiment Setup

Kiasan Implementations: There are two implementations of Kiasan: Bogor/Kiasan (Deng et al 2006) and Sireum/Kiasan. Kiasan was initially implemented in the Bogor framework (Robby et al 2003), thus, the name Bogor/Kiasan. Recently, we have re-implemented Kiasan in the Sireum framework (Robby 2008).

Sireum/Kiasan has many improvements over Bogor/Kiasan. The two most important ones are more bounding strategies and decision procedure support. First, Sireum/Kiasan has implemented both the k-bound and the n-bound while Bogor/Kiasan only has implemented the k-bound. Second, Sireum/Kiasan has more flexible backend plugin architecture so that different backend decision procedures can be plugged in on the fly. As described in Section 2, SymExe relies on decision procedures for path condition satisfiability checking. We developed Bogor/Kiasan and Sireum/Kiasan in Java whereas most high performance decision procedures are implemented in C/C++. While in Bogor/Kiasan only CVC3 (Barrett

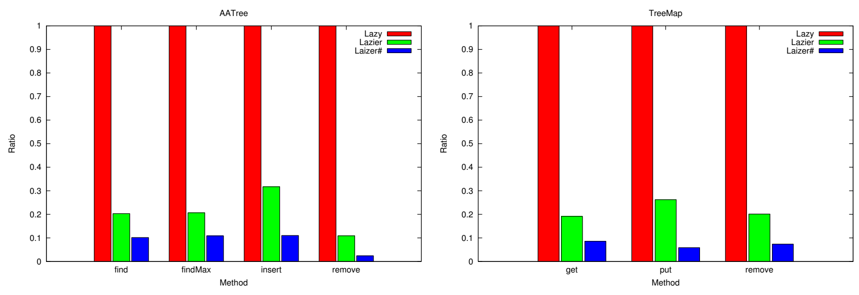

Fig. 10 Ratio of #Paths Explored by the Lazy, Lazier, and Lazier# Initialization Algorithms over #Paths Explored by the Lazy Initialization Algorithm with $k = 3$

and Tinelli 2007) is used through inter-process communication (IPC) by means of a pipe, Sireum/Kiasan can communicate with either CVC3 or Yices (Dutertre and de Moura 2006) through either IPC or the Java Native Interface (JNI).

We have used both Kiasan implementations despite the fact that Sireum/Kiasan supersedes Bogor/Kiasan. While experimental data of Sireum/Kiasan reflects the performance of our latest SymExe implementation, we have used Bogor/Kiasan for the purpose of comparison between the three SymExe algorithms we described earlier. This is because, as will be empirically demonstrated in this Section, the lazier# initialization algorithm outperforms the other two algorithms, and we implemented only the lazier# initialization algorithm in Sireum/Kiasan.

Experiment Environment The experiments were conducted in a machine with dual Xeon Quad-core 2.8 GHz and 16 GiB of Memory running OS X 10.5. And we used Java 1.6, 64-bit with 512 MiB heap.

Examples and Translation Most examples are taken from either the book (Weiss 2006) such as `AATree`, `AvlTree`, and `BinarySearchTree`, LeftistHeap, BinaryHeap and Sort or the package java.util of Java library such as `ArrayDeque`, `ArrayList`, `LinkedList`, `PriorityQueue`, `Stack`, `TreeMap`, `TreeSet`, and `Vector`. Container was earlier introduced in Program 1; GC adapted from a TVLA (Lev-Ami and Sagiv 2000) example is the marking phase of the mark and sweep garbage collection algorithm. For each class, we have added specifications, that is, an executable class invariant (`inv`) and a precondition (`pre`) and postcondition (`post`) for each method to be checked. In order to check these specifications in SymExe, we translate each method `M` into the following form:

```
        assume(inv); assume(pre); M; assert(inv); assert(post);
```

where the `assume`/`assert(exp)` *statements* are executed as follows: the `exp` is evaluated first and the result is pushed onto the top of the operand stack and then the rules for the `assume` and `assert` *instructions* described in Section 4 are applied.

5.2 Comparison of the Lazy, Lazier, and Lazier# Initialization Algorithms

We have performed the experimental study on nine examples listed in Table 1.

We have compared the performance of the lazy, lazier, and lazier# initialization algorithms based on two data: (1) the number of fully explored paths and (2) running time. We also have controlled the k-bound to see how differently this bound affects the performance of each algorithm. We highlight a few points next.

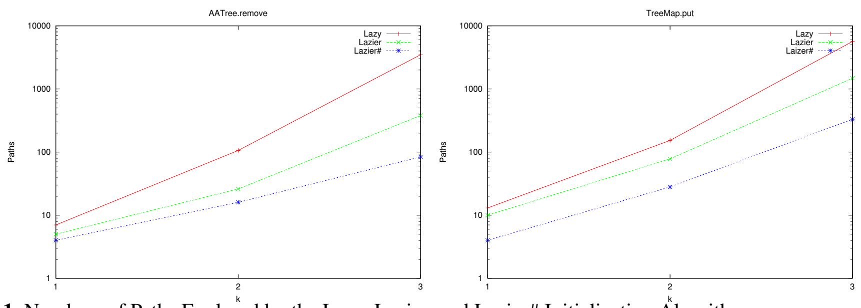

Fig. 11 Numbers of Paths Explored by the Lazy, Lazier, and Lazier# Initialization Algorithms

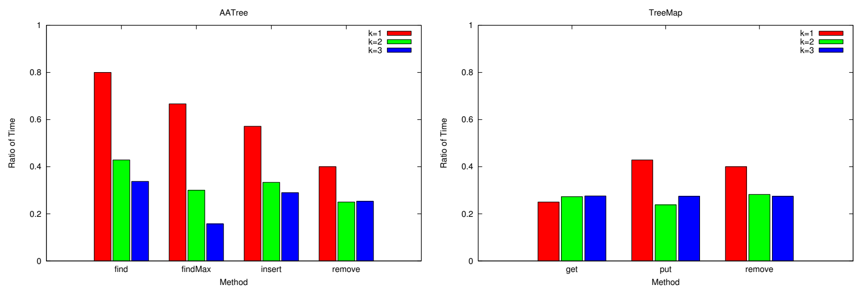

Fig. 12 Ratio of Running Time of Sireum/Kiasan over Running Time of Bogor/Kiasan. In both implementations, the lazier# initialization algorithm and CVC3 through IPC are used.

There is a total order $\geq$ among the number of explored paths of the three algorithms in the order of the lazy, lazier, and lazier# initialization algorithms. In addition, except for Sort and GC, the order is strict. A similar order is observed among the running times of the algorithms assuming a certain margin of error for the Sort and GC examples.

In general, the reduction ratios of fully explored paths by the lazier and lazier# initialization algorithms are very large. For example, as depicted in Figure 10 based on the data in Table 1, the lazier and lazier# initialization algorithms explore 70% to 90% fewer paths than the lazy initialization algorithm for the AATree and TreeMap examples. As a matter of fact, we proved in our other work (Deng et al 2010) that the lazier# initialization algorithm explores the optimal numbers of paths for the search tree examples (i.e., `AATree`, `AvlTree`, `BinarySearchTree`, and `TreeMap`).

Although, in all the algorithms, the number of explored paths grows exponentially as k increases, the increasing rate is the lowest in the lazier# case, and highest in the lazy case as illustrated in Figure 11.

The Sort and GC examples show no improvement due to different reasons. First, the Sort example manipulates an array of integers, not an array of objects, and hence the lazy initialization plays little role. Second, in the GC example, objects are fully expanded, and hence the degree of laziness takes little effect.

We have also measured the performance of Sireum/Kiasan. Due to various optimizations we have applied,[7] Sireum/Kiasan is up to 90% faster, as depicted in Figure 12, than Bogor/Kiasan even without taking advantage of faster SMT solver such as Yices (Dutertre and de Moura 2006).

[7] We defer the description of the optimizations to the future work.

Table 1: Experimental Data Using *k*-bound (Yn – Yices through JNI; Cn – CVC3 through JNI; Cp – CVC3 through IPC; Lz – Lazy; Lr – Lazier; L# – Lazier#; s – seconds; m – minutes)

Example		*k*	Sireum/Kiasan				Bogor/Kiasan					
Class			Paths	Time			Paths			Time		
	Method			Yn	Cn	Cp	Lz	Lr	L#	Lz	Lr	L#
AATree	find	1	4	0.4s	0.4s	0.4s	15	8	4	1.1s	0.7s	0.5s
		2	16	0.7s	0.8s	0.9s	197	32	16	11.4s	3.0s	2.1s
		3	84	2.0s	3.1s	5.5s	827	168	84	1.8m	25.6s	16.3s
	findMax	1	2	0.2s	0.2s	0.2s	5	3	2	0.6s	0.4s	0.3s
		2	4	0.2s	0.3s	0.3s	37	7	4	3.1s	1.3s	1.0s
		3	10	0.5s	1.1s	2.3s	92	19	10	15.3s	7.9s	7.0s
	findMin	1	2	0.2s	0.2s	0.2s	5	3	2	0.6s	0.4s	0.3s
		2	4	0.2s	0.3s	0.3s	41	7	4	3.2s	1.3s	1.0s
		3	10	0.4s	1.0s	2.1s	96	19	10	15.9s	7.9s	7.0s
	insert	1	4	0.3s	0.3s	0.4s	15	10	4	1.4s	1.2s	0.7s
		2	16	0.5s	0.6s	0.8s	96	44	16	8.1s	4.7s	2.4s
		3	84	1.3s	2.5s	5.3s	763	242	84	2.1m	44.4s	18.3s
	remove	1	4	0.2s	0.2s	0.2s	7	5	4	0.6s	0.5s	0.5s
		2	16	0.3s	0.5s	0.5s	106	26	16	6.9s	2.5s	2.0s
		3	84	1.0s	3.1s	4.1s	3488	380	84	19.5m	50.3s	16.2s
AvlTree	find	1	4	0.1s	0.1s	0.2s	6	5	4	0.6s	0.5s	0.4s
		2	21	0.2s	0.3s	0.5s	51	29	21	3.6s	2.5s	2.2s
		3	190	1.6s	4.0s	7.8s	753	275	190	1.4m	33.1s	26.4s
	findMax	1	2	0.1s	0.1s	0.2s	4	3	2	0.5s	0.4s	0.3s
		2	5	0.1s	0.2s	0.2s	19	9	5	2.2s	1.5s	1.1s
		3	20	0.3s	1.4s	2.8s	135	39	20	21.4s	11.1s	9.1s
	findMin	1	2	0.1s	0.1s	0.1s	4	3	2	0.5s	0.4s	0.3s
		2	5	0.1s	0.2s	0.2s	19	9	5	2.2s	1.4s	1.1s
		3	20	0.3s	1.4s	2.7s	135	39	20	21.2s	11.0s	9.1s
	insert	1	4	0.2s	0.2s	0.3s	13	10	4	1.3s	1.2s	0.7s
		2	21	0.3s	0.4s	0.5s	110	58	21	8.8s	5.7s	2.8s
		3	190	1.9s	4.3s	8.4s	1591	550	190	5.1m	1.6m	36.8s
BinarySearchTree	find	1	4	0.1s	0.1s	0.1s	6	5	4	0.5s	0.4s	0.4s
		2	21	0.2s	0.2s	0.3s	51	29	21	3.0s	2.1s	1.8s
		3	236	1.3s	3.3s	7.3s	899	341	236	1.2m	27.2s	21.3s
	findMax	1	2	0.1s	0.1s	0.1s	4	3	2	0.3s	0.3s	0.2s
		2	5	0.1s	0.1s	0.2s	19	9	5	1.8s	1.1s	0.8s
		3	26	0.2s	0.7s	1.7s	171	51	26	15.5s	7.3s	5.9s
	findMin	1	2	0.1s	0.1s	0.1s	4	3	2	0.3s	0.3s	0.2s
		2	5	0.1s	0.1s	0.2s	19	9	5	1.8s	1.1s	0.8s
		3	26	0.2s	0.7s	1.7s	171	51	26	15.7s	7.1s	5.9s
	insert	1	4	0.1s	0.1s	0.1s	13	10	4	1.1s	0.9s	0.5s
		2	21	0.2s	0.3s	0.4s	110	58	21	5.8s	4.0s	2.0s
		3	236	1.3s	3.6s	7.8s	1903	682	236	2.6m	57.1s	23.8s
	remove	1	4	0.1s	0.1s	0.1s	6	5	4	0.4s	0.4s	0.3s
		2	21	0.2s	0.3s	0.3s	76	31	21	3.8s	2.1s	1.7s
		3	236	1.2s	3.2s	6.4s	2347	393	236	3.1m	29.7s	21.1s
LeftistHeap	deleteMin	1	2	0.1s	0.1s	0.1s	4	3	2	0.2s	0.2s	0.2s
		2	5	0.1s	0.1s	0.1s	22	9	5	1.6s	1.0s	0.7s
		3	25	0.3s	0.6s	0.9s	190	49	25	17.8s	6.8s	5.0s
	findMin	1	2	0.1s	0.1s	0.1s	4	3	2	0.3s	0.3s	0.2s
		2	4	0.1s	0.1s	0.1s	16	7	4	1.5s	1.0s	0.7s
		3	12	0.2s	0.5s	0.7s	78	23	12	8.6s	4.5s	3.9s
	insert	1	3	0.1s	0.1s	0.2s	6	6	3	0.7s	0.7s	0.4s
		2	8	0.1s	0.2s	0.2s	16	16	8	2.0s	2.2s	1.3s
		3	31	0.3s	0.7s	1.0s	62	62	31	10.6s	10.2s	6.1s
	merge	1	6	0.1s	0.1s	0.1s	6	6	6	0.6s	0.6s	0.6s
		2	34	0.3s	0.4s	0.6s	34	34	34	4.3s	4.0s	4.3s
		3	588	5.7s	12.0s	20.0s	588	588	588	2.8m	2.7m	3.0m
TreeMap	get	1	4	0.1s	0.1s	0.1s	6	5	4	0.4s	0.4s	0.4s
		2	28	0.2s	0.6s	0.6s	71	39	28	3.5s	2.8s	2.2s
		3	331	2.2s	18.8s	9.9s	3863	739	331	4.3m	55.7s	35.9s
	put	1	4	0.2s	0.2s	0.3s	13	10	4	1.3s	1.7s	0.7s
		2	28	0.4s	1.2s	1.0s	153	78	28	14.5s	9.5s	4.2s
		3	331	4.2s	56.9s	21.4s	5650	1481	331	24.0m	6.2m	1.3m
	remove	1	4	0.1s	0.2s	0.2s	6	5	4	0.6s	0.7s	0.5s
		2	28	0.3s	1.2s	1.1s	121	43	28	11.4s	5.2s	3.9s
		3	331	3.6s	57.5s	21.4s	4495	905	331	15.1m	3.1m	1.3m

Continued on next page

Table 1 – continued from previous page

Example		*k*	Sireum/Kiasan				Bogor/Kiasan					
Class			Paths	Time			Paths			Time		
	Method			Yn	Cn	Cp	Lz	Lr	L#	Lz	Lr	L#
GC	mark	1	306	1.3s	1.3s	1.3s	306	306	306	15.1s	17.0s	16.5s
Container	swap	1	2	0.0s	0.0s	0.0s	20	6	2	1s	0.4s	0.1s
		2	2	0.0s	0.0s	0.0s	20	6	2	1s	0.4s	0.1s
		3	2	0.0s	0.0s	0.0s	20	6	2	1s	0.4s	0.1s
BinaryHeap	deleteMin	1	2	0.0s	0.1s	0.1s	2	2	2	0.2s	0.3s	0.2s
		2	3	0.0s	0.1s	0.1s	3	3	3	0.3s	0.4s	0.3s
		3	5	0.1s	0.1s	0.1s	5	5	5	0.5s	0.6s	0.5s
	findMin	1	2	0.0s	0.0s	0.1s	2	2	2	0.2s	0.2s	0.2s
		2	3	0.0s	0.1s	0.1s	3	3	3	0.3s	0.4s	0.3s
		3	4	0.0s	0.1s	0.1s	4	4	4	0.4s	0.5s	0.4s
	insert	1	2	0.0s	0.1s	0.1s	2	2	2	0.3s	0.3s	0.3s
		2	5	0.1s	0.1s	0.2s	5	5	5	0.5s	0.6s	0.6s
		3	8	0.1s	0.2s	0.4s	8	8	8	0.7s	0.9s	0.9s
Sort	insertionSort	1	1	0.0s	0.0s	0.0s	1	1	1	0.1s	0.2s	0.1s
		2	3	0.0s	0.0s	0.1s	3	3	3	0.2s	0.3s	0.2s
		3	9	0.0s	0.1s	0.2s	9	9	9	0.8s	1.0s	0.8s
	selectionSort	1	1	0.0s	0.0s	0.0s	1	1	1	0.1s	0.2s	0.1s
		2	3	0.0s	0.0s	0.1s	3	3	3	0.3s	0.3s	0.3s
		3	10	0.1s	0.1s	0.2s	10	10	10	1.0s	1.0s	1.0s
	shellsort	1	1	0.0s	0.0s	0.1s	1	1	1	0.2s	0.2s	0.2s
		2	3	0.0s	0.1s	0.2s	3	3	3	0.4s	0.4s	0.4s
		3	9	0.1s	0.3s	0.4s	9	9	9	0.9s	1.0s	1.0s

5.3 Benchmark Experiment Using *n*-bound

To provide benchmarks for other analysis tools, we have also conducted an experimental study using Sireum/Kiasan with *n*-bounding since most similar analysis tools bound the number of heap objects. The backend we used is Yices through JNI since it is the fastest among all backends as shown in Table 1.

The result is shown in Table 2. For each example, we have collected the number of fully explored paths and running time for all the numbers of nodes, *n*, from 5 to 9.

6 Related Work

[Symbolic Execution]

The most closely related work to ours is Symbolic JPF, i.e., the SymExe extension of JPF (Khurshid et al 2003; Anand et al 2007). Kiasan's lazier and lazier# initialization algorithms are improvements over the lazy initialization algorithm of JPF (Khurshid et al 2003) as explained and empirically proved in the paper. In addition, the *k*-bounding technique described in Section 3.4 is a unique feature of Kiasan. Moreover, we have introduced type variables to completely cover the subtyping issue which was not sufficiently covered by Symbolic JPF.

Tools such as Pex (Tillmann and de Halleux 2008) and XRT (Grieskamp et al 2005) represent the heap as pure logic formula and thus require decision procedures (DP) that are able to handle heap structures. In contrast, our algorithms maintain a graphical representation of the visible part of the heap and do not need a decision procedure for heap structures. In fact, our algorithms can be viewed as an algorithmic procedure that implements the functionality of heap structure handling capability of DP. Besides a logic state representation, Smallfoot by Berdine et al (2005) and jStar by Distefano and Parkinson (2008) provide support for

Table 2 Experimental Data Using n-bound (P – Paths, T – Time)

Example		$n=5$		$n=6$		$n=7$		$n=8$		$n=9$	
Class	**Method**	**P**	**T**	**P**	**T**	**P**	**T**	**P**	**T**	**P**	**T**
AATree	contains	56	1.9s	95	3.2s	155	4.8s	240	7.1s	392	10.8s
	findMax	8	1.0s	11	1.7s	15	2.5s	20	3.3s	28	4.5s
	findMin	8	1.0s	11	1.8s	15	2.4s	20	3.4s	28	4.5s
	insert	56	2.1s	95	3.4s	155	4.9s	240	7.6s	392	12.0s
	isEmpty	8	0.9s	11	1.6s	15	2.3s	20	3.2s	28	4.3s
	remove	56	2.1s	95	3.3s	155	4.9s	240	7.2s	392	11.4s
ArrayDeque	addFirst	44	1.2s	77	1.6s	119	2.1s	179	2.8s	251	3.7s
	addLast	44	1.2s	77	1.5s	119	2.1s	179	2.7s	251	3.6s
	isEmpty	56	1.0s	92	1.4s	141	1.8s	205	2.3s	286	3.0s
	removeFirst	56	1.2s	92	1.6s	141	2.1s	205	2.9s	286	3.9s
	removeLast	56	1.3s	92	1.7s	141	2.3s	205	3.0s	286	4.0s
ArrayList	add	6	0.3s	7	0.3s	8	0.3s	9	0.4s	10	0.4s
	get	2	0.2s	2	0.2s	2	0.2s	2	0.2s	2	0.2s
	isEmpty	2	0.1s	2	0.1s	2	0.1s	2	0.1s	2	0.1s
	remove	6	0.4s	7	0.4s	8	0.4s	9	0.5s	10	0.5s
AvlTree	find	123	3.5s	175	4.8s	430	12.4s	974	26.9s	1810	55.4s
	findMax	15	1.4s	19	1.6s	36	3.9s	68	6.3s	112	9.7s
	findMin	15	1.4s	19	1.6s	36	3.9s	68	6.3s	112	9.7s
	insert	123	4.0s	175	5.2s	430	12.9s	974	29.1s	1810	1.0m
	isEmpty	15	1.1s	19	1.3s	36	3.4s	68	5.3s	112	8.0s
BinarySearchTree	find	637	7.2s	2353	26.5s	8788	1.8m	33098	8.6m	125476	41.8m
	findMax	65	1.9s	197	4.1s	626	10.8s	2056	35.6s	6918	2.3m
	findMin	65	1.9s	197	4.1s	626	11.0s	2056	38.0s	6918	2.2m
	insert	637	7.9s	2353	27.4s	8788	1.9m	33098	8.3m	125476	39.8m
	isEmpty	65	1.6s	197	3.3s	626	8.4s	2056	27.2s	6918	1.6m
	remove	637	6.9s	2353	25.8s	8788	1.7m	33098	7.9m	125476	41.5m
LinkedList	add	6	0.4s	7	0.4s	8	0.5s	8	0.5s	8	0.5s
	contains	97	2.4s	147	5.2s	212	19.2s	212	19.2s	212	19.6s
	get	27	0.7s	35	0.8s	44	0.9s	44	1.0s	44	1.0s
	getFirst	6	0.3s	7	0.4s	8	0.4s	8	0.4s	8	0.4s
	getLast	6	0.3s	7	0.4s	8	0.4s	8	0.4s	8	0.4s
	isEmpty	6	0.3s	7	0.3s	8	0.3s	8	0.3s	8	0.3s
	remove	27	0.7s	35	0.8s	44	1.0s	44	1.0s	44	1.0s
	removeFirst	6	0.4s	7	0.4s	8	0.4s	8	0.5s	8	0.5s
	removeLast	6	0.3s	7	0.4s	8	0.4s	8	0.4s	8	0.4s
PriorityQueue	isEmpty	6	0.3s	7	0.3s	8	0.3s	9	0.3s	10	0.4s
	offer	25	0.7s	31	0.8s	38	0.9s	46	1.0s	54	1.1s
	peek	6	0.3s	7	0.3s	8	0.3s	9	0.4s	10	0.4s
	poll	12	0.5s	19	0.6s	27	0.7s	35	0.9s	44	1.0s
Stack	isEmpty	2	0.1s	2	0.1s	2	0.1s	2	0.1s	2	0.1s
	peek	2	0.2s	2	0.2s	2	0.2s	2	0.2s	2	0.2s
	pop	2	0.3s	2	0.3s	2	0.3s	2	0.3s	2	0.3s
	push	6	0.3s	7	0.3s	8	0.4s	9	0.4s	10	0.4s
TreeMap	get	152	4.0s	360	9.0s	855	20.0s	1807	43.8s	3517	1.5m
	isEmpty	18	1.5s	34	3.6s	67	5.9s	123	10.1s	213	17.1s
	lastKey	18	1.7s	34	3.9s	67	6.4s	123	11.0s	213	18.3s
	put	152	4.6s	360	9.8s	855	23.4s	1807	47.9s	3517	1.6m
	remove	152	4.3s	360	9.9s	855	20.7s	1807	48.0s	3517	1.6m
TreeSet	add	152	5.6s	360	11.0s	855	25.0s	1807	54.1s	3517	1.9m
	contains	152	5.1s	360	10.1s	855	24.2s	1807	50.2s	3517	1.8m
	isEmpty	18	2.6s	34	3.9s	67	6.5s	123	11.1s	213	19.2s
	remove	152	5.6s	360	10.4s	855	23.3s	1807	55.1s	3517	1.8m
Vector	add	11	0.4s	13	0.5s	15	0.5s	17	0.5s	19	0.6s
	get	2	0.2s	2	0.2s	2	0.2s	2	0.2s	2	0.2s
	isEmpty	2	0.1s	2	0.1s	2	0.1s	2	0.1s	2	0.1s

separation logic (Reynolds 2002). These tools also require special decision procedures that can handle separation logic expressions.

There is a an interesting approach called concolic execution where symbolic and concrete executions are applied simultaneously, and is demonstrated by tools such as CUTE (Sen and Agha 2005) and Pex (Tillmann and de Halleux 2008). The approach uses the concrete execution to cover branches and the symbolic execution to guide the concrete execution to cover different branches. Essentially, the approach runs the program to be tested multiple times with random inputs for the first run. Then, symbolic execution is used to generate inputs for next concrete execution to cover different branches. For example, given a condition `x!=3`, one concrete execution starts with a random input such as `x=1` and the true branch is covered; then the technique takes the path condition from the symbolic execution, negates it, calls a constraint solver with the negation of the path condition which is $\neg(x \neq 3)$, and gets `x=3`; the next concrete execution will use that input to cover the false branch. The key advantage of the approach is that the concrete execution can assist the symbolic execution for solving certain types of complex arithmetic expressions by replacing symbolic expressions with concrete values. For example, given a condition `y==h(x)` where `x` and `y` are parameters and `h(x)` is a difficult expression for decision procedures, for example, `h(x)=x*x*x*x`. If one concrete execution uses input, say `x=2` and `y=1`, the condition clearly is false. To make the expression true, the symbolic execution replaces `h(x)=x*x*x*x` with `h(2)=16`, that is, `x=2` is used to compute the value of `y` and get `y=h(2)=16`. Therefore, next concrete execution with `x=2` and `y=16` will make the expression true. Similarly, if `h(x)` is native code, the approach works as well. However, this approach does not work for all types of the expressions, for example, it could not solve expression `g(x)==h(x)`, where `g,h` are complex expressions/native code. We believe that this concolic approach can be adapted to our algorithms to handle native code and complex arithmetic.

[Model Checking]

The closest model checking (Clarke et al 2000) approach to Kiasan is explicit-state model checking (which we abbreviate as model checking below) using depth-first exploration strategy: both of them perform a forward path-sensitive analysis and can check temporal properties[8]. Model checking can be classified into two categories: stateless and stateful, depending on whether it stores states. Our SymExe can be seen as a stateless model checking. As a matter of fact, the initial version of Kiasan, Bogor/Kiasan, was built on top of our home-grown software model checker, Bogor (Robby et al 2003). Despite the similarity, there are two major differences between model checking and Kiasan SymExe algorithms:

– Model checking can only work on closed systems, that is, it needs some driver or environment to analyze a module. SymExe is designed for systems with unknown data: it uses symbols for representing unknown values.
– SymExe is more abstract than model checking (that uses no abstractions) since it manipulates symbols instead of concrete values. Furthermore, each path in SymExe corresponds to many concrete paths in model checking.

Another closely related approach is bounded model checking (Biere et al 1999, 2003). Bounded model checkers (BMC) such as CBMC (Clarke et al 2004) and SATURN (Xie

[8] Temporal properties (safety and bounded liveness) can be added in Kiasan by monitoring the finite state automata constructed from the temporal properties similar to the translation by Geilen (2001).

and Aiken 2007) take the approach that directly translates C programs into SAT formulas and leverages the recent technical advancements in propositional SAT solvers such as SATO (Zhang 1997) and CHAFF (Moskewicz et al 2001). To enable the translation, the tools bound loops and recursions. In contrast, Kiasan bounds data first and then loops and recursions.

There have been other BMC techniques and tools, such as Alloy (Jackson 2002), TestEra (Marinov and Khurshid 2001), Korat (Boyapati et al 2002), PIPAL (Darga and Boyapati 2006; Roberson and Boyapati 2010) that bound on data. Similarly to Kiasan, they exhaustively explore some bounded search space. Furthermore, they use various state space reduction techniques to scale to larger bounds. In particular, PIPAL leverages static and dynamic analysis that allow it to safely ignore many states that are similar to the state currently been checked. The most important difference between Kiasan and those tools is the state representation: Kiasan lazily expands the heap while those tools fix the number of objects and their types as well as restricting the range of values for scalars. In addition, Kiasan does not perform state subsumption checking since it may be too expensive, while those tools can compare states since they work on a fixed number of objects where all scalar ranges are restricted. Furthermore, extra effort is required to avoid isomorphic states in those tools, while Kiasan always only considers nonisomorphic states due to the way it expands heap objects.

There is another interesting bounded approach, UDITA (Gligoric et al 2010), which exhaustively explores all the behaviors of test generation programs within some bound using JPF. In contrast to Kiasan which is more of a white-box approach, UDITA is more of a black-box approach that uses some form of specification language to generate test inputs. The underlying algorithm leverages the notion of delayed choice and postpones the choice until it is accessed which, in spirit, is similar to Kiasan's lazier and lazier# algorithms.

[Shape Analysis]

One may think that the analysis performed by our SymExe is very similar to shape analysis though the goal of our analysis is not restricted to searching for the shapes of data structures. Shape analysis searches for a shape-wise invariant at each program point. To cope with potentially infinite number of shapes a program point can have, shape analysis either confines the size of a shape graph or represents a shape with a finite number of access paths. The former, as was done by Chase et al (1990) and Sagiv et al (2002) among many others, lumps the nodes of a shape graph, that are indistinguishable from each other with respect to a certain perspective, into a single node usually called *summary node*. In the latter that was used by Larus and Hilfinger (1988) and Deutsch (1994) among many others, an access path represents all the paths of shape graphs that satisfy a certain access pattern, and as a result, infinite number of shape graphs can be summarized with finite number of access paths. At first glance, a nonprimitive symbol (in short, a symbol from now on) of our analysis such as a symbolic reference and a symbolic object seems to resemble a summary node of shape analysis. Also, those who are familiar with the k-limiting of Jones and Muchnick (1979) may think that our k-bounding is similar to it. They are, however, more opposite than similar to each other.

First, we compare a symbol to a summary node. A symbol intends to represent either NULL or a *single* location of the heap[9]; in contrast, a summary node represents *all* locations

[9] We explain the difference between a symbol and a summary node from the perspective of the lazier# version for the sake of simplicity. Hence a symbol here refers to a symbolic reference.

in the heap that are indistinguishable from each other. This fundamental difference between a symbol and a summary node leads to the difference in the precision of the analysis. The use of summary nodes is the main reason why shape analysis suffers from the loss of precision and is property-dependent (a summarization may work for a property but may not well-suited for a different property), though it also plays an important role in guaranteeing the termination of the analysis without sacrificing the conservativeness of the analysis result. On the other hand, our analysis is precise with respect to any property that a user wants to check because we resolve object-aliasing through case-splitting and never summarize object properties. That is, in our analysis, the non-symbolic portion of a shape graph consisting of non-symbolic locations and arcs between them depicts a precise shape of data structures while symbols characterize unknown parts of data structures. Such a shape graph reveals partial but precise information about the shape of data structures. This is in contrast to a shape graph that is an inclusive but imprecise approximation (i.e., a conservative overapproximation) of all possible shapes.

On the same line of thought, a symbol resolution and the materialization of a summary node are more different than they look. Materialization, suggested by Chase et al (1990), is an effective technique to improve the precision of shape analysis by splitting a summary node, when necessary, into a non-summary node (i.e., a materialized node) and a new summary node that represents an original summary node modulo the materialized node. Although such materialization bears resemblance to the symbol resolution (during lazy initialization) of our analysis, the effects are not the same. A shape graph obtained after materializing a summary node is still an overapproximation, and the improved precision may not be sufficient. This is in contrast to the symbol resolution of our analysis that is always precise assuming that the underlying theorem prover solves the given constraints correctly. Recall that our analysis imposes constraints such as type constraints to exclude infeasible resolutions when resolving a symbol.

The difference between our k-bounding and the k-limiting of Jones and Muchnick can be viewed from the same precision perspective as well. Their k-limiting limits the lengths of node paths of a given shape graph to k. A path longer than k is shrunk to a shorter path that contains one or more of an abstract node which they call an unknown node. Such an unknown node is closer to a summary node than to our symbol in a sense that it may represent more than one node and can point to another node without having to know which node it represents. Therefore, despite the seeming similarity of k-limiting to k-bounding, k-limiting provides overapproximate information about shapes with k-limiting graphs on the contrary that k-bounding coupled with our lazier# initialization algorithm provides precise shape information up to the k bound.

Another main difference of our analysis from conventional shape analysis is the path-sensitivity of SymExe. Most shape analysis methods are based on path-insensitive data flow analysis to guarantee termination. Path-insensitivity is another major cause of losing the precision of analysis. Once again, our analysis has its strength in the precision point.

The trade-off in guaranteeing precision is that our analysis does not terminate on its own if a module under analysis contains, for example, an iterative statement such as a loop. We take a practical approach and address this problem with the bounding technique; shapes beyond a user-specified bound are not considered. Conversely, most shape analysis methods are guaranteed to terminate with all possible shapes considered.

In short, our analysis and conventional shape analysis have different trade-off in terms of the precision and the scope of shapes considered by analysis. Our analysis supports accurate precision for any kind of property, but the analysis scope is limited. Meanwhile, shape analysis guarantees full scope analysis but is property-dependent and lacks precision.

The weakness of an analysis is often compensated for by providing additional information. To improve the precision of shape analysis, additional information such as node sharing, reachability and acyclicity is often exploited in many shape analysis methods. If pre-defined additional information cannot improve the precision enough to filter out false alarms, a user has to provide appropriate additional information as a last resort as was done by Sagiv et al (2002) with instrumentation predicates. Meanwhile, to extend the shape scope of SymExe beyond the bound a bounding technique provides, inductive assertions such as loop invariants should be exploited. Simple patterns of assertions can be inferred automatically. If an automatic assertion inference fails or an obtained assertion is not precise enough, a user has to provide an appropriate one. Overall, a user sometimes needs to intervene and provide additional necessary information in both analysis.

[Java Formal Semantics]

There have been many studies, e.g., Alves-Foss (1999), Drossopoulou and Eisenbach (1998), Bertelsen (2000), Gligoric et al (2010) on formal semantics of Java: Drossopoulou and Eisenbach (1998) provide Java source code semantics to show the type soundness of the language; Bertelsen (2000) defines a formal semantics of JVM bytecode; PIPAL (Gligoric et al 2010) gives a formal semantics of symbolic execution of a Java-like language. The Kiasan's bytecode semantics are very similar to the above operational semantics, but using the semantics to prove the correctness of generalized symbolic executions is unique to this work.

7 Conclusion and Future Work

This paper addresses two unresolved issues of the lazy initialization algorithm: relative inefficiency and lack of solid theoretical foundation. For the first issue, we have described two improved algorithms (lazier and lazier#) that are more efficient than the lazy initialization algorithm. In addition, the improved algorithms have a complete coverage of the Liskov substitution principle which was covered insufficiently by the original lazy initialization algorithm. For the second issue, we have formalized the lazy initialization algorithm as well as the two improved algorithms on a core subset of JVM instructions and proved the relative soundness and completeness of the three algorithms. Furthermore, the algorithms have been realized in the Kiasan framework under the guidance of the formal semantics. Our experimental data on realistic benchmarks show more than ten times reduction of the lazier and lazier# initialization algorithms over the lazy initialization algorithm in terms of analysis time and the number of explored paths, and hence demonstrate the efficiency of our algorithms.

We have two directions of future work: modular/contract reasoning and abstraction. In modular reasoning, contracts can be used to substitute program components. This would allow the tool to scale to larger systems, as the systems can be divided into smaller units more amenable for analysis. We would also like to introduce abstractions to handle commonly used data structures and their properties. For example, when analyzing Java programs using strings, it would be more efficient to use string models/abstractions/theories (Hopcroft and Ullman 1979) than to use the actual `java.lang.String` class implementation in the standard Java library. In short, we believe that both contracts and customized abstract models would be crucial to further scale SymExe.

Acknowledgements This work was supported in part by the US National Science Foundation (NSF) award 0709169 and CAREER award 0644288, the US Air Force Office of Scientific Research (AFOSR), and Rockwell Collins. The second author was also partially supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grant 2010-0001727), and a Korea University Grant. We would also like to thank Kathy Brode for proofreading the draft of this article.

References

Alves-Foss J (ed) (1999) Formal Syntax and Semantics of Java, Lecture Notes in Computer Science, vol 1523, Springer

Anand S, Pasareanu CS, Visser W (2006) Symbolic execution with abstract subsumption checking. In: Valmari A (ed) Model Checking Software, 13th International SPIN Workshop, Vienna, Austria, March 30 - April 1, 2006, Proceedings, Springer, Lecture Notes in Computer Science, vol 3925

Anand S, Orso A, Harrold MJ (2007) Type-dependency analysis and program transformation for symbolic execution. In: Tools and Algorithms for Construction and Analysis of Systems (TACAS)

Barrett C, Tinelli C (2007) CVC3. In: Damm W, Hermanns H (eds) Proceedings of Computer Aided Verification, 19th International Conference, CAV 2007, Springer, Lecture Notes in Computer Science, vol 4590, pp 298–302

Berdine J, Calcagno C, O'Hearn PW (2005) Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer FS, Bonsangue MM, Graf S, de Roever WP (eds) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Springer, Lecture Notes in Computer Science, vol 4111, pp 115–137

Bertelsen P (2000) Dynamic semantics of java bytecode. Future Gener Comput Syst 16:841–850

Biere A, Cimatti A, Clarke EM, Zhu Y (1999) Symbolic model checking without BDDs. In: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99), Springer-Verlag, 193-207, vol 1579, p LNCS

Biere A, Cimatti A, Clarke EM, Strichman O, Zhu Y (2003) Bounded model checking. Advances in Computers 58:117–148

Boyapati C, Khurshid S, Marinov D (2002) Korat: automated testing based on Java predicates. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), ACM, pp 123–133

Brat G, Havelund K, Park S, Visser W (2000) Java PathFinder – a second generation of a Java model-checker. In: Proceedings of the Workshop on Advances in Verification

Chase DR, Wegman M, Zadeck FK (1990) Analysis of pointers and structures. In: Proceedings of the conference on programming language design and implementation (PLDI'90), pp 296–310

Clarke E, Grumberg O, Peled D (2000) Model Checking. MIT Press, Cambridge, MA, USA

Clarke EM, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), Springer-Verlag, LNCS, vol 2988, pp 168–176

Cook SA (1978) Soundness and completeness of an axiom system for program verification. SIAM Journal on Computing 7(1):70–90

Darga PT, Boyapati C (2006) Efficient software model checking of data structure properties. In: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, ACM, New York, NY, USA, OOPSLA '06, pp 363–382

Deng X (2007) Contract-based verification and test case generation for open systems. PhD thesis, Kansas State University

Deng X, Lee J, Robby (2006) Bogor/Kiasan: A *k*-bounded symbolic execution for checking strong heap properties of open systems. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE06), IEEE Computer Society, pp 157–166

Deng X, Robby, Hatcliff J (2007a) Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems. In: Testing: Academic and Industrial Conference – Practice and Research Techniques (TAIC-PART07)

Deng X, Robby, Hatcliff J (2007b) Towards a case-optimal symbolic execution algorithm for analyzing strong properties of object-oriented programs. In: Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM), IEEE Computer Society, London, UK, pp 273–282

Deng X, Walker R, Robby (2010) Case counting analysis for path-sensitive bounded verification techniques on standard data structure operations. Tech. Rep. SAnToS-TR2010-01-19, Kansas State University

Deutsch A (1994) Interprocedural may-alias analysis for pointers: beyond *k*-limiting. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI'94), pp 230–241

Distefano D, Parkinson MJ (2008) jStar: Towards practical verification for Java. In: OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, ACM, pp 213–226

Drossopoulou S, Eisenbach S (1998) Towards an operational semantics and proof of type soundness for java. In: In Formal Syntax and Semantics of Java, Springer-Verlag

Dutertre B, de Moura L (2006) The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf

Geilen M (2001) On the construction of monitors for temporal logic properties. Electr Notes Theor Comput Sci 55(2)

Gligoric M, Gvero T, Jagannath V, Khurshid S, Kuncak V, Marinov D (2010) Test generation through programming in udita. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ACM, New York, NY, USA, ICSE '10, pp 225–234

Grieskamp W, Tillmann N, Schulte W (2005) XRT - exploring runtime for .NET - architecture and applications. Workshop on Software Model Checking (SoftMC05)

Hantler SL, King JC (1976) An introduction to proving the correctness of programs. ACM Computing Surveys (CSUR) 8(3):331–353

Hopcroft JE, Ullman JD (1979) Introduction to automata theory, languages, and computation, 1st edn. Addison-Wesley

Jackson D (2002) Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology (TOSEM) 11(2):256 – 290

Jones ND, Muchnick SS (1979) Flow analysis and optimization of LISP-like structures. In: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'79), ACM Press, pp 244–256

Khurshid S, Păsăreanu CS, Visser W (2003) Generalized symbolic execution for model checking and testing. In: Garavel H, Hatcliff J (eds) Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings, Springer, Lecture Notes in Computer Science, vol 2619, pp 553–568

King JC (1976) Symbolic execution and program testing. Communications of the ACM 19(7):385–394

Larus JR, Hilfinger PN (1988) Detecting conflicts between structure accesses. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI'88), pp 24–31

Leavens GT, Baker AL, Ruby C (1998) JML: a Java modeling language. In: Formal Underpinnings of Java Workshop (at OOPSLA'98), ACM

Lev-Ami T, Sagiv M (2000) TVLA: A framework for kleene-based static analysis. In: Proceedings of the 7th International Static Analysis Symposium (SAS), Springer, Lecture Notes in Computer Science, vol 1694, pp 280–301

Lindholm T, Yellin F (1999) The Java virtual machine specification (2nd edition). Http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

Marinov D, Khurshid S (2001) TestEra: A novel framework for automated testing of Java programs. In: 16th IEEE Conference on Automated Software Engineering (ASE 2001), IEEE Computer Society, p 22

McCarthy J (1962) Towards a mathematical science of computation. Information Processing 62:21–28

Moskewicz MW, Madigan CF, Zhao Y, Zhang L, Malik S (2001) Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th conference on Design automation, ACM Press, pp 530–535

de Moura LM, Bjørner N (2008) Z3: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS08, Springer, Lecture Notes in Computer Science, vol 4963, pp 337–340

MS (2006) Common language infrastructure (CLI). Standard ECMA-335

Păsăreanu CS, Visser W (2004) Verification of Java programs using symbolic execution and invariant generation. SPIN Workshop pp 164–181

Ramalingam G (1994) The undecidability of aliasing. ACM Transactions on Programming Languages and Systems (TOPLAS) 16(5):1467–1471

Reynolds J (2002) Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), IEEE Computer Society, pp 55–74

Robby (2008) Sireum: a software analysis platform. `http://sireum.org`

Robby, Dwyer MB, Hatcliff J (2003) Bogor: An extensible and highly-modular model checking framework. In: Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM, pp 267–276

Roberson M, Boyapati C (2010) Efficient modular glass box software model checking. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications, ACM, New York, NY, USA, OOPSLA '10, pp 4–21

Sagiv M, Reps T, Wilhelm R (2002) Parametric shape analysis via 3-valued logic. TOPLAS 24(3):217–298, a preliminary version appeared in POPL 1999, pages 105–118

Schmidt D (2000) Binary relations for abstraction and refinement. Tech. rep., Kansas State University

Sen K, Agha G (2005) CUTE: A concolic unit testing engine for C. In: Wermelinger M, Gall H (eds) ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), ACM, pp 263–272

Tillmann N, de Halleux J (2008) Pex–white box test generation for .NET. In: Beckert B, Hähnle R (eds) Tests and Proofs, 2nd International Conference (TAP08), Springer, Lecture Notes in Computer Science, vol 4966, pp 134–153

Visser W, Pasareanu CS, Khurshid S (2004) Test input generation in Java Pathfinder. In: Avrunin GS, Rothermel G (eds) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004, ACM, pp 97–107

Weiss MA (2006) Data Structures and Algorithm Analysis in Java, 2nd edn. Addison-Wesley

Xie Y, Aiken A (2007) SATURN: A scalable framework for error detection using boolean satisfiability. ACM Transactions on Programming Languages and Systems (TOPLAS) 29(3)

Zhang H (1997) SATO: An efficient prepositional prover. In: Proceedings of the International Conference on Automated Deduction, Springer, LNCS, vol 1249, pp 272–275

Program 3 The bytecode of the `swap` method

```
public void swap(Container);
  Code:
   0:    aload_0
   1:    getfield        #2; //Field data:Ljava/lang/Object;
   4:    astore_2
   5:    aload_0
   6:    aload_1
   7:    getfield        #2; //Field data:Ljava/lang/Object;
   10:   putfield        #2; //Field data:Ljava/lang/Object;
   13:   aload_1
   14:   aload_2
   15:   putfield        #2; //Field data:Ljava/lang/Object;
   18:   return
```

APPENDIX

A Formalization of the `swap` Example

A.1 Bytecode Execution of `swap` in JVM

We briefly walk through the execution of the bytecode instructions of the `swap` method. The two most important components of the state of JVM are the operand stack and the local variable array. Before a method execution starts, the operand stack is empty. The local variable array is initialized with two values corresponding to the method's two parameters (including the implicit parameter `this`): the value of `this` is stored at array index 0, `n` at index 1. Note local index 2 is reserved for local variable `e`. The Java statement `e = data` is translated into instructions 0, 1, and 4 of Program 3. Instruction 0 (`aload_0`) loads the value of `this` onto the operand stack; instruction 1 (`getfield #2`) reads the value of `this.data` and pushes it onto the operand stack; instruction 4 (`astore_2`) stores the value of `this.data` to the local variable at index 2 which corresponds to variable `e`.

Instructions 5, 6, 7, and 10 correspond to the Java statement `data = n.data`. Instruction 5 (`aload_0`) is the same as instruction 0; instruction 6 (`aload_1`) loads the value of `n` onto the operand stack; instruction 7 (`getfield #2`) loads the value of `n.data` onto the operand stack; instruction 10 (`putfield #2`) writes the value on the top of the stack (i.e., `n.data`) to `this.data`.

Finally, statement `n.data = e` is translated into instructions 13, 14, and 15. Instruction 13 (`aload_1`) loads the value of `n` onto the operand stack; instruction 14 (`aload_2`) loads the value of `e`, which is equal to the value of `this.data` at this time, onto the operand stack; instruction 15 (`putfield #2`) writes the top value from the stack (i.e., the value of `e`) into `n.data`. Now, the swap of `this.data` and `n.data` is done.

A.2 Formalization of States

To illustrate the formalization of states in SEL, SELA, and SELB, we use the `swap` method shown in Program 1 as an example. Recall that the bytecode of the `swap` method is shown in Program 3. We pick one state from each semantics: state 33 in Figure 2, state 22 in Figure 3, and state 11 in Figure 4 for SEL, SELA, and SELB respectively. All the three states are the states after executing the statement `e = data` which corresponds to bytecode instructions 0, 1, and 4. Since the program counter of each state will point to the next instruction to be executed, the program counters of all states should be 5.

State 33 in the lazy symbolic execution tree of the `swap` example can be formalized as follows:

- globals, since the `swap` method does not refer to any static field, then we let the global component be empty: $\emptyset$.
- program counter, $pc = 5$.
- locals, there are two parameters, `this` and `n`, and one local variable, `e`. So locals = $\{0 \mapsto l_0, 1 \mapsto l_1, 2 \mapsto l_1\}$ where l_0 and l_1 are heap locations of α_0 and α_1. Recall that we view a function as a set of pairs.
- stack, the stack is empty, *nil*.
- heap, there are two objects in the heap: $\alpha_0 = \{data \mapsto l_1\}_{\tau_0}$ and $\alpha_1 = \{\}_{\tau_1}$. So the heap = $\{l_0 \mapsto \{data \mapsto l_1\}_{\tau_0}, l_1 \mapsto \{\}_{\tau_1}\}$, where l_0 and l_1 are two arbitrary locations satisfying $l_0 \neq l_1$.

– path condition, there are two type constraints in the path condition: $\phi = \{\tau_0 <: \mathit{Container}, \tau_1 <: \mathit{Container}\}$.

Therefore, the formalization of state 33 is:

$$(\emptyset, 5, \{0 \mapsto l_0, 1 \mapsto l_1, 2 \mapsto l_1\}, \mathit{nil}, \{l_0 \mapsto \{\mathit{data} \mapsto l_1\}_{\tau_0}, l_1 \mapsto \{\}_{\tau_1}\}, \phi),$$

where $\phi = \{\tau_0 <: \mathit{Container}, \tau_1 <: \mathit{Container}\}$.

Similarly, the formalization of state 22 in the lazier symbolic execution tree of the `swap` example shown in Figure 3 is:

$$(\emptyset, 5, \{0 \mapsto l_0, 1 \mapsto \hat{\alpha}_1, 2 \mapsto \hat{\beta}_0\}, \mathit{nil}, \{l_0 \mapsto \{\mathit{data} \mapsto \hat{\beta}_0\}_{\tau}\}, \{\tau <: \mathit{Container}\}),$$

where $\hat{\alpha}_1$ has the type of `Container` and $\hat{\beta}_0$ has the type of `Object`.

The formalization of state 11 in the lazier# symbolic execution tree of the `swap` example shown in Figure 4 is:

$$(\emptyset, 5, \{0 \mapsto l_0, 1 \mapsto \hat{\alpha}_1, 2 \mapsto \bar{\beta}_0\}, \mathit{nil}, \{l_0 \mapsto \{\mathit{data} \mapsto \bar{\beta}_0\}_{\tau}\}, \{\tau <: \mathit{Container}\}),$$

where $\hat{\alpha}_1$ has the type of `Container` and $\bar{\beta}_0$ has the type of `Object`.

A.3 Formalization of Initial States

We will use the `swap` method shown in Program 1 as an example to show the formalizations of initial states in SEL, SELA, and SELB.

In SEL, there are three nonisomorphic initial states:

1. State 1 in Figure 2, $(\emptyset, 0, \{0 \mapsto l, 1 \mapsto \text{NULL}\}, \mathit{nil}, \{l \mapsto \emptyset_{\tau}\}, \{\tau <: \mathit{Container}\})$.
2. State 2 in Figure 2, $(\emptyset, 0, \{0 \mapsto l, 1 \mapsto l\}, \mathit{nil}, \{l \mapsto \emptyset_{\tau}\}, \{\tau <: \mathit{Container}\})$.
3. State 3 in Figure 2, $(\emptyset, 0, \{0 \mapsto l_0, 1 \mapsto l_1\}, \mathit{nil}, \{l_0 \mapsto \emptyset_{\tau_0}, l_1 \mapsto \emptyset_{\tau_1}\}, \phi)$, where $\phi = \{\tau_0 <: \mathit{Container}, \tau_1 <: \mathit{Container}\}$.

In SELA, there are two nonisomorphic initial states:

1. State 1 in Figure 3, $(\emptyset, 0, \{0 \mapsto \hat{\alpha}_0, 1 \mapsto \text{NULL}\}, \mathit{nil}, \emptyset, \emptyset)$, where the type of $\hat{\alpha}_0$ is `Container`. Note that there are three empty sets that appear in the state: globals, the heap, and the path condition. Each empty set has different meaning. The global component is an empty function, that is, the domain set is empty. The heap is a partial function that has nothing defined yet. The path condition has no formula yet which means TRUE.
2. State 2 in Figure 3, $(\emptyset, 0, \{0 \mapsto \hat{\alpha}_0, 1 \mapsto \hat{\alpha}_1\}, \mathit{nil}, \emptyset, \emptyset)$, where $\hat{\alpha}_0$ and $\hat{\alpha}_1$ have the same type, `Container`.

In SELB, there is only one nonisomorphic initial state, $(\emptyset, 0, \{0 \mapsto \hat{\alpha}_0, 1 \mapsto \bar{\alpha}_1\}, \mathit{nil}, \emptyset, \emptyset)$. After applying the *NV* optimization, we get the State 1 in Figure 4, $(\emptyset, 0, \{0 \mapsto \hat{\alpha}_0, 1 \mapsto \hat{\alpha}_1\}, \mathit{nil}, \emptyset, \emptyset)$.

A.4 Formalization of a Lazy Trace

Figure 13 shows the formalization of the highlighted trace (i.e., trace 3-33-334-3341) in Figure 2. Recall that the bytecode of the `swap` example is listed in Program 3.

A.5 Formalization of a Lazier Trace

Figure 14 shows the formalization of the highlighted trace (i.e., trace 2-22-223-2231) in Figure 3.

A.6 Formalization of a Lazier# Trace

Figure 15 shows the formalization of the highlighted trace (i.e., trace 1-11-112-1121) in Figure 4.

$$\begin{aligned}
&(\emptyset, 0, \{0 \mapsto l_0, 1 \mapsto l_1\}, nil, \{l_0 \mapsto \emptyset_{\tau_0}, l_1 \mapsto \emptyset_{\tau_1}\}, \phi)\\
&\overset{AL}{\Rightarrow}_{\mathscr{S}} (\emptyset, 1, \{0 \mapsto l_0, 1 \mapsto l_1\}, l_0 :: nil, \{l_0 \mapsto \emptyset_{\tau_0}, l_1 \mapsto \emptyset_{\tau_1}\}, \phi)\\
&\overset{G4}{\Rightarrow}_{\mathscr{S}} (\emptyset, 4, \{0 \mapsto l_0, 1 \mapsto l_1\}, l_1 :: nil, \{l_0 \mapsto \{data \mapsto l_1\}_{\tau_0}, l_1 \mapsto \emptyset_{\tau_1}\}, \phi)\\
&\overset{AS}{\Rightarrow}_{\mathscr{S}} (\emptyset, 5, \{0 \mapsto l_0, 1 \mapsto l_1, 2 \mapsto l_1\}, nil, \{l_0 \mapsto \{data \mapsto l_1\}_{\tau_0}, l_1 \mapsto \emptyset_{\tau_1}\}, \phi)\\
&\overset{AL}{\Rightarrow}_{\mathscr{S}} (\emptyset, 6, \{0 \mapsto l_0, 1 \mapsto l_1, 2 \mapsto l_1\}, l_0 :: nil, \{l_0 \mapsto \{data \mapsto l_1\}_{\tau_0}, l_1 \mapsto \emptyset_{\tau_1}\}, \phi)\\
&\overset{AL}{\Rightarrow}_{\mathscr{S}} (\emptyset, 7, \{0 \mapsto l_0, 1 \mapsto l_1, 2 \mapsto l_1\}, l_1 :: l_0 :: nil, \{l_0 \mapsto \{data \mapsto l_1\}_{\tau_0}, l_1 \mapsto \emptyset_{\tau_1}\}, \phi)\\
&\overset{G6}{\Rightarrow}_{\mathscr{S}} (\emptyset, 10, \{0 \mapsto l_0, 1 \mapsto l_1, 2 \mapsto l_1\}, l_2 :: l_0 :: nil, \{l_0 \mapsto \{data \mapsto l_1\}_{\tau_0},\\
&\qquad l_1 \mapsto \{data \mapsto l_2\}_{\tau_1}, l_2 \mapsto \emptyset_{\tau_2}\}, \phi \cup \{\tau_2 <: Object\})\\
&\overset{P1}{\Rightarrow}_{\mathscr{S}} (\emptyset, 13, \{0 \mapsto l_0, 1 \mapsto l_1, 2 \mapsto l_1\}, nil, \{l_0 \mapsto \{data \mapsto l_2\}_{\tau_0}, l_1 \mapsto \{data \mapsto l_2\}_{\tau_1},\\
&\qquad l_2 \mapsto \emptyset_{\tau_2}\}, \phi \cup \{\tau_2 <: Object\})\\
&\overset{AL}{\Rightarrow}_{\mathscr{S}} (\emptyset, 14, \{0 \mapsto l_0, 1 \mapsto l_1, 2 \mapsto l_1\}, l_1 :: nil, \{l_0 \mapsto \{data \mapsto l_2\}_{\tau_0}, l_1 \mapsto \{data \mapsto l_2\}_{\tau_1},\\
&\qquad l_2 \mapsto \emptyset_{\tau_2}\}, \phi \cup \{\tau_2 <: Object\})\\
&\overset{AL}{\Rightarrow}_{\mathscr{S}} (\emptyset, 15, \{0 \mapsto l_0, 1 \mapsto l_1, 2 \mapsto l_1\}, l_1 :: l_1 :: nil, \{l_0 \mapsto \{data \mapsto l_2\}_{\tau_0},\\
&\qquad l_1 \mapsto \{data \mapsto l_2\}_{\tau_1}, l_2 \mapsto \emptyset_{\tau_2}\}, \phi \cup \{\tau_2 <: Object\})\\
&\overset{P1}{\Rightarrow}_{\mathscr{S}} (\emptyset, 18, \{0 \mapsto l_0, 1 \mapsto l_1, 2 \mapsto l_1\}, nil, \{l_0 \mapsto \{data \mapsto l_2\}_{\tau_0}, l_1 \mapsto \{data \mapsto l_1\}_{\tau_1},\\
&\qquad l_2 \mapsto \emptyset_{\tau_2}\}, \phi \cup \{\tau_2 <: Object\}),
\end{aligned}$$

where $\phi = \{\tau_0 <: Container, \tau_1 <: Container\}$; AL stands for the ALOAD-S rule; G4 stands for the GETFIELD4-S rule; AS stands for the ASTORE-S rule; G6 stands for the GETFIELD6-S rule; P1 stands for the PUTFIELD1-S rule.

Fig. 13 Formalization of the Trace 3-33-334-3341 in Figure 2

B Concrete Semantic Rules

The concrete JVM bytecode operational semantic rules are divided into five categories: load and store instruction rules, arithmetic instruction rules, object creation and manipulation instruction rules, control transfer instruction rules, and `assume` and `assert` instruction rules. We use the binding $\sigma = (g, pc, \xi, \omega, h, \textsc{True})$ for all the rules.

Load and store instruction rules (shown in Figure 16): Instruction `aload_n` reads the local variable at local index n and puts it onto the stack as illustrated by rule ALOAD-C and `astore_n` stores the top stack value to the local variable at index n as shown in ASTORE-C.

Arithmetic instruction rules (shown in Figure 17): Instruction `iadd` adds two integers from the top of the stack and puts the result back onto the stack. The semantics of `iadd` is represented by rule IADD-C. Similarly, the semantics of `isub` is shown in rule ISUB-C.

Object creation and manipulation instruction rules: We have listed rules for instructions `new` τ, `getfield` f, `putfield` f, `anewarray` τ, `iastore`, `iaload`, `instanceof` τ, and `checkcast` τ in Figures 18 and 19. We will discuss rules for each instruction as follows:

- Instruction `new` τ creates a fresh object of type τ and puts it into the heap. Formal semantics of the instruction is described in rule NEW-C. The fresh object is created by the *new-obj* function (defined in List 2) which initializes each of the field of type τ to its initial value.
- Instruction `getfield` f reads the f field of an object which is indexed by the location on the top of the stack. It has two semantic rules: GETFIELD1-C and GETFIELD2-C. Rule GETFIELD1-C handles the normal case while rule GETFIELD2-C covers the NULL dereference case.

$$\begin{aligned}
&(\emptyset,0,\{0\mapsto\hat{\alpha}_0,1\mapsto\hat{\alpha}_1\},nil,\emptyset,\emptyset)\\
&\overset{AL}{\Rightarrow}_{\mathscr{A}}(\emptyset,1,\{0\mapsto\hat{\alpha}_0,1\mapsto\hat{\alpha}_1\},\hat{\alpha}_0::nil,\emptyset,\emptyset)\\
&\overset{G2A}{\Rightarrow}_{\mathscr{A}}(\emptyset,1,\{0\mapsto l_0,1\mapsto\hat{\alpha}_1\},l_0::nil,\{l_0\mapsto\emptyset_{\tau_0}\},\{\tau_0<:Container\})\\
&\overset{G3A}{\Rightarrow}_{\mathscr{A}}(\emptyset,4,\{0\mapsto l_0,1\mapsto l_1\},\hat{\beta}_0::nil,\{l_0\mapsto\{data\mapsto\hat{\beta}_0\}_{\tau_0}\},\{\tau_0<:Container\})\\
&\overset{AS}{\Rightarrow}_{\mathscr{A}}(\emptyset,5,\{0\mapsto l_0,1\mapsto l_1,2\mapsto\hat{\beta}_0\},nil,\{l_0\mapsto\{data\mapsto\hat{\beta}_0\}_{\tau_0}\},\{\tau_0<:Container\})\\
&\overset{AL}{\Rightarrow}_{\mathscr{A}}(\emptyset,6,\{0\mapsto l_0,1\mapsto\hat{\alpha}_1,2\mapsto\hat{\beta}_0\},l_0::nil,\{l_0\mapsto\{data\mapsto\hat{\beta}_0\}_{\tau_0}\},\{\tau_0<:Container\})\\
&\overset{AL}{\Rightarrow}_{\mathscr{A}}(\emptyset,7,\{0\mapsto l_0,1\mapsto\hat{\alpha}_1,2\mapsto\hat{\beta}_0\},\hat{\alpha}_1::l_0::nil,\{l_0\mapsto\{data\mapsto\hat{\beta}_0\}_{\tau_0}\},\{\tau_0<:Container\})\\
&\overset{G2A}{\Rightarrow}_{\mathscr{A}}(\emptyset,7,\{0\mapsto l_0,1\mapsto l_1,2\mapsto\hat{\beta}_0\},l_1::l_0::nil,\{l_0\mapsto\{data\mapsto\hat{\beta}_0\}_{\tau_0},l_1\mapsto\emptyset_{\tau_1}\},\phi)\\
&\overset{G3A}{\Rightarrow}_{\mathscr{A}}(\emptyset,10,\{0\mapsto l_0,1\mapsto l_1,2\mapsto\hat{\beta}_0\},\hat{\beta}_1::l_0::nil,\{l_0\mapsto\{data\mapsto\hat{\beta}_0\}_{\tau_0},l_1\mapsto\{data\mapsto\hat{\beta}_1\}_{\tau_1}\},\phi)\\
&\overset{P1}{\Rightarrow}_{\mathscr{A}}(\emptyset,13,\{0\mapsto l_0,1\mapsto l_1,2\mapsto\hat{\beta}_0\},nil,\{l_0\mapsto\{data\mapsto\hat{\beta}_1\}_{\tau_0},l_1\mapsto\{data\mapsto\hat{\beta}_1\}_{\tau_1}\},\phi)\\
&\overset{AL}{\Rightarrow}_{\mathscr{A}}(\emptyset,14,\{0\mapsto l_0,1\mapsto l_1,2\mapsto\hat{\beta}_0\},l_1::nil,\{l_0\mapsto\{data\mapsto\hat{\beta}_1\}_{\tau_0},l_1\mapsto\{data\mapsto\hat{\beta}_1\}_{\tau_1}\},\phi)\\
&\overset{AL}{\Rightarrow}_{\mathscr{A}}(\emptyset,15,\{0\mapsto l_0,1\mapsto l_1,2\mapsto\hat{\beta}_0\},\hat{\beta}_0::l_1::nil,\{l_0\mapsto\{data\mapsto\hat{\beta}_1\}_{\tau_0},l_1\mapsto\{data\mapsto\hat{\beta}_1\}_{\tau_1}\},\phi)\\
&\overset{P1}{\Rightarrow}_{\mathscr{A}}(\emptyset,18,\{0\mapsto l_0,1\mapsto l_1,2\mapsto\hat{\beta}_0\},nil,\{l_0\mapsto\{data\mapsto\hat{\beta}_1\}_{\tau_0},l_1\mapsto\{data\mapsto\hat{\beta}_0\}_{\tau_1}\},\phi),
\end{aligned}$$

where $\phi=\{\tau_0<:Container,\tau_1<:Container\}$; AL stands for the ALOAD-S rule; G2A stands for the GETFIELD2-A rule; AS stands for the ASTORE-S rule; G3A stands for the GETFIELD3-A rule; P1 stands for the PUTFIELD1-S rule.

Fig. 14 Formalization of the Trace 2-22-223-2231 in Figure 3

$$\begin{aligned}
&(\emptyset,0,\{0\mapsto\hat{\alpha}_0,1\mapsto\hat{\alpha}_1\},nil,\emptyset,\emptyset)\\
&\overset{AL}{\Rightarrow}_{\mathscr{B}}(\emptyset,1,\{0\mapsto\hat{\alpha}_0,1\mapsto\hat{\alpha}_1\},\hat{\alpha}_0::nil,\emptyset,\emptyset)\\
&\overset{G2A}{\Rightarrow}_{\mathscr{B}}(\emptyset,1,\{0\mapsto l_0,1\mapsto\hat{\alpha}_1\},l_0::nil,\{l_0\mapsto\emptyset_{\tau_0}\},\{\tau_0<:Container\})\\
&\overset{G3B}{\Rightarrow}_{\mathscr{B}}(\emptyset,4,\{0\mapsto l_0,1\mapsto l_1\},\bar{\beta}_0::nil,\{l_0\mapsto\{data\mapsto\bar{\beta}_0\}_{\tau_0}\},\{\tau_0<:Container\})\\
&\overset{AS}{\Rightarrow}_{\mathscr{B}}(\emptyset,5,\{0\mapsto l_0,1\mapsto l_1,2\mapsto\bar{\beta}_0\},nil,\{l_0\mapsto\{data\mapsto\bar{\beta}_0\}_{\tau_0}\},\{\tau_0<:Container\})\\
&\overset{AL}{\Rightarrow}_{\mathscr{B}}(\emptyset,6,\{0\mapsto l_0,1\mapsto\hat{\alpha}_1,2\mapsto\bar{\beta}_0\},l_0::nil,\{l_0\mapsto\{data\mapsto\bar{\beta}_0\}_{\tau_0}\},\{\tau_0<:Container\})\\
&\overset{AL}{\Rightarrow}_{\mathscr{B}}(\emptyset,7,\{0\mapsto l_0,1\mapsto\hat{\alpha}_1,2\mapsto\bar{\beta}_0\},\hat{\alpha}_1::l_0::nil,\{l_0\mapsto\{data\mapsto\bar{\beta}_0\}_{\tau_0}\},\{\tau_0<:Container\})\\
&\overset{G2A}{\Rightarrow}_{\mathscr{B}}(\emptyset,7,\{0\mapsto l_0,1\mapsto l_1,2\mapsto\bar{\beta}_0\},l_1::l_0::nil,\{l_0\mapsto\{data\mapsto\bar{\beta}_0\}_{\tau_0},l_1\mapsto\emptyset_{\tau_1}\},\phi)\\
&\overset{G3B}{\Rightarrow}_{\mathscr{B}}(\emptyset,10,\{0\mapsto l_0,1\mapsto l_1,2\mapsto\bar{\beta}_0\},\bar{\beta}_1::l_0::nil,\{l_0\mapsto\{data\mapsto\bar{\beta}_0\}_{\tau_0},l_1\mapsto\{data\mapsto\bar{\beta}_1\}_{\tau_1}\},\phi)\\
&\overset{P1}{\Rightarrow}_{\mathscr{B}}(\emptyset,13,\{0\mapsto l_0,1\mapsto l_1,2\mapsto\bar{\beta}_0\},nil,\{l_0\mapsto\{data\mapsto\bar{\beta}_1\}_{\tau_0},l_1\mapsto\{data\mapsto\bar{\beta}_1\}_{\tau_1}\},\phi)\\
&\overset{AL}{\Rightarrow}_{\mathscr{B}}(\emptyset,14,\{0\mapsto l_0,1\mapsto l_1,2\mapsto\bar{\beta}_0\},l_1::nil,\{l_0\mapsto\{data\mapsto\bar{\beta}_1\}_{\tau_0},l_1\mapsto\{data\mapsto\bar{\beta}_1\}_{\tau_1}\},\phi)\\
&\overset{AL}{\Rightarrow}_{\mathscr{B}}(\emptyset,15,\{0\mapsto l_0,1\mapsto l_1,2\mapsto\bar{\beta}_0\},\bar{\beta}_0::l_1::nil,\{l_0\mapsto\{data\mapsto\bar{\beta}_1\}_{\tau_0},l_1\mapsto\{data\mapsto\bar{\beta}_1\}_{\tau_1}\},\phi)\\
&\overset{P1}{\Rightarrow}_{\mathscr{B}}(\emptyset,18,\{0\mapsto l_0,1\mapsto l_1,2\mapsto\bar{\beta}_0\},nil,\{l_0\mapsto\{data\mapsto\bar{\beta}_1\}_{\tau_0},l_1\mapsto\{data\mapsto\bar{\beta}_0\}_{\tau_1}\},\phi),
\end{aligned}$$

where $\phi=\{\tau_0<:Container,\tau_1<:Container\}$; AL stands for the ALOAD-S rule; G2A stands for the GETFIELD2-A rule; G3B stands for the GETFIELD3-B rule; AS stands for the ASTORE-S rule; P1 stands for the PUTFIELD1-S rule.

Fig. 15 Formalization of the Trace 1-11-112-1121 in Figure 4

ALOAD-C
$$\frac{\mathit{code}(pc) = \texttt{aload\_n}}{\sigma \Rightarrow_{\mathscr{C}} (g, \mathit{next}(pc), \xi, \xi(n) :: \omega, h, \textsc{True})}$$

ASTORE-C
$$\frac{\mathit{code}(pc) = \texttt{astore\_n} \qquad \omega = v :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} (g, \mathit{next}(pc), \xi[n \mapsto v], \omega', h, \textsc{True})}$$

Fig. 16 Load and Store Instruction Rules in Concrete JVM

IADD-C
$$\frac{\mathit{code}(pc) = \texttt{iadd} \qquad \omega = c_1 :: c_2 :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} (g, \mathit{next}(pc), \xi, (c_1 + c_2) :: \omega', h, \textsc{True})}$$

ISUB-C
$$\frac{\mathit{code}(pc) = \texttt{isub} \qquad \omega = c_1 :: c_2 :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} (g, \mathit{next}(pc), \xi, (c_2 - c_1) :: \omega', h, \textsc{True})}$$

Fig. 17 Rules for Arithmetic Instructions in Concrete JVM

NEW-C
$$\frac{\mathit{code}(pc) = \texttt{new}\ \tau}{\sigma \Rightarrow_{\mathscr{C}} (g, \mathit{next}(pc), \xi, l :: \omega, h[l \mapsto \mathit{new\text{-}obj}(\mathit{symbols}(\sigma), \tau)], \textsc{True})}$$
$$\text{where } l \notin \operatorname{dom} h$$

GETFIELD1-C
$$\frac{\mathit{code}(pc) = \texttt{getfield}\ f \qquad \omega = l :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} (g, \mathit{next}(pc), \xi, h(l)(f) :: \omega', h, \textsc{True})}$$

GETFIELD2-C
$$\frac{\mathit{code}(pc) = \texttt{getfield}\ f \qquad \omega = \textsc{Null} :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} \text{NullPointerException}, (g, pc, \xi, \omega, h, \textsc{True})}$$

PUTFIELD1-C
$$\frac{\mathit{code}(pc) = \texttt{putfield}\ f \qquad \omega = v :: l :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} (g, \mathit{next}(pc), \xi, \omega', h[l \mapsto h(l)[f \mapsto v]], \textsc{True})}$$

PUTFIELD2-C
$$\frac{\mathit{code}(pc) = \texttt{putfield}\ f \qquad \omega = v :: \textsc{Null} :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} \text{NullPointerException}, (g, pc, \xi, \omega, h, \textsc{True})}$$

INSTANCEOF1-C
$$\frac{\mathit{code}(pc) = \texttt{instanceof}\ \tau \qquad \omega = \textsc{Null} :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} (g, \mathit{next}(pc), \xi, 0 :: \omega', h, \textsc{True})}$$

INSTANCEOF2-C
$$\frac{\mathit{code}(pc) = \texttt{instanceof}\ \tau \qquad \omega = l :: \omega' \qquad \alpha_{\tau_1} = h(l) \qquad \tau_1 <: \tau}{\sigma \Rightarrow_{\mathscr{C}} (g, \mathit{next}(pc), \xi, 1 :: \omega', h, \textsc{True})}$$

INSTANCEOF3-C
$$\frac{\mathit{code}(pc) = \texttt{instanceof}\ \tau \qquad \omega = l :: \omega' \qquad \alpha_{\tau_1} = h(l) \qquad \tau_1 \not<: \tau}{\sigma \Rightarrow_{\mathscr{C}} (g, \mathit{next}(pc), \xi, 0 :: \omega', h, \textsc{True})}$$

CHECKCAST1-C
$$\frac{\mathit{code}(pc) = \texttt{checkcast}\ \tau \qquad \omega = \textsc{Null} :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} (g, \mathit{next}(pc), \xi, \omega, h, \textsc{True})}$$

CHECKCAST2-C
$$\frac{\mathit{code}(pc) = \texttt{checkcast}\ \tau \qquad \omega = l :: \omega' \qquad \alpha_{\tau_1} = h(l) \qquad \tau_1 <: \tau}{\sigma \Rightarrow_{\mathscr{C}} (g, \mathit{next}(pc), \xi, \omega, h, \textsc{True})}$$

CHECKCAST3-C
$$\frac{\mathit{code}(pc) = \texttt{checkcast}\ \tau \qquad \omega = l :: \omega' \qquad \alpha_{\tau_1} = h(l) \qquad \tau_1 \not<: \tau}{\sigma \Rightarrow_{\mathscr{C}} \text{ClassCastException}, (g, pc, \xi, \omega, h, \textsc{True})}$$

Fig. 18 Rules for Object Creation and Manipulation Instructions in Concrete JVM

ANEWARRAY1-C
$$\frac{code(pc) = \texttt{anewarray}\ \tau \qquad \omega = c :: \omega' \qquad c \geq 0}{\sigma \Rightarrow_{\mathscr{C}} (g, next(pc), \xi, l :: \omega', h[l \mapsto new\text{-}carr(symbols(\sigma), \tau, c)], \textsc{True})}$$
where $l \notin \operatorname{dom} h$

ANEWARRAY2-C
$$\frac{code(pc) = \texttt{anewarray}\ \tau \qquad \omega = c :: \omega' \qquad c < 0}{\sigma \Rightarrow_{\mathscr{C}} \text{NegativeArraySizeException}, (g, pc, \xi, \omega, h, \textsc{True})}$$

IASTORE1-C
$$\frac{code(pc) = \texttt{iastore} \qquad \omega = c_2 :: c_1 :: l :: \omega' \qquad c_1 < 0 \vee c_1 \geq h(l)(\textsc{len})}{\sigma \Rightarrow_{\mathscr{C}} \text{ArrayIndexOutOfBoundsException}, (g, pc, \xi, \omega, h, \textsc{True})}$$

IASTORE2-C
$$\frac{code(pc) = \texttt{iastore} \qquad \omega = c_2 :: c_1 :: l :: \omega' \qquad 0 \leq c_1 < h(l)(\textsc{len})}{\sigma \Rightarrow_{\mathscr{C}} (g, next(pc), \xi, \omega', h[l \mapsto h(l)[c_1 \mapsto c_2]], \textsc{True})}$$

IASTORE3-C
$$\frac{code(pc) = \texttt{iastore} \qquad \omega = c_2 :: c_1 :: \textsc{Null} :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} \text{NullPointerException}, (g, pc, \xi, \omega, h, , \textsc{True})}$$

IALOAD1-C
$$\frac{code(pc) = \texttt{iaload} \qquad \omega = c :: l :: \omega' \qquad c < 0 \vee c \geq h(l)(\textsc{len})}{\sigma \Rightarrow_{\mathscr{C}} \text{ArrayIndexOutOfBoundsException}, (g, pc, \xi, \omega, h, \textsc{True})}$$

IALOAD2-C
$$\frac{code(pc) = \texttt{iaload} \qquad \omega = c :: l :: \omega' \qquad 0 \leq c < h(l)(\textsc{len})}{\sigma \Rightarrow_{\mathscr{C}} (g, next(pc), \xi, h(l)(c) :: \omega', h, \textsc{True})}$$

IALOAD3-C
$$\frac{code(pc) = \texttt{iaload} \qquad \omega = c :: \textsc{Null} :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} \text{NullPointerException}, (g, pc, \xi, \omega, h, \textsc{True})}$$

Fig. 19 Rules for Object Creation and Manipulation Instruction (2) in Concrete JVM

- Instruction `putfield` f writes a value to field f of an object. The value and location of the object are on the top of the stack. There are two rules for `putfield` f: PUTFIELD1-C and PUTFIELD2-C. PUTFIELD1-C handles the normal case and PUTFIELD2-C is for the case that the object is NULL.
- Instruction `instanceof` τ tests whether the type of an object is a subtype of τ. According to the JVM specification Lindholm and Yellin (1999), if the object is NULL, the test returns FALSE; if the object is non-NULL and the type of the object is a subtype of τ, it returns TRUE; otherwise, it returns FALSE. Rule INSTANCEOF1-C represents the NULL case, and INSTANCEOF2-C and INSTANCEOF3-C handle the non-NULL case. Note that in JVM, 0 is used for FALSE and 1 for TRUE.
- Instruction `checkcast` τ is very similar to instruction `instanceof`. Both of them test whether the type of an object is a subtype of another type. However, there are two differences: first is that if the operand is NULL, the test passes; second is that the instruction does not return any value—if the test passes, it does nothing; otherwise it throws a ClassCastException exception. Rule CHECKCAST1-C represents the NULL case; CHECKCAST2-C and CHECKCAST3-C handle the non-NULL case.
- Instruction `anewarray` τ creates a new array with the length on the top of the operand stack. The new array has all the indexes initialized with the default value of the element type by the *new-carr* function shown in List 2. ANEWARRAY1-C denotes the case of a non-negative length and ANEWARRAY2-C describes the case of a negative length.
- Instruction `iastore` writes an integer value into an integer array. There are three rules for the instruction: IASTORE1-C, IASTORE2-C, and IASTORE3-C. Rule IASTORE1-C is for the array index out of bound case. Rule IASTORE2-C handles the normal case that the index is in bound. IASTORE3-C presents the case that the array is NULL which results in a NullPointerException.
- Instruction `iaload` reads the value from an index of an array. The semantic rules are symmetrical to the rules for instruction `iastore`.

Control transfer instruction rules (shown in Figure 20): Instruction `if_icmplt` checks if the second topmost operand of integer type is less than the topmost operand. If it is the case then the execution jumps to the operand of the instruction (rule IF_ICMPLT1-C); otherwise, the execution will simply move to the next instruction (rule IF_ICMPLT2-C).

IF_ICMPLT1-C $\dfrac{code(pc) = \texttt{if\_icmplt}\ pc' \qquad \omega = c_2 :: c_1 :: \omega' \qquad c_1 < c_2}{\sigma \Rightarrow_{\mathscr{C}} (g, pc', \xi, \omega', h, \text{TRUE})}$

IF_ICMPLT2-C $\dfrac{code(pc) = \texttt{if\_icmplt}\ pc' \qquad \omega = c_2 :: c_1 :: \omega' \qquad c_2 \leq c_1}{\sigma \Rightarrow_{\mathscr{C}} (g, next(pc), \xi, \omega', h, \text{TRUE})}$

IF_ACMPEQ1-C $\dfrac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = v_1 :: v_2 :: \omega' \qquad v_1 \neq v_2}{\sigma \Rightarrow_{\mathscr{C}} (g, next(pc), \xi, \omega', h, \text{TRUE})}$

IF_ACMPEQ2-C $\dfrac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = v_1 :: v_2 :: \omega' \qquad v_1 = v_2}{\sigma \Rightarrow_{\mathscr{C}} (g, pc', \xi, \omega', h, \text{TRUE})}$

IFNULL1-C $\dfrac{code(pc) = \texttt{ifnull}\ pc' \qquad \omega = l :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} (g, next(pc), \xi, \omega', h, \text{TRUE})}$

IFNULL2-C $\dfrac{code(pc) = \texttt{ifnull}\ pc' \qquad \omega = \text{NULL} :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} (g, pc', \xi, \omega', h, \text{TRUE})}$

IFNONNULL1-C $\dfrac{code(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = l :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} (g, pc', \xi, \omega', h, \text{TRUE})}$

IFNONNULL2-C $\dfrac{code(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = \text{NULL} :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} (g, next(pc), \xi, \omega', h, \text{TRUE})}$

Fig. 20 Rules for Control Transfer Instructions in Concrete JVM

ASSUME1-C $\dfrac{code(pc) = \texttt{assume} \qquad \omega = 0 :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} (g, next(pc), \xi, \omega', h, \text{FALSE})}$

ASSUME2-C $\dfrac{code(pc) = \texttt{assume} \qquad \omega = 1 :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} (g, next(pc), \xi, \omega', h, \text{TRUE})}$

ASSERT1-C $\dfrac{code(pc) = \texttt{assert} \qquad \omega = 0 :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} \text{ERROR}, (g, pc, \xi, \omega, h, \text{TRUE})}$

ASSERT2-C $\dfrac{code(pc) = \texttt{assert} \qquad \omega = 1 :: \omega'}{\sigma \Rightarrow_{\mathscr{C}} (g, next(pc), \xi, \omega', h, \text{TRUE})}$

Fig. 21 Rules for `assume` and `assert` Instructions in Concrete JVM

Similar to `if_icmpt`, instruction `if_acmpeq` checks the equality between two object references (which may be NULL) on the top of the stack and the execution will branch if the equality holds.

Instruction `ifnull` does a NULL-ness test of the top of the operand and the execution jumps if it is NULL. IFNULL1-C is for the non-NULL case and IFNULL2-C is for the NULL case.

Instruction `ifnonnull` does the opposite of `ifnull`.

Rules for the `assume` *and* `assert` *instructions (shown in Figure 21):* The semantics for `assume` and `assert` are standard: if the top of the stack is true, `assume` and `assert` do nothing; otherwise, `assume` terminates the execution silently by making path condition FALSE, while `assert` signals an error and terminates the execution.

`Discussion:` We do not use the wraparound semantics for integral types because it complicates the presentation of operational semantics. In addition, we do not check bugs introduced by integer wrapping around in symbolic executions. However, wraparound can be supported by using appropriate decision procedures that model integers using bit-vectors.

IF_ACMPEQ1-A
$$\frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = \hat{\alpha}_\tau :: \hat{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{A}} (g, pc', \xi, \omega', h, \phi)}$$

IF_ACMPEQ2-A
$$\frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = \hat{\alpha}_\tau :: v :: \omega'}{\sigma \Rightarrow_{\mathscr{A}} (g, pc, \xi, \omega, h, \phi \cup \{\tau' <: \tau\})[l/\hat{\alpha}] \qquad \text{where } l \in collect(h), h(l) = \alpha_{\tau'}}$$

IF_ACMPEQ3-A
$$\frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = \hat{\alpha}_\tau :: v :: \omega' \qquad \tau \in \mathbf{RType}}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{A}} (g, pc, \eta_*(\xi), \omega, h[l \mapsto \alpha_{\tau'}], \phi \cup \{\tau' <: \tau\}[l/\hat{\alpha}] \\ \text{where } l \notin \operatorname{dom} h, \alpha_{\tau'} = \textit{new-sym}(symbols(\sigma))\end{array}}$$

IF_ACMPEQ4-A
$$\frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = \hat{\alpha}_\tau :: v :: \omega' \qquad \tau \in \mathbf{AType}}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{A}} (g, pc, \xi, \omega, h[l \mapsto \alpha_{\tau'}], \phi \cup \{\tau' <: \tau, 0 \leq \alpha(\textsc{len})\})[l/\hat{\alpha}] \\ \text{where } l \notin \operatorname{dom} h, \alpha_{\tau'} = \textit{new-sarr}(symbols(\sigma))\end{array}}$$

IF_ACMPEQ5-A
$$\frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = v :: \hat{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{A}} (g, pc, \eta_*(\xi), \omega, h, \phi \cup \{\tau' <: \tau\})[l/\hat{\alpha}] \qquad \text{where } l \in collect(h), h(l) = \alpha_{\tau'}}$$

IF_ACMPEQ6-A
$$\frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = v :: \hat{\alpha}_\tau :: \omega' \qquad \tau \in \mathbf{RType}}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{A}} (g, pc, \xi, \omega, h[l \mapsto \alpha_{\tau'}], \phi \cup \{\tau' <: \tau\})[l/\hat{\alpha}] \\ \text{where } l \notin \operatorname{dom} h, \alpha_{\tau'} = \textit{new-sym}(symbols(\sigma))\end{array}}$$

IF_ACMPEQ7-A
$$\frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = v :: \hat{\alpha}_\tau :: \omega' \qquad \tau \in \mathbf{AType}}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{A}} (g, pc, \xi, \omega, h[l \mapsto \alpha_{\tau'}], \phi \cup \{\tau' <: \tau, 0 \leq \alpha(\textsc{len})\})[l/\hat{\alpha}] \\ \text{where } l \notin \operatorname{dom} h, \alpha_{\tau'} = \textit{new-sarr}(symbols(\sigma))\end{array}}$$

IFNULL-A
$$\frac{code(pc) = \texttt{ifnull}\ pc' \qquad \omega = \hat{\alpha} :: \omega'}{\sigma \Rightarrow_{\mathscr{A}} (g, next(pc), \xi, \omega', h, \phi)}$$

IFNONNULL-A
$$\frac{code(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = \hat{\alpha} :: \omega'}{\sigma \Rightarrow_{\mathscr{A}} (g, pc', \xi, \omega', h, \phi)}$$

Fig. 22 Additional Rules Control Transfer Instructions in SELA

C Lazier Semantic Rules

As described in Section 3, SELA (Symbolic Execution with Lazier Initialization) is distinguished from SEL in the use of symbolic locations. Therefore, in general, SELA semantic rules are the same as the SEL semantic rules unless symbolic locations are involved. Symbolic locations can be used as operands of an instruction or produced by the instruction, `getfield`. For each symbolic location that appears in the operands of an instruction, there are two possibilities: the symbolic location is "consumed" (used) or just "transferred." If a symbolic location is consumed by an instruction, it is resolved to a location except `ifnull` and `ifnonnull` which leverage symbolic locations directly. If a symbolic location is just transferred by an instruction, then the rules in SELA should be the same as the ones in SEL. Recall that the bytecode instructions that we cover are classified into five categories: (1) load and store instruction, (2) arithmetic instruction, (3) object creation and manipulation instruction, (4) control transfer instruction, and (5) `assume` and `assert` instruction. The rules for instructions in (1), (2), and (5) are the same as the ones in SEL since instructions in (1) only transfer symbolic locations; and instructions in (2) and (5) have no symbolic location operand. We only need to discuss rules for (3) and (4). We will first explain rules for (4) and then (3) for clarity. Since the large portions of the SELA semantic rules are shared with the ones of SEL, we present only the additional rules of SELA

GETFIELD1-A
$$\frac{code(pc) = \texttt{getfield}\ f \qquad \omega = \hat{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{A}} (g, pc, \xi, \omega, h, \phi \cup \{\tau' <: \tau\})[l/\hat{\alpha}] \qquad \text{where } l \in collect(h), h(l) = \alpha_{\tau'}}$$

GETFIELD2-A
$$\frac{code(pc) = \texttt{getfield}\ f \qquad \omega = \hat{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{A}} (g, pc, \eta_*(\xi), \omega, h[l \mapsto \alpha_{\tau'}], \phi \cup \{\tau' <: \tau\})[l/\hat{\alpha}] \qquad \text{where } l \notin \mathrm{dom}\, h, \alpha_{\tau'} = new\text{-}sym(symbols(\sigma))}$$

GETFIELD3-A
$$\frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = l :: \omega' \qquad h(l)(f_\tau)\uparrow \qquad \tau \in \mathbf{NPType}}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{A}} (g, next(pc), \xi, \hat{\alpha}_\tau :: \omega', h[l \mapsto h(l)[f_\tau \mapsto \hat{\alpha}_\tau]], \phi) \\ \text{where } \hat{\alpha} \text{ is fresh}\end{array}}$$

PUTFIELD1-A
$$\frac{code(pc) = \texttt{putfield}\ f \qquad \omega = v :: \hat{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{A}} (g, pc, \eta_*(\xi), \omega, h, \phi \cup \{\tau' <: \tau\})[l/\hat{\alpha}] \qquad \text{where } l \in collect(h), h(l) = \alpha_{\tau'}}$$

PUTFIELD2-A
$$\frac{code(pc) = \texttt{putfield}\ f \qquad \omega = v :: \hat{\alpha}_\tau :: \omega'}{\begin{array}{l}\sigma \Rightarrow_{\mathscr{A}} (g, pc, \eta_*(\xi), \omega, h[l \mapsto \alpha_{\tau'}], \phi \cup \{\tau' <: \tau\})[l/\hat{\alpha}] \\ \quad \text{where } l \notin \mathrm{dom}\, h, \alpha_{\tau'} = new\text{-}sym(symbols(\sigma))\end{array}}$$

IASTORE1-A
$$\frac{code(pc) = \texttt{iastore} \qquad \omega = v :: i :: \hat{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{A}} (g, pc, \xi, \omega, h, \phi \cup \{\tau' <: \tau\})[l/\hat{\alpha}] \qquad \text{where } l \in collect(h), h(l) = \alpha_{\tau'}}$$

IASTORE2-A
$$\frac{code(pc) = \texttt{iastore} \qquad \omega = v :: i :: \hat{\alpha}_\tau :: \omega'}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{A}} (g, pc, \xi, \omega, h[l \mapsto \alpha_{\tau'}], \phi \cup \{\tau' <: \tau, 0 \le \alpha(\textsc{len})\})[l/\hat{\alpha}] \\ \text{where } l \notin \mathrm{dom}\, h, \alpha_{\tau'} = new\text{-}sarr(symbols(\sigma))\end{array}}$$

IALOAD1-A
$$\frac{code(pc) = \texttt{iaload} \qquad \omega = i :: \hat{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{A}} (g, pc, \xi, \omega, h, \phi \cup \{\tau' <: \tau\}) \qquad \text{where } l \in collect(h), h(l) = \alpha_{\tau'}}$$

IALOAD2-A
$$\frac{code(pc) = \texttt{iaload} \qquad \omega = i :: \hat{\alpha}_\tau :: \omega'}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{A}} (g, pc, \xi, \omega, h[l \mapsto \alpha_{\tau'}], \phi \cup \{\tau' <: \tau, 0 \le \alpha(\textsc{len})\})[l/\hat{\alpha}] \\ \text{where } l \notin \mathrm{dom}\, h, \alpha_{\tau'} = new\text{-}sarr(symbols(\sigma))\end{array}}$$

Fig. 23 Additional Rules for Object Creation and Manipulation Instructions in SELA

in this subsection. As in the SEL rules, we use the binding $\sigma = (g, pc, \xi, \omega, h, \phi)$, and all the end states with unsatisfiable path conditions are ignored.

Control transfer instruction rules (shown in Figure 22): There are seven additional SELA semantic rules for the `if_acmpeq` instruction. Rule IF_ACMPEQ1-A is to optimize the operation when the two operands are the same symbolic location; there is no need to resolve the symbolic location in this case. The remaining six rules resolve symbolic location operands. Rules IF_ACMPEQ2-A, IF_ACMPEQ3-A, and IF_ACMPEQ4-A handle the case of the first operand being a symbolic location. More specifically, rule IF_ACMPEQ2-A resolves a symbolic location to a location that refers to an existing object in the heap; rule IF_ACMPEQ3-A resolves a symbolic location associated with a record type to a fresh location, which refers to a fresh object; rule IF_ACMEQ4-A resolves a symbolic location associated with an array type to a fresh location, which refers to a fresh array. Symmetrically, rules IF_ACMPEQ5-A, IF_ACMPEQ6-A, and IF_ACMPEQ7-A handle the case of the second operand being a symbolic location. If both operands are locations or NULL, SEL rule IF_ACMPEQ1-S or IF_ACMPEQ2-S is applied.

There is one additional rule for each of the `ifnull` and `ifnonnull` instructions to handle the case of a symbolic location operand. Since a symbolic location can only be resolved to a location which is not NULL,

$$\text{IF\_ACMPEQ1-B} \quad \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = \bar{\alpha}_\tau :: \bar{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{B}} (g, pc', \xi, \omega', h, \phi)}$$

$$\text{IF\_ACMPEQ2-B} \quad \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = \bar{\alpha}_\tau :: v :: \omega'}{\sigma \Rightarrow_{\mathscr{B}} \sigma[\textsc{Null}/\bar{\alpha}]}$$

$$\text{IF\_ACMPEQ3-B} \quad \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = \bar{\alpha}_\tau :: v :: \omega'}{\sigma \Rightarrow_{\mathscr{B}} \sigma[\hat{\alpha}_\tau/\bar{\alpha}_\tau] \qquad \text{where } \hat{\alpha} \text{ is fresh}}$$

$$\text{IF\_ACMPEQ4-B} \quad \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = v :: \bar{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{B}} \sigma[\textsc{Null}/\bar{\alpha}]}$$

$$\text{IF\_ACMPEQ5-B} \quad \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = v :: \bar{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{B}} \sigma[\hat{\alpha}_\tau/\bar{\alpha}_\tau] \qquad \text{where } \hat{\alpha} \text{ is fresh}}$$

Fig. 24 Additional Rules for `if_acmpeq` Instruction in SELB

there is no need for the instructions to resolve the symbolic location operand. This is another advantage of lazier initialization besides being lazier than lazy initialization: symbolic locations are directly leveraged.

For `if_icmplt`, since all the operands are integer values and thus can not be symbolic locations, the rule for the instruction is the same as the one in SEL, IF_ICMPLT-S.

Object creation and manipulation instruction rules (shown in Figure 23): There are three additional rules for the `getfield` and two for `putfield` instructions. Most of those rules are for resolving a symbolic location to a location that refers to one of existing objects in the heap (GETFIELD1-A, PUTFIELD1-A) or a fresh symbolic object (GETFIELD2-A, PUTFIELD2-A). The remaining rule of `getfield`, GETFIELD3-A, demonstrates the essence of lazier initialization: when a field is not defined, a symbolic location is produced, that is, the field is initialized with a fresh symbolic location. Note that the field can be initialized with NULL as well (shown in rule GETFIELD3-S). Therefore, GETFIELD3-A rule overrides SEL rules that initialize an undefined field to a location, i.e., GETFIELD4,5,6-S. In other words, the SELA rules for instruction `getfield` consist of GETFIELD1,2,3-A and GETFIELD1,2,3,7-S.

For the `putfield` instruction, the two additional rules PUTFIELD1-A and PUTFIELD2-A resolve the second operand if it is a symbolic location. Whether the top of the stack is a symbolic location is not examined because the top value is only transferred by the instruction.

For `iastore` and `iaload`, there are two additional rules for each instruction: to resolve a symbolic location to an existing array in the heap or a fresh array.

The rest of the instructions (`anew`, `anewarray`, `instanceof`, and `checkcast`) in this category have the same rules as the ones in SEL because no symbolic location can appear in the operands.

D Lazier# Semantic Rules

SELB (Symbolic Execution with Lazier# Initialization) is distinguished from SELA in the use of symbolic references. Hence, in general, the semantic rules of SELB are the same as those of SELA unless symbolic references are involved. Symbolic references can be used as operands of instructions or produced by the instruction, `getfield`. Depending on the instruction, symbolic references that appear in the operands may be either be consumed or transferred. For each symbolic reference that is consumed by instructions, the symbolic reference is resolved to either NULL or a fresh symbolic location. Once symbolic references are resolved, the rules of SELA and SEL are applied. Similar to SELA, we only discuss additional rules that handle symbolic references for instructions: `if_acmpeq`, `ifnull`, `ifnonnull`, `getfield`, `putfield`, `iastore`, and `iaload` since the rest of the instructions that we cover in this article have the same rules as in SEL. We will use binding $\sigma = (g, pc, \xi, \omega, h, \phi)$.

Figure 24 shows the five additional rules for the `if_acmpeq` instruction. Similar to rule IF_ACMPEQ1-A in SELA, rule IF_ACMPEQ1-B is an optimization for the two operands being the same symbolic reference

IFNULL1-B $$\frac{code(pc) = \texttt{ifnull}\ pc' \qquad \omega = \bar{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{B}} \sigma[\textsc{Null}/\bar{\alpha}]}$$

IFNULL2-B $$\frac{code(pc) = \texttt{ifnull}\ pc' \qquad \omega = \bar{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{B}} \sigma[\hat{\alpha}_\tau/\bar{\alpha}_\tau] \qquad \text{where } \hat{\alpha} \text{ is fresh}}$$

IFNONNULL1-B $$\frac{code(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = \bar{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{B}} \sigma[\textsc{Null}/\bar{\alpha}]}$$

IFNONNULL2-B $$\frac{code(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = \bar{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{B}} \sigma[\hat{\alpha}_\tau/\bar{\alpha}_\tau] \qquad \text{where } \hat{\alpha} \text{ is fresh}}$$

Fig. 25 Additional Rules for `ifnull` and `ifnonnull` Instructions in SELB

GETFIELD1-B $$\frac{code(pc) = \texttt{getfield}\ f \qquad \omega = \bar{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{B}} \sigma[\textsc{Null}/\bar{\alpha}]}$$

GETFIELD2-B $$\frac{code(pc) = \texttt{getfield}\ f \qquad \omega = \bar{\alpha}_\tau :: \omega'}{\sigma \Rightarrow_{\mathscr{B}} \sigma[\hat{\alpha}_\tau/\bar{\alpha}_\tau] \qquad \text{where } \hat{\alpha} \text{ is fresh}}$$

GETFIELD3-B $$\frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = l :: \omega' \qquad h(l)(f_\tau)\uparrow \qquad \tau \in \textbf{NPType}}{\begin{array}{c}\sigma \Rightarrow_{\mathscr{B}} (g, next(pc), \xi, \bar{\alpha}_\tau :: \omega', h[l \mapsto h(l)[f_\tau \mapsto \bar{\alpha}_\tau]], \phi)\\ \text{where } \bar{\alpha} \text{ is fresh}\end{array}}$$

Fig. 26 Additional Rules for `getfield` Instruction in SELB

without having to resolve the symbolic reference. Each of the remaining four rules resolves a symbolic reference operand to either NULL as in IF_ACMPEQ2-B and IF_ACMPEQ4-B, or a fresh symbolic location as in IF_ACMPEQ3-B and IF_ACMPEQ5-B.

Figure 25 shows two additional rules for each of the `ifnull` and `ifnonnull` instructions. The additional rules are for resolving symbolic references to either NULL or a fresh symbolic location.

Figure 26 shows the additional rules for the `getfield` instruction. GETFIELD1-B and GETFIELD2-B are added to resolve symbolic references. Rule GETFIELD3-B initializes an undefined field with a fresh symbolic reference. This SELB rule, GETFIELD3-B, overrides SELA rule GETFIELD3-A and SEL rules GETFIELD3,4,5,6-S. In summary, SELB rules for instruction `getfield` consist of GETFIELD1,2,3-B; GETFIELD1,2-A; and GETFIELD1,2,7-S. Notice that rule GETFIELD3-S, which initializes a field with NULL, is also overridden whereas this rule is not overridden by rule GETFIELD3-A in SELA. This difference explains why SELB is even lazier than SELA: in SELA whether a field value is NULL is decided when it is initialized by the `getfield` instruction; in SELB, this decision is deferred.

The additional rules for `putfield`, `iastore`, and `iaload` instructions are just to resolve the symbolic reference operand and are the same as GETFIELD1-B and GETFIELD2-B; thus the rules are not listed.

E Concretization (γ) Functions

E.1 Substitution Operator

To facilitate the definition of γ functions, we define a substitution operator, $\eta_*(C)$ where η is a function, η : **Value** $\cup$ **Type** $\rightharpoonup$ **Value** $\cup$ **Type**, and C is a construct which can be a value, an expression, a tuple, a function, a set, or a sequence. The inductive definition of C is

$$C ::= c \mid fn : C \to C \mid nil \mid c :: C \mid C\ op\ C \mid (C, C, \ldots, C) \mid \{C, C, \ldots, C\},$$

List 3 Environments

- **SEnv** $= \{E_S \mid E_S : \textbf{PSymbol} \to \textbf{Const}\}$ is the set of all primitive symbol environments.
- **TEnv** $= \{E_T \mid E_T : \textbf{SymType} \to (\textbf{AType} \uplus \textbf{RType})\}$ is the set of all type environments.
- Sym(**Loc**) is the permutation group of **Loc**, that is, the set of bijective functions from **Loc** to **Loc**. Note that Sym means symmetric group here not to be confused with symbolic locations.
- **LEnv** $= \{E_L \mid E_L : \textbf{SymLoc} \to \textbf{Loc}\}$ is the set of all symbolic location environments.
- **REnv** $= \{E_R \mid E_R : \textbf{SymRef} \to (\textbf{SymLoc} \cup \{\textsc{Null}\})\}$ is the set of all symbolic reference environments.

where $c \in \textbf{Value} \cup \textbf{Type} \cup \textbf{PC}$ and $op \in \{+, -, *, /, =, \neq, <, \leq, >, \geq, :>, \vee\}$. The result of $\eta_*(C)$ has the same structure as C except that any component vt that is in the domain of η is replaced by $\eta(vt)$. Formally,

$$\eta_*(C) = \begin{cases} C & \text{if } C \in \textbf{Value} \cup \textbf{Type} \cup \textbf{PC} \wedge C \notin \operatorname{dom} \eta; \\ \eta(C) & \text{if } C \in \textbf{Value} \cup \textbf{Type} \wedge C \in \operatorname{dom} \eta; \\ \bigcup_{d \in \operatorname{dom} C} \{d \mapsto \eta_*(C(d))\} & \text{if } C \text{ is a function;} \\ nil & \text{if } C \text{ is the empty sequence } (nil); \\ \eta_*(d) :: \eta_*(q) & \text{if } C \text{ is a sequence and } C = d :: q; \\ \eta_*(C_1)\ op\ \eta_*(C_2) & \text{if } C = C_1\ op\ C_2; \\ (\eta_*(e_1), \eta_*(e_2), \ldots, \eta_*(e_n)) & \text{if } C = (e_1, e_2, \ldots, e_n) \text{ for some } n \in \mathbb{N}; \\ \bigcup_{e \in C} \{\eta_*(e)\} & \text{if } C \text{ is a set.} \end{cases}$$

List 3 shows a list of environments that are used in the definition of the γ functions. Note that from this now on, we use subscripts c, s, a, and b to denote concrete, SEL, SELA, and SELB state components and domains, respectively.

E.2 Function γ_s

We introduce the following semantic functions:

$$\begin{aligned} \mathbb{V}_s &: \textbf{Value}_s \to ((\textbf{SEnv} \times \mathrm{Sym}(\textbf{Loc})) \to \textbf{Value}_c); \\ \mathbb{O}_s &: \textbf{NPSymbol} \to ((\textbf{TEnv} \times \textbf{SEnv} \times \mathrm{Sym}(\textbf{Loc})) \to \mathscr{P}(\textbf{NPSymbol})); \\ \mathbb{H}_s &: \textbf{Heap}_s \to ((\textbf{TEnv} \times \textbf{SEnv} \times \mathrm{Sym}(\textbf{Loc})) \to \mathscr{P}(\textbf{Heap}_c)); \\ \mathbb{S}_s &: \textbf{State}_s \to ((\textbf{TEnv} \times \textbf{SEnv} \times \mathrm{Sym}(\textbf{Loc})) \to \mathscr{P}(\textbf{State}_c)). \end{aligned}$$

The definitions are listed as follows:

- the $\mathbb{V}_s$ function:

$$\mathbb{V}_s[\![v]\!](E_S, \rho) = \rho_*(E_{S*}(v)).$$

- the $\mathbb{O}_s$ function: $\mathbb{O}_s[\![\alpha_\tau]\!](E_T, E_S, \rho)$ consists of all $\alpha'_{\tau'}$ such that
  1. $\tau' = E_{T*}(\tau)$.
  2. if $\tau' \in \textbf{RType}$, then

$$\forall f \in \textbf{Field}.\alpha(f)\downarrow \wedge f \neq \textsc{conc} \Rightarrow \alpha'(f) = \mathbb{V}_s[\![\alpha(f)]\!](E_S, \rho).$$

  3. if $\tau' \in \textbf{AType}$, then
     (a) the length field of α' is consistent with the length of α. Formally,

$$\alpha'(\textsc{len}) = \mathbb{V}_s[\![\alpha(\textsc{len})]\!](E_S, \rho).$$

     (b) all the initialized indexes of α should appear in α'. Formally,

$$\forall i \in \textit{acc-idx}(\alpha).\alpha'(\mathbb{V}_s[\![i]\!](E_S, \rho)) = \mathbb{V}_s[\![\alpha(i)]\!](E_S, \rho).$$

     (c) If α is a concrete array, then all the uninitialized indexes should be set to the default value. Formally, if $\alpha(\textsc{conc})$ is defined, then for all m satisfying $0 \leq m < \alpha'(\textsc{len})$,

$$m \notin \{\mathbb{V}_s[\![i]\!](E_S, \rho) \mid i \in \textit{acc-idx}(\alpha)\} \Rightarrow \alpha'(m) = \alpha(\textsc{def}).$$

- the $\mathbb{H}_s$ function: $\mathbb{H}_s[\![h_s]\!](E_T, E_S, \rho)$ consists of all h_c such that
  1. each entry in h_s has to appear in h_c, formally,

$$\forall (l, \alpha) \in h_s . \exists \beta \in \mathbb{O}_s[\![\alpha]\!](E_T, E_S, \rho).(\rho(l), \beta) \in h_c.$$

  2. h_c is type correct, that is, for each nonprimitive symbol in h_c, all its fields are mapped to values of compatible types. More specifically, each primitive field is mapped to a constant of its type; each reference type field is mapped to either NULL or a location in h_c which maps to a nonprimitive symbol of a compatible type.
  3. for each entry (l, α_c) in h_c, α_c is well-formed, formally,
     (a) if (l, α_c) is mapped from (l', α_s) in h_s ($l = \rho(l')$ and $\alpha_c \in \mathbb{O}_s[\![\alpha_s]\!](E_T, E_S, \rho)$), and if any field f of α_s is undefined and nonprimitive, $\alpha_c(f)$ has to be one of the following values:
       - NULL.
       - l_1 where $l_1 \notin \rho(\mathrm{dom}\, h_s)$.[10]
       - l_2 where $l_2 \in \rho(\mathrm{dom}\, h_s)$ and $h_s(\rho^{-1}(l_2))(\textsc{conc}) \uparrow$.

     (b) if (l, α_c) is not mapped from any entry in h_s ($l \notin \rho(\mathrm{dom}\, h_s)$), all the nonprimitive fields of α_c can only be one of the three values listed in (a).
- the $\mathbb{S}_s$ function:

$$\mathbb{S}_s[\![(g, pc, \xi, \omega, h, \phi)]\!](E_T, E_S, \rho) = \{ (\rho_*(E_{S*}(g)), pc, \rho_*(E_{S*}(\xi)), \rho_*(E_{S*}(\omega)), h', \\ \textsc{True}) \mid h' \in \mathbb{H}_s[\![h]\!](E_T, E_S, \rho) \}.$$

Finally, making use of the above semantic functions, we define γ_s as follows:

$$\gamma_s(\sigma_s) = \bigcup_{\substack{E_S, E_T, \rho: \\ E_S, E_T \vDash \phi}} \mathbb{S}_s[\![\sigma_s]\!](E_T, E_S, \rho).$$

Note that ϕ is the path condition of the state σ_s.

Property 1 For all $\sigma_s \in \mathbf{State}_s$, if the path condition ϕ_s of σ_s is satisfiable, then $\gamma_s(\sigma_s) \neq \emptyset$.

Proof Since ϕ_s is satisfiable, there exist $E_T \in \mathbf{TEnv}$ and $E_S \in \mathbf{SEnv}$ that satisfy ϕ_s. Then we can construct a state $\sigma_c \in \mathbf{State}_c$ by applying E_T, E_S, the identity permutation ρ to σ_s and letting each undefined field/index in σ_s to be the default value of the type of the field/index. Clearly $\sigma_c \in \gamma_s(\sigma_s)$, and $\gamma_s(\sigma_s) \neq \emptyset$.

E.3 Function γ_a

We define some semantic functions:

$$\mathbb{H}_a : (\mathbf{Heap}_a \times \Phi) \to (\mathscr{P}(\mathbf{Symbol}) \times \mathscr{P}(\mathbf{SymLoc}) \times \mathbf{LEnv} \rightharpoonup (\mathbf{Heap}_s \times \Phi))$$
$$\mathbb{S}_a : \mathbf{State}_a \to \mathbf{LEnv} \rightharpoonup \mathscr{P}(\mathbf{State}_s).$$

The definitions are listed as follows.

- the $\mathbb{H}_a$ function: $\mathbb{H}_a[\![(h_a, \phi)]\!](S, \hat{S}, E_L) = (h_s, \phi')$ for some $h_s \in \mathbf{Heap}_s$ and $\phi' \in \Phi$ if the conditions in (3) hold; otherwise, the function is not defined. The h_s, ϕ', and the conditions are defined as follows:
  1. h_a is well mapped to h_s. More specifically,
     (a) the domain of h_a is mapped correctly,

$$\mathrm{dom}\, h_s = \mathrm{dom}\, h_a \cup E_L(\hat{S}).$$

     (b) each entry in h_a is mapped to h_s. Formally,

$$\forall l \in \mathrm{dom}\, h_a . h_s(l) = E_{L*}(h_a(l)).$$

[10] We adopt a shorthand notation $f(D')$ to represent a function whose domain is restricted to D' that is supposed to be a subset of the domain of f. Formally, if $f : D \to D$ and $D' \subseteq D$ then $f(D') = \{ f(d) \mid d \in D' \}$. For example, $\rho(\mathrm{dom}\, h_s) = \{ \rho(l'') \mid l'' \in \mathrm{dom}\, h_s \}$

(c) for each nonprimitive symbol in h_s that is not mapped from h_a, it must be a newly created symbol. Formally,

$$\forall l \in (\operatorname{dom} h_s - \operatorname{dom} h_a).h_s(l) = \alpha_\tau,$$

where

$$\alpha_\tau = \begin{cases} \mathit{new\text{-}sarr}(S \cup h_s(\operatorname{dom} h_s - \{l\})), & \text{if } E_L^{-1}(l) \text{ is of array type} \\ \mathit{new\text{-}sym}(S \cup h_s(\operatorname{dom} h_s - \{l\})), & \text{otherwise.} \end{cases}$$

2. ϕ' is the smallest set of predicates that satisfies the following conditions:
   (a) $\phi \subseteq \phi'$.
   (b) each symbolic location is mapped to a location that refers to an object of a compatible type. Formally,

$$\forall \hat{\alpha}_\tau \in \hat{S}.(\tau' <: \tau) \in \phi', \text{ where } h_s(E_L(\hat{\alpha})) = \alpha_{\tau'}.$$

   (c) for each array in h_s, ϕ' contains the constraint asserting that the length of the array ≥ 0.

$$\forall \hat{\alpha}_\tau \in \hat{S} \wedge \tau \in \textbf{AType}.(E_L(\hat{\alpha})(\textsc{len}) \geq 0) \in \phi'.$$

3. The function is defined if the following conditions hold:
   (a) each symbolic location is not mapped to a location that refers to a concrete nonprimitive symbol. Formally,

$$\forall \hat{\alpha} \in \hat{S}.\big(h_a(E_L(\hat{\alpha})) \uparrow \vee h_a(E_L(\hat{\alpha}))(\textsc{conc}) \uparrow\big).$$

   (b) ϕ' is satisfiable.
   (c) all symbols in h_a and ϕ are in S.
   (d) all symbolic locations in h_a are in $\hat{S}$.

– the $\mathbb{S}_a$ function (we use binding $\sigma_a = (g, pc, \xi, \omega, h, \phi)$): $\mathbb{S}_a[\![\sigma_a]\!](E_L)$ is not defined if $\mathbb{H}_a[\![(h,\phi)]\!](\mathit{symbols}(\sigma_a), \mathit{sym\text{-}locs}(\sigma_a), E_L)$ is not defined; otherwise,

$$\mathbb{S}_a[\![\sigma_a]\!](E_L) = (E_{L*}(g), pc, E_{L*}(\xi), E_{L*}(\omega), h', \phi'),$$

where $(h', \phi') = \mathbb{H}_a[\![(h,\phi)]\!](\mathit{symbols}(\sigma_a), \mathit{sym\text{-}locs}(\sigma_a), E_L)$.

Finally,

$$\gamma_a(\sigma_a) = \{\, \mathbb{S}_a[\![\sigma_a]\!](E_L) \mid E_L \in \textbf{LEnv} \wedge \mathbb{S}_a[\![\sigma_a]\!](E_L) \text{ is defined} \,\}.$$

Property 2 For all $\sigma_a \in \textbf{State}_a$, if the path condition ϕ_a of σ_a is satisfiable, then $\gamma_a(\sigma_a) \neq \emptyset$.

Proof Define an injective $E_L \in \textbf{LEnv}$ which maps each symbolic location in $\mathit{sym\text{-}locs}(\sigma_a)$ to a fresh location. Since ϕ_a is satisfiable, $\mathbb{S}_a[\![\sigma_a]\!](E_L)$ is defined. Therefore, $\mathbb{S}_a[\![\sigma_a]\!](E_L) \in \gamma_a(\sigma_a)$.

E.4 Function γ_b

$$\gamma_b(\sigma_b) = \begin{cases} \emptyset & \text{if the path condition of } \sigma_b \text{ is } \textsc{False}; \\ \{\, E_{R*}(\sigma_b) \mid E_R \in \mathit{legal\text{-}env}(\sigma_b) \,\} & \text{otherwise,} \end{cases}$$

where $\mathit{legal\text{-}env}(\sigma_b)$ consists of all $E_R \in \textbf{REnv}$ such that

1. E_R does not map any symbolic references to symbolic locations that appear in the state. Formally,

$$\forall \bar{\alpha} \in \mathit{sym\text{-}refs}(\sigma_b).E_R(\bar{\alpha}) \notin \mathit{sym\text{-}locs}(\sigma_b).$$

2. E_R does not map two symbolic references to the same symbolic location. Formally,

$$\forall \bar{\alpha}_1, \bar{\alpha}_2 \in \mathit{sym\text{-}refs}(\sigma_b).\bar{\alpha}_1 \neq \bar{\alpha}_2 \wedge E_R(\bar{\alpha}_1) = E_R(\bar{\alpha}_2) \Rightarrow E_R(\bar{\alpha}_1) = \textsc{Null}.$$

Property 3 For all $\sigma_b \in \textbf{State}_b$, if the path condition ϕ_b of σ_b is satisfiable, then $\gamma_b(\sigma_b) \neq \emptyset$.

Proof Since ϕ_b is satisfiable, then $\gamma_b(\sigma_b) = \{\, E_{R*}(\sigma_b) \mid E_R \in \mathit{legal\text{-}env}(\sigma_b) \,\}$. Define a $E_R \in \textbf{REnv}$ which maps each symbolic reference in $\mathit{sym\text{-}refs}(\sigma_b)$ to $\textsc{Null}$. Clearly, $E_{R*}(\sigma_b) \in \gamma_b(\sigma_b)$.

F Kripke Structure

Definition 1 (Kripke structure Schmidt (2000)) $\mathscr{K} = (\Sigma_{\mathscr{K}}, I_{\mathscr{K}}, \longrightarrow_{\mathscr{K}}, L_{\mathscr{K}})$, where

- $\Sigma_{\mathscr{K}}$ is a set of states;
- $I_{\mathscr{K}}$ is a set of initial states and a subset of $\Sigma_{\mathscr{K}}$;
- $\longrightarrow_{\mathscr{K}} \subseteq \Sigma_{\mathscr{K}} \times \Sigma_{\mathscr{K}}$ is the transition relation; we often call a sequence of transitions a trace.
- $L_{\mathscr{K}} : \Sigma_{\mathscr{K}} \to \mathscr{P}(Atom)$ associates a set of atomic properties, $L_{\mathscr{K}}(s) \subseteq Atom$, to all σ in $\Sigma_{\mathscr{K}}$.

We use unlabeled Kripke structures in the main text where $L_{\mathscr{K}}$ is omitted.

Definition 2 (Simulation of Kripke structures $\lhd_R$ Schmidt (2000)) Given

$$\mathscr{K}_1 = (\Sigma_1, I_1, \longrightarrow_1, L_1) \text{ and } \mathscr{K}_2 = (\Sigma_2, I_2, \longrightarrow_2, L_2),$$

$\mathscr{K}_1 \lhd_R \mathscr{K}_2$ (we read it as "$\mathscr{K}_1$ is simulated by $\mathscr{K}_2$") for $R \in \Sigma_1 \times \Sigma_2$ if and only if

$$\forall \sigma_1 \in \Sigma_1, \sigma_2 \in \Sigma_2.(\sigma_1, \sigma_2) \in R \wedge \sigma_1 \longrightarrow_1 \sigma_1' \Rightarrow \exists \sigma_2' \in \Sigma_2.\sigma_2 \longrightarrow_2 \sigma_2' \wedge (\sigma_1', \sigma_2') \in R.$$

The above simulation relation essentially states that if two states are related by a certain relation, the end states obtained by applying their own transitions are related by the same relation as well.

Definition 3 (Power Kripke structure Schmidt (2000)) Given any Kripke structure, $\mathscr{K} = (\Sigma_{\mathscr{K}}, I_{\mathscr{K}}, \longrightarrow_{\mathscr{K}}, L_{\mathscr{K}})$, the power Kripke structure of $\mathscr{K}$ is

$$\mathscr{P}(\mathscr{K}) = (\mathscr{P}(\Sigma_{\mathscr{K}}), \mathscr{P}(I_{\mathscr{K}}), \overset{\bullet}{\longrightarrow}_{\mathscr{K}}, L_{\mathscr{P}(\mathscr{K})}),$$

satisfying the following condition: for two sets of states $S, S' \subseteq \Sigma_{\mathscr{K}}$, $S \overset{\bullet}{\longrightarrow}_{\mathscr{K}} S'$ only if $\forall \sigma' \in S'.\exists \sigma \in S.\sigma \longrightarrow_{\mathscr{K}} \sigma'$.

We also use unlabeled power Kripke structures in the main text where $L_{\mathscr{P}(\mathscr{K})}$ is omitted.

G Proof of Lemma 1

Proof We prove part (1) by showing

$$I_{\mathscr{C}} \subseteq \bigcup_{\sigma_s \in I_{\mathscr{S}}} \gamma_s(\sigma_s) \quad \text{and} \quad I_{\mathscr{C}} \supseteq \bigcup_{\sigma_s \in I_{\mathscr{S}}} \gamma_s(\sigma_s).$$

$\subseteq$ direction: It is sufficient to show that for all $\sigma_c \in I_{\mathscr{C}}$ there exists a state $\sigma_s \in I_{\mathscr{S}}$ such that $\sigma_c \in \gamma_s(\sigma_s)$. Suppose that $\sigma_c = (g_c, pc_{init}, \xi_c, \emptyset, h_c, \text{TRUE})$ is in $I_{\mathscr{C}}$. We first construct a state $\sigma_s = (g_s, pc_s, \xi_s, \omega_s, h_s, \phi)$, and a primitive symbol environment E_S and a type environment E_T to facilitate the proof of $\sigma_c \in \gamma_s(\sigma_s)$ as follows.

1. Globals g_s are almost the same as g_c except that each primitive global, f, is replaced by a fresh primitive symbol. In addition, we add to E_S the mapping from $g_s(f)$ to $g_c(f)$.
2. $pc_s = pc_{init}$.
3. Locals ξ_s are treated the same as the globals.
4. $\omega_s = \emptyset$.
5. The heap h_s contains only the mapping from locations that appear in ξ_c and g_c. Furthermore, for each location l in the domain of h_s, $h_s(l)$ is mapped to a fresh nonprimitive symbol with a fresh symbolic type. Let $\alpha'_{\tau'} = h_s(l)$ and $\alpha_\tau = h_c(l)$. If τ is in **RType**, then all fields in α' are undefined; if the type is in **AType**, then only the LEN field of α' is defined as a fresh primitive symbol, and its indexes are undefined. In addition, we add to E_T the mapping from τ' to τ.
6. The path condition ϕ: if there is a reference from a global or a local of type τ to a nonprimitive symbol $\alpha'_{\tau'}$ in h_s, we add $\tau' <: \tau$ to ϕ; if this global or local is of array type ($\tau \in$ **AType**), then we also add to ϕ a constraint $\alpha'_{\tau'}(\text{LEN}) \geq 0$.

It is clear that $\sigma_s \in I_{\mathscr{S}}$. In addition, if we apply $\mathbb{S}_s$ to σ_s, aforementioned two environments E_S and E_T, and the identity permutation ρ, then get

$$\sigma_c \in \mathbb{S}_s[\![\sigma_s]\!](E_S, E_T, \rho).$$

Since $\gamma_s(\sigma_s) = \bigcup_{\substack{E_S, E_T, \rho: \\ E_S, E_T \models \phi}} \mathbb{S}_s[\![\sigma_s]\!](E_T, E_S, \rho)$, we can conclude that $\sigma_c \in \gamma_s(\sigma_s)$.

$\supseteq$ direction: From the definition of initial states introduced in Section 4.1 and the definition of γ_s, it is easy to deduce that for all $\sigma_s \in I_{\mathscr{S}}$, $\gamma_s(\sigma_s) \subseteq I_{\mathscr{C}}$. Therefore, we can conclude that $I_{\mathscr{C}} \supseteq \bigcup_{\sigma_s \in I_{\mathscr{S}}} \gamma_s(\sigma_s)$.

The proofs of part (2) and (3) can be done similarly.