Kiasan/KUnit: Automatic Test Case Generation and Analysis Feedback for Open Object-oriented Systems

Xianghua Deng, Robby, John Hatcliff
Department of Computing and Information Sciences, Kansas State University
234 Nichols Hall, Manhattan, KS 66506, USA
(deng, robb, hatcliff)@cis.ksu.edu

ABSTRACT
Significant progresses have been made on using symbolic execution to generate unit test suites, but existing frameworks are still struggling with several issues including poor performance on heap data, lack of coverage goals tied to the space of heap configurations, and lack of support for dealing with open object-oriented systems and generating mock objects.

In this paper, we demonstrate how a static analysis feedback and unit test case generation framework, KUnit, built on the Bogor/Kiasan symbolic execution engine addresses these issues by: (a) providing an effective unit test case generation for sequential heap-intensive Java programs (whose computation structures are incomplete – open systems), (b) showing how the scope and cost of Kiasan/KUnit’s analysis and test case generation can be controlled via notions of heap configuration coverage, and (c) leveraging method contract information to better deal with open object-oriented systems and to support automatic mock object creation. In a broad experimental study on twenty-two Java data structure modules, we show that KUnit is able to: (a) achieve 100% feasible branch coverage on almost all methods by using only small heap configurations, (b) improve on competing tools for coverage achieved, size of test suites, and time to generate test suites.

1. INTRODUCTION

Unit testing – a quality assurance method in which individual software system modules (i.e., units) are tested in isolation, is an integral part of many development processes. Successful unit testing requires several important steps.

First, a suite of unit tests must be created that call the public methods of the unit with method input parameter values that drive the execution of the unit down particular execution paths. The number of unit tests and choice of parameter values are usually driven by multiple factors including: (1) the need to cover all the different combinations of values that might get passed into the unit when it is deployed within the context of a complete system (i.e., context value coverage), and (2) the need to obtain a certain level of code coverage for the unit (e.g., certain levels of statement or branch coverage). Factor (1) may be aided by documentation about a unit’s required behavior or by formal or informal unit interface contract specifications (e.g., such as those that result for a Design-By-Contract methodology [22]). In the presence of such specifications, the test suite construction goals associated with (1) can be presented as aiming to achieve a certain level of specification coverage.

Second, one must design a collection of test oracles that can be invoked to automatically determine if the output of a test is correct. As with (1) above, the construction of oracles can be guided by the presence of documentation or contract specifications (e.g., the post-condition of a method contract can often be directly translated into a test oracle).

Third, because units are tested in isolation, some sort of executable behavior must be defined for any method invocations that the unit makes into the enclosing context (we view such units as having incomplete computation structures – open systems). Such behavior is often implemented in what are termed mock objects – dummy objects with method stubs that implement fragments of the functionality that exists in the real objects that the unit interacts with when deployed in a system context. The partial functionality of the mock object might: (a) use assertions to check for properties of parameter values flowing from the unit into the mock object, (b) simulate side effects that might occur during the execution of the context method, (c) generate simulated method return values that get passed from the context back into the unit. Methodologies for constructing mock objects are very ad-hoc (e.g., there is hardly ever an attempt to relate the behaviors of mock and real objects as over/under-approximations), and the degree to which the functionality of real object is captured by the mock object varies widely.

The burden of constructing a test suite that satisfies the conditions above is often onerous, yet research in recent years has made substantial progress toward automating one or more aspects of unit test suite construction. A particularly interesting line of work uses symbolic execution (SymEx) as an alternative analysis framework for testing and bug finding [21, 17, 32, 18, 27, 28, 10]. One key advantage of symbolic execution over real/concrete execution (e.g., traditional testing) is that one can avoid the burden of constructing numerous concrete input parameter values and instead use symbolic values and constraints to compactly represent sets of possible input values.

Despite significant progress, existing symbolic execution based frameworks are still struggling with several issues.

- Poor performance on heap-data: Many existing frameworks [15, 14, 16] focus primarily on non-object-oriented systems and often do not perform well at all on code that is heap-intensive.

- Coverage goals not tied to space of heap configurations: Among those that do treat heap data, some methods [27, 10] aim to generate test suites that achieve certain levels of branch coverage instead of stronger notions of heap configuration coverage. Thus, these techniques miss errors that result from aliasing scenarios and other heap configurations

This work was supported in part by the US National Science Foundation (NSF) awards 0453438, 0429141, and CAREER award 0644288, the US Air Force Office of Scientific Research (AFOSR), and Lockheed Martin Advanced Technology Laboratories.
whose conditions are not directly exposed in branch tests.

**Search-bounds not tied to coverage bounds:** It seems preferable to have techniques in which increasing the bounds (and cost) of the analysis search directly corresponds to increasing the amount of coverage achieved, since this would allow testers to more confidently judge the merits of decreasing or increasing analysis costs. Yet, existing strategies almost always bound the search by the number of steps in depth-first search rather than other notions more strongly tied to behavioral coverage.

**Little or no support for mock object creation:** Most approaches focus on generating mock objects for open systems whose inputs are scalar and data structure values. That is, the input values do not contain behavior such as interface implementations, which is crucial for systems, for example, developed using design patterns [13].

In previous work, we developed Kiasan [7] – a symbolic execution engine for sequential Java programs built on the Bogor model checking framework. In this paper, we extend Kiasan with a unit test case generation framework called KU nit that automatically generates both JUnit tests from contract-annotated method implementations as well as visualizations of heap objects flowing in and out of methods that can be very useful to developers for understanding complex methods and diagnosing the cause of program errors.

Others [27] have argued that the systematic exploration of heap configurations used by Kiasan/KUnit and others [17] is computationally intractable for large or complex units. Indeed, previous work on symbolic execution for Java has included very little in the way of experimental results (e.g., only one example is treated in [30]) that would contradict these claims. One of the primary contributions of the work in this paper is to present an experimental study involving twenty-two Java data structure examples that assesses the performance not only Kiasan but also of two of the algorithms used in the most closely related tools jCUTE [27] and JPF [30] – this is, by far, the most comprehensive study of symbolic execution techniques for object-oriented programs undertaken to this point. Counter to the arguments of [27], our experiments show that for the examples that we considered, KU nit’s performance is almost never worse than that of jCUTE [27] and JPF [30], and in most cases it is significantly better (e.g., reducing time to achieve high levels of coverage from multiple hours down to a few minutes).

The specific findings and contributions of our work are as follows:

- **Effective with heap data:** We show that KU nit can generate JUnit test suites that obtain 100% feasible branch coverage on almost all of our 22 heap-intensive Java data structure modules.
- **Bounded controlled heap configuration coverage:** KU nit uses an enhancement of earlier work on lazy initialization [17] to systematically enumerate heap configurations. In contrast to earlier work, however, KU nit uses a search bounding strategy that allows direct control over the coverage of heap configurations by bounding the search based on the length of heap reference chains. Our studies show that a relatively small reference-chain length bound of 2 is sufficient to achieve 100% coverage of feasible branches for almost all of the examples that we consider.
- **Rigorous foundations:** KU nit’s path sensitive exploration has been proven sound (guaranteed to generate tests for all execution paths up to its bounding strategy), and avoids sources of unsoundness present

![Figure 1: A Symbolic Execution Example](image)

in other frameworks based on lazy initialization [17]. Our recent work [9] demonstrated that Kiasan/KU nit’s lazier initialization algorithm generates a minimal number of non-isomorphic heap states for several of the most complicated examples considered in works on Java symbolic execution. This implies that KU nit avoids redundant test case generation and provides a minimal test suite with respect to the number of non-isomorphic heap configurations generated.

- **Strong interface contract integration:** KU nit makes extensive use of method contracts (phrased as executable specifications) to prune test cases that would not be generated by valid system contexts.
- **Contract-based mock object generation:** KU nit can automatically generate mock objects summarizing execution cases based on method contracts.

This rest of the paper is organized as follows. Section 2 summarizes the Kiasan symbolic execution framework. Section 3 describes the KU nit approach for deriving concrete heap and scalar values from symbolic path information. Section 4 explains how these concrete values are presented as object diagrams, JUnit test cases, and presents our approach for generating mock objects, Section 5 presents experimental results, Section 6 compares our approach to related work, and Section 7 concludes. We encourage the reader to consult the Papers section of the Bogor web-site which includes: (a) an accompanying technical report [8] that provides the detailed presentation and discussion of our experimental results, KU nit’s formal semantics, and relative soundness and completeness proofs, and (b) the source code, specifications, generated test suites and heap graph visualizations for all our experiments.

2. BACKGROUND

Symbolic Execution Basics: King proposed symbolic execution [19] as a technique for program testing and debugging. One key advantage of symbolic execution over real/concrete execution (e.g., traditional testing) is it can reason about unknown values which are represented as symbols instead of concrete values (e.g., integers). While executing a program, it maintains the relationship on symbol as a constraint (φ). Figure 1 illustrates the symbolic computation tree of an example method abs (each tree node is a symbolic state (x, φ) that associates a symbol to x and predicate φ that constrains the symbols). 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 a constraint φ set to true (no constraints imposed yet). When executing line 2, the symbolic execution does not have sufficient information to decide which branch to take because φ =⇒ (x < 0) and φ =⇒ ¬(x < 0) are both 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 the
public class Node<E> {
  /* @ ensures data == old(n.data) */
  @ &.n.data == old(data); @ /
  public void swap(@NonNull Node<E> n) {
    E e = n.data; data = n.data; n.data = e; }
  private Node<E> next; E data;
  // @ ensures n.data == old(data) &. data == e;
  public void foo(@NonNull Node<E> n, @NonNull E e) {
    n.data = data; data = e; }
  /* @ requires acyclicOnNext() &. ...;
    @ ensures sorted(c)
    @ &. elements().equals(\old(elements())); @ */
  public void sort(@NonNull Comparator<E> c) {
    ... c.compare(... ... ) ... }
}

public interface Comparator<E> { ...
  // @ensures \result>0 | | result==0 | | \result<0;
  @Pure int compare(E e1, E e2); // total order
}

Figure 2: A List Example

path condition as it represents the conditions on variables that would be necessary for execution to flow down the current path. A collection of concrete values for program variables that satisfies the path condition can be used to form a test case that drives execution along the path. If the path condition becomes false, the path is infeasible hence abandoned. The example shows that the true branch at line 4 is infeasible.

Handling Objects: An extension of symbolic execution to handle heap objects is “lazy initialization” [17]. The lazy initialization algorithm starts with no or partial knowledge of object values (i.e., symbolic objects whose fields are unitialized) referenced by program variables. As the program executes and accesses object fields, it “discovers” (i.e., materializes) the field values on an on-demand basis (i.e., hence the term “lazy initialization”). When an unmaterialized field is read, if the field’s type is a scalar type, then a fresh symbol is created for that scalar value. Otherwise, for an unmaterialized reference field, the algorithm (based on an underlying model checking engine) systematically (safely) explores all possible points-to relationships by non-deterministically choosing between the following values for the reference: (a) NULL, (b) any existing symbolic object whose type is compatible with the field’s type, or (c) a fresh symbolic object (whose type is constrained to be equal to or a subtype of the field’s type).

In [9], we introduced the lazier# algorithm which significantly improves the lazy algorithm. The lazier# algorithm uses three forms of abstractions for objects: (1) a symbolic object (as in the lazy algorithm) represents a concrete object that has been materialized to the point where storage for the object and a reference (pointer) to the object has been introduced (but some fields of the object may have yet to be materialized), (2) a symbolic reference represents a placeholder for a reference (non-null) that has yet to be introduced – it thus represents a pending choice over a set of symbolic objects, and (3) a symbolic value represents a pending choice between either a NULL value or a symbolic reference. Thus, the lazier# algorithm can be described as follows. Step 1, when an unmaterialized variable is read, it is initialized with a fresh symbolic value. Step 2, if the symbolic value is compared to NULL, symbolic value is non-deterministically replaced with either NULL, or a fresh symbolic reference. If a field of a symbolic value is accessed, the symbolic value is non-deterministically replaced with NULL (which results in raising a null dereference exception), or a fresh symbolic reference and the field access proceeds on the symbolic reference. Step 3, accessing a field of a symbolic reference causes a non-deterministic choice over symbolic objects as in the lazy initialization algorithm. Step 1 can be bypassed by directly using a fresh symbolic reference if the variable is known to be non-NULL (e.g., as specified in a contract pre-condition). Note that purpose of moving from a single abstraction form in lazy initialization to three forms in lazier# is to delay the non-deterministic choice/branching over symbolic objects as long as possible (or avoid it altogether in some cases), thus reducing both the number of states and the number of paths through in the symbolic execution tree. This optimization yield orders of magnitude improvement over the lazy initialization algorithm (as will be discussed in Section 5).

To illustrate the lazier# initialization algorithm, let us examine the swap example shown in Figure 2. Figure 3 illustrates the symbolic execution tree using the lazier# algorithm and an trace (along with its states and their sibling states). Symbolic references are annotated with '; and symbolic (reference) values are annotated with ‘#’. The algorithm starts with one state (State 1). Notice that n refers to the symbolic reference n1 instead of a symbolic value because of the non-NULL precondition on n allows the symbolic value step to be bypassed. When executing swap’s first statement, n0 is replaced with a fresh symbolic object n0, and its data field is initialized with a fresh symbolic value e0, thus resulting in State 11. Continuing on with executing swap’s second statement, n1 is replaced with either the existing symbolic object n0 or a fresh symbolic object n1. In the former case, n0’s data field has been initialized, thus no special treatment is needed (111). In the latter case, n1’s data field is initialized with a fresh symbolic value e1 (112). From 112, it produces 1121. As can be observed, 1121 satisfies swap’s postcondition expressed using the JML (Java Modelling language) [20] (\old(e) refers to the evaluation of e at the effective prestate; the construction of the effective prestate is described in 3).

We have formalized the semantics of the lazier# algorithm and proved that it simulates Kiasan’s (relatively sound and complete lazy/lazier initialization algorithms [7], and thus simulating a concrete execution. Interested readers are referred to [9] for detailed discussion on the lazier# algorithm (semantics) and its associated empirical study.

Controlling heap coverage via k-bounding: Due to the exponential nature of symbolic execution with respect to program paths, various approaches have been proposed to control its analysis cost. [17, 27] use bounding on the length of execution path. While useful, it is difficult to assess the behavior coverage that they achieve on heap-oriented properties. That is, if no errors are found, one does not gain confidence about the true absence of errors in a nicely characterized portion of the system state-space. One only knows that errors are absent in the states encountered along the particular execution traces that the tool happens to choose, e.g., to achieve some form of coverage (e.g., statement and branch coverage). [17, 4] use bounding on the number of objects of each type (or in general, the number of objects in the heap). While this is better suited for object-oriented programs than the length of program path, it still does not give a reasonable coverage on heap structures (it is hard to characterize heap configurations that are explored).

In contrast, Kiasan uses a k-bounding strategy on the length of lazy initialization (reference) chain that gives a fine-grained control over behavior coverage for strong heap-
oriented properties (more appropriate for object-oriented properties) while still allowing one to manage the analysis cost [7]. For example, with this strategy, one can specify the analysis to consider programs manipulating list, trees, and other structures up to a certain size. Based on the relative soundness of Kiasan, one can conclude that if it does not find an error, no error exists for any heap configuration up to the size determined by the $k$-bound. This gives us a methodology to apply Kiasan on an incremental basis where we systematically increase the portion of the heap state space in which we are completely confident that no error exists.

Thus, this strategy provides a quantifiable behavior coverage on the space of points-to/aliasing relationships that goes beyond traditional branch/statement coverage.

Note that a stateless state-space exploration with $k$-bounding may not terminate. To address this, we employ a loop (recursive) bounding strategy. When used, this approach cannot produce conclusive behavior guarantee. However, Kiasan is able to notify users when such case occurs; it can even point out the program point and the state under which such condition occurs. Thus, users can be properly warned when errors may be missed due to bound exhaustion. On the other hand, Kiasan can notify users when its ($k$ and loop) bounds are not exhausted, which means that it achieves complete behavior coverage (e.g., for the swap method in Figure 2).

Optimality: We have presented rigorous justifications that the lazier# algorithm is case-optimal on a number of complex data structures such as binary search tree, AVL tree, and red-black tree [9]. That is, it generates the minimum number of non-isomorphic state configurations for a given $k$-bound. This means that test-case generation techniques semantically consistent with the lazier# algorithm will give the minimum number of non-isomorphic test inputs (i.e., they are not redundant with respect to heap shapes generated); we introduce such a semantically consistent technique in the next section.

To demonstrate the case-optimality of the lazier# algorithm, we leverage a standard combinatoric technique called generating function [31] to count the number of non-isomorphic state configurations (cases) for any given $k$-bound. The generating function-based counting method generates the same numbers of cases as the lazier# algorithm explored for the three tree examples. Interested readers are referred to [9] for a detailed presentation of our case counting method.

3. FOUNDATIONS FOR KUNIT

To generate test cases, we need to concretize the input configuration for each symbolic path. This is not straightforward to do when using a lazy initialization algorithm because the initial symbolic state does not yet have sufficient heap information (recall that heap structure is discovered in an on-demand basis); one needs to reconstruct the heap structure at a method (unit) entry (pre-state) by using the information along the path to the method’s exit points (post-states). This suggests that it might be possible to work backward from a post-state to the initial state. Fortunately, Kiasan is implemented on top of the extensible Bogor model checking framework [24] that provides a backtracking capability for “reverse” execution. However, the backtracking facility cannot be used as is; otherwise, one would get the same exact initial state that the algorithm begins with.

We have formalized (and proved its consistency with Kiasan’s algorithms) a modified reverse execution algorithm for KUnit that, given a path, produces a corresponding symbolic pre-state with sufficient heap information for concretization. In addition, we have formalized KUnit’s concretization algorithm for producing a concrete state that refines a given symbolic pre-state. This section presents the intuition behind KUnit’s algorithms; interested readers are referred to [8] for KUnit’s formal semantics, consistency, and refinement proofs. In the next two paragraphs, we describe the two main steps in our approach: (1) constructing symbolic input states that preserve materialized heap structures by using a modified backtracking algorithm, and (2) concretizing the symbolic input states into concrete states.

Constructing effective symbolic input state: Intuitively, for any symbolic execution trace $s_1 \Rightarrow \cdots \Rightarrow s_n$, we can give the path condition at the end state $s_n$ to a constraint solver and get an instance of assignments of scalar values to primitive symbols. As mentioned above, we work backward from the end of each path and propagate the heap materialization to the initial state to obtain what we will call the effective initial state. Destructive updates to the heap complicate this further, however, these are naturally handled using the infrastructure of the backtracking facility employed in Bogor. To illustrate why a modified approach to backtracking is needed, consider that when using the normal backtracking approach, one would undo the materialization that resulted from lazy initialization and restore the current state to the pre-state of a transition. Since our goal is to actually propagate backward such materialization, we keep the lazily initialized heap objects intact in the undo step and do not “de-materialize” them. In addition, we keep the state’s path condition intact. Although the path condition includes symbols (and their associated constraints) that are not mapped by variables at the input state, they do not harm the process of concretizing values (one can remove such irrelevant symbols/constraints using transitive dependence information to determine if they are required for reasoning about variables at the input state). For example, consider Figure 4 that illustrates the construction of swap’s expanded symbolic input state from State 1121. From 1121, the algorithm backtracks to 112′, then to 11′, and finally to 1′. Notice that when backtracking the first three statements of swap, we do not de-materialize lazily initialized objects. In addition, notice that State $x'$ in Figure 4 refines State $x$ in
Concretizing the effective symbolic input state: As mentioned above, when concretizing the effective symbolic input state, we use a constraint solver to get non-heap related value instantiations. For concretizing the remaining symbolic heap structures, there are four possible symbolic forms present in the state: (1) un-initialized fields, (2) symbolic values, (3) symbolic references, and (4) symbolic objects. For uninitialized fields, we use default values according to the field's type. For each symbolic value/reference/object, we use a fresh object whose type satisfies the type constraints in the state path condition, and all of its fields' values are default values according to the fields' type. Note that this strategy works well with contract specifications because Kiasan embeds executable, effective (e.g., invariants transformed as parts of pre-/post-conditions) contracts along with the code [7]. That is, if a field is unconstrained even under the specified contracts, then using any value is fine. For the swap example, the symbolic objects n0 and n1 are concretized as Node objects. The uninitialized fields n0.next and n1.next are concretized with the NULL value, and the symbolic values e0 and e1 become fresh java.lang.Object.

4. TEST CASES AND OBJECT GRAPHS
Generating JUnit Test Cases: KUnit generates JUnit test cases using the concretization algorithm described in the previous section. Following the Design-by-Contract (DBC) methodology [22], each test case has the following structure:

```
"assume" effective pre-condition
invoke the method being tested
assert effective post-condition,
```

where "assume" is the code that builds the method input state using calls to Java's reflection APIs. While using reflection produces hard-to-read code, one can address this by adopting the JUnit-Objects [35] approach that uses XML descriptors to describe objects to be created.

Visualizing Effects using Object Graphs: To accompany each JUnit test case generated, KUnit visualizes a method's behavior by producing object graphs of the method's pre-/post-states. This is useful to provide a quick view of each of the test cases. It is interesting to note that as a consequence of the lazier# algorithm, the object graphs are focused on the heap objects that are accessed by Kiisan (this is in contrast to [4] that generates heap structures whose parts may not be accessed). Figure 5 presents a pre-/post-state pair for one test case of a put operation.

Figure 5: Red-Black Tree put Pre-/Post-states

Figure 3 for $x \in \{1, 11, 12\}$.

public class SortTest extends TestCase {
  int index = 2;

  public void testSort() throws Exception {
    Comparator c = new Comparator() {
      public int compare(Object o1, Object o2) {
        index -= ...;
        switch (index) {
          case 1: return c1(this, o1, o2);
          case 0: return c0(this, o1, o2);
          default: throw new Error();
        }
      }
    }
    // build input state using reflection
    // call sort with c on input state
    // check post-condition
    ...
  }

  static int c0(Comparator c, Object o1, Object o2) {
    return -10;
  }
}

Figure 6: A sort Test with Mock Object (excerpts)

tions red-black tree implementation. As can be observed, the value field in the two entry objects at the pre-state are missing. We chose to not show such fields to indicate that those fields are not accessed by the method (the JUnit test case input state has those fields' values equal to NULL, or default values in general). We believe this visualization has wide applicability for code inspection and understanding. For example, one can use this feature to try to understand the behavior of a fragment of "legacy" code by having KUnit quickly generate a variety of input/output pairs (which can be viewed as use cases of the code fragment). This is also useful for understanding how a given contract constrains the input states of a method (e.g., helpful when drafting contracts).

Closing Open Systems using Mock Objects: To generate test cases for open systems with incomplete computation structure, one also needs to generate method implementations required to close the unit. For example, we need to generate an implementation of the Comparator interface passed as an argument to the sort method in Figure 2. This essentially amounts to generating mock objects.

In our approach, we generate mock objects by following the DBC methodology. Thus, each mock method m has the following basic structure:

```
assert m's effective pre-condition
"assume" m's effective post-condition
```

That is, the "assume" part simulates the effect of the mock method. For example, considering compare's contract in Figure 2, we can initially design a mock object by simply having a method body that non-deterministically returns a negative, zero, or a positive value.

However, one complicating factor when generating mock objects is that mock methods may be called multiple times using different calling contexts (even within a single test case). Thus, one has to summarize the behavior of the mock method for all the contexts. To address this, we leverage information about the sequence of method invocations from the symbolic execution path to create mock methods. That is, we remember the order number of when a mock method is invoked, and we use the order number to index a behavior that simulates the effect of that particular invocation. We store the behavior of each invocation in a separate helper method. The overall mock method behavior is then memoized [23] by indexing the invocation to the helper methods. For example, Figure 6 illustrates a test case generated for the
sort method in Figure 2. In this case, there are two objects in the input state, thus, there are two compare invocations. One is inside sort and another one is in its post-condition (i.e., sorted). For each invocation, KUnit creates a helper method such as c0 (which returns -10), illustrating that in the last invocation, compare determines that the first argument is less than the second. Thus, the behavior of the Comparator c is spread to the helper methods c0 and c1 (not shown). To keep track of the ordering, we use an indexing counter, which decides which helper method should be invoked as implemented in the compare method in Figure 6.

Note that since compare is a pure method, we do not need to simulate side-effects using reflection. In general, side-effects are simulated using the same strategy used to build the input state, however, we do not need to rebuild the entire state. It is enough to construct parts of the state that are affected according to the specified contracts. Without contracts, KUnit assumes that there is no side-effect (i.e., best case). However, fresh symbolic (scalar/reference) values are used for return values (i.e., worst case). In short, KUnit’s soundness is relative to user-supplied contracts.

The indexing strategy for memoizing behavior works well for demonstrating the paths explored by symbolic execution. We believe this approach already provides significant automation for developers as well as providing incentive to adopt the DBC methodology. That is, by using contracts instead of writing numerous test cases manually (which is labor intensive and difficult to get good coverage with), one can benefit from Kissan’s strong static checking, and KUnit’s effective visualization and automatic test case generation. Moreover, contracts can serve as a formal basis for documentation purposes instead of using often ambiguous natural language descriptions.

Ideally, however, one would like to generate mock method behavior based on the context. That is, instead of memoizing the result value of each invocation, one would prefer to decide the result value based on the context. For example, instead of returning -10 in c0, one would prefer to decide the return value based on c1 and c2. This would make the test case insensitive to method invocation orderings, thus, it opens the possibility of reusing test cases when the method being analyzed (e.g., sort) is modified (e.g., when the sequence of mock method invocations may be altered).

In the context of open systems with incomplete computation structures and weak contracts such as the one for compare that does not specify the relationship between return values/side-effects and its contexts, this cannot easily be done. One can employ a heuristic strategy, for example, by using Integer objects when creating the input state of sort and leveraging the natural orderings of Integers when creating an implementation of compare. However, the problem in general is difficult (i.e., heuristics can only be applied to a known set of interfaces), and we believe it is still a challenging research issue in open object-oriented systems. Even in practice, it is still hard to determine which test cases become stale after code modification, let alone generating test cases that are impervious to code modification.

In short, we believe that our approach serves the purpose for generating test cases. In the case invocation orderings are changed, the test cases may fail, but we encourage users to leverage contracts to statically re-check the modified code. The generated test cases themselves are evidence/provide feedback that our static analysis performs as it is supposed to (even when it does not find errors). We believe that our approach can be improved by employing heuristics strategy for most commonly used interface, while further investigations are needed to address the general problem.

5. EVALUATION

To evaluate the approach presented in the previous sections, we collected and analyzed twenty-two different Java data structure modules (many modules include multiple classes for implementing different components of the same data structure) from both standard libraries such as classes in the Java 5 Collection Framework and in textbooks on Java data structures. The total example collection contains about 150 (public and helper) methods. For each module considered, the methodology that we followed included writing a single invariant (repD) that characterized the correctness of the data structure representation. Using the invariant information, our approach guarantees that all generated tests have test inputs that satisfy the invariant, and the output of each test is automatically checked against the invariant. Additional lightweight contract information for reference non-NULL-ness was also used. Our approach generates tests for the public methods of each class. Private helper methods are covered in the path exploration and coverage goals, but only as they are invoked from public classes. Test cases are not generated directly for private helper methods because the class invariant sometime does not hold at the pre/post-state of private methods (as these are intended to work at intermediate private states in which the data structure invariant does not hold).

Our experiments aimed to answer the following questions:

- **Coverage levels**: What level of branch coverage is achieved as the value of the k-bound increases? In particular, what value of k is necessary to achieve 100% branch coverage? Can we always achieve 100% branch coverage?
- **Computation cost**: What is the cost (time) as we increase the scope of the analysis? Can high levels of coverage be obtained within time constraints that are feasible within a typical development context?
- **Size of test suite**: How many test cases result from different values of k? How many test cases are needed to achieve high levels of branch coverage?
- **Annotation burden**: What is the comparative size of required invariants (specifications) and method source code?
- **Comparison to other approaches**: How does Kissan/KUnit’s performance (time, achieved coverage, size of test suite), compare to related approaches?

Table 1 presents excerpts of the data from our experiments conducted using a 2.4 GHz Opteron Linux workstation with 512 MB Java heap size. Please note: the columns for test cases and total time labeled lazy are for KUnit data whereas the columns labeled lazy contain data for the lazy initialization algorithm of JPF; otherwise the columns pertain to KUnit only. All discussion in the text below pertains to KUnit data (JPF data is discussed in a separate section below). Due to space constraints, we are only able to show the data for selected modules and selected methods from within these modules. We have omitted data for simpler modules; for these omitted modules, KUnit was able to achieve 100% feasible branch coverage, and there were no cases in which the processing of a method exceeded 10 seconds (to achieve 100% feasible branch coverage). Note that since our analysis is done at the bytecode level, our branch coverage is actually multiple condition coverage (MCC) (i.e., the Java compiler transforms/short-circuits && and || to ifs and gos). For the shown modules, we show the
Table 1: KUnit Experiment Data (excerpts); with Lazy columns giving data for the JPF algorithm

```java
private void fixAfterInsertion(Entry<Y> x) {
    x.color = RED;
    while (x != null && x != root && x.parent.color == RED) {
      Entry<Y> y = rightOf(x.parentOf(x.parentOf(x)));
      if (colorOf(y) == RED) {
        setColor(x.parentOf(x), BLACK);
        setColor(x, BLACK);
        setColor(x.parentOf(x), RED);
        x = parentOf(x.parentOf(x));
      } else {
        if (x == rightOf(x.parentOf(x))) {
          rotateLeft(x);
          rotateRight(x.parentOf(x));
        } else {
          rotateRight(x.parentOf(x));
          rotateLeft(x);
        }
      }
    }
}
```

Figure 7: TreeMap.fixAfterInsertion method (excerpts) data for the 1-3-3 complex methods from the respective module. The complete data set, source code, generated object graphs, and test cases for all methods and modules can be found on the Bogor website [34]. The columns in the table report on the size of k, number of test cases generated, bytecode-level branch and instruction coverage, total running time (excluding time for generating object graph visualizations) followed by a breakdown of this time spent in the CVC Lite theorem prover [3], time to run the concretization algorithm (reverse execution \(\Rightarrow^r\) and POOC [26]), and time to form the JUnit tests. Finally, the last column gives the additional time not included in the total time column to generate the object graph visualizations.

Coverage Levels: As expected, the cost of analysis and number of test cases grows exponentially as \(k\) increases. Thus, we were encouraged (and somewhat surprised) to find that a value of \(k = 2\) was sufficient for achieving 100% branch coverage in almost all cases, and \(k = 3\) was required to get the highest levels of coverage for only the TreeMap and binary heap examples. We have confirmed and documented in [8] that almost all cases in which 100% branch coverage is not achieved represent pathological cases of infeasible branches/dead code. Only for TreeMap did we encounter feasible branches (2 for put, 3 for remove) that were not covered at \(k = 3\). As an example, Figure 7 shows the fixAfterInsertion helper function (called from put after a node is inserted in the red-black tree). The inserted node \(x\) is colored as RED as shown in line 2. The goal of this method is to adjust the inserted tree to satisfy the red-black tree invariant. The only red-black tree invariant that the new tree could violate is “all children of a RED node have to be BLACK” because the parent node of the inserted node could be RED too. KUnit reports (and we confirmed) that this method contains three infeasible paths related to the loop invariant. The main loop from line 4 (we only show half here and the other half is symmetric) deals with the case that node \(x\) is RED and its parent also RED. Part of the loop invariant is that \(x\) is non-NULL and RED before entering the loop; inside the loop, \(x\) is either unchanged or moved up the tree to the parent of parent of \(x\) which can not be NULL because the parent of \(x\) is RED and the root of the tree is BLACK and further \(x\) is colored with RED. Thus, \(x\) is NULL at line 4 and the continue.
ditional in line 19 are always true. Symmetrical to line 19, there is a similar infeasible path not included in Figure 7 due to space constraints.

**Computation Cost:** The java.util.TreeMap example is the most complex from our study. The two most complicated methods of this example required 2.2 minutes in both cases for \( k = 3 \). The total running time for a vast majority of the methods is under a few seconds. When considering future work, it is important to note that typically one-half to two-thirds of the time is spent in the theorem prover. There are significant opportunities for optimizations (e.g., caching results across theorem proving calls), but these will require us to work with the developers of CVC Lite to design a new collection of top-level APIs.

**Size of Test Suite:** As noted earlier, the data clearly indicates exponential growth in the number of tests as \( k \) increases. In most cases (considering omitted data as well), the number of tests generated for a \( k \)-bound that yields 100% feasible branch coverage is under 10 seconds. Note that our numbers of tests are what results from considering all heap configurations within the \( k \) bound -- not the number of tests required to achieve 100% branch coverage. In some situations, test cases beyond what are necessary to achieve 100% branch coverage are generated. It is relatively easy to modify Kiasan/KUnit so that it avoids exploring a path after branch coverage has been achieved. We are implementing this modification for the purpose of measuring the number of test cases beyond those required for 100% branch coverage.

**Annotation Burden:** The size of the required invariant method is relatively small compared to the overall size of the classes. For instance, in our most complicated example (TreeMap), the invariant is 74 non-comment source statements (NCSS) while the total NCSS is 414 (invariant is 17.87% of total). Across all of our examples, the invariant is typically 10-18% of the entire class NCSS. It is worth noting that from a single invariant, one is able to obtain a huge benefit in the form of automatically generated test suites. We believe the disparity between the effort required to write an invariant and the effort required to write tests manually (which iteratively trying to achieve a coverage) is so great as to render negligible any complaints about the effort required to provide the invariant annotations.

**Comparison to jCUTe [27]:** We have been able to consider jCUTe’s capabilities against all the examples for which KUnit has been applied. Due to space constraints, we simply summarize the main findings of our experiments: the complete data and associated assessment can be found in [8]. jCUTe was not able to handle roughly half of our examples because it does not currently handle arrays where the size is unspecified at compile time. For those examples that jCUTe was able to handle, the performance of the tool was similar on simple examples, but on more complicated examples, it was often the case that: (a) the coverage achieved by jCUTe fell far below the coverage achieved by KUnit, and (b) the run-time of jCUTe was significantly longer than that of KUnit. Again, we do not conclude that the performance of KUnit is uniformly better than that of jCUTe on all programs; we are corresponding with jCUTe creators to identify examples in which jCUTe might perform better.

Figure 2 presents the jCUTe experiment data (\( d \) is the search depth used; \( \#r \) is the number of path iterations; \( \#e \) is the number of tests generated). For BinarySearchTree (remove), with a runtime approximately the same as KUnit, jCUTe was able to obtain coverage for 15 of the 16 branches (same as KUnit). jCUTe needed to explore 321 paths, while KUnit’s only needed to explore 236. However, jCUTe has an (orthogonal) post-processing phase that KUnit currently omits that is able to find a collection of only 11 test cases that are able to achieve 15/16 coverage. For AVLTree, jCUTe performance is comparable to KUnit on methods find, findMax, and findMin. For each of these cases, the numbers for KUnit are actually better, but still within the same order of magnitude. For example, for find, KUnit achieves 100% branch coverage on \( k = 1 \) in 1.4s (compared to 5.9s for jCUTe) and generates 4 tests instead of 13. For findMax, KUnit achieves 100% branch coverage on \( k = 2 \) in 3.7s (compared to 13s for jCUTe) and generates 5 tests instead of 11. However, the performance diverges significantly for the insert method. jCUTe obtains only 12/18 branch coverage (compared to 18/18 for KUnit), and ran for over 1 hour (compared to 8.8s for \( k = 2 \) for KUnit -- the point at which 18/18 coverage was reached). For TreeMap.remove, after running for over one hour with a depth bound of 25, jCUTe was only able to cover 16 of 86 (73 feasible) branches (compared to KUnit’s coverage of 70/86 in 2.2 minutes). For TreeMap.put, after running for 1 hour and 40 minutes with a depth bound of 25, jCUTe was only able to cover 15 of 52 (44 feasible) branches (KUnit covers 42/52 in 2.2 minutes).

In contrast to Kiasan, jCUTe does not systematically explore all points-to combinations, thus, it may miss errors due to aliasing relationships that are not exposed in the tests of conditionals (jCUTe drives the exploration of different paths based solely on predicates occurring in branch tests). For example, consider the two methods in Figure 2 that have slightly different than swap, but its post-condition can be violated under a certain input configuration (i.e., this=n). KUnit will generate a test that exposes this post-condition violation, but jCUTe would not because the method contains no tests that expose the points-to scenario that leads to the violation. It is still unclear to us how often such situations arise in real code, and we are currently devising experiments to further clarify the benefits of the extra degree of heap configuration coverage provided by Kiasan.

**Comparison to JPF [17]:** Direct comparison to JPF is non-trivial for several reasons. First, to run JPF’s symbolic execution, one must hand-instrument programs to incorporate the symbolic execution functionality which makes large case studies like the ones that we have carried out infeasible. (Recent work by Anand et al. [1] provides an automated transformation for JPF, but that tool component has not been released yet.) Moreover, JPF does not actually generate test cases at this point (it only emits information about symbolic states). To enable a direct comparison between the underlying algorithmic strategies in both tools, we have implemented the “lazy initialization” approach of JPF originally introduced in [17]. Having corresponded closely.
with NASA Ames personnel, we are confident that our implementation of lazy initialization reflects that strategy implemented in JPF. Given the underlying lazy initialization algorithm, we then had to decide which bounding strategy to use: should we choose the JPF depth-bounded strategy or KUnit’s k-bounded strategy? Because one of our primary objectives is to demonstrate the benefits of our lazier# initialization approach over lazy initialization, we wanted to focus on exactly that factor in the experiments. Thus, we implemented a k-bounding strategy for lazy initialization (removing the additional k-bounding/depth-bounding factor) with data for number of test cases and total time given in the Lazy columns of Table 1. In general, lazier# produces significantly smaller state-spaces (and thus significantly smaller test suites as reflected in the Test Cases column). In most complex examples, lazier# gives several orders of magnitude reduction in time to generate test suites, e.g., reducing time required to achieve the highest levels of coverage from over 24 hours down to 2.2 minutes for TreeMap.put. The reductions grow exponentially as k increases.

Comparison to Commercial Tools: The commercial test case generation tool CodePro performed abysmally on all of our examples (even ones without heap data) because it includes no underlying technique for reasoning about linear arithmetic or heap data. Recently, we were able to obtain an evaluation license for AgitarOne – an award winning unit test generation tool based on underlying Daikon [11] technology. Although we have only been able to consider a few examples thus far, AgitarOne achieves around 20% branch coverage for the java.util.TreeMap example. Essentially, AgitarOne does not use sophisticated reasoning about heap structures due to its Daikon foundations (Daikon infers invariants for scalar types). By manually creating multiple initial test cases (called “helper objects”) in the Agitar terminology, Agitar engineers reported to us that they were able to achieve higher levels of coverage, but we were not able to obtain detailed metrics from their efforts.

6. RELATED WORK

Throughout the paper we have contrasted our approach to others, thus we limit ourselves to a concise discussion here. The closest work to ours is the symbolic execution engine of JPF [30] which originally inspired our work. JPF uses a map to record lazily initialized values of fields in symbolic objects in order to remember which heap objects should not be de-materialized. The authors of [30] adopted this approach because they use bytecode instrumentation to facilitate symbolic execution. In contrast, we leverage information readily available from the backtracking algorithm in an explicit-state model checking because Kiasan directly interprets bytecode for symbolic execution. We have already discussed the benefits of our lazier# initialization approach compared to the lazy initialization of JPF; [30] demonstrates their approach using one example due to tediousness of the instrumentation process which they had to do manually. Recent work by [1] provides an automated translation that addresses this problem. JPF does not consider open systems with incomplete computation structure (in fact, most symbolic execution approaches do not do so).

There is an interesting line of work starting with [15] that uses simultaneous symbolic and concrete executions focusing on non-heap-intensive C programs. A recent extension [14] provides a compositional approach in which method behaviors are summarized as pre/post-conditions. Building on [15], CUTE [27] refers to the combination of symbolic and concrete execution as “concolic execution”, and uses a branch condition-driven approach with a heap abstraction that keeps track of reference equality and inequalities. KUnit only uses concrete execution to test feasibility of generated test cases (due to incompleteness of constraint solvers). We believe our approaches are complementary; that is, Kiasan’s systematic exploration of all heap configurations under a bound is suitable for heap intensive programs, and the concolic approach can be used to handle complex arithmetic constraints and native library calls (although this causes unsoundness). Further study is needed to compare the use of pre/post-conditions in our work with that of [14].

Parametric Unit Testing (PUT) [29], CUTE [27], and KeV [10] use a logical representation of heap structure, and they depend on theorem provers to decide assertions. While this does not introduce explicit non-deterministic choices as is the case with Kiasan, such non-deterministic choices are done by the theorem prover. Thus, it is not clear which approach is better without further investigation. In contrast to explicit representation [25], it has not been demonstrated that one can reason about strong heap-oriented properties when using logical heap representation. In addition, Kiasan’s stateless search is easily parallelizable, which can benefit from the recent trend to multi-core processor architectures. One can fork Kiasan when a non-deterministic choice happens. In addition, KUnit’s concretization algorithm is independent of the main symbolic execution, thus, it can be forked as well. Furthermore, we focus more on establishing strong heap coverage, while they focus on branch coverage; as mentioned previously, k = 2 is often enough to reach 100% feasible multiple condition coverage (MCC).

Symstra [32] uses sequencing of public method calls to test a class. It uses symbolic primitive values and concrete heap structures, and it employs state subsumption to generate non-isomorphic end states. Our approach uses complete symbolic representation of scalar and heap structures and leverage contracts to generate input states of programs. TestEra [21] and Korat [4] can generate non-isomorphic complex heap structures a priory instead of using lazy initialization algorithm (less efficient). Moreover, they focus on generating heap structure configurations without scalar data. In contrast to all of the above (except [10]), Kiasan/KUnit fully supports the DBC methodology. ESC/Java-like frameworks [12, 5, 2] are the most popular contract-based checking tools for object-oriented programs and are usually based on weakest precondition calculi. One problem with such approach is that it is difficult to generate counter-examples for contract violations (i.e., test cases illustrating violations). Recent work on [6] tried to address this issue by processing ESC/Java failed proof attempts, and then running programs with random inputs to check whether the warnings are false alarm (if not, it found a test case). This seems to work well for scalar data, however it does not work with heap intensive programs and contracts (since ESC/Java itself targets lightweight properties). We believe that Kiasan/KUnit provides an alternative solution for contract-based static checking that is able to reason about strong heap-oriented properties, while the work presented in this paper takes us further in term of analysis feedback compared to [6].

7. CONCLUSION

We have presented, KUnit, a framework built on the Kiasan symbolic execution engine for unit test case generation,
input/output object graph visualization, and mock object generation. Our efforts have highlighted the concept and utility of guaranteeing complete coverage of heap configurations up to bounds on the length of reference chains in data structures. This strengthens previous testing approaches by guaranteeing full soundness (no errors will be missed) for any data whose size is within the supplied Kiasan 4-bound. Using a broad range of heap-intensive examples, we have shown that our approach can achieve 100% feasible branch coverage (improving on related techniques) by using only small heap configurations and with minimal number of generated tests (again, improving on previous approaches by reducing number of tests generated). From another viewpoint, these capabilities add significant values to Kiasan as a contract checking tool because they dramatically improve upon the quality of feedback provided by error trace and counterexample information in other tools for ESC/Java [6].

Acknowledgments: The authors wish to thank Koushik Sen for his assistance in running his JCUTE tool. Agitar Corporation for providing an evaluation license for AgitarOne, Corina Păsăreanu and Willem Visser for discussions regarding the JPF’s symbolic execution, and Andrew King and Ashish Panday for their help in running our experiments.

8. REFERENCES
Bibliography


Appendix A

Experiment Data
<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>k</th>
<th>Test Cases</th>
<th>Branch Coverage</th>
<th>Bytecode Coverage</th>
<th>Total (+dot)</th>
<th>CVC Lite</th>
<th>=\gamma&amp; \approx \theta</th>
<th>JUnit Gen.</th>
<th>GraphViz dot (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>find</td>
<td>1</td>
<td>2</td>
<td>2/2=100%</td>
<td>8/8=100%</td>
<td>0.6s</td>
<td>0.2s</td>
<td>0.0s</td>
<td>0.1s</td>
<td>0.4s</td>
</tr>
<tr>
<td>AirTree</td>
<td>findMax</td>
<td>1</td>
<td>2</td>
<td>7/8=87%</td>
<td>38/42=90%</td>
<td>1.6s</td>
<td>0.8s</td>
<td>0.2s</td>
<td>0.1s</td>
<td>0.6s</td>
</tr>
<tr>
<td>DoubleLinkedList</td>
<td>insert</td>
<td>1</td>
<td>5</td>
<td>12/18=66%</td>
<td>119/160=74%</td>
<td>3.0s</td>
<td>1.3s</td>
<td>0.2s</td>
<td>0.1s</td>
<td>0.95</td>
</tr>
<tr>
<td>GC</td>
<td>Mark</td>
<td>1</td>
<td>306</td>
<td>12/12=100%</td>
<td>64/64=100%</td>
<td>46.2s</td>
<td>44.8s</td>
<td>2.2s</td>
<td>10.8s</td>
<td>59.6s</td>
</tr>
<tr>
<td>StackAr</td>
<td>classify</td>
<td>1</td>
<td>15</td>
<td>16/16=100%</td>
<td>54/54=100%</td>
<td>1.7s</td>
<td>1.7s</td>
<td>0.2s</td>
<td>0.5s</td>
<td>2.4s</td>
</tr>
<tr>
<td>A.P.</td>
<td>partition</td>
<td>1</td>
<td>1</td>
<td>1/1=100%</td>
<td>18/8=26%</td>
<td>0.3s</td>
<td>0.3s</td>
<td>0.1s</td>
<td>0.1s</td>
<td>0.3s</td>
</tr>
<tr>
<td>BinaryHeap</td>
<td>deleteMin</td>
<td>1</td>
<td>2</td>
<td>6/14=42%</td>
<td>71/120=59%</td>
<td>0.5s</td>
<td>0.7s</td>
<td>0.1s</td>
<td>0.3s</td>
<td>0.4s</td>
</tr>
<tr>
<td>Sort</td>
<td>insertionSort</td>
<td>1</td>
<td>2</td>
<td>1/6=16%</td>
<td>9/44=20%</td>
<td>0.4s</td>
<td>0.5s</td>
<td>0.0s</td>
<td>0.1s</td>
<td>0.2s</td>
</tr>
</tbody>
</table>

Table A.1: KUnit Experiment Data (excerpts); (with Lazy columns giving data for the JPF algorithm) (1)
<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Test Cases</th>
<th>Branch Coverage</th>
<th>Bytecode Coverage</th>
<th>Total (dot)</th>
<th>CVC Lite</th>
<th>∇ = \frac{J\text{Unit Gen.}}{\text{GraphViz dot (+)}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>DisjSets</td>
<td>find</td>
<td>Lazy</td>
<td>8/8=100%</td>
<td>65/65=100%</td>
<td>3.2</td>
<td>1.2s</td>
<td>0.2s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lazy#</td>
<td>8/8=100%</td>
<td>65/65=100%</td>
<td>1.8</td>
<td>1.7m</td>
<td>1.1m</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8/8=100%</td>
<td>65/65=100%</td>
<td>1.2</td>
<td>1.1m</td>
<td>1.2s</td>
</tr>
<tr>
<td>BinarySearchTree</td>
<td>findMax</td>
<td>Lazy</td>
<td>5/6=83%</td>
<td>5/6=83%</td>
<td>1.1</td>
<td>0.7s</td>
<td>0.8s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lazy#</td>
<td>5/6=83%</td>
<td>5/6=83%</td>
<td>3.5</td>
<td>2.1s</td>
<td>0.2s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5/6=83%</td>
<td>5/6=83%</td>
<td>1.3</td>
<td>0.7s</td>
<td>0.8s</td>
</tr>
<tr>
<td>Stacks</td>
<td>find</td>
<td>Lazy</td>
<td>8/8=100%</td>
<td>65/65=100%</td>
<td>2.9</td>
<td>6.5s</td>
<td>1.0s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lazy#</td>
<td>8/8=100%</td>
<td>65/65=100%</td>
<td>1.8</td>
<td>1.7m</td>
<td>1.1m</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8/8=100%</td>
<td>65/65=100%</td>
<td>1.2</td>
<td>1.1m</td>
<td>1.2s</td>
</tr>
<tr>
<td>LinkedList</td>
<td>insert</td>
<td>Lazy</td>
<td>6/6=100%</td>
<td>63/63=100%</td>
<td>3.9</td>
<td>1.7s</td>
<td>0.3s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lazy#</td>
<td>6/6=100%</td>
<td>63/63=100%</td>
<td>3.1</td>
<td>2.2s</td>
<td>0.3s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6/6=100%</td>
<td>63/63=100%</td>
<td>4.8</td>
<td>2.2s</td>
<td>0.3s</td>
</tr>
<tr>
<td>TreeMap</td>
<td>remove</td>
<td>Lazy</td>
<td>8/12=66%</td>
<td>68/95=71%</td>
<td>4.4</td>
<td>1.0s</td>
<td>0.6s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lazy#</td>
<td>8/12=66%</td>
<td>68/95=71%</td>
<td>3.1</td>
<td>2.2s</td>
<td>0.3s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8/12=66%</td>
<td>68/95=71%</td>
<td>4.8</td>
<td>2.2s</td>
<td>0.3s</td>
</tr>
</tbody>
</table>

Table A.2: KUnit Experiment Data (excerpts); (with Lazy columns giving data for the JPF algorithm) (2)
Appendix B

java.util.TreeMap Coverage Report

The class java.util.TreeMap from Java 1.5 library is a red-black tree implementation. We tested two most important methods from the class: put (inserting an element into the tree) and remove (deleting an element from the tree) using KUnit. For \( k = 3 \), KUnit reports a branch coverage of 40/52 for the put method and 70/86 for the remove method.

So for testing of the remove method, there are 16 uncovered branches. We organize the uncovered branches according the feasibility as follows.

- Inherent infeasible paths, 6: one path in each of setColor, parentOf, rightOf, and leftOf methods and two paths in deleteEntry method. The setColor method shown in Figure B.1 has a test of null-ness of the argument at line 2 which is always true. The setColor method is private and only called from fixAfterInsertion, shown in Figure B.3, and fixAfterDeletion, shown in Figure B.4, methods. The calling contexts always guarantee that the argument is non-null. The same is for the parentOf, rightOf, and leftOf methods.

For method deleteEntry shown in Figure B.5, the conditional at line 39 is always true because line 33 had already done the test; similarly, the conditional at line 42 is always true because of a previous conditional at line 40.

- Infeasible paths due to specification, 1: the compare method tests the null-ness of field comparator at line 5 of Figure B.2. And we put “comparator!=null” in the specification for both methods.

- Infeasible paths due to context, 6. All 6 paths are in successor method shown in Figure B.6. In our testing of remove, the successor method is only called from deleteEntry at line 8 with an argument that has both left and right children. So as shown in Figure B.6, the conditional at lines 5 is always false and the conditional at line 7 is always true. Furthermore, lines 13-29 are unreachable code which contain 4 branches. So there are total 6 infeasible branches due to context.

- Feasible paths, 3: one in each of rotateRight, rotateLeft, and deleteEntry methods.

For testing of the put method, there are 10 uncovered branches. We organize the uncovered branches according the feasibility as follows.
Inherent infeasible paths, 7: one path in each of setColor, parentOf, rightOf, and leftOf methods and three paths in fixAfterInsertion method. The reason for setColor, parentOf, rightOf, and leftOf methods is the same as explained above.

The three infeasible paths in fixAfterInsertion shown in Figure B.3 are more involved and related to the loop invariant of the main loop in the method body.

fixAfterInsertion helper function (called from put after a node is inserted in the red-black tree). The inserted node x is colored as RED as shown in line 2. The goal of this method is to adjust the inserted tree to satisfy the red-black tree invariant. The only red-black tree invariant that the new tree could violate is “all children of a RED node have to be BLACK” because the parent node of the inserted node could be RED too. KUnit reports (and we confirmed) that this method contains three infeasible paths related to the loop invariant. The main loop from line 4 (we only show half here and the other half is symmetric) deals with the case that node x is RED and its parent also RED. Part of the loop invariant is that x!=null and the color of x is RED. x is non-NULL and RED before entering the loop; inside the loop, x is either unchanged or moved up the tree to the parent of parent of x which can not be NULL because the parent of x is RED and the root of the tree is BLACK and further x is colored with RED. Thus, x!=null at line 4 and the conditional in line 19 are always true. Symmetrically, the conditional at line 36 is always true.

Infeasible paths due to specification, 1: the compare method tests the null-ness of field comparator and we put “comparator!=null” in the specification for both methods.

Feasible paths, 2: one in each of rotateRight and rotateLeft methods.

The uncovered branches in rotateRight and rotateLeft methods are different in testings of remove and put methods. In summary, for all the feasible branches, there is only one in deleteEntry not covered by KUnit.

```java
private static <K, V> void setColor(Entry<K, V> p, boolean c) {
    if (p != null)
        p.color = c;
}
```

**Figure B.1: Method setColor**

```java
/**
 * Compares two keys using the correct comparison method for this TreeMap.
 */
private int compare(K k1, K k2) {
    return (comparator == null ? ((Comparable<? super K>) k1).compareTo(k2)
        : comparator.compare((K) k1, (K) k2));
}
```

**Figure B.2: Method compare**
private void fixAfterInsertion (Entry<K, V> x) {
  x.color = RED;
  
  while (x != null && x != root && x.parent.color == RED) {
    Entry<K, V> y = rightOf(parentOf(parentOf(x)));
    if (colorOf(y) == RED) {
      setColor(parentOf(x), BLACK);
      setColor(y, BLACK);
      setColor(parentOf(parentOf(x)), RED);
      x = parentOf(parentOf(x));
    } else {
      if (x == rightOf(parentOf(x))) {
        x = parentOf(x);
        rotateLeft(x);
      } else {
        setColor(parentOf(x), BLACK); // bug seeded
        setColor(parentOf(parentOf(x)), RED);
        if (parentOf(parentOf(x)) != null)
          rotateRight(parentOf(parentOf(x)));
      }
    }
    Entry<K, V> y = leftOf(parentOf(parentOf(x)));
    if (colorOf(y) == RED) {
      setColor(parentOf(x), BLACK);
      setColor(y, BLACK);
      setColor(parentOf(parentOf(x)), RED);
    } else {
      if (x == leftOf(parentOf(x))) {
        x = parentOf(x);
        rotateRight(x);
      } else {
        setColor(parentOf(x), BLACK);
        setColor(parentOf(parentOf(x)), RED);
        if (parentOf(parentOf(x)) != null)
          rotateLeft(parentOf(parentOf(x)));}
    }
  }
  root.color = BLACK;
}

Figure B.3: Method fixAfterInsertion
```java
private void fixAfterDeletion(Entry<K, V> x) {
    while (x != root && colorOf(x) == BLACK) {
        Entry<K, V> sib = rightOf(parentOf(x));
        if (colorOf(sib) == RED) {
            setColor(sib, BLACK);
            setColor(parentOf(x), RED);
            rotateLeft(parentOf(x));
            sib = rightOf(parentOf(x));
        }
        if (colorOf(leftOf(sib)) == BLACK
            && colorOf(rightOf(sib)) == BLACK) {
            setColor(sib, RED);
            x = parentOf(x);
        } else { // symmetric
            Entry<K, V> sib = leftOf(parentOf(x));
            if (colorOf(sib) == RED) {
                setColor(sib, BLACK);
                setColor(parentOf(x), RED);
                rotateRight(parentOf(x));
                sib = leftOf(parentOf(x));
            }
            if (colorOf(rightOf(sib)) == BLACK
                && colorOf(leftOf(sib)) == BLACK) {
                setColor(sib, RED);
                x = parentOf(x);
            } else {
                if (colorOf(leftOf(sib)) == BLACK) {
                    setColor(rightOf(sib), BLACK);
                    setColor(sib, RED);
                    rotateLeft(sib);
                    sib = leftOf(parentOf(x));
                }
                setColor(sib, colorOf(parentOf(x)));
                setColor(parentOf(x), BLACK);
                setColor(leftOf(sib), BLACK);
                rotateRight(parentOf(x));
                x = root;
            }
        }
    }
    setColor(x, BLACK);
}
```

**Figure B.4: Method fixAfterDeletion**
private void deleteEntry(
    Entry<K, V> p) {
    decrementSize();

    // If strictly internal, copy successor's element to p and then make p
    // point to successor.
    if (p.left != null && p.right != null) {
        Entry<K, V> s = successor(p);
        p.key = s.key;
        p.value = s.value;
        p = s;
    } // p has 2 children

    // Start fixup at replacement node, if it exists.
    Entry<K, V> replacement = (p.left != null ? p.left : p.right);

    if (replacement != null) {
        // Link replacement to parent
        replacement.parent = p.parent;
        if (p.parent == null)
            root = replacement;
        else if (p == p.parent.left)
            p.parent.left = replacement;
        else
            p.parent.right = replacement;

        // Null out links so they are OK to use by fixAfterDeletion.
        p.left = p.right = p.parent = null;

        // Fix replacement
        if (p.color == BLACK)
            fixAfterDeletion(replacement);
        else if (p.parent == null) { // return if we are the only node.
            root = null;
        } else { // No children. Use self as phantom replacement and unlink.
            if (p.color == BLACK)
                fixAfterDeletion(p);
            else if (p.parent != null) {
                if (p == p.parent.left)
                    p.parent.left = null;
                else if (p == p.parent.right)
                    p.parent.right = null;
                p.parent = null;
            }
        }
    }

    Figure B.5: Method DeleteEntry
/**
 * Returns the successor of the specified Entry, or null if no such.
 */

private Entry<K, V> successor(Entry<K, V> t) {
    if (t == null)
        return null;
    else if (t.right != null) {
        Entry<K, V> p = t.right;
        while (p.left != null)
            p = p.left;
        return p;
    } else {
        Entry<K, V> p = t.parent;
        Entry<K, V> ch = t;
        while (p != null && ch == p.right) {
            ch = p;
            p = p.parent;
        }
        return p;
    }
}  

Figure B.6: Method Successor
# Appendix C

## jCUTE Experiment

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>( d )</th>
<th>Time</th>
<th>Branch Coverage</th>
<th>Paths</th>
<th>Test</th>
<th>Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>AvlTree</td>
<td>find</td>
<td>10</td>
<td>5.9s</td>
<td>10/10=100%</td>
<td>738</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>findMax</td>
<td>10</td>
<td>15s</td>
<td>8/8=100%</td>
<td>30</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>findMin</td>
<td>10</td>
<td>14s</td>
<td>4/4=100%</td>
<td>30</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>27</td>
<td>1h 1m</td>
<td>12/18=67%</td>
<td>6848</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>BinaryHeap</td>
<td>deleteMin</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>BinarySearchTree</td>
<td>insert</td>
<td>8</td>
<td>7s</td>
<td>6/6=100%</td>
<td>18</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>remove</td>
<td>17</td>
<td>2.5m</td>
<td>15/16=94%</td>
<td>321</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>DisjSets</td>
<td>find</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>union</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DisjSetsFast</td>
<td>find</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>union</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DoubleLinkedList</td>
<td>addBefore</td>
<td>600</td>
<td>1h 4m</td>
<td>0/1=0%</td>
<td>2064</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>indexOf</td>
<td>1000</td>
<td>1h 24m</td>
<td>0/10=0%</td>
<td>2593</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>remove</td>
<td>1000</td>
<td>1h 24m</td>
<td>0/10=0%</td>
<td>2593</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clear</td>
<td>1000</td>
<td>58m</td>
<td>0/1=0%</td>
<td>1199</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lastIndexOf</td>
<td>1000</td>
<td>1h 34m</td>
<td>0/10=0%</td>
<td>2596</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>toArray</td>
<td>1000</td>
<td>57m</td>
<td>0/2=0%</td>
<td>1199</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>java.util.TreeMap</td>
<td>lastKey</td>
<td>25</td>
<td>1h 22m</td>
<td>5/6=83%</td>
<td>9077</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>remove</td>
<td>25</td>
<td>1h 23m</td>
<td>16/86=19%</td>
<td>9079</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>put</td>
<td>25</td>
<td>1h 22m</td>
<td>15/52=29%</td>
<td>9079</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>java.util.Vector</td>
<td>add</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ensureCapacity</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>indexOf</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>insertElementAt</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lastIndexOf</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>removeElementAt</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Stack Array</td>
<td>push</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pop</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Stack List</td>
<td>push</td>
<td>1000</td>
<td>1h 26m</td>
<td>0/1=0%</td>
<td>3902</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pop</td>
<td>3</td>
<td>2s</td>
<td>2/2=100%</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Sort</td>
<td>insertionSort</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>shellsort</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>GC</td>
<td>Mark</td>
<td>19</td>
<td>51s</td>
<td>12/12=100%</td>
<td>107</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>IntLinkedList</td>
<td>merge</td>
<td>29</td>
<td>32s</td>
<td>10/10=100%</td>
<td>17</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>ABS</td>
<td>abs</td>
<td>3</td>
<td>3s</td>
<td>2/2=100%</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>triangleClassification</td>
<td>classify</td>
<td>5</td>
<td>5s</td>
<td>3/16=19%</td>
<td>8</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>ArrayPartition</td>
<td>partition</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

**Table C.1: jCUTE Data**
C.1 Experiment Methodology

Recall that a theme of the Kiasan approach is controlling the scope and cost of the analysis via the k-bound on the length of reference chains. In contrast, jCUTE aims to achieve as high degree of branch coverage as possible with the costs and scope of the analysis controlled by a depth bound. One interesting point of comparison between the Kiasan and jCUTE is to consider the amount of branch coverage (and the performance in achieving that coverage) that the respective tools can achieve on different examples. In the Kiasan examples, we highlighted the bound value required for achieving 100% branch coverage, and in cases where 100% coverage could not be achieved, we reported the coverage achieved.

We conduct jCUTE experiments in an analogous way driven by the desire to maximize coverage. In particular, we seek to find the minimal depth bound required to achieve 100% branch coverage. In cases where 100% is not achieved, we attempt to determine that amount of coverage that can be achieved in roughly 1 hour. In jCUTE it is possible to constraint the number of “iterations” of the algorithm (equivalent to the number of paths explored). For simplicity, we do not constrain the number of iterations, that is, we set the depth and let jCUTE run to finish. The experiment data is recorded if any of following condition is true:

1. reached desired branch coverage (100% coverage of feasible branches);
2. depth reached 1000;
3. run time more than 1 hour;
4. jCUTE error occurred (as the TriangleClassification example).

Table C.1 displays the results of the experiments. jCUTE currently does not handle arrays of variable size (situations where the size of array is not known statically). This means that jCUTE cannot process a number of the examples used in our experiments. These situations are indicated in Table C.1 with the symbol X.

C.2 Summary of jCUTE Performance

1. AVL Tree: jCUTE performance is comparable to Kiasan on methods find, findMax, and findMin. For each of these cases, the numbers for Kiasan are actually better, but still within the same order of magnitude. For example, for find, Kiasan achieves 100% branch coverage on $k = 1$ in 1.5s (compared to 5.9s for jCUTE) and generates 4 tests instead of 13. For findMax, Kiasan achieves 100% branch coverage on $k = 2$ in 3.6s (compared to 15s for jCUTE) and generates 5 tests instead of 11.

However, the performance diverges significantly for the insert method. jCUTE obtains only 12/18 branch coverage (compared to 18/18 for Kiasan), and ran for over one hour (compared to 8.8s for $k = 2$ for Kiasan – the point at which 18/18 coverage was reached).
2. **Binary Heap**, jCUTE cannot handle this example with dynamically sized arrays so no data is reported.

3. **Binary Search Tree**, jCUTE was executed for methods `insert` and `remove`. jCUTE performance is comparable to Kiasan on all methods. Kiasan is almost better on all the data. For example, for `insert`, Kiasan takes 1.7s (compared to 7s for jCUTE) to reach 100% branch coverage with $k = 1$ and generates 6 (compared to 9 for jCUTE) test cases. For `remove`, Kiasan achieves 15/16 branch coverage in 1.5m (compared to 2.5m for jCUTE) with $k = 3$ and generated 236 (compared to 11 for jCUTE) test cases. However, jCUTE explores 321 paths to achieve this coverage.

4. **Disjoint Set (Original/Fast)**, jCUTE cannot handle these two examples with dynamically sized arrays so no data is reported.

5. **Double Linked List** is taken from java.util.LinkedList. jCUTE executed for methods `addBefore`, `indexOf`, `remove`, `clear`, and `lastIndexOf`. In all the example, Kiasan does much better than jCUTE: jCUTE spent more than or about one hour for each method but the coverage is 0%. We are currently corresponding with jCUTE creators to determine why jCUTE is not able to make any progress on this example.

6. **TreeMap**, jCUTE executed for methods `lastKey`, `remove`, and `put`. Kiasan performs significantly better than jCUTE. For example, for `remove`, Kiasan achieve 69/84 branch coverage in 2m 12s with $k = 3$ while jCUTE takes more than 1 hour to reach 16/84 coverage. The other two method comparisons are similar.

7. **Vector**, jCUTE cannot handle this example with dynamically sized arrays so no data is reported.

8. **Stack Array Implementation**, jCUTE cannot handle this example with dynamically sized arrays so no data is reported.

9. **Stack List Implementation**, jCUTE executed for methods `push` and `pop`. For `pop`, jCUTE’s performance is comparable with Kiasan. Kiasan achieves 2/2 branch coverage with 0.3s (compared to 2s for jCUTE) with $k = 1$ and generates 2 (the same as jCUTE does) test cases.

   However, for `push`, jCUTE performs much worse: after more than 1 hour without covering any of the tested code while Kiasan uses 0.5s to achieve 100% branch coverage with $k = 1$.

10. **Sort**, jCUTE cannot handle this example with dynamically sized arrays so no data is reported.

11. **GC**, jCUTE executed for method `Mark`. jCUTE performs better than Kiasan but still within order of magnitude. jCUTE achieves 100% branch coverage with 51s (compared to 1m 44.8s for Kiasan) and generates 10 (compared to 306 for Kiasan) test cases.
12. **IntLinkedList.** jCUTE executed for method `merge`. Kiasan and jCUTE are comparable in `merge` method. Kiasan takes 7.4s to achieve 100% branch coverage (compared to 32s for jCUTE) with \( k = 3 \) and generates 30 (compared to 15 for jCUTE) test cases.

13. **ABS.** jCUTE executed for method `abs`. jCUTE and Kiasan performance are similar. Kiasan is actually better but within same magnitude. Kiasan achieve 100% branch coverage in 0.1s (compared to 3s for jCUTE) and generates 2 (compared to 3 for jCUTE) test cases.

14. **TriangleClassification.** jCUTE executed for method `classify`. jCUTE can not handle this example and throws 4 internal errors. All the errors are the same and the error message is listed as follows:

   Error in Thread[main,5,main] null
   java.lang.ArrayIndexOutOfBoundsException: 1
   at cute.concolic.a.g.a(ArithmeticExpression.java:91)
   at cute.concolic.a.g.a(ArithmeticExpression.java:128)
   at cute.concolic.b.b.a(ComputationStack.java:273)
   at cute.concolic.b.c.a(ComputationStacks.java:94)
   at cute.concolic.Call.branchPos(Call.java:299)
   at TriangleClassification.classify(TriangleClassification.java:42)
   at TriangleClassification.main(TriangleClassification.java:59)
   at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
   at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
   at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
   at java.lang.reflect.Method.invoke(Method.java:585)
   at cute.RunOnce.main(RunOnce.java:242)

15. **Array Partition.** jCUTE cannot handle this example with dynamically sized arrays so no data is reported.
Appendix D

Formalization of Kiasan Symbolic Execution

D.1 Substitution Functions

First we will define some substitution functions. Assume that $D, D'$ are some domains and $\text{Seq}(D)$ is the set of all sequences of elements in $D$:

- the substitution function: $\text{sub} : D \times (D \to D) \to D$ as
  \[
  \text{sub}(d, g) = \begin{cases} 
  g(d) & \text{if } d \in \text{dom } g; \\
  d & \text{otherwise}.
  \end{cases}
  \]

- the function substitution function $\text{sub-fun} : (D' \to D) \times (D \to D) \to (D' \to D)$ as
  $\text{sub-fun}(f, g) = f'$ where $\text{dom } f = \text{dom } f'$ and $\forall d \in \text{dom } f. f'(d) = \text{sub}(f(d), g)$.

- the one-step function substitution function $\text{sub-fun}_1 : (D' \to D) \times D \times D \to (D' \to D)$ as
  $\text{sub-fun}_1(f, d, d') = \text{sub-fun}(f, \{(d, d')\})$.

- the sequence substitution function: $\text{sub-seq} : \text{Seq}(D) \times (D \to D) \to \text{Seq}(D)$ as
  $\text{sub-seq}(\text{nil}, g) = \text{nil}$ and $\text{sub-seq}(d::q, g) = \text{sub}(d, g)::\text{sub-seq}(q, g)$.

- the one-step sequence substitution function: $\text{sub-seq}_1 : \text{Seq}(D) \times D \times D \to \text{Seq}(D)$ as
  $\text{sub-seq}_1(q, d, d') = \text{sub-seq}(q, \{(d, d')\})$.

- the functional substitution function $\text{sub-fun2} : (D'' \to D' \to D) \times (D \to D) \to (D'' \to D' \to D)$ as
  $\text{sub-fun2}(f, g) = f'$ where $\text{dom } f = \text{dom } f'$ and $\forall d'' \in \text{dom } f. f'(d'') = \text{sub-fun}(f(d''), g)$.

- the one-step functional substitution function $\text{sub-fun2}_1 : (D'' \to D' \to D) \times D \times D \to (D'' \to D' \to D)$ as
  $\text{sub-fun2}_1(f, d, d') = \text{sub-fun2}(f, \{(d, d')\})$.

Then we introduce some simple properties of the substitution functions:
Lemma 1. Suppose partial function $g : D \rightarrow D$ for some domain $D$ satisfies $\text{dom } g \cap \text{ran } g = \emptyset$. Then for any $(d, d') \in g$ and function $f : D' \rightarrow D$, sequence $q : \text{Seq}(D)$, and functional $f^{\rho} : D'' \rightarrow D'$, we have $\begin{align*}
\text{sub-fun}(f, g) &= \text{sub-fun}(\text{sub-fun}_1(f, d, d'), g), \\
\text{sub-seq}(q, g) &= \text{sub-seq(} \text{sub-seq}_1(q, d, d'), g), \\
\text{sub-fun2}(f^{\rho}, g) &= \text{sub-seq}(\text{sub-fun2}_1(f^{\rho}, d, d'), g).
\end{align*}$

Lemma 2. If $R$ be the range of $f : D' \rightarrow D$, the set of elements in a sequence $q : \text{Seq}(D)$ or the second range of $f^{\rho} : D'' \rightarrow D'$, then for any $g : D \rightarrow D$, $\begin{align*}
\text{sub-fun}(f, g) &= \text{sub-fun}(f, g \mid _{R \cap \text{dom } g}), \\
\text{sub-seq}(q, g) &= \text{sub-seq}(g \mid _{R \cap \text{dom } g}), \\
\text{sub-fun2}(f^{\rho}, g) &= \text{sub-fun2}(f^{\rho}, g \mid _{R \cap \text{dom } g}).
\end{align*}$

D.2 Operational Semantics

This section presents the formal operational semantics of Kiasan’s symbolic execution with lazier initialization and lazier initialization, as well as a concrete execution semantics for Java bytecode instructions.

D.2.1 Operational Semantics of Symbolic Execution with Lazy Initialization

We will discuss the core symbolic execution (with lazy initialization) operational semantics of JVM bytecode with additional two instructions, assume and assert. First, the semantics domains are introduced. Then some auxiliary functions that facilitate the definition of semantic rules are defined. Finally the semantic rules for bytecode instructions and assume/assert are presented.

Semantic Domains

The semantic/syntactical domains are listed as following:

- the set of primitive types, $\text{Types}_{\text{prim}}$, consisting of INT, CHAR, etc.,
- the set of array types, $\text{Types}_{\text{array}}$,
- the set of record types, $\text{Types}_{\text{record}}$,
- the set of symbolic types, $\text{SymTypes}$,
- the set of non-primitive types, $\text{Types}_{\text{non-prim}} = \text{Types}_{\text{record}} \cup \text{Types}_{\text{array}} \cup \text{SymTypes}$,
- the set of all types, $\text{Types} = \text{Types}_{\text{prim}} \cup \text{Types}_{\text{non-prim}}$,
- the set of program counters, PCs
- the set of boolean expressions, $\Phi$,
- the set of locations, $\text{Locs}$,

\footnote{$\cup$ denotes disjoint union}
• the set of natural numbers, \( \mathbb{N} \),
• the set of constants, \( \text{Consts} \), including \( \mathbb{N}, \text{True}, \text{False}, \text{null}, \text{etc.} \),
• the set of fields, \( \text{Fields} \), including \( \text{len}, \text{def}, \text{conc}, \text{etc.} \),
• the set of integer symbols, \( \text{Symbols}_{\text{INT}} \),
• the set of primitive symbols, \( \text{Symbols}_{\text{prim}} \), including \( \text{Symbols}_{\text{INT}} \),
• the set of values, \( \text{Values} = \text{Consts} \cup \text{Locs} \cup \text{Symbols}_{\text{prim}} \),
• the set of indexes, \( \text{Indexes} = \text{Fields} \cup \mathbb{N} \cup \text{Symbols}_{\text{INT}} \),
• the set of non-primitive symbols, \( \text{Symbols}_{\text{non–prim}} = \{ X_{\tau}^{m,n} \mid X_{\tau}^{m,n} : \text{Indexes} \rightarrow \text{Values} \} \),
• the set of symbols, \( \text{Symbols} = \text{Symbols}_{\text{prim}} \cup \text{Symbols}_{\text{non–prim}} \),
• the set of globals, \( \text{Globals} = \{ g \mid g : \text{Fields} \rightharpoonup \text{Values} \} \),
• the set of operand stacks, \( \text{Stacks} = \{ \omega \mid \omega : \text{Seq(Values)} \} \), all sequences of values,
• the set of locals, \( \text{Locals} = \{ l \mid l : \mathbb{N} \rightharpoonup \text{Values} \} \),
• the set of heaps, \( \text{Heaps} = \{ h \mid h : \text{Locs} \rightharpoonup \text{Symbols}_{\text{non–prim}} \} \),
• the set of bytecode instruction with additional \text{assert} and \text{assume} instructions, \( \text{Instrs} \),

We follow Java type system in the semantic domains: we use \( \text{Types}_{\text{prim}} \) to model the primitive types and \( \text{Types}_{\text{non–prim}} \) for the reference types which are divided into object types (\( \text{Types}_{\text{record}} \)), array types (\( \text{Types}_{\text{array}} \)), and symbolic types (\( \text{SymTypes} \)). \( \text{SymTypes} \) is used to model the variable real types of the non-primitive input parameters and global fields.\(^2\) PCs denotes the set of program counters or indexes of code arrays. A special program counter, \( \text{eor} \), is introduced to indicate that the end of code array is reached and execution stops. Similar to types, \( \text{Symbols} \) are divided into two types: primitive symbols, such as symbolic integers, symbolic floats, etc.; and reference symbols including symbolic objects and symbolic arrays. Concrete values are modeled by the \( \text{Consts} \) domain. For simplicity, we unify concrete objects and all symbolic values into the \( \text{Symbols} \) domain. Each member of \( \text{Symbols}_{\text{non–prim}} \) domain, \( X_{\tau}^{m,n} \), has three properties (we often omit properties when they are not important/applicable): \( \tau \) is the type of the symbol, \( m \) is the object field or array element expansion bound, and \( n \) is the number of array elements bound. (We will discuss the difference between \( m \) and \( n \) for arrays at the end of this section.) And each non-primitive symbol, \( X_{\tau}^{m,n} \), is modeled as a partial mapping from its fields to values. Each primitive symbol \( X_{\tau} \) or field \( f_{\tau} \) also has a property of its type \( \tau \). Since arrays are also modeled by \( \text{Symbols} \), the domain (\( \text{Indexes} \)) of the partial mapping of array \( X \) includes natural numbers and symbolic integers. Concrete objects created during the execution are represented as non-primitive symbols too, but their field are all initialized (see the \text{new-obj} auxiliary function). On the other hand, fields

\(^2\)In fact, all the non-primitive symbolic objects are created with symbolic types.
of symbolic objects may have not been initialized (initially created using the \textit{new-sym} function). Fields of the array include indexes and length, \texttt{len}, (which is always defined). Symbolic arrays and concrete arrays are created using the \textit{new-sarr} and \textit{new-arr} functions respectively. \texttt{Locs} represents the set of addresses in the heap.

\textbf{State} \hspace{1em} Since we only consider single threaded programs modularly (one method at a time), we represent symbolic state with only one stack frame element (the stack frame element of the method being analyzed). A state is represented as a tuple of global variables, program counter, locals, operand stack, and heap following the Java Virtual Machine specification \cite{4}; we add path condition \(\phi\) (as a conjunctive-set of formulas) as another state component. So the definition of the set of symbolic states is:

\[ \Sigma_s = \text{Globals} \times \text{PCs} \times \text{Locals} \times \text{Stacks} \times \text{Heaps} \times \Phi \]

and we let \(\sigma\) ranges over \(\Sigma_s\).

We will follow the convention that

- \(\tau\) ranges over types, \texttt{Types},
- \(pc\) ranges over program counters, \texttt{PCs},
- \(\phi\) ranges over boolean expressions, \texttt{Phi},
- \(i\) and \(j\) range over locations, \texttt{Locs},
- \(m\), \(n\), and \(k\) range over natural numbers, \(\mathbb{N}\),
- \(c\) and \(d\) range over constants, \texttt{Consts},
- \(f\) ranges over fields, \texttt{Fields},
- \(X\), \(Y\), and \(Z\) range over symbols, \texttt{Symbols},
- \(v\) ranges over values, \texttt{Values},
- \(\iota\) ranges over indexes, \texttt{Indexes},
- \(g\) ranges over globals, \texttt{Globals},
- \(\omega\) ranges over operand stacks, \texttt{Stacks},
- \(l\) ranges over locals, \texttt{Locals},

The meta-variables used to range over the semantic domains may be primed or subscripted.
Auxiliary Functions

We define some auxiliary functions to facilitate the definition of operational semantics:

- default value function, default : Types → Values as \( \lambda \tau . v \), where \( v \) is the default value of \( \tau \);
- fields of a type function, fields : Types → \( \mathcal{P}(\text{Fields}) \) as \( \lambda \tau . \{ f_r | f_r \text{ is a field in } \tau \} \);
- subtype function, \( \tau’ <: \tau : \text{Types} \times \text{Types} \rightarrow \text{Boolean} \) as \( \tau’ \) is a subtype of \( \tau \) (reflexive);
- defined integral indexes of a non-primitive symbol function, acc-idx : Symbols\(_{\text{non-prim}} \rightarrow \mathcal{P}(\mathbb{N} \cup \text{Symbols\(_{\text{int}}\))) \) as \( \lambda X. \{ i \in \mathbb{N} \cup \text{Symbols\(_{\text{int}}\)} | X(i) \downarrow \} \);
- locations that map to symbolic objects function, collect : Heaps → \( \mathcal{P}(\text{Locs}) \) as \( \lambda h. \{ i | h(i)(\text{conc}) \uparrow \} \);
- the set of all symbols in a state function, symbols : \( \Sigma_s \rightarrow \mathcal{P}(\text{Symbols}) \) as \( \lambda \sigma. \{ X | X \text{ appears in } \sigma \} \);
- new primitive symbol function, new-prim-sym : \( \text{Types}_{\text{prim}} \times \mathcal{P}(\text{Symbols}) \rightarrow \text{Symbols}_{\text{prim}} \) as \( \lambda (\tau, ss). X_\tau, X \not\in ss \);
- new symbolic type function, new-sym-type : \( \mathcal{P}(\text{Symbols}) \rightarrow \text{SymTypes} \) as \( \lambda ss. \tau \) s.t. \( \tau \in \text{SymTypes} \) and \( \tau \) does not appear in \( ss \);
- new array type function, array-type : Types → \( \text{Types}_{\text{array}} \) as \( \lambda \tau. \tau’ \), where \( \tau’ \) is the array type of element type \( \tau \);
- new symbolic record function, new-sym : \( \mathcal{P}(\text{Symbols}) \times \mathbb{N} \times \mathbb{N} \rightarrow \text{Symbols}_{\text{non-prim}} \) as \( \lambda (ss, m, n). X_\tau^{m,n} \), s.t. \( X \not\in ss \land \tau = \text{new-sym-type}(ss) \land \forall t \in \text{Indexes}.X(t) \uparrow \);
- new symbolic array function, new-sarr : \( \mathcal{P}(\text{Symbols}) \times \mathbb{N} \times \mathbb{N} \rightarrow \text{Symbols} \) as \( \lambda (ss, m, n). \text{new-sym}(ss \cup \{ X \}, m, n)[\text{LEN} \mapsto X] \) where \( X = \text{new-prim-sym}(\text{int}, ss) \);
- new concrete object function, new-obj : \( \mathcal{P}(\text{Symbols}) \times \text{Types}_{\text{record}} \rightarrow \text{Symbols} \) as \( \lambda (ss, \tau). X_\tau^{0,0} \), s.t. \( X \not\in ss \land \forall f_r \in \text{fields}(\tau).X(f_r) = \text{default}(\tau’) \);
- new concrete array function, new-arr : \( \mathcal{P}(\text{Symbols}) \times \text{Types} \times (\mathbb{N} \cup \text{Symbols}_{\text{int}}) \times \mathbb{N} \rightarrow \text{Symbols} \) as \( \lambda (ss, \tau, v, n). X_\tau^{0,n}, X \not\in ss \land \tau’ = \text{array-type}(\tau) \land \text{dom} X = \{ \text{DEF}, \text{LEN}, \text{CONC} \} \land X(\text{DEF}) = \text{default}(\tau) \land X(\text{LEN}) = v \).

Semantics Rules

Given an array of instructions, we define a function \( \text{code} : \text{PCs} \rightarrow \text{Instrs} \) which takes in a program counter and returns the corresponding instruction that is pointed to by the input program counter.

Operational semantic rules are in the format of

\[
\begin{align*}
\text{pre} \\
\sigma \Rightarrow_S \sigma_1[[\parallel \sigma_2]] \mid \text{Exception}, \sigma’|\text{Error}, \sigma''
\end{align*}
\]

19
that shows how a state is changed by one bytecode instruction to multiple normal states, or an exception raised, or an error occurred due to non-determinism. More specifically, given a state \( \sigma \), if \( \text{pre} \) is satisfied, after executing instruction pointed by the program counter component of \( \sigma \), the resulting state is \( \sigma_1 \) or nondeterministically \( \sigma_1 \) or \( \sigma_2 \); or an exception thrown with a state \( \sigma' \); or Error with a state \( \sigma'' \). Exceptions are handled the same way as JVM specification [4] does. If an error occurred, then the program stops. For simplicity, we assume that garbage collection is performed after each transition. Moreover, we stop exploring paths whose state’s path condition is unsatisfiable.

Each symbolic semantics rule name is the format of xxxx#-S where xxx is the instruction name and since there may be multiple rules for one instruction, we use number # (from 1 to n) to distinguish the rules for same instruction. Due to limit of space, we only present semantics for some representative JVM instructions and the instructions are divided into following categories:

- Arithmetic instructions: Instruction \texttt{iadd} adds two integers from the top of the stack and puts result back into the stack. \texttt{iadd} is represented by rule IADD-S. A fresh symbolic integer is introduced as the result and a constraint added to the path condition stating that the fresh symbolic integer equals to the sum of two operands.

- Object creation and manipulation instructions: \texttt{new \tau}, \texttt{getfield \ f}, \texttt{putfield \ f}, \texttt{instanceof \ \tau}, and \texttt{checkcast \ \tau} are presented. Accesses to symbolic objects (e.g., \texttt{getfield \ f}) operate according to the lazy initialization algorithm described previously. Similar to [3], we limit the choosing range to symbolic objects/arrays by introducing an additional field, \texttt{conc}, which is defined for concrete objects while undefined for symbolic objects. This eliminates false alarms in the case where freshly created objects (using the \texttt{new \ \tau} instruction during the execution) are reachable through object expansion; concretely, this only happens through assignments.

  - Instruction \texttt{new \ \tau} creates a fresh object of type \( \tau \) and put it into heap. By the definition of \texttt{new-obj}, all the fields including \texttt{conc} are initialized. This guarantees that the newly created object will not put in the range of lazy initialization.

  - Instruction \texttt{getfield \ f} reads the \( f \) field of an object which is indexed by the address on the top of the stack. Semantics rules GETFIELD(1..7)-S are for \texttt{getfield}. Rules GETFIELD1-S and GETFIELD7-S are the default behavior of the \texttt{getfield \ f}: GETFIELD1-S is for the case of the field of the object is defined; GETFIELD7-S is for the case of the object reference is \texttt{null}. Rules GETFIELD(2..6)-S demonstrate the lazy initialization algorithm when the field is undefined. GETFIELD2-S handles the subcase of primitive field type. A new symbol is created and the field is initialized with the fresh symbol. GETFIELD3-S lazily initializes a non-primitive field to \texttt{null}. GETFIELD4-S lazily initializes a non-primitive field by nondeterministically choosing from existing symbolic objects (with \texttt{conc} undefined) from heap with compatible types. Rules GETFIELD5-S and GETFIELD6-S show the field is initialized with a new symbolic object or array respectively if the object bound is not exhausted (greater than zero).
– Instruction putfield $\tau$ writes a value to a field of an object. The value and object address are in the top of the stack. There are two rules for putfield $\tau$: PUTFIELD1-S and PUTFIELD2-S. PUTFIELD1-S handles the normal case and PUTFIELD2-S deals with the case of the object is null.

– Instruction instanceof $\tau$ tests whether an object is a type of $\tau$. According the JVM specification [4], if the object is null, the test returns true. If the object is non-null, returns true if the type of the object is a subtype of $\tau$, false otherwise. Rule INSTANCEOF1-S represents the null case and INSTANCEOF2-S does the non-null case.

– Instruction checkcast $\tau$ is very similar to the instruction instanceof except that it does not return true or false instead it does nothing if the test passes otherwise throws a ClassCastException.

- Array manipulation instructions: anewarray $\tau$, iastore, and iaload are presented. As mentioned previously, symbolic arrays require a special treatment: fields of symbolic objects are fixed by their types but elements of symbolic arrays are not fixed because the length may be unknown; this includes arrays explicitly created with a symbolic length. To address this, we introduce another bound $n$ on symbol $X^{m,n}$ that limits the number of distinct array elements that can be lazily initialized; each symbolic array allows lazy initializations up to $n$ kinds of distinct elements. If an array element is accessed through a symbolic index (e.g., iaload):

  1. the index maybe out of bounds,
  2. the index is equal to one of the accessed indexes (from the acc-idx function), or
  3. $n$ is decremented if the above does not hold, the number of distinct indexes accessed so far is less than the length of array, and $n$ is greater than zero.

Elements of local arrays (created by anewarray) should have default values, but we cannot simply assign default values to all elements to a local array because the array length maybe unknown. Instead, we keep a default value for the array on its len field and lazily initialize an accessed index with it.

– Instruction anewarray $\tau$ creates a new array with length on the top of the stack. Because the way we bound arrays, there are two rules for this instruction: ANEWARRAY1-S for fixed (concrete) array length then the array bound is the same as the length; ANEWARRAY2-S for symbolic array length.

– Instruction iastore writes an integer value into an integer array. Rule IASTORE1-S is for the array index out of bound case and IASTORE4-S presents the case of array is null. Rule IASTORE2 is for the case of the index equals to one of accessed index. Rule IASTORE3 creates a new index in the array.

– Instruction iaload reads the value from an index of an array. Similar to getfield, lazy initialization is applied when an index is undefined (Rule IALOAD3-S). The rest of rules are similar to the rules for instruction iastore.
- Control transfer instructions: we list semantic rules for instructions \texttt{if\_icmplt} and \texttt{if\_acmpeq}.

  - Instruction \texttt{if\_icmplt} compares the top two integral values on the stack. Since the two compared values may be symbolic and thus cannot decide the ordering, rule IF\_ICMPLT-S has two end states to cover both the true and the false branches if the top of the stack cannot be determined to greater than the one below it (if one branch can be determined, then the other branch will have inconsistent path condition, which will then be ignored).

  - Instruction \texttt{if\_acmpeq} compares two object references on the top of the stack. Since Kiasan maintains a precise visible heap, the two references are either equal or not equal. Thus there are two rules for \texttt{if\_acmpeq}: IF\_ACMPEQ1-S for not equal case and IF\_ACMPEQ2-S for the equal case.

- Instructions assume and assert instructions: the semantics for assume and assert are standard: if the top of the stack is true, assume and assert does nothing; otherwise, assume terminates the execution silently by making path condition \texttt{FALSE}, while assert signals an error and terminates the execution.

We use the binding, $\sigma = (g, pc, l, w, h, \phi)$, for all the rules. And $k$ is used as both the object bound and the array bound.

\[
\text{IADD-S} \quad \begin{aligned} \text{code}(pc) &= \text{iadd} \quad \omega &= v_1 :: v_2 :: \omega' \\
\sigma \Rightarrow_S (g, \text{next}(pc), l, Y :: \omega', h, \phi \cup \{Y = v_1 + v_2\}) \quad \text{where } Y = \text{new-prim-sym}(\text{int}, \text{symbols}(\sigma)) \end{aligned}
\]

\[
\text{IF\_ICMPLT-S} \quad \begin{aligned} \text{code}(pc) &= \text{if\_icmplt} pc' \quad \omega &= v_1 :: v_2 :: \omega' \\
\sigma \Rightarrow_S (g, \text{next}(pc), l, \omega', h, \phi \cup \{v_2 \neq v_1\}) \quad \text{if } \{ v_2 < v_1 \} \end{aligned}
\]

\[
\text{NEW-S} \quad \begin{aligned} \sigma \Rightarrow_S (g, \text{next}(pc), l, i :: \omega, h[i \mapsto \text{new-obj}(\text{symbols}(\sigma), \tau)], \phi) \\
\text{code}(pc) &= \text{getfield} f_i \quad \omega = i :: \omega' \\
\end{aligned}
\]

\[
\text{GETFIELD1-S} \quad \begin{aligned} \sigma \Rightarrow_S (g, \text{next}(pc), l, h(i)(f_i) :: \omega', h, \phi) \\
\text{code}(pc) &= \text{getfield} f_i \quad \omega = i :: \omega' \\
\end{aligned}
\]

\[
\text{GETFIELD2-S} \quad \begin{aligned} \sigma \Rightarrow_S (g, \text{next}(pc), l, X :: \omega', h[i \mapsto h(i)[f \mapsto X]], \phi) \\
\text{where } X = \text{new-prim-sym}(\tau, \text{symbols}(\sigma)) \\
\text{code}(pc) &= \text{getfield} f_i \quad \omega = i :: \omega' \\
\end{aligned}
\]

\[
\text{GETFIELD3-S} \quad \begin{aligned} \sigma \Rightarrow_S (g, \text{next}(pc), l, \text{null} :: \omega', h[i \mapsto h(i)[f \mapsto \text{null}]], \phi) \\
\text{code}(pc) &= \text{getfield} f_i \\
\end{aligned}
\]

\[
\text{GETFIELD4-S} \quad \begin{aligned} \omega = i :: \omega' \\
\text{h(i)(f_i)\uparrow} \\
\tau \in \text{Types}_{\text{non-prim}} \\
j \in \text{collect}(h) \\
Z_{\tau} = h(j) \quad Z_{\tau'} = h(j) \\
\sigma \Rightarrow_S (g, \text{next}(pc), l, j :: \omega', h[i \mapsto h(i)[f \mapsto j]], \phi \cup \{\tau' < \tau\}) \\
\text{code}(pc) &= \text{getfield} f_i \quad \omega = i :: \omega' \\
\text{h(i)(f_i)\uparrow} \\
\tau \in \text{Types}_{\text{array}} \\
y^{m,n} = h(i) \\
m > 0 \\
j \notin \text{dom } h \\
\end{aligned}
\]

\[
\text{GETFIELD5-S} \quad \begin{aligned} \sigma \Rightarrow_S (g, \text{next}(pc), l, j :: \omega', h[i \mapsto h(i)[f \mapsto j]], Z_{\tau'}, \phi \cup \{\tau' < \tau\}, Z(\text{LEN}) \geq 0) \\
\text{where } Z_{\tau'} = \text{new-sarr}(\text{symbols}(\sigma), m - 1, k) \\
\end{aligned}
\]
\[ code(pc) = \text{getfield } f_r \quad \omega = i::\omega' \]

**GETFIELD6-S**
\[ h(i)(f_r) \uparrow \quad \tau \in \text{Types}_{\text{record}} \quad Y^{m,n} = h(i) \quad m > 0 \quad j \notin \text{dom } h \]
\[ \sigma \Rightarrow_S (g, \text{next}(pc), l, j::\omega', h[i \mapsto h(i)[f \mapsto j]][j \mapsto Z_{\tau'}], \phi \cup \{\tau' <: \tau\}) \]
where \( Z_{\tau'} = \text{new-sym}(\text{symbols}(\sigma), m-1, k) \)

**GETFIELD7-S**
\[ code(pc) = \text{getfield } f_r \quad \omega = \text{null}::\omega' \]
\[ \sigma \Rightarrow_S \text{NullPointerException}, (g, pc, l, \omega', h, \phi) \]

**PUTFIELD1-S**
\[ code(pc) = \text{putfield } f \quad \omega = v::i::\omega' \]
\[ \sigma \Rightarrow_S (g, \text{next}(pc), l, \omega', h[i \mapsto h(i)[f \mapsto v]], \phi) \]

**PUTFIELD2-S**
\[ code(pc) = \text{putfield } f \quad \omega = v::\text{null}::\omega' \]
\[ \sigma \Rightarrow_S \text{NullPointerException}, (g, pc, l, \omega', h, \phi) \]

**ANEWARRAY1-S**
\[ code(pc) = \text{anewarray } \tau \quad \omega = m::\omega' \quad i \notin \text{dom } h \]
\[ \sigma \Rightarrow_S (g, \text{next}(pc), l, \omega', h[i \mapsto \text{new-arr}(\text{symbols}(\sigma), \tau, m, m)], \phi) \]

**ANEWARRAY2-S**
\[ code(pc) = \text{anewarray } \tau \quad \omega = X::\omega' \quad i \notin \text{dom } h \]
\[ \sigma \Rightarrow_S (g, \text{next}(pc), l, \omega', h[i \mapsto \text{new-arr}(\text{symbols}(\sigma), \tau, X, k)], \phi \cup \{X \geq 0\}) \]
\( \text{NegativeArraySizeException}, (g, pc, l, \omega', h, \phi \cup \{X < 0\}) \)

**ISTORE1-S**
\[ code(pc) = \text{iastore} \quad \omega = v::i::\omega' \]
\[ \sigma \Rightarrow_S \text{ArrayIndexOutOfBoundsException}, (g, pc, l, \omega', h, \phi \cup \{i < 0 \lor i \geq h(i)(\text{LEN})\}) \]
\[ code(pc) = \text{iastore} \quad \omega = m::\omega' \quad Z = h(i) \quad \tau' \in \text{acc-idx}(Z) \]

**ISTORE2-S**
\[ code(pc) = \text{iastore} \quad \omega = v::i::\omega' \]
\[ \sigma \Rightarrow_S (g, \text{next}(pc), l, \omega', h[i \mapsto Z[\tau' \mapsto v]], \phi \cup \{i = \tau'\}) \]
\[ \sigma \Rightarrow_S \text{NullPointerException}, (g, pc, l, \omega', h, \phi) \]

**ILOAD1-S**
\[ code(pc) = \text{iaload} \quad \omega = i::\omega' \quad Z = h(i) \quad \tau' \in \text{acc-idx}(Z) \]
\[ \sigma \Rightarrow_S \text{ArrayIndexOutOfBoundsException}, (g, pc, l, \omega', h, \phi \cup \{i < 0 \lor h(i)(\text{LEN}) \leq i\}) \]

**ILOAD2-S**
\[ code(pc) = \text{iaload} \quad \omega = i::\omega' \quad Z = h(i) \quad \tau' \in \text{acc-idx}(Z) \]
\[ \sigma \Rightarrow_S (g, \text{next}(pc), l, Z(\tau')::\omega', h, \phi \cup \{i = \tau'\}) \]
\[ \sigma \Rightarrow_S \text{NullPointerException}, (g, pc, l, \omega', h, \phi) \]

**ILOAD3-S**
\[ code(pc) = \text{iaload} \quad \omega = i::\omega' \quad Z = h(i) \quad \tau' \in \text{acc-idx}(Z) \]
\[ \sigma \Rightarrow_S (g, \text{next}(pc), l, v::\omega', h[i \mapsto Z[m,n-1][i \mapsto v]], \phi \cup \{i \neq \tau' \mid \tau' \in I\}) \]
\[ \sigma \Rightarrow_S \text{NullPointerException}, (g, pc, l, \omega', h, \phi) \]
\[ \text{where } v = \begin{cases} \text{Z}^{m,n}(\text{DEF}) & \text{if } Z^{m,n}(\text{DEF}) \downarrow \\ \text{new-prim-sym}(\text{INT}, \text{symbols}(\sigma)) & \text{if } Z^{m,n}(\text{DEF}) \uparrow \end{cases} \]

**ILOAD4-S**
\[ code(pc) = \text{iaload} \quad \omega = i::\text{null}::\omega' \]
\[ \sigma \Rightarrow_S \text{NullPointerException}, (g, pc, l, \omega', h, \phi) \]
D.2.2 Operational Semantics of Symbolic Execution with Lazier Initialization

First we introduce a new semantic domain: the set of symbolic locations, \textbf{SymLocs}, to model unknown non-\texttt{null} references. We let \(\delta\) ranges over symbolic locations and each \(\delta^{\text{\textit{m}}n}\) has the same three properties as non-primitive symbols do. Clearly, we need to add the symbolic locations into values. So we have \(\textbf{Values} = \textbf{Consts} \cup \textbf{Locs} \cup \textbf{Symbols}_{\text{\textit{prim}}} \cup \textbf{SymLocs}\). We use \(\Sigma_a\)\(^3\) to denote the set of lazier states. The only difference between lazier and symbolic states is that the lazier states can have symbolic location. Thus \(\Sigma_a \supset \Sigma_s\).

\(^3\)Subscript \(a\) denotes that the component is a part of lazier state.
Auxiliary Functions

We introduce some auxiliary functions to facilitate the definition of operational semantics of lazier initialization. *init-loc-heap* returns the modified heap and new constraints introduced by initializing a symbolic location to a location. *init-sym-loc* transforms a lazier state into a new lazier state by initializing a symbolic location into a location. *init-sym-loc* takes in a lazier state and a symbolic location and returns a set of states which are end states of input state with the symbolic location is initialized.

\[
\begin{align*}
\text{init-loc-heap} : (\text{Heaps}_a \times \mathcal{P}(\text{Symbols}) \times \text{SymLocs} \times \text{Locs}) \rightarrow (\text{Heaps}_a \times \Phi) \\
\text{init-sym-loc} : \Sigma_a \times \text{SymLocs} \times \text{Locs} \rightarrow \Sigma_a \\
\text{init-sym-loc}^* : \Sigma_a \times \text{SymLocs} \rightarrow \mathcal{P}(\Sigma_a). 
\end{align*}
\]

The definitions are listed as follows with binding \(\sigma_a = (g, pc, l, h, \phi)\):

- the *init-loc-heap* function: \(\text{init-loc-heap}(h_a, ss, \delta_r^{m,n}, i) = (h'_a, \phi')\) where
  
  - if \(i \in \text{dom } h_a\): \(h'_a = \text{sub-fun}_4(h_a, \delta, i)\) and \(\phi' = \{\tau' <: \tau\}\) where \(h_a(i) = X_{\tau'}\).

  - if \(i \notin \text{dom } h_a\):
    \[
    \text{dom } h'_a = \text{dom } h_a \cup \{i\}
    \]
    and
    \[
    \forall j \in \text{dom } h_a, h'_a(j) = \text{sub-fun}_4(h_a(j), \delta_r, i)
    \]
    and \(h'_a(i) = X_{\tau'}\) where
    \[
    X_{\tau'} = \begin{cases} 
    \text{new-sarr}(ss, m, k) & \text{if } \tau \in \text{Types}_{\text{array}} \\
    \text{new-sym}(ss, m, k) & \text{if } \tau \in \text{Types}_{\text{record}} 
    \end{cases}
    \]
    and
    \[
    \phi' = \begin{cases} 
    \{X(\text{LEN}) \geq 0, \tau <: \tau'\} & \text{if } \tau \in \text{Types}_{\text{array}} \\
    \{\tau' <: \tau\} & \text{if } \tau \in \text{Types}_{\text{record}} 
    \end{cases}
    \]

- *init-sym-loc* function,

\[
\text{init-sym-loc} = \lambda(\sigma_a, \delta_r^{m,n}, i).\{(\text{sub-fun}_4(g, \delta, i), pc, \text{sub-fun}_4(l, \delta, i), \text{sub-seq}_1(\omega, \delta, i), \#1(\text{init-loc-heap}(h, \text{symbols}(\sigma_a), \delta_r^{m,n}, i)), \#2(\text{init-loc-heap}(h, \text{symbols}(\sigma_a), \delta_r^{m,n}, i) \cup \phi))\}
\]

- *init-sym-loc*\(^*\) function,

\[
\text{init-sym-loc}^* = \lambda(\sigma_a, \delta_r^{m,n}, i).\{\text{init-sym-loc}(\sigma_a, \delta_r^{m,n}, i) \mid i \in \text{collect}(h) \}
\]

\[
\text{or } i \in (\text{Locs} \setminus \text{dom } h) \text{ if } m \geq 0). \]
In general, the lazier initialization semantic rules are the same as symbolic execution with lazy initialization semantics rules if all the operands are not symbolic locations; otherwise, initializations of the symbolic locations in the operands will be done first. We show the lazier initialization semantic rules for instructions if_acmpeq and getfield below. There are two notable features in the operational semantics for lazier initialization. First, the rules are “small step”. For example, there are three semantics rules for the if_acmpeq instruction: the two rules just initialize the operand if either operand is a symbolic location (the program counter does not change); if two operands are locations, then rule IF_ACMPEQ1-S or IF_ACMPEQ2-S will apply. Second, instead of using a symbolic location to represent all the candidates (null, existing objects, and a new symbolic object) for return, the getfield rule treats null case separately, thus for a reference field access, the getfield will return a non-deterministic choice between null (rule GETFIELD3-S) and a symbolic location which denotes a non-null unknown reference (rule GETFIELD2-A). This is because there are usually a lot of null-ness tests in Java code and specifications; and we still want to take advantage of lazier initialization after a null-ness test. So, for getfield, the rules GETFIELD1,2,3,7-S stay the same in the lazier initialization and rules GETFIELD4,5,6-S are replaced by GETFIELD2-A.

Similar to the symbolic semantics rules, we use the binding \( \sigma = (g, pc, l, \omega, h, \phi) \) and all the end states with path conditions unsatisfiable are ignored.

- **IF_ACMPEQ1-A**
  \[
  \text{code}(pc) = \text{if_acmpeq } pc' \quad \omega = \delta_{r}^{m,n}::\delta_{r}^{m,n}::\omega'
  \]

- **IF_ACMPEQ2-A**
  \[
  \sigma \Rightarrow \lambda (g, pc', \omega', h, \phi)
  \quad \text{code}(pc) = \text{if_acmpeq } pc' \quad \omega = \delta^{m,n}::\omega'
  \]

- **IF_ACMPEQ3-A**
  \[
  \sigma \Rightarrow \lambda \quad \sigma' \quad \text{where } \sigma' \in \text{init-sym-loc}^{*}(\sigma, \delta_{r}^{m,n})
  \quad \text{code}(pc) = \text{if_acmpeq } pc' \quad \omega = \delta_{r}^{m,n}::\omega'
  \]

- **IFNULL-A**
  \[
  \text{code}(pc) = \text{ifnull } pc' \quad \omega = \delta::\omega'
  \]

- **IFNONNULL-A**
  \[
  \sigma \Rightarrow \lambda (g, pc', l, \omega', h, \phi)
  \quad \text{code}(pc) = \text{ifnonnull } pc' \quad \omega = \delta::\omega'
  \]

- **GETFIELD1-A**
  \[
  \sigma \Rightarrow \lambda \quad \sigma' \quad \text{where } \sigma' \in \text{init-sym-loc}^{*}(\sigma, \delta_{r}^{m,n})
  \quad \text{code}(pc) = \text{getfield } f_{r} \quad \omega = \delta_{r}^{m,n}::\omega'
  \]

- **GETFIELD2-A**
  \[
  \omega = i::\omega'
  \quad Y^{m,n} = h(i) \quad Y(f_{r}) \uparrow \quad \tau \in \text{Types}_{\text{non-prim}}
  \quad \text{code}(pc) = \text{getfield } f_{r}
  \quad \sigma \Rightarrow \lambda (g, \text{next}(pc), l, \delta_{r}^{m,1,k}::\omega', h[i \mapsto Y^{m,n}[f_{r} \mapsto \delta_{r}^{m,1,k}]], \phi)
  \]

D.2.3 Operational Semantics of Symbolic Execution with Lazier# Initialization

First we introduce a new semantic domain: the set of symbolic references, SymRefs, to model unknown non-null references or null. We let \( \delta \) ranges over SymRefs and each \( \delta^{m,n} \) just like \( \delta_{r}^{m,n} \) except that it can be initialized to null. Clearly, we need to add the new domain into the domain
Values. So we have

\[
Values = \text{Consts} \cup \text{Locs} \cup \text{Symbols}_{\text{prim}} \cup \text{SymLocs} \cup \text{SymRefs}.
\]

We use \(\Sigma_b^4\) to denote the set of lazier\# states. Clearly \(\Sigma_b \supset \Sigma_a\).

Auxiliary Functions

Similar to lazier initialization, we introduce some auxiliary functions to facilitate the definition of operational semantics of lazier\# initialization:

\[
\begin{align*}
\text{init-sym-ref} : \Sigma_b \times \text{SymRefs} \times (\text{SymLocs} \cup \{\text{NULL}\}) &\rightarrow \Sigma_b \\
\text{init-sym-ref}^* : \Sigma_b \times \text{SymRefs} &\rightarrow \mathcal{P}(\Sigma_b).
\end{align*}
\]

The definitions are listed as follows with binding \(\sigma_b = (g, pc, l, \omega, h, \phi)\):

- **init-sym-ref** function,

\[
\text{init-sym-ref}(\sigma_b, \hat{\delta}, \text{NULL}) = \{ (\text{sub-fun}_1(g, \hat{\delta}, \text{NULL}), pc, \text{sub-fun}_1(l, \hat{\delta}, \text{NULL}), \\
\quad \text{sub-seq}_1(\omega, \hat{\delta}, \text{NULL}), \text{sub-fun}_2(h, \hat{\delta}, \text{NULL}, \phi) \}
\]

and

\[
\text{init-sym-ref}(\sigma_b, \hat{\delta}, \delta) = \{ (\text{sub-fun}_1(g, \hat{\delta}, \delta), pc, \text{sub-fun}_1(l, \hat{\delta}, \delta), \text{sub-seq}_1(\omega, \hat{\delta}, \delta), \\
\quad \text{sub-fun}_2(h, \hat{\delta}, \delta, \phi) \}
\]

- **init-sym-ref** function,

\[
\text{init-sym-ref}^*(\sigma_b, \hat{\delta}) = \{ \text{init-sym-ref}(\sigma_b, \hat{\delta}, \delta) \mid \delta \notin \text{collect-loc}(\sigma_b) \} \\
\quad \cup \{ \text{init-sym-ref}(\sigma_b, \hat{\delta}, \text{NULL}) \}.
\]

In general, the lazier\# initialization semantic rules are the same as symbolic execution with lazier initialization semantics rules if all the operands are not symbolic references; otherwise, initializations of the element in symbolic references in the operands will be done first. We show the lazier\# initialization semantic rules for instructions if, acmp, and getfield below. Compared to lazier initialization, there is difference in the operational semantics for lazier\# initialization. For instruction getfield, instead of returns a non-deterministic choice between NULL and a symbolic location, rule GETFIELD2-B just returns a fresh symbolic reference. So, for getfield, the rules GETFIELD1,2,7-S and GETFIELD1-A stay the same in the lazier\# initialization and rules GETFIELD3, 4,5,6-S are replaced by GETFIELD2-A.

Similar to the symbolic semantics rules, we use the binding \(\sigma = (g, pc, l, \omega, h, \phi)\) and all the end states with path conditions unsatisfiable are ignored.

\(^{4}\)Subscript \(b\) denotes that the component is a part of lazier\# state.
### D.2.4 Bytecode Concrete Execution Semantics

To prove properties of our symbolic execution, we need to formalize the concrete bytecode execution. Thus, we introduce concrete states:

\[
\sigma_c \in \Sigma_c = \text{Globals} \times \text{PCs} \times \text{Locals} \times \text{Stacks} \times \text{Heaps} \times \text{Boolean}.
\]

Compared to the symbolic states, there are three restrictions in concrete states: first, no \( X \in \text{Symbols}_{\text{prim}} \) appears in concrete states; second, no \( \text{SymTypes} \) appears in the concrete states; third, for all \( X_\tau \in \text{Symbols}_{\text{non-prim}} \) which appears in concrete states, all the fields of type \( \tau \) are defined and there is no bound associated with \( X \). Furthermore, \( \text{def} \) and \( \text{conc} \) are removed from the \text{Fields} domain.

We also need to change the definition of \text{new-arr} to \( \text{new-arr}_\tau : \mathcal{P}(\text{Symbols}) \times \text{Types} \times \mathbb{N} \rightarrow \text{Symbols}_{\text{non-prim}} = \lambda (ss, \tau, m).X'_{\tau}, \text{s.t. } X \not\in ss \land \tau' = \text{array-type}(\tau) \land \forall 0 \leq j < m.X_{\tau'}(j) = \text{default}(\tau) \land X_{\tau'}(\text{LEN}) = m. \)

The concrete JVM bytecode operational semantics is listed below. We use the binding \( \sigma = (g, pc, l, \omega, h, \text{TRUE}) \) for all the rules. When the last component of the end state is \text{FALSE}, the transition is ignored. Note that we do not use the wrap around semantics for integral types because it complicates the operational semantics presentation. In addition, we do not concern ourselves to check bugs introduced by integer wrap arounds in our symbolic execution. However, wrap arounds can be supported by using appropriate decision procedures that model integers using bit-vectors.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Code</th>
<th>Concrete Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF_ACMPEQ_1-B</td>
<td>( \text{code}(pc) = \text{if_acmpeq } pc' \quad \omega = \delta^m_n \odot \delta^m_l \odot \omega' ) ( \sigma \Rightarrow (g, pc', \omega', h, \phi) )</td>
<td>( g, pc', \omega', h, \phi ) ( g, pc', \omega', h, \phi )</td>
</tr>
<tr>
<td>IF_ACMPEQ_2-B</td>
<td>( \text{code}(pc) = \text{if_acmpeq } pc' \quad \omega = v \odot \delta^m_n \odot \omega' ) ( \sigma \Rightarrow \sigma' \quad \text{where } \sigma' \in \text{init-sym-ref}^\theta (\sigma, \delta^m_n) )</td>
<td>( g, pc', \omega', h, \phi ) ( g, pc', \omega', h, \phi )</td>
</tr>
<tr>
<td>IF_ACMPEQ_3-B</td>
<td>( \text{code}(pc) = \text{if_acmpeq } pc' \quad \omega = \delta^m_n \odot v \odot \omega' ) ( \sigma \Rightarrow \sigma' \quad \text{where } \sigma' \in \text{init-sym-ref}^\theta (\sigma, \delta^m_n) )</td>
<td>( g, pc', \omega', h, \phi ) ( g, pc', \omega', h, \phi )</td>
</tr>
<tr>
<td>GETFIELD_1-B</td>
<td>( \text{code}(pc) = \text{getfield } f'_\tau \quad \omega = \delta^m_n \odot \omega' ) ( \sigma \Rightarrow \sigma' \quad \text{where } \sigma' \in \text{init-sym-ref}^\theta (\sigma, \delta^m_n) )</td>
<td>( g, pc', \omega', h, \phi ) ( g, pc', \omega', h, \phi )</td>
</tr>
<tr>
<td>GETFIELD_2-B</td>
<td>( \omega = i \odot \omega' \quad Y^m_n = h(i) \quad Y(f_i) \uparrow \quad \tau \in \text{Types}_{\text{non-prim}} \quad \hat{\delta} \text{ is fresh} )</td>
<td>( g, next(pc), l, \delta^m_{1,k} \odot \omega', h[i \mapsto Y^m_n[f_i \mapsto \delta^m_{1,k}]], \phi )</td>
</tr>
</tbody>
</table>

\( \sigma_c \in \Sigma_c = \text{Globals} \times \text{PCs} \times \text{Locals} \times \text{Stacks} \times \text{Heaps} \times \text{Boolean} \)
\[
\begin{align*}
\text{IF\_ICMPLT\_2\_C} & : \quad \text{code}(pc) = \text{if\_icmplt}(pc') \quad \omega = \text{d}\coloneq\text{c} \cdot \omega' \quad c \neq d \\
\text{IF\_ACMPEQ\_1\_C} & : \quad \sigma \Rightarrow_C (g, \text{next}(pc), l, \omega', h, \text{TRUE}) \\
\text{IF\_ACMPEQ\_2\_C} & : \quad \text{code}(pc) = \text{if\_acmpeq}(pc') \quad \omega = i \cdot j \cdot \omega' \quad i \neq j \\
\text{IF\_NULL\_1\_C} & : \quad \sigma \Rightarrow_C (g, \text{next}(pc), l, \omega', h, \phi) \\
\text{IF\_NULL\_2\_C} & : \quad \text{code}(pc) = \text{if\_nonnull}(pc') \quad \omega = \text{NULL} \cdot \omega' \\
\text{IF\_NON\_NULL\_1\_C} & : \quad \sigma \Rightarrow_C (g, pc', l, \omega', h, \phi) \\
\text{IF\_NON\_NULL\_2\_C} & : \quad \text{code}(pc) = \text{if\_nonnull}(pc') \quad \omega = \text{NULL} \cdot \omega' \\
\text{ANE\_WARR\_1\_C} & : \quad \sigma \Rightarrow_C (g, \text{next}(pc), l, \omega', h, \phi) \\
\text{ANE\_WARR\_2\_C} & : \quad \text{code}(pc) = \text{ane\_warray}(\tau) \quad \omega = c \cdot \omega' \quad c \geq 0 \quad i \notin \text{dom} h \\
\text{NEW\_C} & : \quad \sigma \Rightarrow_C (g, \text{next}(pc), l, i \cdot \omega, h[i \mapsto \text{new\_obj}(\text{symbols}(\sigma), \tau)], \text{TRUE}) \\
\text{IA\_STORE\_1\_C} & : \quad \sigma \Rightarrow_C \text{ArrayIndexOutOf BoundsException}, (g, pc, l, \omega', h, \text{TRUE}) \\
\text{IA\_STORE\_2\_C} & : \quad \text{code}(pc) = \text{iastore} \quad \omega = d \cdot c \cdot \omega' \quad 0 \leq c < h(i)(\text{LEN}) \\
\text{IA\_STORE\_3\_C} & : \quad \sigma \Rightarrow_C \text{NullPointerException}, (g, pc, l, \omega', h, \text{TRUE}) \\
\text{IA\_LOAD\_1\_C} & : \quad \sigma \Rightarrow_C \text{ArrayIndexOutOf BoundsException}, (g, pc, l, \omega', h, \text{TRUE}) \\
\text{IA\_LOAD\_2\_C} & : \quad \text{code}(pc) = \text{iaload} \quad \omega = c \cdot \omega' \quad 0 \leq c < h(i)(\text{LEN}) \\
\text{IA\_LOAD\_3\_C} & : \quad \sigma \Rightarrow_C \text{NullPointerException}, (g, pc, l, \omega', h, \text{TRUE}) \\
\text{GET\_FIELD\_1\_C} & : \quad \sigma \Rightarrow_C (g, \text{next}(pc), l, h(i)(c) \cdot \omega', h, \text{TRUE}) \\
\text{GET\_FIELD\_2\_C} & : \quad \text{code}(pc) = \text{getfield} \quad \omega = c \cdot \omega' \quad 0 \leq c < h(i)(\text{LEN}) \\
\end{align*}
\]

29
\[
\begin{align*}
\text{PUTFIELD1-C} & \quad \text{code}(pc) = \text{putfield}\ f \quad \omega = v::i::\omega' \\
& \quad \sigma \Rightarrow_C (g, \text{next}(pc), l, \omega', h[i \mapsto h(i)[f \mapsto v]], \text{TRUE}) \\
\text{PUTFIELD2-C} & \quad \text{code}(pc) = \text{putfield}\ f \quad \omega = v::\text{null}::\omega' \\
& \quad \sigma \Rightarrow_C \text{NullPointerException}, (g, pc, l, \omega', h, \text{TRUE}) \\
\text{INSTANCEOF1-C} & \quad \text{code}(pc) = \text{instanceof}\ \tau \quad \omega = \text{null}::\omega' \\
& \quad \sigma \Rightarrow_C (g, \text{next}(pc), l, 1::\omega', h, \text{TRUE}) \\
\text{INSTANCEOF2-C} & \quad \text{code}(pc) = \text{instanceof}\ \tau \quad \omega = i::\omega' \quad X_{\tau_1} = h(i) \quad \tau_1 <: \tau \\
& \quad \sigma \Rightarrow_C (g, \text{next}(pc), l, 1::\omega', h, \text{TRUE}) \\
\text{INSTANCEOF2-C} & \quad \text{code}(pc) = \text{instanceof}\ \tau \quad \omega = i::\omega' \quad X_{\tau_1} = h(i) \quad \tau_1 \not<: \tau \\
& \quad \sigma \Rightarrow_C (g, \text{next}(pc), l, 0::\omega', h, \text{TRUE}) \\
\text{CHECKCAST1-C} & \quad \text{code}(pc) = \text{checkcast}\ \tau \quad \omega = \text{null}::\omega' \\
& \quad \sigma \Rightarrow_C (g, \text{next}(pc), l, \text{null}::\omega', h, \text{TRUE}) \\
\text{CHECKCAST2-C} & \quad \text{code}(pc) = \text{checkcast}\ \tau \quad \omega = i::\omega' \quad X_{\tau_1} = h(i) \quad \tau_1 <: \tau \\
& \quad \sigma \Rightarrow_C (g, \text{next}(pc), l, i::\omega', h, \text{TRUE}) \\
\text{CHECKCAST2-C} & \quad \text{code}(pc) = \text{checkcast}\ \tau \quad \omega = i::\omega' \quad X_{\tau_1} \not= h(i) \quad \tau_1 \not<: \tau \\
& \quad \sigma \Rightarrow_C \text{ClassCastException}, (g, pc, l, \omega', h, \text{TRUE}) \\
\text{ASSUME1-C} & \quad \text{code}(pc) = \text{assume} \quad \omega = 0::\omega' \\
& \quad \sigma \Rightarrow_C (g, \text{next}(pc), l, \omega', h, \text{FALSE}) \\
\text{ASSUME2-C} & \quad \text{code}(pc) = \text{assume} \quad \omega = 1::\omega' \\
& \quad \sigma \Rightarrow_C (g, \text{next}(pc), l, \omega', h, \text{TRUE}) \\
\text{ASSERT1-C} & \quad \text{code}(pc) = \text{assert} \quad \omega = 0::\omega' \\
& \quad \sigma \Rightarrow_C \text{ERROR}, (g, pc, l, \omega', h, \text{TRUE}) \\
\text{ASSERT2-C} & \quad \text{code}(pc) = \text{assert} \quad \omega = 1::\omega' \\
& \quad \sigma \Rightarrow_C (g, \text{next}(pc), l, \omega', h, \text{TRUE})
\end{align*}
\]

D.3 Formal Proofs

In this section, we will prove the soundness and completeness for symbolic execution relates to concrete execution and lazier symbolic execution relates to symbolic execution.

D.3.1 Relative Soundness and Completeness of Basic Symbolic Execution

In this section, we relate symbolic execution (non-compositional) and concrete execution under the assumption the bounds \( k \) are sufficient large. First we will define a concretization function \( \gamma_s \)
\(^5\) to relate symbolic states and concrete state. Second, we will introduce binary relations between

\(^5\)Since we assume the ideal case: the object bound and array length bounds \( k \) are sufficient large, any symbol/array always has bounds greater than 0.
concrete states and symbolics and prove simulation between concrete state-space and symbolic state-space. Finally, we will prove the relative sound and completeness of symbolic execution regards to concrete execution within one method and no invocation in the body of the method.

**Definition of $$\gamma_s$$**

Let us start with some definitions:

- the set of all environments, $$Env = \{ E \mid E : \text{Symbols}_{prim} \to \text{Consts} \}$$;
- the set of all type environments, $$\Gamma = \{ T \mid T : \text{SymTypes} \to (\text{Types}_{array} \cup \text{Types}_{record}) \}$$;
- the group of all permutations of Locations, Sym(Locs).

Then we introduce some semantic functions\(^\text{1}\) to facilitate the definition of $$\gamma_s$$.

$$\mathcal{V}_s : \text{Values}_s \to ((Env \times \text{Sym(Locs)}) \to \text{Values}_s)$$

$$O_s : \text{Symbols}_{non-prim} \to ((\Gamma \times Env \times \text{Sym(Locs)}) \to \mathcal{P}(\text{Symbols}_{non-prim}))$$

$$\mathcal{H}_s : \text{Heaps}_s \to ((\Gamma \times Env \times \text{Sym(Locs)}) \to \mathcal{P}(\text{Heaps}_s))$$

$$ST_s : \Sigma_S \to ((\Gamma \times Env \times \text{Sym(Locs)}) \to \mathcal{P}(\Sigma_C)).$$

Here are the definitions ($$\forall T \in \Gamma, E \in Env, \rho \in \text{Sym(Locs)}$$):

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

$$\mathcal{V}_s \llbracket V \rrbracket (E, \rho) = \text{sub}(\text{sub}(v, E), \rho)$$

- the $$O_s$$ function:

$$O_s \llbracket X'(\tau) \rrbracket (T, E, \rho) = \{ X'_{\tau} \mid \tau' = \text{sub}(\tau, T) \land \text{mapfields}(X, X'_{\tau}, E, \rho) \},$$

where

$$\text{mapfields}(X, X'_{\tau}, E, \rho) \overset{\text{def}}{=} \forall \lambda.X(\lambda) \downarrow \Longrightarrow X'(\lambda) = \mathcal{V}_s \llbracket X(\lambda) \rrbracket (E, \rho), \text{ if } \tau' \in \text{Types}_{record}$$

$$\text{mapfields}(X, X'_{\tau}, E, \rho) \overset{\text{def}}{=} X'(\text{LEN}) = \mathcal{V}_s \llbracket X(\text{LEN}) \rrbracket (E, \rho) \land \forall \lambda \in \text{acc-idx}(X).$$

$$X'(\mathcal{V}_s \llbracket t \rrbracket (E, \rho)) = \mathcal{V}_s \llbracket X(t) \rrbracket (E, \rho) \land (X(\text{DEF}) \downarrow \Longrightarrow \forall(0 \leq m < X'(\text{LEN})$$

$$\land m \notin \{ \mathcal{V}_s \llbracket t \rrbracket (E, \rho) \mid t \in \text{acc-idx}(X) \}).X'(m) = X(\text{DEF}))$$, if $$\tau' \in \text{Types}_{array}$$

- the $$\mathcal{H}_s$$ function\(^\text{2}\):

$$\mathcal{H}_s \llbracket h_s \rrbracket (T, E, \rho) = \{ h_c \mid \text{contains}(h_c, h_s, T, E, \rho) \land \text{well-typed}(h_c)$$

$$\land \text{well-formed}(h_c, h_s, T, E, \rho) \},$$

\(^\text{1}\)From this point on, we use subscript $$s$$ to denote symbolic state components/domains and $$c$$ for concrete state components/domains.

\(^\text{2}\)An alternative view of functions as sets of pairs may be taken.
where \( \text{contains}(h_c, h_s, T, E, \rho) \) if and only if
\[
\forall (i, X) \in h_s, \exists Y \in O_s[X](T, E, \rho). (\rho(i), Y) \in h_c.
\]

\( \text{well-typed}(h_c) \) if and only if for each non-primitive symbol in \( h_c \) must have all its fields mapped to values of their types. More specifically, each primitive field is mapped to a constant of its type; each reference type field is mapped to either \text{null} or a location in \( h_c \) which is mapped a non-primitive symbol of a compatible type.

\( \text{well-formed}(h_c, h_s, T, E, \rho) \) if and only if for each entry \((i, X_c)\) in \( h_c, X_c \) is well-formed, that is,

1. if \((i, X_c)\) is mapped from \((j, X_s)\) in \( h_s \) \((i = \rho(j) \) and \( X_c \in O_s[X_s](T, E, \rho))\), and if any field \( f \) of \( X_s \) is undefined and non-primitive, \( X_c(f) \) has to be one of following values:
   - \text{null}
   - \( i' \) where \( i' \notin \rho(\text{dom} \ h_s) \).
   - \( i'' \) where \( i'' \in \rho(\text{dom} \ h_s) \) and \( h_s(\rho^{-1}(i''))(\text{conc}) \uparrow \).
2. if \((i, X_c)\) is not mapped from any entry in \( h_s \) \((i \notin \rho(\text{dom} \ h_s))\), all the fields of \( X_c \) are treated as the ones with corresponding undefined fields in \( h_s \).

- the \( ST_s \) function:

\[
ST_s[(g, pc, l, \omega, h, \phi)](T, E, \rho) = \{ (\text{sub-fun}(\text{sub-fun}(g, E), \rho), pc, \text{sub-fun}(\text{sub-fun}(l, E), \rho), \text{sub-seq}(\text{sub-seq}(\omega, E), \rho), h', \text{TRUE}) | h' \in T_h[h](T, E, \rho) \}.
\]

Finally, the definition of \( \gamma_s : \Sigma_s \to \mathcal{P}(\Sigma_c) \) is
\[
\gamma_s(\sigma_s) = \bigcup_{\forall E,T,\phi,\mu} ST_s[(\sigma_s)](T, E, \rho).
\]

**Concrete and Symbolic Kripke Structures**

Given a method \( m \), we have a set of global variables \( G \) and local variables \( L \) (ordered from 0..\( n \)). We use Kripke structures \(^8\) \( C = (\Sigma_C, I_C, \rightarrow_C, L_C) \) and \( S = (\Sigma_S, I_S, \rightarrow_S, L_S) \) to model the state-spaces from the concrete and the symbolic executions, respectively. Each component is defined as following

- states,

\[
\Sigma_C = \Sigma_c \cup (\text{Exception} \times \Sigma_c) \cup (\text{Error} \times \Sigma_c).
\]

\[
\Sigma_S = \Sigma_s \cup (\text{Exception} \times \Sigma_s) \cup (\text{Error} \times \Sigma_s).
\]

Furthermore, we require that all the \( \Sigma_C \) and \( \Sigma_S \) are well typed according to the signature of \( m \).

---

\(^8\)Appendix ?? presents definitions of Kripke structures and simulations on Kripke structures adapted from [6] for a quick reference.
• initial states, according to JVM specification [4], the initial states have empty operand stacks and all the arguments are stored in local. So

\[ I_C = \{ (g_c, pc_{\text{init}}, l_c, \text{nil}, h_c, \text{True}) \mid \text{dom}(g_c) = G \land \text{dom}(l_c) = L \} \],

where \( pc_{\text{init}} \) is the start program counter of the method.

\[ I_S = \{ (g_s, pc_{\text{init}}, l_s, \text{nil}, h_s, \{\text{True}\}) \mid \text{dom}(g_s) = G \land \text{dom}(l_s) = L \} \]

and each local and global is initialized as follows: if it is primitive type, a symbolic primitive symbolic is created; otherwise, it is nondeterministically initialized as a symbolic object with all its fields undefined or \text{null} with all the possible aliasing.

• transition relations,

\[
\begin{align*}
c_1 \rightarrow_C c_2 & \iff c_1 \Rightarrow_C c_2 \land \text{last component of } c_2 \text{ is True.} \\
s_1 \rightarrow_S s_2 & \iff s_1 \Rightarrow_S s_2 \land \text{the path condition of } s_2 \text{ is satisfiable.}
\end{align*}
\]

• labels, we will not use this component. So let them undefined.

Function \( \gamma_s \) is trivially extended to \( \gamma^*_s : \Sigma_S \rightarrow \mathcal{P}(\Sigma_C) \) as

\[
\gamma^*_s(s) = \begin{cases} 
\gamma_s(\sigma_s), & \text{if } s = \sigma_s \text{ for some } \sigma_s \in \Sigma_s; \\
\{ (\text{Exception}, \sigma_c) \mid \sigma_c \in \gamma_s(\sigma_s) \}, & \text{if } s = (\text{Exception}, \sigma_s) \text{ for some } \sigma_s \in \Sigma_s; \\
\{ (\text{Error}, \sigma_c) \mid \sigma_c \in \gamma_s(\sigma_s) \}, & \text{if } s = (\text{Error}, \sigma_s) \text{ for some } \sigma_s \in \Sigma_s.
\end{cases}
\]

**Simulation Relations**

To show the relationship between \( C \) and \( S \), we define a relation.

**Definition 1.** \( \mathcal{R} \subseteq \Sigma_C \times \Sigma_S \), as follows: \( c \mathcal{R} s \iff c \in \gamma^*_s(s) \).

For any \( \sigma_s \) with path condition (\( \phi \)) satisfiable, there exists one \( \sigma_c \) such that \( \sigma_c \mathcal{R} \sigma_s \) since there exist some \( E \) and \( T \) which satisfy \( \phi \).

Clearly, for all \( c_0 \in I_C \), there exists \( s_0 \in I_S \) such that \( c_0 \mathcal{R} s_0 \).

**Proposition 1.** \( C \models_\mathcal{R} S \).

**Proof.** It is sufficient to show that for all \( \sigma_c \in \Sigma_C, \sigma_s \in \Sigma_S \) if \( \sigma_c \rightarrow_C \sigma'_c \) and \( \sigma_c \mathcal{R} \sigma_s \) then there exists \( \sigma'_s \in \Sigma_S \) such that \( \sigma_s \rightarrow_S \sigma'_s \) and \( \sigma'_c \mathcal{R} \sigma'_s \).

We will proceed with the rule induction on \( \rightarrow_C \).

• Rule IADD-C: Let \( \sigma_c = (g_c, pc, l_c, d :: c :: \omega_c, h_c, \text{True}) \), then \( \sigma'_c = (g_c, \text{next}(pc), l_c, (c + d) :: \omega_c, h_c, \text{True}) \). Suppose \( \sigma_c \mathcal{R} \sigma_s \). We need to show that there exists \( \sigma'_s \in \Sigma_S \) such that \( \sigma_s \rightarrow_S \sigma'_s \) and \( \sigma'_c \mathcal{R} \sigma'_s \). Since \( \sigma_c \mathcal{R} \sigma_s \), we have \( \sigma_c \in \gamma_s(\sigma_s) \). The symbolic state \( \sigma_s \) must have the form of \( (g_s, pc, l_s, v_1 :: v_2 :: \omega_s, h_s, \phi) \) for some \( T, E, \rho \) with \( T, E \vdash \phi, \mathcal{V}_s [v_1](E, \rho) = \text{true} \). This implies that the symbolic state \( \sigma_s \) corresponds to a concrete state \( \gamma_s(\sigma_s) \) which is related to \( \sigma'_s \) by some transition rule. The symbolic state \( \sigma'_s \) must satisfy the condition that it can reach \( \sigma'_c \) through the path condition (\( \phi \)).
c, \mathcal{V}_s[v_2](E, \rho) = d, \text{ sub-fun}(\text{sub-fun}(g_s, E), \rho) = g_c, \text{ sub-fun}(\text{sub-fun}(l_s, E), \rho) = l_c, \text{ sub-seq}(\text{sub-seq}(\omega_s, E), \rho) = \omega_c, \text{ and } h_c \in \mathcal{H}_s[h_j](T, E, \rho). \]

Using the rule IADD-S, we get

\[ \sigma_s \rightarrow_s \sigma'_s \text{ with } \sigma'_s = (g_s, \text{next}(pc), l_s, \omega_s, h_s, \phi \cup \{Y = v_1 + v_2\}) \text{ where } Y \text{ is fresh. We only need to show } \sigma'_s \in \gamma_s(\sigma_s'), \text{ that is, to find } T', E', \rho' \text{ such that } \sigma'_s \in \text{ST}_s[\sigma'_s](T', E', \rho'). \]

We claim that

\[ T' = T, E' = E[Y \mapsto c + d], \text{ and } \rho' = \rho \text{ are the right choice. Since } Y \text{ is fresh,} \]

\[ \text{sub-fun}(\text{sub-fun}(g_s, E'), \rho') = \text{sub-fun}(\text{sub-fun}(g_s, E), \rho) = g_c, \text{ sub-fun}(\text{sub-fun}(l_s, E'), \rho') = \text{sub-fun}(\text{sub-fun}(l_s, E), \rho) = l_c, \text{ sub-seq}(\text{sub-seq}(\omega_s, E), \rho') = \text{sub-seq}(\text{sub-seq}(\omega_s, E), \rho) = \omega_c, \text{ and } h_c \in \mathcal{H}_s[h_j](T', E', \rho'). \]

Furthermore, since \[ \mathcal{V}_s[Y](E', \rho) = c + d = \mathcal{V}_s[v_1](E, \rho) + \mathcal{V}_s[v_2](E, \rho), \]

we get \[ T, E' \not\equiv (\phi \cup \{Y = v_1 + v_2\}). \]

Therefore, \[ \sigma'_c \in \text{ST}_s[\sigma'_s](T, E', \rho) \subseteq \gamma_s(\sigma'_s). \]

- Rule IF.ICMPLT2-C: Let \( \sigma_c = (g_c, \text{pc}, l_c, c \mapsto \omega_c, h_c, \text{TRUE}) \), then \( c \geq d \) and \( \sigma'_c = (g_c, \text{next}(pc), l_c, \omega_c, h_c, \text{TRUE}) \). Suppose \( \sigma_c \not\models \sigma_s \). We need to show that there exists \( \sigma'_s \in \Sigma_s \) such that \( \sigma_s \rightarrow_s \sigma'_s \) and \( \sigma'_c \not\models \sigma'_s \). Since \( \sigma_c \not\models \sigma_s \), we have \( \sigma_c \in \gamma_s(\sigma_s) \). The symbolic state \( \sigma_s \) must have the form of \( (g_s, \text{pc}, l_s, v_1 \mapsto v_2 \mapsto \omega_s, h_s, \phi) \) for some \( T, E, \rho \) with \( T, E \models \phi \), \( \mathcal{V}_s[v_1](E, \rho) = c, \mathcal{V}_s[v_2](E, \rho) = d, \text{ sub-fun}(\text{sub-fun}(g_s, E), \rho) = g_c, \text{ sub-fun}(\text{sub-fun}(l_s, E), \rho) = l_c, \text{ sub-seq}(\text{sub-seq}(\omega_s, E), \rho) = \omega_c, \text{ and } h_c \in \mathcal{H}_s[h_j](T, E, \rho). \]

Using the IF.ICMPLT-S rule, we get \[ \sigma_s \rightarrow_s \sigma'_s \text{ with } \sigma'_s = (g_s, \text{next}(pc), l_s, \omega_s, h_s, \phi \cup \{v_1 \geq v_2\}) \] (the first end state). We only need to show \( \sigma'_c \in \gamma_s(\sigma'_s). \) Since \( \mathcal{V}_s[v_1](E, \rho) = c, \mathcal{V}_s[v_2](E, \rho) = d, \) and \( c \geq d \), we get \( T, E \not\equiv \phi \cup \{v_1 \geq v_2\}. \)

Therefore, \[ \sigma'_c \in \text{ST}_s[\sigma'_s](T, E, \rho) \subseteq \gamma_s(\sigma'_s). \]

- Rule ANEWARRAY1-C: Suppose \( \sigma_c = (g_c, \text{pc}, l_c, c \mapsto \omega_c, h_c, \text{TRUE}) \). Then \( c \geq 0 \) and \( \sigma'_c = (g_c, \text{next}(pc), l_c, \omega_c, h'_c, \text{TRUE}) \) where \( i \) is fresh and \( h'_c = h_c[i \mapsto \text{new-arr}_c(\text{symbols}(\sigma_c), \tau, c)] \). Suppose \( \sigma_c \not\models \sigma_s \). We need to show that there exists \( \sigma'_s \in \Sigma_s \) such that \( \sigma_s \rightarrow_s \sigma'_s \) and \( \sigma'_c \not\models \sigma'_s \). Since \( \sigma_c \not\models \sigma_s \), we have \( \sigma_c \in \gamma_s(\sigma_s) \). The symbolic state \( \sigma_s \) has the form of \( (g_s, \text{pc}, l_s, v_1 \mapsto \omega_s, h_s, \phi) \) for some \( T, E, \rho \) with \( T, E \models \phi \), \( \mathcal{V}_s[v_1](E, \rho) = c, \text{ sub-fun}(\text{sub-fun}(g_s, E), \rho) = g_c, \text{ sub-fun}(\text{sub-fun}(l_s, E), \rho) = l_c, \text{ sub-seq}(\text{sub-seq}(\omega_s, E), \rho) = \omega_c, \text{ and } h_c \in \mathcal{H}_s[h_j](T, E, \rho). \)

Using the ANEWARRAY2-S rule, we get \[ \sigma_s \rightarrow_s \sigma'_s \text{ with } \sigma'_s = (g_s, \text{next}(pc), l_s, \omega_s, h'_s, \phi \cup \{v \geq 0\}) \] where \( h'_s = h_s[j \mapsto \text{new-arr}(\text{symbols}(\sigma_s), \tau, X, k)] \) and \( j \) is fresh (the first end state). We only need to show \( \sigma'_c \in \gamma_s(\sigma'_s). \) Define \( \rho' = \rho[j \mapsto i][\rho^{-1}(i) \mapsto \rho(j)]. \) It is clear that \( \rho' \in S \) and for location \( i' \notin \{j, \rho^{-1}(i)\}, \rho'(i') = \rho(i') \).

Since \( i \) is fresh in \( \sigma_c \) and \( \sigma_c \not\models \sigma_s \), \( \rho^{-1}(i) \) must be fresh in \( \sigma_s \) (not in dom \( h_s \)) too. Thus we get \( \text{sub-fun}(\text{sub-fun}(g_s, E), \rho') = g_c, \text{ sub-fun}(\text{sub-fun}(l_s, E), \rho') = l_c, \text{ and } \omega_c \). From \( c \geq 0 \), \( \mathcal{V}_s[v](E, \rho') \geq 0 \), that is, \( T, E \not\equiv \phi \cup \{v \geq 0\}. \) It remains to show \( h'_c \in \mathcal{H}_s[h_j](T, E, \rho') \). Clearly well-typed \( h'_c \) because \( i \) is fresh in \( h_c \). Then we show that \( \text{contains}(h'_c, h'_c, T, E, \rho'). \)

For any entry \( (i', X') \in h_c \), since \( j \) and \( \rho^{-1}(i) \) are fresh in \( h_c \), we get \( O_s[X'](T, E, \rho') = O_s[X](T, E, \rho) \). Furthermore, since \( \text{new-arr}_c(\text{symbols}(\sigma_c), \tau, c) \in O_s(\text{new-arr}(\text{symbols}(\sigma_c), \tau, X, k))(T, E, \rho') \), we can get \( \text{contains}(h'_c, h'_c, T, E, \rho') \). Next we need to show well-formed \( (h'_c, h'_c, T, E, \rho'). \) Since \( \text{new-arr}(\text{symbols}(\sigma_c), \tau, X, k)(\text{conc} \downarrow, \text{symbol} \text{new-arr}_c(\text{symbols}(\sigma_c), \tau, c) \text{ of entry } (i, \text{new-arr}_c(\text{symbols}(\sigma_c), \tau, c)) \text{ in } h'_c \) is well-formed under \( E \) and \( \rho' \). For any symbol \( Y \) in the range of \( h_c \), if \( Y \) has a reference field \( j \) whose corresponding field is not defined in \( h_s \), by the well-formed \( (h'_c, h'_c, T, E, \rho') \), \( f \) can not be any location that points to concrete object in \( h_c \). But \( h'_c \) has only one extra concrete object at \( i \) than \( h_c \).
and i is fresh in $h_c$. Therefore, $f$ can not point to $i$, that is, symbol new-arr$(\text{symbols}(\sigma_c), \tau, c)$ is well-formed. We get well-formed$(h_c', h'_s, T, E, \rho')$. Thus $h'_c \in \mathcal{H}_s[h'_s](T, E, \rho')$. Finally, $\sigma'_c \in ST_s[\mathcal{H}_s[h'_s]](T, E, \rho') \subseteq \gamma_s(\sigma'_s)$.

- **Rule GETFIELD1-C:** Suppose $\sigma_c = (g_c, pc, l_c, i : \omega_c, h_c)$, then $\sigma'_c = (g_c, \text{next}(pc), l_c, v :: \omega_c, h_c)$ where $X = h_c(i), v = X(f)$. Let $\tau_i$ be the real type of symbol $h_c(i)$. Suppose $\sigma_c \mathcal{R} \sigma_s$. We need to show that exists $\sigma'_s \in \Sigma_S$ such that $\sigma_s \rightarrow_S \sigma'_s$ and $\sigma'_c \mathcal{R} \sigma'_s$. Since $\sigma_c \mathcal{R} \sigma_s$, we have $\sigma_c \in \gamma_s(\sigma_s)$. The symbolic state $\sigma_s$ must have the form of $(g_s, pc, l_s, l' :: \omega_s, h_s, \phi)$ for some $T, E, \rho$ with $T \models \phi$, $\rho(\tau_i) = i$, sub-fun(sub-fun($g_s, E, \rho$)) = $g_c$, sub-fun(sub-fun($l_s, E, \rho$)) = $l_c$, sub-seq(sub-seq($\omega_s, E, \rho$)) = $\omega_c$, and $h_c \in \mathcal{H}_s[h'_s](T, E, \rho)$. WLOG, assume that the type of $f$, $\tau$, is a record type and $f$ is not in the domain of $h_s(i)$. We will proceed with a case analysis according to the value of $v$ by well-formed($h_c, h_s, T, E, \rho$):

- **case $v = \text{null}$:** We will apply the GETFIELD3-S rule and get $\sigma'_s = (g_c, \text{next}(pc), l, \text{null} :: \omega, h'_s, \phi), \text{where } h'_s = h_s(i \mapsto h_s(i)[f \mapsto \text{null}])$. It suffices to show contains$(h_s, h'_s, T, E, \rho)$ and well-formed$(h_s, h'_s, T, E, \rho)$. Since $\rho(\tau_i) = i$ and $\sigma_c \mathcal{R} \sigma_s$, $X \in O_s[Y](T, E, \rho)$. Furthermore, Since $h_c \in \mathcal{H}_s[h'_s](T, E, \rho)$ and $X \in O_s[h_s(i)[f \mapsto \text{null}]](T, E, \rho)$ by $X(f) = \text{null} = h_s(i)(f)$, we get contains$(h_s, h'_s, T, E, \rho)$ and well-formed$(h_s, h'_s, T, E, \rho)$ hold. We get $h_c \in \mathcal{H}_s[h'_s](T, E, \rho)$. Then $\sigma'_c \in \gamma_s(\sigma'_s)$.

- **case $v \in \rho(\text{dom } h_s) \land h_s(\rho^{-1}(v))$ (conc) $\uparrow$:** We will apply the rule GETFIELD4-S and get $\sigma'_s = (g_c, \text{next}(pc), l, v' :: \omega, h'_s, \phi \cup \{\tau' <: \tau\})$ where $h'_s = h_s(i \mapsto h_s(i)[f \mapsto j])$ and $Z_{\tau'} = h_s(v')$. We also have $v' = \rho^{-1}(v)$ ($v' \in \text{collect}(h_s)$ because $v \in \rho(\text{dom } h_s)$ and $h_s(\rho^{-1}(v))$ (conc) $\uparrow$). Since well-typed($h_s$), the type of $h_s(v)$, $\tau_v$, is a subtype of $\tau$. Furthermore, since $h_s(v) \in O_s[Z_{\tau'}](T, E, \rho)$, we arrive at $T \models \tau' <: \tau$. Thus $T, E \models \phi \cup \{\tau' <: \tau\}$. The rest of the proof is similar to the null case.

- **case $v \in \text{Locs} \land v \notin \rho(\text{dom } h_s)$:** We will apply the rule GETFIELD6-S (because we assume that bound $k$ is sufficient large and $m > 0$) and get $\sigma'_s = (g_c, \text{next}(pc), l, v :: \omega, h'_s, \phi \cup \{\tau' <: \tau\})$ where $h'_s = h_s(i \mapsto h_s(i)[f \mapsto j])$ and $Z_{\tau'} = \text{new-sym}($symbols$(\sigma), m-1, k)$. Define $\rho' = \rho[j \mapsto v]$ and $T' = T[\tau' \mapsto \tau_v]$. Since well-typed($h_s$), we get $\tau_v <: \tau$. Furthermore, since $\rho' = \rho[j \mapsto v]$ and $T' = T[\tau' \mapsto \tau_v]$, $T' \models \tau' <: \tau$. Thus $T', E \models \phi \cup \{\tau' <: \tau\}$. Since $j$ is fresh in $h_s$, sub-fun(sub-fun($g_s, E, \rho'$)) = sub-fun(sub-fun($g_s, E, \rho$)), sub-fun(sub-fun($l_s, E, \rho'$)) = sub-fun(sub-fun($l_s, E, \rho$)), and sub-seq(sub-seq($\omega_s, E, \rho'$)) = sub-seq(sub-seq($\omega_s, E, \rho$)) hold. It remains to show contains($h_c, h'_s, T, E, \rho$) and well-formed($h_c, h'_s, T, E, \rho$). Since $X \in O_s[Y_{mn}[f \mapsto v]](T', E, \rho')$, and $h_s(v) \in O_s[Z_{\tau'}](T', E, \rho')$, contains($h_c, h'_s, T', E, \rho$) holds. Since $v \notin \rho(\text{dom } h_s)$, $h_s(v)$ is well-formed. Since the new symbol $Z_{\tau'}$ in $h'_s$ has conc field undefined, the rest of symbols in $h_c$ are well-formed. Thus we get well-formed($h_c, h'_s, T, E, \rho$) and further, $h_c \in \mathcal{H}_s[h'_s](T, E, \rho')$. Therefore, $\sigma'_c \in ST_s[\mathcal{H}_s[h'_s]](T, E, \rho') \subseteq \gamma_s(\sigma'_s)$.

\[ \square \]

**Definition 2.** $\mathcal{R}_c \subseteq \Sigma_S \times \mathcal{P}(\Sigma_C)$, as $\sigma_s \mathcal{R}_c \sigma_c \iff \gamma_s(\sigma_s) = S_c$. 

35
The relation \( \mathcal{R}_s \) is left total by definition. Also it is clear that for \( \sigma_s \mathcal{R}_s S_c, S_c \) is not empty, if and only if the path condition \( \phi \) of \( \sigma_s \) is satisfiable. Furthermore, for any \( \sigma_s \in I_S \) and \( \sigma_s \mathcal{R}_s S_c \), it is clear that \( S_c \subseteq I_C \) by the definition of \( \gamma_s \) function.

**Proposition 2.** \( S \prec \mathcal{R}_s \mathcal{P}(C) \).

**Proof.** It is sufficient to show that for all \( \sigma_s, \sigma'_s \in \Sigma_S, S_c, S'_c \in \mathcal{P}(\Sigma_C) \), if \( \sigma_s \rightarrow_S \sigma'_s \), \( \sigma_s \mathcal{R}_s S_c \), and \( \sigma'_s \mathcal{R}_s S'_c \) then \( S_c \rightarrow_C S'_c \).

We will prove by rule induction on symbolic operational semantics transitions, \( \rightarrow_S \).

- Rule IADD-S: \( \sigma_c = (g_s, pc, l_s, v_1 : v_2 : \omega_s, h_s, \phi) \). Then \( \sigma'_s = (g_s, next(pc), l_s, Z :: \omega_s, h_s, \phi \cup \{Z = v_1 + v_2\}) \) where \( Z \) is fresh. Suppose \( \sigma_s \mathcal{R}_s S_c \) and \( \sigma'_s \mathcal{R}_s S'_c \). We need to show that \( S_c \rightarrow_C S'_c \), that is, for any \( \sigma'_c \in S'_c \), there exists some \( \sigma_c \in S_c \) such that \( \sigma_c \rightarrow_C \sigma'_c \). Suppose \( \sigma'_c \in S'_c \), that is, \( \sigma'_c \in \gamma_s(s'_c) \). Then \( \sigma'_c \) must be in the form of \( (g'_c, next(pc), l'_c, \omega'_c, h'_c, \text{TRUE}) \) with some \( T, E, \rho \) such that \( T, E \vdash \phi \cup \{Z = v_1 + v_2\} \), \( \forall \sigma S | \sigma E, \rho \), \( g'_c, sub-fun(sub-fun(g_s, E), \rho) = g_s, sub-fun(sub-fun(l_s, E), \rho) = l'_c, \rho, \rho \leq \omega'_c \). Clearly \( \sigma'_c \rightarrow_C \sigma_c \). We only need to show that \( \sigma_c \in \gamma_s(\sigma_s) \). Since \( Z \) is fresh, \( T, E \vdash \phi \). Thus \( \sigma_c \in ST_s[\sigma_s](T, E, \rho) \leq \gamma_s(\sigma_s) \).

- Rule IFJCOMPLT-S: \( \sigma_c = (g_s, pc, l_s, v_1 : v_2 : \omega_s, h_s, \phi) \) and \( \sigma'_c = (g_s, next(pc), l_s, \omega_s, h_s, \phi \cup \{v_2 \geq v_1\}) \). (We only consider one end state, the other end state is symmetric.) Suppose \( \sigma_s \mathcal{R}_s S_c \) and \( \sigma'_s \mathcal{R}_s S'_c \). We need to show that \( S_c \rightarrow_C S'_c \), that is, for any \( \sigma'_c \in S'_c \), there exists some \( \sigma_c \in S_c \) such that \( \sigma_c \rightarrow_C \sigma'_c \). Suppose \( \sigma'_c \in S'_c \), that is, \( \sigma'_c \in \gamma_s(s'_c) \). Then \( \sigma'_c \) must be in the form of \( (g'_c, next(pc), l'_c, \omega'_c, h'_c, \text{TRUE}) \) with some \( T, E, \rho \) such that \( T, E \vdash \phi \cup \{v_2 \geq v_1\} \), \( sub-fun(sub-fun(g_s, E), \rho) = g'_c, sub-fun(sub-fun(l_s, E), \rho) = l'_c, sub-seq(sub-seq(\omega_s, E), \rho) = \omega'_c \), and \( h'_c \in \mathcal{H}_{\{\omega_s\}}(T, E, \rho) \). Take \( \sigma_c = (g'_c, pc, l'_c, V_{\{v_1\}}(E, \rho) \vdash \omega'_c, h'_c, \text{TRUE}) \). Clearly \( \sigma_c \rightarrow_C \sigma'_c \). We conclude that \( \sigma_c \in ST_s[\sigma_s](T, E, \rho) \leq \gamma_s(\sigma_s) \).

- Rule ANEWARRAY2-S: Suppose \( \sigma_c = (g_s, pc, l_s, X : \omega_s, h_s, \phi) \). We only consider that non-exceptional end state here. Then \( \sigma'_c = (g_s, next(pc), l_s, i : \omega_s, h_s[i \mapsto \text{new-arr}(\text{symbols}(\sigma_s), \tau, X, k)], \phi \cup \{X \geq 0\}) \) where \( i \) is fresh. Suppose \( \sigma_s \mathcal{R}_s S_c \) and \( \sigma'_s \mathcal{R}_s S'_c \). We need to show that \( S_c \rightarrow_C S'_c \), that is, for any \( \sigma'_c \in S'_c \), there exists some \( \sigma_c \in S_c \) such that \( \sigma_c \rightarrow_C \sigma'_c \). Suppose \( \sigma'_c \in S'_c \), that is, \( \sigma'_c \in \gamma_s(s'_c) \). Then \( \sigma'_c \) must be in the form of \( (g'_c, next(pc), l'_c, j : \omega'_c, h'_c, \text{TRUE}) \) with some \( T, E, \rho \) such that \( T, E \vdash \phi \cup \{X \geq 0\} \), \( \rho(i) = j \), \( sub-fun(sub-fun(g_s, E), \rho) = g'_c, sub-fun(sub-fun(l_s, E), \rho) = l'_c, sub-seq(sub-seq(\omega_s, E), \rho) = \omega'_c \), and \( h'_c \in \mathcal{H}_{\{i \mapsto \text{new-arr}(\text{symbols}(\sigma_s), \tau, X, k)\}}(T, E, \rho) \). We need to find a \( \sigma_c \in \Sigma_S \) such that \( \sigma_c \in \gamma_s(\sigma_s) \) and \( \sigma_c \rightarrow_C \sigma'_c \). We claim that \( \sigma_c = (g'_c, pc, l'_c, E(\alpha) : \omega'_c, h'_c, \text{TRUE}) \) where \( h'_c = h'_c \setminus \{j \} \) satisfies the above two conditions. Since \( T, E \vdash \phi \cup \{X \geq 0\} \), \( T, E \vdash \phi \). To show \( \sigma_c \in \gamma_s(\sigma_s) \), it suffices to show that \( h_c \in \mathcal{H}_{\{i \mapsto \text{new-arr}(\text{symbols}(\sigma_s), \tau, X, k)\}}(T, E, \rho) \). Since \text{new-arr}(\text{symbols}(\sigma_s), \tau, X, k) \) will return a symbol with conc field defined and \( h'_c \in \mathcal{H}_{\{i \mapsto \text{new-arr}(\text{symbols}(\sigma_s), \tau, X, k)\}}(T, E, \rho) \), symbols in \( h'_c \) such that their corresponding symbols in \( h_c \) have conc fields not defined or do not have corresponding symbols can not contains \( j \).
Furthermore, since $i$ is fresh in $h_s$, $h_c$ does not have any symbol such that $j$ is in its range. Therefore, well-typed($h_c$), contains($h_c, h_s, T, E, \rho$), and well-formed($h_c, h_s, T, E, \rho$). We get $\sigma_c \in ST_s[\sigma_s](T, E, \rho) \subseteq \gamma_s(\sigma_s)$. Clearly $\sigma_c \rightarrow_C \sigma_c'$.

- Rule GETFIELD3-S: Suppose $\sigma_s = (g_s, pc, l_s, i :: \omega_s, h_s, \phi)$. Then $f$ is not defined in $h_s(i)$ and $\sigma_c' = (g_s, next(pc), l_s, \text{null} :: \omega_s, h_s', \phi')$ where $h_s' = h_s[i \mapsto h_s[i][f \mapsto \text{null}]]$. Suppose $\sigma_s \mathcal{R}_c S_c$ and $\sigma_c' \mathcal{R}_c S_c'$. We need to show that $S_c \rightarrow_C S_c'$, that is, for any $\sigma_c' \in S_c'$, there exists some $\sigma_c \in S_c$ such that $\sigma_c \rightarrow_C \sigma_c'$. Suppose $\sigma_c' \in S_c'$, that is, $\sigma_c' \in \gamma_s(\sigma_c)$. Then $\sigma_c$ must be in the form of $(g_s', next(pc), l_s', \text{null} :: \omega_s', h_s', \phi')$ where $\omega_s' = \omega_s$, $h_s' = h_s[i \mapsto h_s[i][f \mapsto \text{null}][j \mapsto Z_{s'}]]$, $\phi' = \phi \cup \{\tau' :: \tau\}$ where $Z_{s'} = \text{new-sym}(\text{symbols}(\sigma_c), m, 1, k)$ and $j \notin \text{dom} h_s$. Suppose $\sigma_s \mathcal{R}_c S_c$ and $\sigma_c' \mathcal{R}_c S_c'$. We need to show that $S_c \rightarrow_C S_c'$, that is, for any $\sigma_c' \in S_c'$, there exists some $\sigma_c \in S_c$ such that $\sigma_c \rightarrow_C \sigma_c'$. Suppose $\sigma_c' \in S_c'$, that is, $\sigma_c' \in \gamma_s(\sigma_c)$. Then $\sigma_c$ must be in the form of $(g_s', next(pc), l_s', \text{null} :: \omega_s', h_s', \phi')$ where $\omega_s' = \omega_s$, $h_s' = h_s[i \mapsto h_s[i][f \mapsto \text{null}][j \mapsto Z_{s'}]]$, $\phi' = \phi \cup \{\tau' :: \tau\}$ where $Z_{s'} = \text{new-sym}(\text{symbols}(\sigma_c), m, 1, k)$ and $j \notin \text{dom} h_s$. Suppose $\sigma_s \mathcal{R}_c S_c$ and $\sigma_c' \mathcal{R}_c S_c'$. We need to show that $S_c \rightarrow_C S_c'$, that is, for any $\sigma_c' \in S_c'$, there exists some $\sigma_c \in S_c$ such that $\sigma_c \rightarrow_C \sigma_c'$. Define $\rho' = \rho[j \mapsto v']$ and $\sigma_c = (g_s', pc, l_s', i :: \omega_s', h_s', \phi')$. From $h_s' \in \mathcal{H}_s[h_s'](T, E, \rho)$, it is clear that $\sigma_c \rightarrow_C \sigma_c'$. Since $T, E \in \phi \cup \{\tau' :: \tau\}$, $T, E \in \phi$. Then it suffices to show $h_s' \in \mathcal{H}_s[h_s'](T, E, \rho')$. Since $h_s' \in \mathcal{H}_s[h_s'](T, E, \rho)$, well-typed($h_s'$) and contains($h_s', h_s, T, E, \rho$) hold. Since $h_s'(j) = Z$ and $\text{conc} \notin \text{dom} Z$, well-formed($h_s', h_s, T, E, \rho$) hold. Finally, $\sigma_c \in ST_s[\sigma_s](T, E, \rho') \subseteq \gamma_s(\sigma_s)$.

\[ \square \]

Relative Soundness and Completeness

The soundness means that if there is an error in the concrete execution, then the symbolic execution will be able to find it. And the completeness is the converse. We use a theorem prover to decide the satisfiability of path conditions. But in general, theorem provers are neither sound nor complete for the first order logic with integer and float arithmetics. But in this section, we proceed to show the symbolic execution is sound and complete with assumption that the underlying theorem prover is sound and complete. This is why we called it “Relative Soundness and Completeness”.

**Proposition 3** (Soundness). Given any concrete trace $c_1 \rightarrow_C c_2 \rightarrow_C \cdots \rightarrow_C c_n$ with $c_1 \in I_C$, there is a corresponding symbolic trace $s_1 \rightarrow_S s_2 \rightarrow_S \cdots \rightarrow_S s_n$ with $s_1 \in I_S$ such that $c_k \mathcal{R} s_k$ for all $1 \leq k \leq n$. 

37
Proof. We get $s_1$ by the simulation relation between $C$ and $S$. Then we proceed by mathematical induction on $n$ using Proposition 1.

**Proposition 4 (Completeness).** Given any symbolic trace $s_1 \rightarrow_S s_2 \rightarrow_S \cdots \rightarrow_S s_n$ with $s_1 \in I_S$, there is a corresponding concrete trace $c_1 \rightarrow_C c_2 \rightarrow_C \cdots \rightarrow_C c_n$ such that $c_k \mathcal{R} s_k$ for all $1 \leq k \leq n$ and $c_1 \in I_C$.

Proof. Since the $\phi$ of $s_1$ is not false, $C_1 = \gamma'(s_1) \neq \emptyset$. Then we show there exists a trace in $\mathcal{P}(C)$, $C_1 \rightarrow_C C_2 \rightarrow_C \cdots \rightarrow_C C_n$ such that $s_k \mathcal{R}_S C_k$ for all $1 \leq k \leq n$ by mathematical induction on $n$ using Proposition 2. Since the $\phi$ of $s_n$ is satisfiable, then $C_n \neq \emptyset$. Pick any $c_n \in C_n$ and use the definition of $\rightarrow_C$, we get the corresponding concrete trace $c_1 \rightarrow C c_2 \rightarrow C \cdots \rightarrow C c_n$. □

### D.3.2 Relative Soundness and Completeness of Symbolic Execution with Lazier Initialization

Following the outline of Section D.3.1, we relate the lazier initialization symbolic execution in Section D.2.2 and symbolic execution in Section D.2.1. First, we define a function $\gamma_a$ which given a lazier symbolic state, it returns all the symbolic states that have the same shape and only change symbolic locations to concrete locations. Then we introduce binary relations between symbolic states (power) and lazier symbolic state-spaces. Finally, we will prove the relative sound and completeness of lazier symbolic execution with regards to symbolic execution intra-procedurally.

**Definition of $\gamma_a$**

Let us first introduce a definition: The set of all symbolic variable environments

$$\Pi = \{ F \mid F : \text{SymLocs} \rightarrow \text{Locs} \}. \quad (D.1)$$

Then we define some semantics functions with subscript $a$ denoting lazier symbolic domain-components:

$$\mathcal{H}_a : (\text{Heaps}_a \times \Phi) \rightarrow (\mathcal{P}(\text{Symbols}) \times \mathcal{P}(\text{SymLocs}) \times \Pi) \rightarrow \mathcal{P}(\text{Heaps}_a \times \Phi))$$

$$\mathcal{S}T_a : \Sigma_a \rightarrow \Pi \rightarrow \mathcal{P}(\Sigma_a).$$

The definitions \(^9\) are listed as follows ($\forall F \in \Pi$).

- the $\mathcal{H}_a$ function:

  $$\mathcal{H}_a[(h_a, \phi)](ss, \Delta, F) = \{(h_s, \phi') \mid \text{well-mapped}(\Delta, h_a, F) \land \text{heap}(ss, \Delta, h_a, h_s, F)$$
  $$\land \text{pc}(\phi', \phi, h_s, F) \land \phi' \text{ is satisfiable} \},$$

  where $\text{well-mapped} : \mathcal{P}(\text{SymLocs}) \times \text{Heaps}_a \times \Pi \rightarrow \text{BOOLEAN}$ with $\text{well-mapped}(\Delta, h_a, F)$ if and only if

  $$\forall \delta \in \Delta. (h_a(F(\delta)) \uparrow \lor h_a(F(\delta))(\text{conc}) \uparrow);$$

\(^9\)Subscript $a$ is frequently used to indicate a component in the lazier symbolic states.
heap : \mathcal{P}(\text{Symbols}) \times \mathcal{P}(\text{SymLocs}) \times \text{Heaps} \times \Pi \to \text{Boolean} \text{ with } heap(ss, \Delta, h_s, h_s, F)

if and only if

\begin{align*}
dom h_s = & \ dom h_s \cup F(\Delta) \land \forall i \in dom h_s.h_s(i) = \text{sub-fun}(h_s(i), F) \\
& \land \forall i \in (dom h_s - dom h_s).h_s(i) = X_\tau,
\end{align*}

where \( X_\tau = \begin{cases} 
\text{new-sarr}(ss \cup h_s(\text{Locs} - \{i\}), k, k), & \text{if } \exists \delta', \tau'' \in F^{-1}(i) \text{ such that } \tau'' \in \text{Types}_{\text{array}} \\
\text{new-sym}(ss \cup h_s(\text{Locs} - \{i\}), k, k) & \text{otherwise};
\end{cases} \)

pc : \Phi \times \Phi \times \text{Heaps} \times \Pi \times \mathcal{P}(\text{SymLocs}) \to \text{Boolean} \text{ with } pc(\phi', \phi, h_s, F, \Delta) \text{ if and only if } \phi' \text{ is the least set of predicates that satisfies following conditions:}

\phi \subseteq \phi' \land \forall \delta \in \Delta. \tau' < \tau \in \phi' \land X(\text{len}) \geq 0 \in \phi' \text{ if } \tau \in \text{Types}_{\text{array}} \text{ where } h_s(F(\delta)) = X_{\tau'}.

Note: similar the property of substitution, Lemma 2,

\[ \mathcal{H}_a[(h_a, \phi)](ss, \Delta, F) = \mathcal{H}_a[(h_a, \phi)](ss, \Delta, F \mid \Delta), \]

for any \( F \). The \( \mathcal{H}_a \) function either returns a empty set which means contradicting \( F \) or a set with a single element.

- the \( ST_a \) function (we use binding \( \sigma_a = (g, pc, l, \omega, h, \phi) \)):

\[ ST_a[\sigma_a](F) = \{ \text{sub-fun}(g, F), pc, \text{sub-fun}(l, F), \text{sub-seq}(\omega, F), h', \phi' \mid (h', \phi') \in \mathcal{H}_a[(h, \phi)](\text{symbols}(\sigma_a), \text{collect-sym-locs}(\sigma_a), F) \}, \]

where \( \text{collect-sym-locs} \) takes in a state and returns the set of symbolic locations that appear in the state. In the light of the return of \( \mathcal{H}_a \) function can only be \( \emptyset \) or a singleton, \( ST_a \) function return \( \emptyset \) or a singleton too.

Finally, the definition of \( \gamma_a : \Sigma_a \to \mathcal{P}(\Sigma_s) \) is

\[ \gamma_a(\sigma_a) = \bigcup_{F \in \Pi} ST_a[\sigma_a](F). \]

Properties of \( \gamma_a \)

**Definition 3.** A location \( i \) is a legal value for \( \delta \) regarding to a lazier symbolic state \( \sigma_a = (g, pc, l, \omega, h, \phi) \) if and only if the following conditions hold:

1. \( \delta \in \text{collect-sym-locs}(\sigma_a) \);
2. \( i \notin \text{dom } h \text{ or } h(i)(\text{conc}) \uparrow; \)
3. \( (h', \phi') = \text{init-loc-heap}(h, \text{symbols}(\sigma_a), \delta, i) \) with \( \phi' \) is satisfiable.
Lemma 3. Let \( \sigma_a \in \Sigma_a \) and \( F \in \Pi \). Suppose \( \sigma_s \in ST_a[\sigma_a](F) \). For any \( (\delta, i) \in F \), if \( \sigma'_a \in \text{init-sym-loc}(\sigma_a, \delta, i) \) and \( i \) is a legal value for \( \delta \) regarding to \( \sigma_a \), then \( \sigma_s \in ST_a[\sigma'_a](F) \).

Proof. Suppose \( \sigma_a = (g_a, pc, l_a, \omega_a, h_a, \phi) \) and \( \sigma_s = (g_s, pc, l_s, \omega_s, h_s, \phi_s) \). By the definition of \( \text{init-sym-loc} \), \( \sigma'_a = (\text{sub-fun}_1(g_a, \delta, i), pc, \text{sub-fun}_1(l_a, \delta, i), \text{sub-seq}_1(\omega_a, \delta, i), h'_a, \phi') \), where \( \sigma_s \in ST_a[\sigma_a](F) \), we have \( g_s = \text{sub-fun}(\text{sub-fun}_1(g_a, \delta, i), F) \), \( l_s = \text{sub-fun}(\text{sub-fun}_1(l_a, \delta, i), F) \), and \( \omega_s = \text{sub-seq}(\text{sub-seq}_1(\omega_a, \delta, i), F) \) by Lemma 1. It remains to show that

\[
(h_s, \phi_s) \in H_a[(h'_a, \phi')] (\text{symbols}(\sigma'_a), \text{collect-sym-locs}(\sigma'_a), F).
\]

We know that \( (h_s, \phi_s) \in H_a[(h_a, \phi)] (\text{symbols}(\sigma_a), \text{collect-sym-locs}(\sigma_a), F) \). We will proceed by the definition of \( H_a \). Since \( \sigma'_a \) has one fewer symbolic location \( (\delta) \) than \( \sigma_a \), the predicate \( \text{well-mapped}(\text{symbols}(\sigma'_a), \text{collect-sym-locs}(\sigma'_a), F) \) holds. Also it is easy to see that both \( \text{heap}(\text{collect-sym-locs}(\sigma'_a), h'_a, h_s, F) \) and \( pc(\phi, \phi', h_s, F, \text{collect-sym-locs}(\sigma'_a)) \) hold. We conclude that \( \sigma_s \in ST_a[\sigma'_a](F) \) holds. \( \square \)

Lemma 4. Let \( \sigma_a \in \Sigma_a \) and \( F \in \Pi \). For any \( (\delta, i) \in F \) where \( i \) is a legal value for \( \delta \) regarding to \( \sigma_a \), if \( \sigma'_a \in \text{init-sym-loc}(\sigma_a, \delta, i) \) and \( \sigma_s \in ST_a[\sigma'_a](F) \), then \( \sigma_s \in ST_a[\sigma_a](F) \).

Proof. Similar to Lemma 3, the difficult part is to show that

\[
(h_s, \phi_s) \in H_a[(h_a, \phi)] (\text{symbols}(\sigma_a), \text{collect-sym-locs}(\sigma_a), F).
\]

We know that \( (h_s, \phi_s) \in H_a[(h'_a, \phi')] (\text{symbols}(\sigma'_a), \text{collect-sym-locs}(\sigma'_a), F) \). We will proceed by the definition of \( H_a \). Since \( \sigma_a \) has one more symbolic location \( (\delta) \) than \( \sigma'_a \) and by \( i \) is legal for \( \delta \) regarding to \( \sigma_a \), the predicate \( \text{well-mapped}(\text{symbols}(\sigma_a), \text{collect-sym-locs}(\sigma_a), F) \) holds. Also it is easy to see that both \( \text{heap}(\text{collect-sym-locs}(\sigma_a), h_a, h_s, F) \) and \( pc(\phi, \phi', h_s, F, \text{collect-sym-locs}(\sigma_a)) \) hold. We conclude that \( \sigma_s \in ST_a[\sigma_a](F) \) holds. \( \square \)

Lazier Kripke Structure

For any given method \( m \), we have a set of global variables \( G \) and local variables \( L \) (ordered from 0..n). We use Kripke structure \( \mathcal{A} = (\Sigma_{\mathcal{A}}, I_{\mathcal{A}}, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}}) \) to model the state-space from the lazier initialization symbolic executions. The components are defined as follows:

- states, \( \Sigma_{\mathcal{A}} = \Sigma_a \cup (\text{Exception} \times \Sigma_a) \cup (\text{Error} \times \Sigma_a) \).

- initial states,

\[
I_{\mathcal{A}} = \{ (g_a, pc_{\text{init}}, l_a, \text{nil}, h_a, \text{TRUE}) \mid \text{dom}(g_a) = G \land \text{dom}(l_a) = L \},
\]

and each local and global is initialized as follows: if it is primitive type, a symbolic primitive symbolic is created; otherwise, it is nondeterministically initialized as a fresh symbolic location or \text{null}.

- transition relation, \( a \xrightarrow{\mathcal{A}} a' \iff a \xrightarrow{\mathcal{A}} a_2, a_2 \xrightarrow{\mathcal{A}} a_3, \ldots, a_n \xrightarrow{\mathcal{A}} a' \) for some \( n \in \mathbb{N} \) with program counters of \( a, a_2, \ldots, a_n \) are the same and the program counter of \( a \) and \( a' \) are different and the path condition of \( a' \) is satisfiable.
• labels, we do not use this part and thus they are ignored.

Similar to $\gamma_s$, function $\gamma_a$ is trivially extended to $\gamma_a^*: \Sigma_{\mathcal{A}} \rightarrow \mathcal{P}(\Sigma_S)$ as

$$
\gamma_a^*(a) = \begin{cases} 
\gamma_a(\sigma_a), & \text{if } a = \sigma_a \text{ for some } \sigma_a \in \Sigma_a; \\
\{ (\text{Exception}, \sigma_a) \mid \sigma_s \in \gamma_a(\sigma_a) \}, & \text{if } a = (\text{Exception}, \sigma_a) \text{ for some } \sigma_a \in \Sigma_a; \\
\{ (\text{Exception}, \sigma_a) \mid \sigma_s \in \gamma_a(\sigma_a) \}, & \text{if } a = (\text{Error}, \sigma_a) \text{ for some } \sigma_a \in \Sigma_a.
\end{cases}
$$

**Simulation Relations**

We introduce a relation $\mathcal{R}'$ between lazier symbolic states $\Sigma_{\mathcal{A}}$ and $\Sigma_S$ as follows:

**Definition 4.** $\sigma_s \mathcal{R}' \sigma_a \iff \sigma_s \in \gamma_a^*(\sigma_a)$.

Clearly, for all $s_0 \in I_S$, there exists a $a_0 \in I_{\mathcal{A}}$ such that $s_0 \mathcal{R}' a_0$.

**Proposition 5.** $S \preceq_{\mathcal{R}'} \mathcal{A}$.

**Proof.** It is sufficient to show that for all $\sigma_s, \sigma'_s \in \Sigma_S, \sigma_a \in \Sigma_{\mathcal{A}}$ if $\sigma_s \rightarrow_S \sigma'_s$ and $\sigma_s \mathcal{R}' \sigma_a$ then there exists $\sigma'_a \in \mathcal{A}$ such that $\sigma_a \rightarrow_{\mathcal{A}} \sigma'_a$ and $\sigma'_s \mathcal{R}' \sigma'_a$. We will proceed with the rule induction on $\rightarrow_S$.

- **Rule IF-ACMPEQ1-S:** Let $\sigma_s = (g_s, pc, l_s, i :: j :: \omega_s, h_s, \phi)$. Then $i \neq j$ and $\sigma'_s = (g_s, \text{next}(pc), l_s, \omega_s, h_s, \phi)$. Suppose $\sigma_s \mathcal{R}' \sigma_a$, we need to show there exists $\sigma'_a \in \mathcal{A}$ such that $\sigma_a \rightarrow_{\mathcal{A}} \sigma'_a$ and $\sigma'_s \mathcal{R}' \sigma'_a$. Since $\sigma_s \mathcal{R}' \sigma_a$, we have $\sigma_s \in \gamma_a(\sigma_a)$. WLOG, suppose that $\sigma_a$ has the form of $(g_a, pc, l_a, \delta, \delta', \omega_a, h_a, \phi')$ for some $F$ with $\mathcal{V}'[\delta](F) = i$, $\mathcal{V}'[\delta'](F) = j$, and $\sigma_s \in \mathcal{S}\overline{T}_a[\sigma_a](F)$. After taking the IF-ACMPEQ3-A rule, we get an invisible state $t_1 = (g'_a, pc, l'_a, i :: \delta', \omega'_a, h'_a, \phi''')$ with $t_1 \in \text{init-sym-loc}(\sigma_a, \delta, i)$. By Lemma 3, we have $\sigma_s \in \mathcal{S}\overline{T}_a[t_1](F)$. After taking the IF-ACMPEQ2-A rule, we get another invisible state $t_2 = (g''_a, pc, l''_a, i :: j :: \omega''_a, h''_a, \phi''''')$ with $t_2 \in \text{init-sym-loc}(t_1, \delta', j)$. By Lemma 3, we have $\sigma_s \in \mathcal{S}\overline{T}_a[t_2](F)$. Finally, we take the IF-ACMPEQ1-S rule and get $\sigma'_a = (g''_a, \text{next}(pc), l''_a, \omega'_a, h''_a, \phi''''')$. Now it is sufficient to show that $\sigma'_a \in \gamma_a(\sigma_a)$. Clearly $\text{sub-fun}(g''_a, F) = \text{sub-fun}(g_a, F) = g_s, \text{sub-fun}(l''_a, F) = \text{sub-fun}(l_a, F) = l_s$, and $\text{sub-seq}(\omega''_a, F) = \text{sub-seq}(\omega_a, F) = \omega$, by applying Lemma 1 twice. It remains to show that $(h_s, \phi) \in \mathcal{H}_a[(h'_a, \phi''')]((\text{symbols}(\sigma'_a), \text{collect-sym-locs}(\sigma'_a), F))$. Since $\text{symbols}(\sigma'_a) = \text{symbols}(t_2)$ and $\text{collect-sym-locs}(\sigma'_a) = \text{collect-sym-locs}(t_2) = \text{collect-sym-locs}(\sigma_a) \setminus \{\delta, \delta'\}$, we get $(h_s, \phi) \in \mathcal{H}_a[(h''_a, \phi''''')((\text{symbols}(\sigma'_a), \text{collect-sym-locs}(\sigma'_a), F))$. Therefore, $\sigma'_a \in \gamma_a(\sigma_a)$.

- **Rule GETFIELD3-S:** Suppose $\sigma_s = (g_s, pc, l_s, i :: \omega_s, h_s, \phi)$. Then $\sigma'_s = (g_s, \text{next}(pc), l_s, \text{null} :: \omega_s, h'_s, \phi)$ where $h_s(i) = Y$ and $h'_s[i \mapsto Y[f], f \mapsto \text{null}]$. Suppose $\sigma_s \mathcal{R}' \sigma_a$, we need to show there exists $\sigma'_a \in \mathcal{A}$ such that $\sigma_a \rightarrow_{\mathcal{A}} \sigma'_a$ and $\sigma'_s \mathcal{R}' \sigma'_a$. Since $\sigma_s \mathcal{R}' \sigma_a$, we have $\sigma_s \in \gamma_a(\sigma_a)$. WLOG, suppose that $\sigma_a$ has the form of $(g_a, pc, l_a, \delta, \delta' :: \omega_a, h_a, \phi')$ for some $F$ with $\mathcal{V}'[\delta](F) = i$ and $\sigma_s \in \mathcal{S}\overline{T}_a[\sigma_a](F)$. After taking the GETFIELD1-A rule, we get an invisible state $t = (g'_a, pc, l'_a, i :: \omega'_a, h'_a, \phi'')$ with $t \in \text{init-sym-loc}(\sigma_a, \delta, i)$. By Lemma 3, we have $\sigma_s \in \mathcal{S}\overline{T}_a[t](F)$. Finally, we take the rule GETFIELD3-S and get $\sigma'_a =$
(g′, next(pc), l′, null :: [ω′, h′[i → h′(i)[f′ → null]], φ′]). We need to show σ′ ∈ γa(σa).
By Lemma 1, sub-fun((g′, F) = sub-fun((g, F) = g, sub-fun(l, F) = sub-fun(l′, F) = l′, and
sub-seq(ω′, F) = sub-seq(ω, F) = ω, hold. It is sufficient to show that (h[i → h(i)[f′ → null]], φ) ∈ HA[[h[i → h′(i)[f′ → null]], φ′]](symbols(σa), collect-sym-locs(σa), F).
Since symbols(σa) = symbols(t) and collect-sym-locs(σa) = collect-sym-locs(t) = collect-sym-locs(σa \ {δ}), and h′(i)(f) = h(i)(f) = null, (h[i → h′(i)[f′ → null]], φ) ∈ HA[[h′(i)[f′ → null]], φ′]](symbols(σa), collect-sym-locs(σa), F) by the definition of HA.

• Rule GETFIELD6-S Suppose σa = (g, pc, l, i :: ω, h, φ). Then σ′ = (g, next(pc), l′, j :: ω, h′, φ′) where h′(i) = Y\[m,n\] and h′ = h[i → Y\[m,n\][f → j]][j → Z], φ′ = φ \ {τ′ < τ} where Z\[τ\] = new-sym(\(symbols(σ), m − 1, k\) and j \∈ dom h. Suppose σa \∈ A, we need to show there exists σ′ ∈ A such that σa → σ′ and σ′ \∈ A. Since σa \∈ A, we have σa \∈ γa(σa). WLOG, suppose that σa has the form of (g, pc, l, δ :: ω, h, φ) for some F with \(V(δ)](F) = i\) and σa \∈ ST\[a][σa](F). After taking the GETFIELD1-A rule, we get an invisible state t = (g, pc, l, i :: ω, h, φ) with t \∈ init-sym-loc(σa, δ, i). By Lemma 3, we get σa \∈ ST\[a][t](F). Finally, we can take GETFIELD3-S transition rule and get σ′ = (g′, next(pc), l′, i :: ω′, h′, φ′) where δ′ = δ \∈ collect-sym-locs(t). Let F′ = F[δ′ → j]. Since δ′ is fresh in t, sub-fun(g′, F′) = sub-fun(g, F) = g, sub-fun(l′, F′) = sub-fun(l, F) = l, and sub-seq(ω′, F′) = sub-seq(ω, F) = ω, it remains to show (h[i → h′(i)[f → j]][j → Z], φ′ \{τ′ < τ\}) \∈ HA[[h′(i)[f → δ′]], φ′]](symbols(σa), collect-sym-locs(σa)). Since we already have (h, φ) \∈ HA[[h′, φ′]](symbols(t), collect-sym-locs(t), F), According to the definition of HA function, we only need to consider the extra elements: δ′, j, and Z. Since j is not in the domain of h, j is not in the domain of h′. So j is not in the domain of h′[i → h′(i)[f → δ′]]. We get well-mapped(collect-sym-locs(t) \ {δ′}, h′[i → h′(i)[f → δ′]], F′). Since Z = new-sym(\(symbols(σ), m − 1, k\) and F′ = j, we have heap(collect-sym-locs(t) \ {δ′}, h′[i → h′(i)[f → δ]], h[i → h′(i)[f → j]][j → Z], collect-sym-locs(t) \ {δ′}, F)), which holds. Thus (h[i → h′(i)[f → j]][j → Z], φ \{τ′ < τ\}) \∈ HA[[h′(i)[f → δ]], φ′]](symbols(σa), collect-sym-locs(σa), F), which holds.

□

Next we define a relation.

Definition 5. \( \mathcal{R}_a \subseteq \Sigma_I \times \mathcal{P}(\Sigma_S)\), as follows:

\[ \sigma_a \mathcal{R}_a S_s \iff \gamma_a(\sigma_a) = S_s \]

Clearly, \( \mathcal{R}_a \) is left total. Since \( \mathcal{R}_a \) is right total, then for all σa, if σa \mathcal{R}_a S_s, then S_s \neq \emptyset. Furthermore, for any σa \∈ I_a and σa \mathcal{R}_a S_s, it is clear that S_s \subseteq I_s by the definition of γa function.

Proposition 6. \( \mathcal{A} \prec_{\mathcal{R}_a} \mathcal{P}(\Sigma) \).

Proof. It is sufficient to show that for all σa \∈ \Sigma_I, S_s \in \mathcal{P}(\Sigma_S) if σa → a \sigma_a and σa \mathcal{R}_a S_s and σ′ \mathcal{R}_s S′_s then S_s → S S′_s.

We will prove by rule induction on transitions, → a.
• Rule if_acmpeq: Suppose, WLOG, \( \sigma_a = (g_a, pc, l_a, \delta, \delta', \omega_a, h_a, \phi') \). Then by the definition of \( \rightarrow_{\mathcal{A}} \), the rule consists of three lazier symbolic transitions rules: IF_ACMEPQ3-A, IF_ACMEPQ2-A, and IF_ACMEPQ1-S or IF_ACMEPQ2-S. After taking IF_ACMEPQ3-A rule, we get an invisible state \( t_1 = (g'_a, pc, l_a', i :: \delta'_i :: \omega_a, h_a', \phi''') \) for some \( i \in \text{Locs} \) and \( t_1 \in \text{init-sym-loc}(\sigma_a, \delta, i) \). Then after taking IF_ACMEPQ2-A rule, we get another invisible state \( t_2 = (g''_a, pc, l''_a, i :: j :: \omega'_a, h'_a', \phi''') \) for some \( j \in \text{Locs} \) and \( t_2 \in \text{init-sym-loc}(t_2, \delta', j) \). WLOG, suppose \( i \neq j \) (the \( i = j \) case is symmetric). Finally, we take IF_ACMEPQ1-S rule and get \( \sigma'_a = (g'''_a, \text{next}(pc), l''_a', \delta'_i, \omega'_a, h'_a', \phi''') \). Suppose \( \sigma_a, R'_a, S_s \) and \( \sigma'_a, R'_a, S'_s \). We need to show that \( S_s \rightarrow_S S'_s \), that is, for any \( \sigma'_s \in S'_s \), there exists some \( \sigma_s \in S_s \) such that \( \sigma_s \rightarrow_S \sigma'_s \). Suppose \( \sigma'_s \in S'_s \), that is, \( \sigma'_s \in \gamma_a(\sigma'_s) \). Then \( \sigma'_s \) must be in the form of \( \sigma'_s = (g'_s, \text{next}(pc), l'_s, \omega'_s, h'_s, \phi') \) for some \( F \) and \( \sigma'_s \in \mathcal{ST}_a[\sigma_a](F) \). Define \( \sigma_s = (g'_s, \text{next}(pc), l'_s, i :: j :: \omega'_s, h'_s, \phi) \). It is clear that \( \sigma_s \rightarrow_S \sigma'_s \). We only need to show \( \sigma_s \in S_s \), that is, \( \sigma_s \in \gamma_a(\sigma_a) \). Define \( F' = F[\delta \mapsto i][\delta' \mapsto j] \). We will show \( \sigma_s \in \mathcal{ST}_a[\sigma_s](F') \). Since \( \delta \) and \( \delta' \) do not appear in \( \sigma'_a \), thus \( t_2 \), we have \( \sigma_s \in \mathcal{ST}_a[t_2](F') \) by Lemma 2 and property of \( \mathcal{H}_a \). By applying Lemma 4 twice, we get \( \sigma_s \in \mathcal{ST}_a[\sigma_a](F') \).

• Rule getfield \( f_t \): Suppose, WLOG, \( \sigma_a = (g_a, pc, l_a, \delta, \omega_a, h_a, \phi') \) and \( \tau \in \text{Types}_{\text{record}} \). By the definition of \( \rightarrow_{\mathcal{A}} \), the transition consists of two lazier rules: GETFIELD1-A and (GETFIELD2-A, GETFIELD3-A, or GETFIELD1-S). After taking the GETFIELD1-A rule, we get an invisible state \( t = (g'_a, pc, l'_a, i :: \omega_a', h'_a', \phi''') \) for some \( i \in \text{Locs} \) and \( t = \text{init-sym-loc}(\sigma_a, \delta, i) \). WLOG, assume that \( f \) is undefined in \( h_a(i) \). We take the GETFIELD2-A rule and get \( \sigma'_a = (g''_a, \text{next}(pc), l''_a, \delta', \omega'_a, h'_a', \phi'') \). Suppose \( \sigma'_s \in S'_s \), that is, \( \sigma'_s \in \gamma_a(\sigma'_a) \). Then \( \sigma'_s \) must be in the form of \( (g'_s, \text{next}(pc), l'_s, i :: \omega'_s, h'_s, \phi) \) for some \( F \) such that \( F(\delta') = j \) and \( \sigma'_s \in \mathcal{ST}_a[\sigma_s](F) \). Define \( h_s \) as \( h'_s \) after following two operations:

1. remove \( h'_s(i)(f) \). So the field \( f \) of \( h_s(i) \) becomes undefined.
2. if no symbol in \( h_s \) has a field points to \( h'_s(i)(f) \), then the entry at location \( h'_s(i)(f) \) is removed from \( h_s \).

Define \( \phi_s \) as satisfying \( pc(\phi_s, \phi'', h_s, F, \text{collect-sym-locs}(t)) \), so \( \phi_s \cup \{ \tau' :: \tau \} = \phi \) where \( Z_{\tau'} = h'_s(F(\delta')) \). Define \( \sigma_s = (g'_s, \text{next}(pc), l'_s, i :: \omega'_s, h'_s, \phi_s) \). We will first show \( \sigma_s \in S_s \) and then \( \sigma_s \rightarrow_S \sigma'_s \). To show \( \sigma_s \in S_s \), it suffices to show \( \sigma_s \in \mathcal{ST}_a[t](F) \) (then we can apply Lemma 4 with \( F[\delta \mapsto i] \)). Now we use the definition of \( \mathcal{H}_a \) to show \( (h_s, \phi_s) \in \mathcal{H}_a[(h'_s, \phi''')](\text{symbols}(t), \text{collect-sym-locs}(t), F) \). Since \( pc \) predicate obviously holds by construction of \( \phi_s \), it suffices to show the well-mapped and heap predicates. Since \( h'_s \) has one less symbolic location \( (\delta') \) than \( h'_s[i \mapsto h'_s(i)(f \mapsto \delta')] \), well-mapped(\text{collect-sym-locs}(t), h'_s, F) holds. We will prove the heap predicate by an cases analysis according the freshness of \( F(\delta') \):

\[- F(\delta') \notin F(\text{collect-sym-locs}(t)) \cup \text{dom } h'_s : \text{then the entry } (F(\delta'), h'_s(F(\delta'))) \text{ is removed from } h_s. \] Since heap(\text{collect-sym-locs}(\sigma'_a), h'_s[i \mapsto h'_s(i)(f \mapsto \delta')], h'_s, F) holds and
\begin{align*}
\text{collect-sym-locs}(\sigma'_i) - \text{collect-sym-locs}(t) = \{\delta'\}, \text{ we have} & \\
\text{heap}(\text{collect-sym-locs}(t), h'_a, F) \text{ holds.} & \\
\text{– otherwise: so the entry } (F(\delta'), h'_a(F(\delta'))) \text{ is not removed from } h_a \text{ by the definition of } h_a. & \\
\text{We are done because } \text{heap}(\text{collect-sym-locs}(\sigma'_i), h'_a[i \mapsto h'_a(i)[f_r \mapsto \delta']]), h'_a, F) \text{ holds.} & \\
\end{align*}

So we have proved \((h_s, \phi_s) \in \mathcal{H}_a[(h_a, \phi')] (\text{symbols}(t), \text{collect-sym-locs}(t), F)\). Thus \(\sigma_s \in \mathcal{ST}_a[t](F)\) holds and further, \(\sigma'_s \in S_s\). It remains to show that \(\sigma_s \rightarrow_S \sigma'_s\). There are two cases:

\begin{itemize}
    \item \(h_s(F(\delta'))\) is not defined: Since the \(\sigma'_s\) has only \(\delta'\) that is not in \(\sigma_o\), so \(h'_a(F(\delta'))\) is a fresh symbol. We can take the GETFIELD6-S rule and get \(\sigma_s \rightarrow_S \sigma'_s\).
    \item \(h_s(F(\delta'))\) is defined: By the wellmappedness of \(h_a, h_s(F(\delta'))\)(conc) is not defined. So we can take the GETFIELD4-S rule and get \(\sigma_s \rightarrow_S \sigma'_s\).
\end{itemize}

\end{proof}

\section*{Soundness and Completeness}

\textbf{Proposition 7} (Soundness). \textit{Given any symbolic trace } \(s_1 \rightarrow_S s_2 \rightarrow_S \cdots \rightarrow_S s_n\) \textit{with } \(s_1 \in I_S\), \textit{there is a corresponding lazier symbolic trace } \(a_1 \rightarrow_A a_2 \rightarrow_A \cdots \rightarrow_A a_n\) \textit{with } \(a_1 \in I_A\) \textit{such that } \(s_k \overset{R'}{\rightarrow} a_k\) \textit{for all } \(1 \leq k \leq n\).

\begin{proof}
We proceed by mathematical induction on \(n\) using Proposition 9.
\end{proof}

\textbf{Proposition 8} (Completeness). \textit{Given any lazier symbolic trace } \(a_1 \rightarrow_A a_2 \rightarrow_A \cdots \rightarrow_A a_n\) \textit{with } \(a_1 \in I_A\), \textit{there is a corresponding symbolic trace } \(s_1 \rightarrow_S s_2 \rightarrow_S \cdots \rightarrow_S s_n\) \textit{such that } \(s_k \overset{R}{\rightarrow} a_k\) \textit{for all } \(1 \leq k \leq n\) \textit{and } \(s_1 \in I_S\).

\begin{proof}
It is easy to show that there exists a trace in \(\mathcal{P}(S), s_1 \overset{*}{\rightarrow}_S s_2 \overset{*}{\rightarrow}_S \cdots \overset{*}{\rightarrow}_S s_n\) such that \(a_k \overset{R}{\rightarrow} s_k\) for all \(1 \leq k \leq n\) by mathematical induction on \(n\) using Proposition 6. Since \(S_n \neq \emptyset\), we can pick a \(s_n \in S_n\) and use the definition of \(\overset{*}{\rightarrow}_S\), then get the corresponding symbolic trace \(s_1 \rightarrow_S s_2 \rightarrow_S \cdots \rightarrow_S s_n\).
\end{proof}

\section*{D.3.3 Relative Soundness and Completeness of Symbolic Execution with Lazier\# Initialization}

Following the outline of Section D.3.2, we relate the lazier\# initialization symbolic execution in Section D.2.3 and lazier symbolic execution in Section D.2.2. First, we define a function \(\gamma_h\) which given a lazier\# symbolic state, it returns all the lazier symbolic states that have the same shape and only change symbolic references to either \texttt{null} or symbolic locations. Then we introduce binary relations between lazier symbolic states (power) and lazier\# symbolic state-spaces. Finally, we will prove the relative sound and completeness of lazier\# symbolic execution with regards to lazier symbolic execution intra-procedurally.
Definition of \( \gamma_b \)

Let us first introduce a definition: The set of all symbolic reference environments

\[ \Xi = \{ G \mid G : \text{SymRefs} \rightarrow (\text{SymLocs} \cup \{\text{null}\}) \}. \tag{D.2} \]

Then we define a function: \( \text{legal-env} : \Sigma_b \rightarrow \mathcal{P}(\Xi) \) as

\[ \text{legal-env}(\sigma_b) = \{ G \in \Xi \mid G(\text{collect-sym-refs}(\sigma_b)) \cap \text{collect-sym-locs}(\sigma_b) = \emptyset \land \forall \hat{\delta}_1 \neq \hat{\delta}_2 \in \text{collect-sym-refs}(\sigma_b). G(\hat{\delta}_1) = G(\hat{\delta}_2) \implies G(\hat{\delta}_1) = \text{null} \}, \]

where \( \text{collect-sym-refs} \) collects all the symbolic references in a state.

And \( ST_b : \Sigma_b \times \Xi \rightarrow \Sigma_a \) as

\[ ST_b[\sigma_b](G) = (\text{sub-fun}(g, G), pc, \text{sub-fun}(l, G), \text{sub-seq}(\omega, G), \text{sub-fun}2(h, G), \phi), \]

with binding \( \sigma_b = (g, pc, l, \omega, h, \phi) \).

The definition of \( \gamma_b : \Sigma_b \rightarrow \mathcal{P}(\Sigma_a) \) is

\[ \gamma_b(\sigma_b) = \bigcup_{G \in \text{legal-env}(\sigma_b)} ST_b[\sigma_b](G). \]

Properties of \( \gamma_b \)

**Lemma 5.** Let \( \sigma_b \in \Sigma_b \) and \( G \in \text{legal-env}(\sigma_b) \). Suppose \( \sigma_a = ST_b[\sigma_b](G) \) and \( \sigma_a = (g_a, pc, l_a, \omega_a, h_a, \phi_a) \). For any \( (\hat{\delta}, v) \in G \), if \( \sigma_b' = \text{init-sym-ref}(\sigma_b, \hat{\delta}, v) \), then \( (g_a, pc', l_a, \omega_a, h_a, \phi_a) \in ST_b[\sigma_b'](G) \).

**Proof.** Suppose \( \sigma_b = (g_b, pc, l_b, \omega_b, h_b, \phi) \). By the definition of \( \text{init-sym-ref} \), \( \sigma_b' = (\text{sub-fun}_1(g_b, \hat{\delta}, v), pc', \text{sub-fun}_1(l_b, \hat{\delta}, v), \text{sub-seq}(\omega_b, \hat{\delta}, v), \text{sub-fun}2(h_b, \hat{\delta}, v), \phi) \). Since \( \sigma_a = ST_b[\sigma_b](G) \), we have \( g_a = \text{sub-fun}(\text{sub-fun}_1(g_b, \hat{\delta}, v), G), l_a = \text{sub-fun}(\text{sub-fun}_1(l_b, \hat{\delta}, v), G), \omega_a = \text{sub-seq}(\text{sub-seq}(\omega_b, \hat{\delta}, v), G), \) and \( h_a = \text{sub-fun}2(\text{sub-fun}2(h_b, \hat{\delta}, v), G), \) by Lemma 1. We conclude that \( (g_a, pc', l_a, \omega_a, h_a, \phi_a) \in ST_b[\sigma_b'](G) \) holds. \( \square \)

**Lemma 6.** Let \( \sigma_b = (g_b, pc, l_b, \omega_b, \phi) \in \Sigma_b \) and \( G \in \text{legal-env}(\sigma_b) \). For any \( (\hat{\delta}, v) \in G \), if \( \sigma_b' = \text{init-sym-ref}(\sigma_b, \hat{\delta}, v) \) and \( (g_a, pc', l_a, \omega_a, h_a, \phi) = ST_b[\sigma_b'](G) \), then \( (g_a, pc, l_a, \omega_a, h_a, \phi) = ST_b[\sigma_b](G) \).

**Proof.** Proof is similar to Lemma 5. \( \square \)

**Improved Lazier Kripke Structure**

For any given method \( m \), we have a set of global variables \( \text{Globals} \) and local variables \( \text{Locals} \) (ordered from 0..\( n \)). We use Kripke structure \( B = (\Sigma_{\beta}, I_{\beta}, \longrightarrow_{\beta}, L_{\beta}) \) to model the state-space from the lazier# initialization symbolic executions. The components are defined as follows:

- states, \( \Sigma_{\beta} = \Sigma_b \cup (\text{Exception} \times \Sigma_b) \cup (\text{Error} \times \Sigma_b) \).
• initial states,

\[ I_B = \{ (g_b, pc_{\text{init}}, b, nil, h_b, \{\text{TRUE}\}) \mid \text{dom}(g_b) = \text{Globals} \land \text{dom}(h_b) = \text{Locals} \}, \]

and each local and global is initialized as follows: if it is primitive type, a primitive symbol is created; otherwise, it is initialized as a fresh symbolic reference. Furthermore, \( h_b \) is the empty heap.

• transition relation, \( b \xrightarrow{g} b' \iff b \xrightarrow{g} b_2, b_2 \xrightarrow{g} b_3, \ldots, b_n \xrightarrow{g} b' \) for some \( n \in \mathbb{N} \) with program counters of \( b, b_2, \ldots, b_n \) are the same and the program counter of \( b \) and \( b' \) are different and the path condition of \( b' \) is satisfiable.

• labels, we do not use this part and thus it is ignored.

Similar to \( \gamma_a \), function \( \gamma_b \) is trivially extended to \( \gamma_b^* : \Sigma_B \rightarrow \mathcal{P}(\Sigma_A) \)

\[
\gamma_b^*(b) = \begin{cases} 
\gamma_b(\sigma_b), & \text{if } b = \sigma_b \text{ for some } \sigma_b \in \Sigma_b; \\
\{ (\text{Exception}, \sigma_a) \mid \sigma_a \in \gamma_b(\sigma_b) \}, & \text{if } b = (\text{Exception}, \sigma_b) \text{ for some } \sigma_b \in \Sigma_b; \\
\{ (\text{Exception}, \sigma_a) \mid \sigma_a \in \gamma_b(\sigma_b) \}, & \text{if } b = (\text{ERROR}, \sigma_b) \text{ for some } \sigma_b \in \Sigma_b.
\end{cases}
\]

**Simulation Relations**

We introduce a relation \( R'' \) between lazier# symbolic states \( \Sigma_B \) and \( \Sigma_A \) as follows:

**Definition 6.** \( \sigma_a \stackrel{R''}{\rightarrow} \sigma_b \iff \sigma_a \in \gamma_b^*(\sigma_b) \).

Clearly, for all \( a_0 \in I_A \), there exists a \( b_0 \in I_B \) such that \( a_0 \stackrel{R''}{\rightarrow} b_0 \).

**Proposition 9.** \( A \triangleleft_{R''} B \).

**Proof.** It is sufficient to show that for all \( \sigma_a, \sigma_a' \in \Sigma_A, \sigma_b \in \Sigma_B \) if \( \sigma_a \xrightarrow{A} \sigma_a' \) and \( \sigma_a \stackrel{R''}{\rightarrow} \sigma_b \), then there exists \( \sigma_b' \in \Sigma_B \) such that \( \sigma_b \xrightarrow{B} \sigma_b' \) and \( \sigma_a \stackrel{R''}{\rightarrow} \sigma_b' \). We will proceed with the rule induction on \( \xrightarrow{A} \).

• Rule if_acmpeq: Suppose, WLOG, \( \sigma_a = (g_a, pc, l_a, \delta_t : \delta'_t : \omega_a, h_a, \phi') \). Then by the definition of \( \xrightarrow{A} \), the rule consists of three lazier symbolic transitions rules: IF_ACMPEQ3-A, IF_ACMPEQ2-A, and IF_ACMPEQ1-S or IF_ACMPEQ2-S. After taking IF_ACMPEQ3-A rule, we get an invisible state \( t_1 = (g_a', pc', l_a', i : \delta'_i : \omega_a', h_a', \phi''') \) for some \( i \in \text{Locs} \) and \( t_1 \in \text{init-sym-loc}(\sigma_a, \delta, i) \). Then after taking IF_ACMPEQ2-A rule, we get another invisible state \( t_2 = (g_a'', pc', l_a'', i : j : \omega_a'', h_a'', \phi''') \) for some \( j \in \text{Locs} \) and \( t_2 \in \text{init-sym-loc}(t_2, \delta', j) \). WLOG, suppose \( i \neq j \) (the \( i = j \) case is symmetric). Finally, we take IF_ACMPEQ1-S rule and get \( \sigma_a'' = (g_a'', \text{next}(pc), l_a'', \omega_a'', h_a'', \phi''') \). Suppose \( \sigma_a \stackrel{R''}{\rightarrow} \sigma_b' \). We need to show that there exists any \( \sigma_b'' \in \Sigma_B \) such that \( \sigma_b \xrightarrow{B} \sigma_b'' \). WLOG, suppose that \( \sigma_b'' = (g_b, pc, l_b, \hat{\delta} : \delta' : \omega_b, h_b, \phi) \). Since \( \sigma_a \stackrel{R''}{\rightarrow} \sigma_b' \), there exists \( G \in \text{legal-env}(\sigma_b') \) such that \( \sigma_a = ST_b[\sigma_b'](G) \). Clearly \( G(\hat{\delta}) = \delta \). We take rule IF_ACMPEQ3-B and get a state \( t'_0 = \text{init-sym-ref}(\sigma_b', \hat{\delta}, \delta) \) with stack \( \delta : \delta' :: \text{sub-seq}(\omega_b, \hat{\delta}, \delta) \). By Lemma 5, we get \( \sigma_a \stackrel{R''}{\rightarrow} t'_0 \). Then we take
IF_ACMPSEQ3-A, IF_ACMPSEQ2-A, and IF_ACMPSEQ1-S. We get \( t_1' = \text{init-sym-loc}(t_0', \delta, i) \) after rule IF_ACMPSEQ3-A, \( t_2' = \text{init-sym-loc}(t_1', \delta', j) \) after rule IF_ACMPSEQ2-A, and \( \sigma' \) after IF_ACMPSEQ1-S. Since all the rules do not involve any symbolic references, it clearly that \( t_1 \mathcal{R} t_1' \), and \( t_2 \mathcal{R} t_2' \), and finally \( \sigma' \mathcal{R} \sigma' \).

- **Rule getfield \( f_f \):** Suppose, WLOG, \( \tau \in \text{Types}_{\text{non-prim}} \) and \( \sigma_a = (g_a, pc, l_a, i :: \omega_a, h_a, \phi) \) and \( Y^{m,n} = h_a(i) \) and \( Y(f) \uparrow. \) Assume that rule GETFIELD2-A is taken. We get \( \sigma'_a = (g_a, \text{next}(pc), l_a, \sigma^m \langle k :: \omega_a, h_a \rangle[i \mapsto Y^{m,n}[f_t \mapsto \delta^m \langle -1 \rangle]], \phi) \) where \( \delta \) is fresh. Suppose \( \sigma_a \mathcal{R} \sigma_b \) and \( \sigma_a = ST_b[\sigma_a](G) \) for some \( G \in \text{legal-env}(\sigma_b). \) WLOG, assume \( \sigma_b = (g_b, pc, l_b, i :: \omega_b, h_b, \phi) \). Clearly we have \( X^{m,n} = h_b(i) \) for some \( X \) and \( X(f) \uparrow. \) After rule GETFIELD2-B, we get \( \sigma'_b = (g_b, \text{next}(pc), l_b, \tilde{\delta}^m \langle k :: \omega', h_b \rangle[i \mapsto Y^{m,n}[f_t \mapsto \delta^m \langle -1 \rangle]], \phi) \) and \( \tilde{\delta} \) is fresh in \( \sigma_b. \) It is easy to see that \( G[\tilde{\delta} \mapsto \delta] \in \text{legal-env}(\sigma'_b) \). Thus we have \( \sigma'_a = ST_b[\sigma'_a](G[\tilde{\delta} \mapsto \delta]) \), that is, \( \sigma_a \mathcal{R} \sigma' \).

Next we define a relation.

**Definition 7.** \( \mathcal{R} \subseteq \Sigma_b \times \mathcal{P}(\Sigma_a), \) as follows:

\[
\sigma_b \mathcal{R} S_a \iff \gamma_b(\sigma_b) = S_a
\]

Clearly, \( \mathcal{R} \) is left total. Since \( \mathcal{R} \) is right total, then for all \( \sigma_b, \) if \( \sigma_b \mathcal{R} S_a \), then \( S_a \neq \emptyset. \) Furthermore, for any \( \sigma_b \in I_b \) and \( \sigma_b \mathcal{R} S_a, \) it is clear that \( S_a \subseteq I_a \) by the definition of \( \gamma_b \) function.

**Proposition 10.** \( \mathcal{B} \models_{\mathcal{R}^\ast} \mathcal{P}(\mathcal{A}). \)

**Proof.** It is sufficient to show that for all \( \sigma_b \in \Sigma_b, S_a \in \mathcal{P}(\Sigma_a) \) if \( \sigma_b \rightarrow_{\mathcal{B}} \sigma'_b \) and \( \sigma_b \mathcal{R} S_a \) and \( \sigma'_b \mathcal{R} S'_a \), then \( S_a \rightarrow_{\mathcal{A}} S'_a. \)

We will prove by rule induction on transitions, \( \rightarrow_{\mathcal{B}}. \)

- **Rule if_acmpseq:** Suppose, WLOG, \( \sigma_b = (g_b, pc, l_b, \delta_1 :: \delta_2 :: \omega_b, h_b, \phi). \) Then by the definition of \( \rightarrow_{\mathcal{B}}, \) the rule consists of five transitions rules: IF_ACMPSEQ3-B, IF_ACMPSEQ2-B, IF_ACMPSEQ3-A, IF_ACMPSEQ2-A, and IF_ACMPSEQ1-S or IF_ACMPSEQ2-S. After taking the IF_ACMPSEQ3-B rule, we get an invisible state \( t_1 = \text{init-sym-ref}(\sigma_b, \delta, 1) \) and then IF_ACMPSEQ2-B rule, we get \( t_2 = \text{init-sym-ref}(t_1, \delta_2, 2) \). Then by IF_ACMPSEQ3-A rule, we get to \( t_3 \) and by IF_ACMPSEQ2-A rule, we arrive at \( t_4 \) where \( \delta_1 \) and \( \delta_2 \) are fresh. WLOG, suppose we take IF_ACMPSEQ1-S rule and get \( \sigma'_b = (\text{sub-fun}_1(\text{sub-fun}_1(g_b, \delta_1, \delta_1), \delta_2, \delta_2), \text{next}(pc), \text{sub-fun}_2(\text{sub-fun}_2(h_b, \delta_1, \delta_1), \delta_2, \delta_2), \text{sub-seq}_4(\text{sub-seq}_4(h_b, \delta_1, \delta_1), \delta_2, \delta_2), \phi). \) Suppose \( \sigma_b \mathcal{R} S_a \) and \( \sigma'_b \mathcal{R} S'_a. \) We need to show that \( S_a \rightarrow_{\mathcal{A}} S'_a \), that is, for any \( \sigma_a \in S'_a \), there exists some \( \sigma_a \in S_a \) such that \( \sigma_a \rightarrow_{\mathcal{A}} \sigma_a \). Suppose \( \sigma_a \in S'_a \) that is, \( \sigma'_a \in \gamma_a(\sigma'_b). \) Then \( \sigma_a \) must be in the form of \( (g'_a, \text{next}(pc), l'_a, \omega'_a, h'_a, \phi) \) for some \( G \) and \( \sigma' \in ST_b[\sigma'_a](G). \) Define \( G' = G[\delta_1 \mapsto \delta_1][\delta_2 \mapsto \delta_2]. \) Clearly \( G' \in \text{legal-env}(\sigma_b). \) Define \( \sigma_a = ST_b[\sigma_a](G'). \) We need to show
that $\sigma_a \rightarrow_\mathcal{A} \sigma'_a$. After applying Lemma 5 twice, we get $\sigma_a = ST_b[t_2](G')$. Since $\sigma_a$ only differs from $t_2$ by some symbolic references which are not operands of the instruction, $\sigma_a$ can takes exactly the same rules and get to $\sigma'_a$. We conclude that $\sigma_a \rightarrow_\mathcal{A} \sigma'_a$.

- Rule `getfield` $f_i$: Suppose, WLOG, $\sigma_b = (g_b, pc, I_b, \delta, \phi)$ and $\tau \in $ Types$_{record}$. By the definition of $\rightarrow_\mathcal{B}$, the transition multiple lazier# rules. The first one is GETFIELD1-B. WLOG, assume that the invisible state after GETFIELD1-B is $t_1 = (\text{sub-fun1}(g_b, \delta), pc, \text{sub-fun1}(l_b, \delta))$, and $\text{sub-seq1}(\omega_b, \delta, \phi)$ for some fresh $\delta$. Then rule GETFIELD1-A is taken and get an invisible state $t_2 = (g_2, pc, l_2, \omega_2, \delta, \phi^2) = \text{init-sym-loc}(t_1, \delta, i)$ for some $i \in $ Locs. WLOG, assume that $f$ field is undefined in $h_2(i)$. We take the GETFIELD2-B rule and get $\sigma'_b = (g_2, next(pc), l_2, \delta', \omega_2, h_2[i \mapsto h_2(i)[f \mapsto \delta']], \phi_2)$, where $\delta'$ is fresh in $t_2$.

Suppose $\sigma_b \mathcal{R}_b' S_a$ and $\sigma'_b \mathcal{R}_b' S'_a$$. We need to show that $S_a \rightarrow_\mathcal{A} S'_a$, that is, for any $\sigma'_a \in S'_a$, there exists some $\sigma_a \in S_a$ such that $\sigma_a \rightarrow_\mathcal{A} \sigma'_a$. Suppose $\sigma'_a \in S'_a$, that is, $\sigma'_a \in \mathcal{G}(\sigma'_b)$. Then $\sigma_a$ must be in the form of $(g'_a, next(pc), l'_a, \delta', \omega'_a, h'_a, \phi)$ for some $G$ such that $G(\delta') = \delta$ and $\sigma'_a \in ST_b[\sigma_b][G](G')$. Define $G' = G[\delta \mapsto \delta]$. Clearly $G' \in $ legal-env($\sigma_b$). Let $\sigma_a = ST_b[\sigma_b][G'](G')$. Using Lemma 5, we get $\sigma_a = ST_b[t_1](G')$. Since $t_1$ only has more symbolic references than $\sigma_a$, rule GETFIELD1-A is applicable and get $s'_2$. Since $\delta'$ is fresh in $t_2$ and $G(\delta') = \delta'$, $\delta'$ is fresh in $s'_2$. Therefore, we can apply GETFIELD2-A and get $\sigma'_a$. We conclude that $\sigma_a \rightarrow_\mathcal{A} \sigma'_a$.

\[
\]

**Soundness and Completeness**

**Proposition 11** (Soundness). *Given any lazier symbolic trace $a_1 \rightarrow_\mathcal{A} a_2 \rightarrow_\mathcal{A} \cdots \rightarrow_\mathcal{A} a_n$ with $a_1 \in I_\mathcal{A},$ there is a corresponding lazier# symbolic trace $b_1 \rightarrow_\mathcal{B} b_2 \rightarrow_\mathcal{B} \cdots \rightarrow_\mathcal{B} b_n$ with $b_1 \in I_\mathcal{B}$ such that $a_k \mathcal{R} b_k$ for all $1 \leq k \leq n$.\n
**Proof.** We proceed by mathematical induction on $n$ using Proposition 9.

**Proposition 12** (Completeness). *Given any lazier# symbolic trace $b_1 \rightarrow_\mathcal{B} b_2 \rightarrow_\mathcal{B} \cdots \rightarrow_\mathcal{B} b_n$ with $b_1 \in I_\mathcal{B},$ there is a corresponding symbolic trace $a_1 \rightarrow_\mathcal{A} a_2 \rightarrow_\mathcal{A} \cdots \rightarrow_\mathcal{A} a_n$ such that $a_k \mathcal{R} b_k$ for all $1 \leq k \leq n$ and $a_1 \in I_\mathcal{A}$.\n
**Proof.** It is easy to show that there exists a trace in $\mathcal{P}(\mathcal{A})$, $S_1 \rightarrow_\mathcal{A} S_2 \rightarrow_\mathcal{A} \cdots \rightarrow_\mathcal{A} S_n$ such that $b_k \mathcal{R}_b' S_k$ for all $1 \leq k \leq n$ by mathematical induction on $n$ using Proposition 10. Since $S_n \neq \emptyset$, we can pick a $a_n \in S_n$ and use the definition of $\rightarrow_\mathcal{A}$, then get the corresponding lazier symbolic trace $a_1 \rightarrow_\mathcal{A} a_2 \rightarrow_\mathcal{A} \cdots \rightarrow_\mathcal{A} a_n$.\n
48
Appendix E

Formalization of KUnit

E.1 Test Input Generation Algorithms and Proofs

In this section, we show that the test input generation algorithms and the proof of path coverage. Similar to the sound and completeness proofs, we will take three steps: first step is for symbolic execution with lazy initialization; second step is for symbolic execution with lazier initialization; last step is for symbolic execution with laziest initialization. Each step takes a similar procedure: first we will define inverse rules for symbolic operational semantics rules, the backtracking rules; then a default concretization function, finally the input generation algorithm and the path coverage proof.

E.1.1 Backtracking Rules for Symbolic Execution with Lazy Initialization

Backtracking rules are the inverse rules of transition rules $\rightarrow_S$ (suppose we take one path at a time). Then after any transition $\sigma_1 \Rightarrow_S \sigma_2$, we can apply the corresponding backtracking rule which takes $\sigma_2$ and returns $\sigma_1$. The simple way to achieve backtracking functionality is to save old state $\sigma_1$. But this is inefficient in practice, since there is usually few changes from $\sigma_1$ to $\sigma_2$. Instead, only the changes ($\delta$) are stored in practice. Since we are doing theory, we just take the simple approach and define backtracking rules as $(\Sigma_1 \times \Sigma_2) \Rightarrow_S^{-1} \Sigma_1$. Specifically, assume $\sigma_1 \Rightarrow_S \sigma_2$ by rule FOO-S, then its backtracking rule FOO-S-BACK is $\langle \sigma_1, \sigma_2 \rangle \Rightarrow_S^{-1} \sigma_1$.

In order to generate concrete test inputs, we modify two kinds of backtracking rules:

1. rules that involve lazy initialization, such as `getfield` and `iaload`. The lazy initialized fields are kept in the return state.

2. rules that add new constraints into path condition. The added constraints are kept in the return state.

Formally, each backtracking rule is in the format of

$$\frac{\text{premises}}{\langle \sigma_1, \sigma_2 \rangle \Rightarrow_S^{-1} \sigma_3}.$$
where $\sigma_1, \sigma_2, \sigma_3 \in \Sigma_S$.

We use the bindings, $\sigma = (g, pc, l, \omega, h, \phi)$ and $\sigma^' = (g^', pc^', l^', \omega^', h^', \phi^')$, for all the backtracking rules.

\[
\text{IADD-S-BACK} \quad \begin{aligned}
\text{code}(pc) &= \text{iadd} \\
\omega &= v_1 :: v_2 :: \omega_1 \\
\omega^' &= v^' :: \omega_2 \\
\text{pc}^' &= \text{next}(pc)
\end{aligned}
\]

\[
\langle \sigma, \sigma^' \rangle \Rightarrow_{S}^1 (g^', pc^', l^', v_1 :: v_2 :: \omega_2, h^', \phi')
\]

\[
\text{IF_ICMPLT-S-T-BACK} \quad \begin{aligned}
\omega &= v_1 :: v_2 :: \omega_1 \\
v_2 &< v_1 \in \phi^' \\
\text{pc}^' &= \text{next}(pc)
\end{aligned}
\]

\[
\langle \sigma, \sigma^' \rangle \Rightarrow_{S}^1 (g^', pc^', l^', v_1 :: v_2 :: \omega_2, h^', \phi')
\]

\[
\text{IF_ICMPLT-S-F-BACK} \quad \begin{aligned}
\omega &= v_1 :: v_2 :: \omega_1 \\
v_2 &\not< v_1 \in \phi^' \\
\text{pc}^' &= \text{next}(pc)
\end{aligned}
\]

\[
\langle \sigma, \sigma^' \rangle \Rightarrow_{S}^1 (g^', pc^', l^', v_1 :: v_2 :: \omega_2, h^', \phi')
\]

\[
\text{NEW-S-BACK} \quad \begin{aligned}
\omega &= v_1 :: v_2 :: \omega_1 \\
\text{pc}^' &= \text{next}(pc)
\end{aligned}
\]

\[
\langle \sigma, \sigma^' \rangle \Rightarrow_{S}^1 (g^', pc^', l^', v_1 :: v_2 :: \omega_2, h^', \phi')
\]

\[
\text{GETFIELD-S-BACK} \quad \begin{aligned}
\omega &= i :: \omega_1 \\
\text{pc}^' &= \text{next}(pc)
\end{aligned}
\]

\[
\langle \sigma, \sigma^' \rangle \Rightarrow_{S}^1 (g^', pc^', l^', i :: \omega_2, h^', \phi')
\]

\[
\text{GETFIELD-S-Ex-BACK} \quad \begin{aligned}
\omega &= v :: i :: \omega_1 \\
\text{pc}^' &= \text{next}(pc)
\end{aligned}
\]

\[
\langle \sigma, \sigma^' \rangle \Rightarrow_{S}^1 (g^', pc^', l^', v :: i :: \omega, h^'[i \mapsto h^'(i)[f \mapsto h(i)(f)]], \phi')
\]

\[
\text{PUTFIELD-S-BACK1} \quad \begin{aligned}
\omega &= v :: i :: \omega_1 \\
\text{pc}^' &= \text{next}(pc)
\end{aligned}
\]

\[
\langle \sigma, \sigma^' \rangle \Rightarrow_{S}^1 (g^', pc^', l^', v :: i :: \omega, h^'[i \mapsto h^'(i) \backslash \{(f, h(i)(f))\}], \phi')
\]

\[
\text{PUTFIELD-S-BACK2} \quad \begin{aligned}
\omega &= v :: i :: \omega_1 \\
\text{pc}^' &= \text{next}(pc)
\end{aligned}
\]

\[
\langle \sigma, \sigma^' \rangle \Rightarrow_{S}^1 (g^', pc^', l^', v :: i :: \omega, h^'[i \mapsto h^'(i) \backslash \{(f, h(i)(f))\}], \phi')
\]

\[
\text{PUTFIELD-S-Ex-BACK} \quad \begin{aligned}
\sigma^' &= \text{NullPointerException} \\
\omega &= v :: v :: \text{null} :: \omega_1 \\
\text{pc}^' &= \text{next}(pc)
\end{aligned}
\]

\[
\langle \sigma, \sigma^' \rangle \Rightarrow_{S}^1 (g^', pc^', l^', v :: v :: \text{null} :: \omega, h^', \phi')
\]

**Figure E.1:** Backtracking Rules for Symbolic Execution with Lazy Initialization

Figure E.1 presents the backtracking rules:

- **rule IADD-S-BACK** is the corresponding backtracking rule for IADD-S. The IADD-S-BACK replaces the top element of the stack of second state with the top two elements of the stack of the first state and returns the second state. It can be seen as popping the result of addition and pushing the two operands to stack.

- **rules IF_ICMPLT-S-T-BACK and IF_ICMPLT-S-F-BACK** are the corresponding backtracking rules for IF_ICMLET-S. There are two backtracking rules because rule IF_ICMLET-S has two
possible resulting states depending on whether the True branch or the False branch is taken. So when the top element less than the the element below it in the stack of first state (condition is True), IF_ICMPLT-S-T-BACK applies and returns the second state with two operands in the top of the first state being pushed onto the stack of the second state. Otherwise, rule IF_ICMPLT-S-F-BACK applies.

- rule NEW-S-BACK is the corresponding backtracking rule for NEW-S. The backtracking rule removes the newly created heap entry indexed by the top of the stack of the second state.

- rule GETFIELD-S-BACK is the corresponding backtracking rule for rules GETFIELD[1..6]-S. This is because we preserve the results of lazy initialization.

- rule GETFILEDS-Ex-BACK is the corresponding backtracking rule for GETFIELD7-S which throws a NullPointerException.

- rules PUTFIELD-S-BACK1 and PUTFIELD-S-BACK2 are the corresponding backtracking rules for the rule PUTFIELD1-S. Both backtracking rules undo the written value to a field by rule PUTFIELD1-S. When the field is defined in the first state, PUTFIELD-S-BACK1 applies; otherwise, PUTFIELD-S-BACK2 applies.

- rule PUTFIELD-S-Ex-BACK is the corresponding backtracking rule for PUTFIELD2-S.

**Lemma 7.** For any \( \sigma_1 \rightarrow_S \sigma_2 \), obviously we can apply the corresponding backtracking rule \( \langle \sigma_1, \sigma_2 \rangle \Rightarrow_S^{-1} \sigma_3 \) for some state \( \sigma_3 \). Then \( \sigma_3 \rightarrow_S \sigma_2 \) and the transition rule is not one of the lazy initialization rule (GETFIELD3-S, GETFIELD4-S, GETFIELD5-S, GETFIELD6-S, etc.).

**Proof.** We will proceed with rule induction on operational semantics rules of symbolic execution, \( \rightarrow_S \) (we only present representative rules):

- Rule IADD-S, assume that \( \sigma_1 = (g, pc, l, v_1 :: v_2 :: \omega, h, \phi) \). Then \( \sigma_2 = (g, next(pc), l, Z :: \omega, h, \phi \cup \{Z = v_1 + v_2\}) \) where \( Z \) is fresh. After applying rule IADD-S-BACK, we get \( \sigma_3 = (g, pc, l, v_1 :: v_2 :: \omega, h, \phi \cup \{Z = v_1 + v_2\}) \). Clearly, \( \sigma_3 \rightarrow_S \sigma_2 \) by rule IADD-S. For any \( E, T \) that satisfy \( \phi \cup \{Z = v_1 + v_2\} \), we have \( V_{s[v_1]}(T,E) + V_{s[v_2]}(T,E) = V_{s[Z]}(T,E) \). The other components of \( \sigma_2 \) and \( \sigma_3 \) are the same.

- Rule IF_ICMPLT-S, let \( \sigma_1 = (g, pc, l, v_1 :: v_2 :: \omega, h, \phi) \). WLOG, assume that the False branch is taken. Then \( \sigma_2 = (g, next(pc), l, Z :: \omega, h, \phi \cup \{v_2 \geq v_1\}) \). After applying the IF_ICMPLT-F-BACK rule, we get \( \sigma_3 = (g, pc, l, v_1 :: v_2 :: \omega, h, \phi \cup \{v_2 \geq v_1\}) \). Clearly, \( \sigma_3 \rightarrow_S \sigma_2 \) by rule IF_ICMPLT-S (the True branch is infeasible).

- Rule NEW-S, let \( \sigma_1 = (g, pc, l, \omega, h, \phi) \). Then \( \sigma_2 = (g, next(pc), l, i :: \omega, h[i \mapsto \text{new-obj}({\text{symbols}(\sigma_1), \tau})], \phi) \) where \( i \) is fresh in \( h \). After applying the rule NEW-S-BACK, we get \( \sigma_3 = (g, pc, l, \omega, h, \phi) = \sigma_1 \). Clearly, \( \sigma_3 \rightarrow_S \sigma_2 \).

- Rule GETFIELD3-S, let \( \sigma_1 = (g, pc, l, i :: \omega, h, \phi) \). Then \( f \) is not defined in \( h(i) \) and \( \sigma_2 = (g, next(pc), l, \text{null} :: \omega, h', \phi) \) where \( h' = h[i \mapsto h(i)[f, \text{null}]] \). After applying rule GETFIELD-S-BACK, we get \( \sigma_3 = (g, pc, l, i :: \omega, h', \phi) \). Clearly \( \sigma_3 \rightarrow_S \sigma_2 \) by rule GETFIELD1-S.

51
• Rule GETFIELD6-S, \( \sigma_1 = (g_s, pc, l_s, i :: \omega_s, h_s, \phi) \). Then \( f \) is not defined in \( h_s(i) \) and \( \sigma_2 = (g_s, \text{next}(pc), l_s, j :: \omega_s, h'_s, \phi') \) where \( h_s(i) = Y^{m,n} \), \( h'_s = h_s[i \mapsto Y^{m,n}[f \mapsto j][j \mapsto Z_r]] \), \( \phi' = \phi \cup \{ \tau' <: \tau \} \) for \( Z_r = \text{new-sym}(\text{symbols}(\sigma_3), m - 1, k) \) and \( j \notin \text{dom} h_s \). After applying rule GETFIELD-S-BACK, we get \( \sigma_3 = (g_s, pc, l_s, i :: \omega_s, h'_s, \phi') \). Clearly \( \sigma_3 \rightarrow_S \sigma_2 \) by rule GETFIELD1-S.

We can have an even stronger property of the backtracking rules:

**Lemma 8.** Given backtracking rule \( (\sigma_1, \sigma_2) \Rightarrow_S^1 \sigma_3 \) for some state \( \sigma_1, \sigma_2, \sigma_3 \). Then \( \sigma_3 \rightarrow_S \sigma_2 \) and the transition rule is not one of the lazy initialization rule (GETFIELD3-S, GETFIELD4-S, GETFIELD5-S, GETFIELD6-S, etc.).

**Proof.** We proceed with rule induction of \( \Rightarrow_S^1 \).

### E.1.2 Default Concretization Function for Symbolic Execution with Lazy Initialization

We define a default concretization function \( \text{default-concr} : \Gamma \times \text{Env} \times \Sigma_S \rightarrow \Sigma_C \). The goal of \( \text{default-concr} \) is to generate a default concrete state for a symbolic state given \( E \) and \( T \) satisfying the path condition of the symbolic state. Intuitively, the concrete state is generated by substituting symbolic values with \( E \) and all the undefined fields are initialized with default values.

Then we introduce some semantic functions to facilitate the definition of \( \text{default-concr} \). \( O_d \) maps a symbolic object/array into a concrete object or array. \( \mathcal{H}_d \) maps a symbolic heap into a concrete heap by mapping each symbol in its range to a concrete object using \( O_d \).

\[
\begin{align*}
O_d : & \text{Symbols}_{\text{non-prim}} \rightarrow ((\Gamma \times \text{Env}) \rightarrow \text{Symbols}_{\text{non-prim}}); \\
\mathcal{H}_d : & \text{Heaps}_s \rightarrow ((\Gamma \times \text{Env}) \rightarrow \text{Heaps}_c).
\end{align*}
\]

Here are the definitions (\( e \) is the identity permutation of locations):

- the \( O_d \) function:

\[
O_d[X](T, E) = X'_t,
\]

where \( t' = \text{sub}(t, T) \) and if \( t' \in \text{Types}_{\text{record}} \): for all \( f_{t''} \in \text{fields}(t') \),

\[
X'(f_{t''}) = \begin{cases} 
\mathcal{V}_s[X(f)](E, e) & \text{if } X(f) \downarrow \\
\text{default}(t'') & \text{otherwise}
\end{cases}
\]

if \( t' \in \text{Types}_{\text{array}} \): \( X'(\text{LEN}) = \mathcal{V}_s[X(\text{LEN})](E, e) \land \forall t \in \text{acc-idx}(X), X'(\mathcal{V}_s[t](E, e)) = \mathcal{V}_s[X(t)](E, e) \)

and \( \forall (0 \leq m < X'(\text{LEN}) \land m \notin \{ \mathcal{V}_s[t](E, e) \mid t \in \text{acc-idx}(X) \}) \).

\[
X'(m) = \begin{cases} 
X(\text{DEF}) & \text{if } X(\text{DEF}) \downarrow \\
\text{default}(t'') & \text{otherwise}
\end{cases}
\]

where \( t'' \) is the element type of \( t' \).
• the $\mathcal{H}_d$ function:

$$\mathcal{H}_d[h_s](T, E) = h_c$$

where

$$\text{dom } h_s = \text{dom } h_c \land \forall (i, X) \in h_s, (i, O_d[X](T, E)) \in h_c.$$  

Finally the default-conc function:

$$dc(T, E, (g, pc, l, \omega, h, \phi)) = (\text{sub-fun}(\text{sub-fun}(g, E), e), pc, \text{sub-fun}(\text{sub-fun}(l, E), e), \text{sub-seq}(\text{sub-seq}(\omega, E), e), \mathcal{H}_d[h](T, E), \text{TRUE}).$$

$$\text{default-conc}(T, E, s) = \begin{cases} dc(T, E, \sigma_s) & \text{if } s = \sigma_s; \\
(\text{Exception}, dc(T, E, \sigma_s)) & \text{if } s = (\text{Exception}, \sigma_s); \\
(\text{Error}, dc(T, E, \sigma_s)) & \text{if } s = (\text{Error}, \sigma_s). \end{cases}$$

It is easy to show that for all $E, T$ that satisfy $\phi$ of $\sigma_s$, $\text{default-conc}(T, E, \sigma_s) \in \gamma_s(\sigma_s)$.

**Lemma 9.** Suppose $\sigma_1 \rightarrow_S \sigma_2$ and the transition rule is not one of the lazy initialization rules. For any $E, T$ satisfy $\phi_2$ of $\sigma_2$, $\text{default-conc}(T, E, \sigma_1) \rightarrow_C \text{default-conc}(T, E, \sigma_2)$

**Proof.** Proceed by rule induction on $\rightarrow_S$. \qed

### E.1.3 Input Generation Algorithm and Proof

Given any trace $s_1 \rightarrow_S s_2 \rightarrow_S \cdots s_n \rightarrow_S s_{n+1}$. Define a sequence of states $s_i'$ for $1 \leq i \leq n + 1$ as

$$s_{n+1}' = s_{n+1},$$

$$\langle s_n, s_{n+1}' \rangle \Rightarrow^1_S s_n',$$

$$\vdots$$

$$\langle s_2, s_2' \rangle \Rightarrow^1_S s_2',$$

$$\langle s_1, s_1' \rangle \Rightarrow^1_S s_1'.$$

The applicability of backtracking rules is shown by Lemma 12. Then we apply the default concretization function for $s_i'$, where $1 \leq i \leq n + 1$ for any $E, T \models \phi_{n+1}$ with $\phi_{n+1}$ is the path condition of $s_{n+1}$. And we get $c_1 = \text{default-conc}(T, E, s_1'), \ldots, c_{n+1} = \text{default-conc}(T, E, s_{n+1}')$.

**Proposition 13.** $c_1 \rightarrow_C c_2 \cdots \rightarrow_C c_{n+1}$ and $c_i \in \gamma_s(s_i)$ for all $1 \leq i \leq n + 1$;

To prove this main theorem, we need one additional definition and some lemma.

**Definition 8.** Relation $\prec_S: \Sigma_S \times \Sigma_S$ as $s_1 \prec_S s_2$ if and only if $s_1$ is similar to $s_2$ except that there are some fields of symbolic objects/arrays in $s_1$ are defined but not in $s_2$.
Precisely, \( s_1 <_S s_2 \) for \( s_1 = (g_1, pc_1, l_1, \omega_1, h_1, \phi_1) \) and \( s_2 = (g_2, pc_2, l_2, \omega_2, h_2, \phi_2) \) if and only if
\[
g_1 = g_2 \land pc_1 = pc_2 \land l_1 = l_2 \land \omega_1 = \omega_2 \land \text{refineheap}(pc_1, h_1, h_2) \land \phi_1 \succeq \phi_2,
\]
where \( \text{refineheap}(pc_1, h_1, h_2) \) if and only if following conditions hold

1. \( \text{dom } h_2 \subseteq \text{dom } h_1 \);
2. for all \( i \in \text{dom } h_2, (\forall t. h_2(i)(t) \downarrow \implies (h_1(i)(t) \downarrow \land h_2(i)(t) = h_1(i)(t)) \land h_2(i)(\text{conc}) \downarrow \iff h_1(i)(\text{conc}) \downarrow ) \);
3. \( \text{code}(pc_1) = \text{getfield} f \implies \omega_2 = i :: \omega'_2 \land h_1(i)(f) \downarrow ^1; \)
4. for all \( i \in \text{dom } h_1 \setminus \text{dom } h_2, h_1(i)(\text{conc}) \uparrow \land \forall t \in \text{acc-idx}(h_1(i)) . \text{non-concrete}(h_1, i, t) ; \)
5. for all \( i \in \text{dom } h_2, \text{for all } t. (h_1(i)(t) \downarrow \land h_2(i)(t) \uparrow ) \implies \text{non-concrete}(h_1, i, t), \)

where
\[
\text{non-concrete}(h_1, i, t) \equiv h_1(i)(t) = j \text{ for some } j \in \text{Locs} \text{ implies } h_1(j)(\text{conc}) \uparrow .
\]

Clearly, \( \gamma_s(s_1) \subseteq \gamma_s(s_2) \).

**Lemma 10.** Suppose \( \sigma_1 \rightarrow_S \sigma_2 \). Let \( \sigma_3 \) be the outcome of backtracking from \( \sigma_2 \), that is, \( \langle \sigma_1, \sigma_2 \rangle \Rightarrow_S^{-1} \sigma_3 \). Then \( \sigma_3 <_S \sigma_1 \).

**Proof.** This can be shown easily by rule induction on \( \rightarrow_S \). \( \square \)

**Lemma 11.** Suppose we have \( \sigma_4 <_S \sigma_2 \) and \( \sigma_1 \rightarrow_S \sigma_2 \). Let \( \sigma_3 \) be the outcome of backtracking from \( \sigma_4 \), that is, \( \langle \sigma_1, \sigma_4 \rangle \Rightarrow_S^{-1} \sigma_3 \). Then \( \sigma_3 <_S \sigma_1 \).

**Proof.** We will prove by rule induction of symbolic semantics rules: \( \rightarrow_S \). We will use the default bindings: \( \sigma_1 = (g_1, pc_1, l_1, \omega_1, h_1, \phi_1), \sigma_2 = (g_2, pc_2, l_2, \omega_2, h_2, \phi_2), \sigma_3 = (g_3, pc_3, l_3, \omega_3, h_3, \phi_3), \) and \( \sigma_4 = (g_4, pc_4, l_4, \omega_4, h_4, \phi_4) \).

- Rule IADD-S: Since \( \sigma_1 \rightarrow_S \sigma_2 \) by rule IADD-S, we know \( \omega_1 = v_1 :: v_2 :: \omega'_1, \text{code}(pc_1) = \text{iadd}, \omega_2 = v' :: \omega'_2, \text{pc}_2 = \text{next}(pc_1), g_1 = g_2, l_1 = l_2, h_1 = h_2, \) and \( \phi_1 \subseteq \phi_2 \). Since \( \sigma_4 <_S \sigma_2 \), we have \( g_4 = g_2, pc_4 = pc_2, l_4 = l_2, \omega_2 = \omega_4, \text{refineheap}(pc_4, h_4, h_2), \phi_2 \subseteq \phi_4 \). By rule IADD-S-BACK, we get \( \sigma_3 = (g_4, pc_1, l_4, v_1 :: v_2 :: \omega'_4, h_4, \phi_4) \), where \( \omega_4 = v_4 :: \omega'_4 \). Then we get \( g_3 = g_4 = g_1, pc_3 = pc_1, l_3 = l_4 = l_2 = l_1, \text{refineheap}(pc_3, h_3, h_1), \phi_1 \subseteq \phi_2 \subseteq \phi_4 = \phi_3 \). Thus we conclude that \( \sigma_3 <_S \sigma_1 \).

- Rule IF,ICMPLT-S: WLOG, assume that false branch is taken. Similar to the IADD-S-BACK case.

---

\(^1\)Similar properties should hold for iaload and aaload. For the purpose of simpler presentation, it is not listed.
• Rule NEW-S: The proof of \( g_3 = g_1, \text{pc}_3 = \text{pc}_1, \omega_3 = \omega_1, \) and \( \phi_1 \subseteq \phi_3 \) is similar to the IADD-S case. The interesting part is to show that \( \text{refineheap}(\text{pc}_3, h_3, h_1) \). We know \( \text{refineheap}(\text{pc}_4, h_4, h_2), \ h_3 = h_4 \setminus \{(i', h_4(i'))\} \) and \( h_1 = h_2 \setminus \{(i', h_2(i'))\} \). First of all, we need to show that \( \sigma_3 \) is a well-defined symbolic state, that is, \( i' \) is not referred in \( \sigma_3 \). Since \( g_3 = g_4 = g_1 \) and \( i' \) is fresh in \( \sigma_1 \), \( g_3 \) does not refer to \( i' \). Similarly, \( l_3 \) and \( \omega_3 \) do not refer to \( i' \). Since \( \text{refineheap}(\text{pc}_4, h_4, h_2), h_4 \setminus \{(i', h_4(i'))\} \) does not refer to \( i' \) by Properties 4 and 5 of \( \text{refineheap} \). Thus \( \sigma_3 \) is well-defined symbolic state. Then it is clear that \( \text{refineheap}(\text{pc}_3, h_3, h_1) \). Thus we conclude \( \sigma_3 <_S \sigma_1 \).

• Rule GETFIELD2-S: Similar to the Rule NEW-S case, the interesting part is to show that \( \text{refineheap}(\text{pc}_3, h_3, h_1) \). Suppose \( \omega_1 = i :: \omega'_1 \) and \( \text{code}(\text{pc}_1) = \text{getfield } f \). Then \( h_2(i)(f) \downarrow \). By \( \sigma_4 <_S \sigma_2, h_4(i)(f) \downarrow \). Using the GETFIELD-S-BACK rule, we get \( h_4 = h_3 \), in particular, \( h_3(i)(f) \downarrow \). Thus we get \( \text{refineheap}(\text{pc}_3, h_3, h_1) \). The other GETFIELDx-S rules are similar.

• Rule PUTFIELD1-S: Again, the interesting part is to show \( \text{refineheap}(\text{pc}_3, h_3, h_1) \). Suppose \( \text{code}(\text{pc}_1) = \text{putfield } f, \omega_1 = v :: i :: \omega'_1, \) and \( h_1(f) \uparrow \). From PUTFIELD1-S rule, we know \( h_2(i)(f) = v \). Since \( \sigma_4 <_S \sigma_2, h_4(i)(f) = v \) and \( v \) can not point to \( \text{dom} h_4 \setminus \text{dom} h_2 \). Then we arrive at \( \text{refineheap}(\text{pc}_3, h_3, h_1) \).

\[ \square \]

**Lemma 12.** \( s'_i <_S s_i \) for all \( 1 \leq i \leq n + 1 \).

**Proof.** Since \( s_{n+1} = s'_{n+1} \), we get \( s'_n <_S s_n \) by Lemma 10. Then by induction: going backward with Lemma 11 for inductive step.

\[ \square \]

**Lemma 13.** \( s'_1 \rightarrow_S s'_2 \rightarrow_S \cdots \rightarrow_S s'_{n+1} \).

**Proof.** It suffices to show for all \( 1 \leq i \leq n \), \( s'_i \rightarrow_S s'_{i+1} \). Since \( \langle s_i, s'_{i+1} \rangle \Rightarrow_{S} \langle \cdot \rangle \), we have \( s'_i \rightarrow_S s'_{i+1} \) by Lemma 8.

\[ \square \]

Finally the proof of main theorem: Proposition 13:

**Proof.** By Lemma 12, we know the transition rule, \( s'_i \rightarrow_S s'_{i+1} \), is not one of the lazy initialization rules. Then by Lemma 13 and Lemma 9, we get \( c_i = \text{default-concr}(T, E, s'_i) \rightarrow_c \text{default-concr}(T, E, s_{i+1}) = c_{i+1} \) for all \( 1 \leq i \leq n \). Since \( c_i \in \gamma_i(s'_i) \subseteq \gamma_i(s_i) \), we have \( c_i \in \gamma_i(s_i) \) for all \( 1 \leq i \leq n + 1 \).

\[ \square \]

Using the soundness of symbolic execution, we know that each path is covered by some symbolic trace. Since for each symbolic trace, we generate a concrete trace which covers the path, we have achieved complete path coverage.
E.1.4 Backtracking Rules for Symbolic Execution with Lazier Initialization

The backtracking rule FOO-A-BACK for rule FOO is defined as \( \Sigma_R \times \Sigma_R \Rightarrow \Sigma_R \). We modify the standard lazier backtracking rules as follows: if a symbolic location is initialized, the initialization is kept in the return state. For example, the IF_ACMPEQ2-A and IF_ACMPEQ3-A rules have the same backtracking rule IF_ACMPEQ-A-BACK. The GETFIELD1-A-BACK is the backtracking rule for the GETFIELD1-A rule. The backtrack rule for rule GETFIELD2-A is GETFIELD-S-BACK. The other backtracking rules are the same as the backtracking rules in the symbolic execution with lazy initialization shown in Section E.1.1.

We use the bindings, \( \sigma = (g, pc, l, \omega, h, \phi) \) and \( \sigma' = (g', pc', l', \omega', h', \phi') \), for all the backtracking rules.

\[
\begin{align*}
\text{IF_ACMPEQ-A-BACK} & \quad \text{code}(pc) = \text{if_acmpeq} \ pc \quad pc = pc' \\
\sigma, \sigma' & \quad \Rightarrow \sigma' \\
\text{GETFIELD1-A-BACK} & \quad \text{code}(pc) = \text{getfield} \ f \quad pc = pc' \\
\sigma, \sigma' & \quad \Rightarrow \sigma'
\end{align*}
\]

Since each \( \Rightarrow \) transition may consist of multiple \( \Rightarrow \) transitions, the backtracking of \( \Rightarrow \) will start from the last one and proceed backward. Suppose that \( \sigma \Rightarrow \sigma' \) consists of \( n + 1 \) transitions \( \sigma \Rightarrow \sigma_1 \sigma_2 \Rightarrow \sigma_2 \ldots \sigma_n \Rightarrow \sigma' \) in \( \Rightarrow \). We know that the first \( n \) transitions just initialize symbolic locations and only the last transition does the real computation. By the backtracking rules of lazier symbolic execution, the initialization of symbolic location will backtrack to the second input state \( (\langle a, a' \rangle \Rightarrow \sigma' \) ). Thus the net effect of backtracking \( \sigma \Rightarrow \sigma' \) from \( \sigma' \) is the same as the just backtracking the last \( \Rightarrow \) rule from \( \sigma' \), that is, \( (\sigma_n, \sigma') \Rightarrow \sigma'' \).

Lemma 14. For any \( a_1 \Rightarrow a_2 \), after backtracking from \( a_3 \) we get an input state \( a'_1 \) (by possible several backtrackings). Then \( a'_1 \Rightarrow a_3 \) must hold. Further, \( a'_1 \Rightarrow a_3 \) and the transition rule is not one of those involving lazy/lazier initialization.

\[\text{Proof.} \quad \text{We proceed by rule induction on the transition instruction of } \Rightarrow. \text{ If the last } \Rightarrow \text{ transition in } a_1 \Rightarrow a_2 \text{ is } t \Rightarrow_S a_2 \text{ for some state } t. \text{ Then we know that } (t, a_2) \Rightarrow \sigma'_1. \text{ Use Lemma 7, we get } a'_1 \Rightarrow_S a_2 \text{ and the transition rule is not one of those involving lazy initialization. Otherwise, the last transition rule can only be IFNULL-A, IFNONNULL-A, or GETFIELD2-A. It is easy to show that the conclusion holds. Thus } a'_1 \Rightarrow a_3 \text{ holds and the transition does not involve lazier/lazy initialization.}\]

E.1.5 Default Concretization Function for Symbolic Execution with Lazier Initialization

We now define a default concretization function \( \text{default-sym} : \Pi \times \Sigma_R \rightarrow \Sigma_S \) as \( \text{default-sym}(F, \sigma_a) = \sigma_s \) with \( \sigma_s \in ST_a[\sigma_a](F) \) for some \( F \) satisfies \( \text{default-sym-map}(F, \sigma_a) \). And \( \text{default-sym-map}(F, \sigma_a) \) if and only if

\[\text{Lazy initialization refers to getfield and the field is not defined. Lazier initialization refers to initialize symbolic location to location.}\]
• $\forall \delta \in \text{collect-sym-locs}(\sigma_a). F(\delta) \notin \text{dom} h_a$;

• $\forall \delta_1, \delta_2 \in \text{collect-sym-locs}(\sigma_a). F(\delta_1) = F(\delta_2) \implies \delta_1 = \delta_2$.

The idea is to create a new symbolic object for each symbolic location and add constraints to the path condition accordingly. It is obvious that $\text{default-sym}(F, \sigma_a) \in \gamma_a(\sigma_a)$.

**Lemma 15.** Suppose $a_1 \rightarrow_{\mathcal{A}} a_2$ and the transition rule does not involve lazy/lazier initialization. Then $\text{default-sym}(F, a_1) \rightarrow_S \text{default-sym}(F, a_2)$ for some $F$ such that $\text{default-sym-map}(F, a_2)$ and the transition does not involve lazy initialization.

**Proof.** We proceed with rule induction on $\rightarrow_{\text{Laziest}}$. $\square$

### E.1.6 Input Generation Algorithm for Symbolic Execution with Lazier Initialization and Proof

Given any trace $a_1 \rightarrow_{\mathcal{A}} a_2 \rightarrow_{\mathcal{A}} \cdots a_n \rightarrow_{\mathcal{A}} a_{n+1}$. Suppose $t_i \rightarrow_{\mathcal{A}} a_{i+1}$ is the last $\rightarrow_{\mathcal{A}}$ transition in $a_i \rightarrow_{\mathcal{A}} a_{i+1}$ for all $1 \leq i \leq n$. Define a sequence of states $a'_i$ for $1 \leq i \leq n+1$ as

$$
\begin{align*}
& a'_{n+1} = a_{n+1} \\
& \langle t_n, a'_{n+1} \rangle \rightarrow_{\mathcal{A}} a'_n, \\
& \vdots \\
& \langle t_2, a'_2 \rangle \rightarrow_{\mathcal{A}} a'_1, \\
& \langle t_1, a'_2 \rangle \rightarrow_{\mathcal{A}} a'_1.
\end{align*}
$$

The applicability of backtracking rules is shown by Lemma 17. Then we apply the default concretization function for $a'_i$, where $1 \leq i \leq n+1$ for some $F$ which satisfies $\text{default-sym-map}(F, a_{n+1})$. And we get $s_1 = \text{default-sym}(F, a'_1), \ldots, s_{n+1} = \text{default-sym}(F, a'_{n+1})$.

**Proposition 14.** 1. $s_1 \rightarrow_S s_2 \cdots \rightarrow_S s_{n+1}$ and $s_i \in \gamma_a(a_i)$ for all $1 \leq i \leq n+1$;

2. $\text{default-concr}(T, E, s_1) \rightarrow_C \text{default-concr}(T, E, s_2) \cdots \rightarrow_C \text{default-concr}(T, E, s_{n+1})$ for some $T, E$ satisfy the path condition of $s_{n+1}$.

To prove this main theorem, we need an additional definition and some lemma.

**Definition 9.** Relation $\prec_{\mathcal{A}}: \Sigma_\mathcal{A} \times \Sigma_\mathcal{A}$ as $a_1 \prec_{\mathcal{A}} a_2$ iff $a_1$ is similar to $a_2$ except that there are some fields of symbolic objects/arrays in $a_1$ are defined but not in $a_2$ and some of symbolic locations are initialized in $a_1$ but not in $a_2$.

Precisely, $a_1 \prec_{\mathcal{A}} a_2$ for $a_1 = (g_1, pc_1, l_1, \omega_1, h_1, \phi_1)$ and $a_2 = (g_2, pc_2, l_2, \omega_2, h_2, \phi_2)$ if and only if following conditions hold

1. There exists some sequence of $(\delta_1, i_1), \ldots, (\delta_m, i_m)$ for some $m \geq 0$ where each $i_i$ is legal value for $\delta_i$ regarding to $a_2$ and let $t_1 = \text{init-sym-loc}(a_2, \delta_1, i_1)$, $t_2 = \text{init-sym-loc}(t_1, \delta_2, i_2)$, $\ldots$, $t_m = \text{init-sym-loc}(t_{m-1}, \delta_m, i_m)$. Suppose $t_m = (g_m, pc_1, l_m, \omega_m, h_m, \phi_m)$, we have $g_m = g_1, pc_1 = pc_2, l_1 = l_m, \omega_1 = \omega_m, \phi_m \subseteq \phi_1$, and $\text{refineheap}(pc_1, h_m, h_1)$. $57$
2. If code(pc₁) accesses heap, then the operands (in top of stack) a₁ can not be symbolic locations. For example, if code(pc₂) = getfieldf then \( \omega_{i} = i::\omega'_{i} \) for some \( i \in \text{Locs} \).

It is clearly that \( \gamma_{a}(a_{1}) \subseteq \gamma_{a}(a_{2}) \).

**Lemma 16.** Suppose we have

1. \( \sigma_{4} \prec_{\mathcal{R}} \sigma_{2} \) or
2. \( \sigma_{4} = \sigma_{2} \)

and \( \sigma_{1} \rightarrow_{\mathcal{R}} \sigma_{2} \). Let \( \sigma_{3} \) be the outcome of backtracking from \( \sigma_{4} \). Then \( \sigma_{3} \prec_{\mathcal{R}} \sigma_{1} \).

**Proof.** WLOG, we assume that \( \sigma_{4} \prec_{\mathcal{R}} \sigma_{2} \). We will prove by rule induction of lazier semantics transitions: \( \rightarrow_{\mathcal{R}} \). We will use bindings:

\( \sigma_{1} = (g_{1}, pc_{1}, l_{1}, \omega_{1}, h_{1}, \phi_{1}) \),
\( \sigma_{2} = (g_{2}, pc_{2}, l_{2}, \omega_{2}, h_{2}, \phi_{2}) \),
\( \sigma_{3} = (g_{3}, pc_{3}, l_{3}, \omega_{3}, h_{3}, \phi_{3}) \),
\( \sigma_{4} = (g_{4}, pc_{4}, l_{4}, \omega_{4}, h_{4}, \phi_{4}) \).

- Rule IADD: Since \( \sigma_{1} \rightarrow_{\mathcal{R}} \sigma_{2} \) by rule IADD-S, we know \( \omega_{1} = v_{1} :: v_{2} :: \omega'_{1} \), code(pc₁) = iadd, \( \omega_{2} = v' :: \omega'_{1} \), \( pc_{2} = \text{next}(pc_{1}) \), \( g_{1} = g_{2}, l_{1} = l_{2}, h_{1} = h_{2}, \) and \( \phi_{1} \subseteq \phi_{2} \). Since \( \sigma_{4} \prec_{\mathcal{R}} \sigma_{2} \) and the top element of \( \omega_{4} \) is a type of integer, we get \( \sigma_{3} \prec_{\mathcal{R}} \sigma_{1} \).

- Rule IF_ACMEQ: WLOG, assume that \( \sigma_{1} \rightarrow_{\mathcal{R}} t_{1} \) by rule IF_ACMEQ2-A; then \( t_{1} \rightarrow_{\mathcal{R}} t_{2} \) by rule IF_ACMEQ3-A; finally \( t_{2} \rightarrow_{\mathcal{R}} \sigma_{2} \) by rule IF_ACMEQ1-S. Then \( \omega_{1} = \delta' :: \omega'_{1} \) for some \( \delta', \delta'' \), and \( \omega'_{1} \). Suppose the stack of \( t_{2} \) is \( \omega_{1} = i :: j :: \omega'_{1} \). Since \( \sigma_{4} \prec_{\mathcal{R}} \sigma_{2} \), there exists some sequence of \( (\delta_{i}, i) \) satisfies condition 1. We can just append \((\delta', i)\) and \((\delta'', j)\) in front of the sequence \( (\delta_{i}, i) \) for \( \sigma_{3} \) and \( \sigma_{1} \). By the backtracking rules IF_ACMEQ-A-BACK and IF_ACMEQ-S-F-BACK, we get \( \sigma_{3} \prec_{\mathcal{R}} \sigma_{1} \).

- Rule GETFIELD f: WLOG, assume that \( \sigma_{1} \rightarrow_{\mathcal{R}} t \) by rule GETFIELD1-A and \( t \rightarrow_{\mathcal{R}} \sigma_{2} \) by rule GETFIELD2-S. Then \( \omega_{1} = \delta :: \omega'_{1} \) for some \( \delta \) and \( \omega'_{1} \) and the stack of \( t \), \( \omega_{i} = i :: \omega'_{i} \). Since \( \sigma_{4} \prec_{\mathcal{R}} \sigma_{2} \), there exists some sequence \( (\delta_{i}, i) \) satisfies condition 1. We can just append \((\delta', i)\) in front of sequence \( (\delta_{i}, i) \) for \( \sigma_{3} \) and \( \sigma_{1} \). By the backtracking rules GETFIELD1-A-BACK and GETFIELD-S-BACK, we get \( \sigma_{3} \prec_{\mathcal{R}} \sigma_{1} \).

\[\blacksquare\]

**Lemma 17.** \( a'_{i} \prec_{\mathcal{R}} a_{i} \) for all \( 1 \leq i \leq n + 1 \).

**Proof.** Since \( a_{n+1} = a'_{n+1} \), we have \( a'_{n} \prec_{\mathcal{R}} a_{n} \) by case (2) of Lemma 16. Then by induction: going backward with case (1) of Lemma 16 for inductive step. \[\blacksquare\]

**Lemma 18.** \( a'_{1} \rightarrow_{\mathcal{R}} a'_{2} \rightarrow_{\mathcal{R}} \cdots \rightarrow_{\mathcal{R}} a'_{n+1} \). Further, all the rules are \( \Rightarrow_{S} \) and not involving lazy initialization.

**Proof.** It suffices to show for all \( 1 \leq i \leq n, a'_{i} \rightarrow_{\mathcal{R}} a'_{i+1} \). Since \( a'_{i} \prec_{\mathcal{R}} a_{i}, a'_{i+1} \prec_{\mathcal{R}} a_{i+1} \), and \( a_{i} \rightarrow_{\mathcal{R}} a_{i+1} \), we have \( a'_{i} \rightarrow_{\mathcal{R}} a'_{i+1} \). By the definition of \( \prec_{\mathcal{R}} \), we know that all the transitions rules are \( \Rightarrow_{S} \) and not involving lazy initialization. \[\blacksquare\]
Finally the proof of main theorem: Proposition 13:

Proof. By Lemma 18 and Lemma 15, we get \( s_i = \text{default-sym}(F, a'_i) \rightarrow_S \text{default-sym}(F, a'_i) \rightarrow s_{i+1} \) for all \( 1 \leq i \leq n \). Since \( s_i \in \gamma_\sigma(a'_i) \subseteq \gamma_\sigma(a_i) \), we get \( s_i \in \gamma_\sigma(a_i) \) for all \( 1 \leq i \leq n + 1 \). Since \( a'_i \Rightarrow a'_{i+1} \) does not involve lazy initialization, \( s_i \Rightarrow_S s_{i+1} \) does not involve lazy initialization. By Lemma 9, we have \( \text{default-concr}(T, E, s_i) \rightarrow_C \text{default-concr}(T, E, s_i) \rightarrow C \rightarrow_C \text{default-concr}(T, E, s_{i+1}) \) for some \( T, E \) satisfying the path condition of \( s_{i+1} \).

Since \( \text{default-concr}(T, E, s_i) \in \gamma_\sigma(s_i) \) and \( s_i \in \gamma_\sigma(a_i) \), \( \text{default-concr}(T, E, s_i) \in \bigcup_{t \in \gamma_\sigma(a_i)} \gamma_\sigma(t) \). Therefore, for any lazier symbolic trace \( a_1 \rightarrow a_2 \cdots \rightarrow a_n \), we generate a corresponding concrete trace \( c_1 \rightarrow c_2 \cdots \rightarrow c_n \) and \( c_i \in \gamma_\sigma(\gamma_\sigma(a_i)) \) for all \( 1 \leq i \leq n \). Using the soundness of symbolic execution with lazier initialization, we get the complete path coverage.

### E.1.7 Backtracking Rules for Symbolic Execution with Lazier# Initialization

The backtracking rule FOO-B-BACK for rule FOO is defined as \( \Sigma_{b} \times \Sigma_{b} \Rightarrow_{b}^{-1} \Sigma_{b} \). We modify the standard lazier# backtracking rules as follows: if a symbolic reference is initialized, the initialization is kept in the return state. For example, the IF_ACMPAQ2-B and IF_ACMPAQ3-B rules have the same backtracking rule IF_ACMPAQ-B-A-BACK. The GETFIELD1-B-BACK is the backtracking rule for the GETFIELD1-B rule. The backtrack rule for rule GETFIELD2-B is GETFIELD-S-BACK. The other backtracking rules are the same as the backtracking rules in the symbolic execution with lazy/lazier# initialization shown in Section E.1.1 and Section E.1.7.

Since each \( \rightarrow_{b} \) transition may consist of multiple \( \Rightarrow_{b} \) transitions, the backtracking of \( \rightarrow_{b} \) will start from the last one and proceed backward. Suppose that \( \sigma \rightarrow_{b} \sigma' \) consists of \( n + 1 \) transitions \( \sigma \Rightarrow_{b} \sigma_1, \sigma_2 \Rightarrow_{b} \sigma_2, \ldots, \) and \( \sigma_n \Rightarrow_{b} \sigma' \) in \( \Rightarrow_{b} \). We know that the first \( n \) transitions just initialize symbolic locations/references and only the last transition does the real computation. By the backtracking rules of lazier# symbolic execution, the initialization of symbolic location/reference will backtrack to the second input state \( \langle b, b' \rangle \Rightarrow_{b}^{-1} b' \). Thus the net effect of backtracking \( \sigma \rightarrow_{b} \sigma' \) from \( \sigma' \) is the same as the just backtracking the last \( \Rightarrow_{b} \) rule from \( \sigma' \), that is, \( \langle \sigma_n, \sigma' \rangle \Rightarrow_{b}^{-1} \sigma'' \).

Lemma 19. For any \( b_1 \rightarrow_{b} b_2 \), after backtracking from \( b_2 \), we get an input state \( b'_1 \) (possible by several backtracings). Then \( b'_1 \Rightarrow_{b} b_2 \) must hold. Further, the transition rule is not one of those involving lazy initialization.

Proof. Suppose the last \( \Rightarrow_{b} \) transition in \( b_1 \rightarrow_{b} b_2 = t \Rightarrow_{b} b_2 \) for some state \( t \). Then we know that \( \langle t, b_2 \rangle \Rightarrow_{b}^{-1} b'_1 \). We can use rule induction on the instruction and get \( b'_1 \Rightarrow_{b} b_2 \) and the transition rule is not one of those involving lazy initialization. Thus \( b'_1 \rightarrow_{b} b_2 \) holds.

### E.1.8 Default Concretization Function for Symbolic Execution with Lazier# Initialization

We now define a default concretization function \( \text{default-lazier} : \Sigma \times \Sigma_{b} \rightarrow \Sigma_{b} \) as \( \text{default-lazier}(G, \sigma_b) = \sigma_a \) with \( \sigma_a \in \text{ST}_{b} (\sigma_b) \) (G) for some \( G \) satisfies \( \text{default-symref-map}(G, \sigma_b) \) and \( \text{default-symref-map}(G, \sigma_b) \).
if and only if $\forall \hat{\delta} \in \text{collect-sym-refs}(r_b).G(\hat{\delta}) = \text{null}$. The idea is set each symbolic reference to null.

**Lemma 20.** Suppose $b_1 \rightarrow_G b_2$ and $b_1 \Rightarrow_G b_2$ the transition rule does not involve lazy initialization. Then $\text{default-lazier}(G, b_1) \rightarrow_S \text{default-lazier}(G, b_2)$ for some $G$ such that $\text{default-symref-map}(G, b_2)$.

**Proof.** Proof by rule induction on $\rightarrow_G$. □

### E.1.9 Input Generation Algorithm for Symbolic Execution with Lazier Initialization and Proof

Given any trace $(b_1 \rightarrow_G b_2 \rightarrow_G \ldots \rightarrow_G b_n \rightarrow_G b_{n+1})$. Suppose $t_i \Rightarrow_A b_{i+1}$ is the last $\Rightarrow_G$ transition in $b_i \rightarrow_G b_{i+1}$ for all $1 \leq i \leq n$. Define a sequence of states $b'_i$ for $1 \leq i \leq n+1$ as

$$b'_{n+1} = b_{n+1}$$

$$\langle t_n, b'_{n+1} \rangle \Rightarrow_A b'_n,$$

$$\vdots$$

$$\langle t_2, b'_3 \rangle \Rightarrow_A b'_2,$$

$$\langle t_1, b'_2 \rangle \Rightarrow_A b'_1.$$  

The applicability of backtracking rules is shown by Lemma 22. Then we apply the default concretization function for $b'_i$, where $1 \leq i \leq n+1$ for some $G$ which satisfies $\text{default-symref-map}(G, b_{n+1})$. And we get $a_1 = \text{default-lazier}(G, b'_i)$, $\ldots$, $a_{n+1} = \text{default-lazier}(G, b'_{n+1})$. Define $s_1 = \text{default-sym}(F, a_1)$, $\ldots$, $s_{n+1} = \text{default-sym}(F, a_{n+1})$ where $F$ satisfies $\text{default-sym-map}(F, a_{n+1})$.

**Proposition 15.** 1. $a_1 \rightarrow_A a_2 \rightarrow_A \cdots \rightarrow_A a_{n+1}$.  
2. $s_1 \rightarrow_S s_2 \rightarrow_S \cdots \rightarrow_S s_{n+1}$ and $s_i \in \gamma_A(a_i)$ for all $1 \leq i \leq n+1$;  
3. $\text{default-conc}(T, E, s_1) \rightarrow_C \text{default-conc}(T, E, s_2) \rightarrow_C \cdots \rightarrow_C \text{default-conc}(T, E, s_{n+1})$ for some $T, E$ satisfy the path condition of $s_{n+1}$.

To prove this main theorem, we need an additional definition and some lemma.

**Definition 10.** Relation $<_G: \Sigma_B \times \Sigma_B$ as $b_1 <_G b_2$ iff $b_1$ is similar to $b_2$ except that there are some fields of symbolic objects/arrays in $b_1$ are defined but not in $b_2$; some of symbolic locations are initialized in $b_1$ but not in $b_2$; some of symbolic references are initialized in $b_1$ but not in $b_2$.

Precisely, $b_1 <_G b_2$ for $b_1 = (g_1, pc_1, l_1, \omega_1, h_1, \phi_1)$ and $b_2 = (g_2, pc_2, l_2, \omega_2, h_2, \phi_2)$ if and only if following conditions hold

1. There exists a subset of symbolic references, $\text{psr} \subseteq \text{SymRefs}$ and a state $t = \text{ST}_{\text{sym}}[b_2](G | \text{psr})$ such that $G \in \text{legal-env}(b_2)$. Further, there exists some sequence of $(\delta_1, i_1), \ldots, (\delta_m, i_m)$ for some $m \geq 0$ where each $i_j$ is legal value for $\delta_j$ regarding to $t$ and let $t_1 = \text{init-sym-loc}(t, \delta_1, i_1)$, $t_2 = \text{init-sym-loc}(t_1, \delta_2, i_2), \ldots, t_m = \text{init-sym-loc}(t_{m-1}, \delta_m, i_m)$. Suppose $t_m = (g_{m}, pc_1, l_m, \omega_m, h_m, \phi_m)$, we have $g_m = g_1, pc_1 = pc_2, l_1 = l_m, \omega_1 = \omega_m, \phi_m \subseteq \phi_1$, and $\text{refineheap}(pc_1, h_m, h_1)$. 

Lemma 21. Suppose we have

1. \( \sigma_4 \prec_B \sigma_2 \) or

2. \( \sigma_4 = \sigma_2 \)

and \( \sigma_1 \rightarrow_B \sigma_2 \). Let \( \sigma_3 \) be the outcome of backtracking from \( \sigma_4 \). Then \( \sigma_3 \prec_B \sigma_1 \).

Proof. WLOG, we assume that \( \sigma_4 \prec_B \sigma_2 \). We will prove by rule induction of lazier# semantics transitions: \( \rightarrow_B \). We will use bindings: \( \sigma_1 = (g_1, p_{c_1}, l_1, o_1, h_1, \phi_1) \), \( \sigma_2 = (g_2, p_{c_2}, l_2, o_2, h_2, \phi_2) \), \( \sigma_3 = (g_3, p_{c_3}, l_3, o_3, h_3, \phi_3) \), and \( \sigma_4 = (g_4, p_{c_4}, l_4, o_4, h_4, \phi_4) \).

- Rule IADD: Since \( \sigma_1 \Rightarrow A \sigma_2 \) by rule IADD-S, we know \( \omega_1 = v_1 :: v_2 :: \omega'_1 \), \( \text{code}(p_{c_1}) = \text{iadd} \), \( \omega_2 = \nu' :: \omega'_1, p_{c_2} = \text{next}(p_{c_1}) \), \( g_1 = g_2, l_1 = l_2, h_1 = h_2 \), and \( \phi_1 \subseteq \phi_2 \). Since \( \sigma_4 \prec_B \sigma_2 \) and the top element of \( \omega_4 \) is a type of integer, we get \( \sigma_3 \prec_B \sigma_1 \).

- Rule IF_ACMEQP: WLOG, assume that \( \sigma_1 \Rightarrow B \sigma_1 \) by rule IF_ACMEQP2-B; then \( \sigma_1 \Rightarrow B \sigma_2 \) by rule IF_ACMEQP2-A; then \( \sigma_2 \Rightarrow B \sigma_3 \) by rule IF_ACMEQP3-A; finally \( \sigma_3 \Rightarrow B \sigma_4 \) by rule IF_ACMEQP1-S. Then \( \omega_1 = \delta' :: \delta'' :: \omega'_1 \) for some \( \delta', \delta'' \), and \( \omega'_1 \). Suppose the stack of \( t_3 \) is \( \omega_3 = \delta' :: \delta'' :: \omega'_3 \) and the stack of \( t_5 \) is \( \omega_5 = i :: j :: \omega'_5 \). Since \( \sigma_4 \prec_B \sigma_2 \), there exist \( \text{psr}, G \), and some sequence of \( (\delta_i, i) \) satisfies condition 1. We can just let \( \text{psr}' = \text{psr} \cup \{ \delta', \delta'' \} \) and \( G' = G[\delta' \mapsto \delta'] \cup \delta'' \mapsto \delta'' \) and append \( (\delta', i) \) and \( (\delta'', j) \) in front of the sequence \( (\delta_i, i) \) for \( \sigma_3 \) and \( \sigma_1 \). By the backtracking rules IF_ACMEQP-A_BACK and IF_ACMEQP-S-F_BACK, we get \( \sigma_3 \prec_B \sigma_1 \).

- Rule GETFIELD f: WLOG, assume that \( \sigma_1 \Rightarrow B \sigma_1 \) by rule GETFIELD1-B and \( \sigma_1 \Rightarrow B \sigma_2 \) and \( \sigma_2 \Rightarrow B \sigma_3 \) by rule GETFIELD2-S. Then \( \omega_1 = \delta :: \omega'_1 \) for some \( \delta \) and \( \omega'_1 \) and the stack of \( t_1 \), \( \omega_1 = \delta :: \omega'_1 \) and the stack of \( t_2 \), \( \omega_2 = i :: \omega'_2 \). Since \( \sigma_4 \prec_B \sigma_2 \), there exist \( \text{psr}, G \), and some sequence \( (\delta_i, i) \) satisfy condition 1. We can let \( \text{psr}' = \text{psr} \cup \{ \delta \} \), \( G' = G[\delta \mapsto \delta] \), and append \( (\delta, i) \) in front of sequence \( (\delta_i, i) \) for \( \sigma_3 \) and \( \sigma_1 \). By the backtracking rules GETFIELD1-A_BACK and GETFIELD-S_BACK, we get \( \sigma_3 \prec_B \sigma_1 \).

Lemma 22. \( b'_i \prec_B b_i \) for all \( 1 \leq i \leq n + 1 \).

Proof. Since \( b_{n+1} = b'_{n+1} \), we have \( b'_n \prec_B b_n \) by case (2) of Lemma 21. Then by induction: going backward with case (1) of Lemma 21 for inductive step.

Lemma 23. \( b_1 \rightarrow_B b_2 \rightarrow_B \cdots \rightarrow_B b'_{n+1} \). Further, all the transition consists of just one \( \Rightarrow_B \) and not involving lazy initialization.
Proof. It suffices to show for all \(1 \leq i \leq n\), \(b'_i \rightarrow_{b} b'_{i+1}\). Since \(b'_i \prec b_i\) and \(b'_i \rightarrow_{b} b'_{i+1}\), we have \(b'_i \rightarrow_{b} b'_{i+1}\). By the definition of \(\prec\), we know that all the transitions rules are one step and not involving lazy initialization. \(\square\)

Finally the proof of main theorem: Proposition 15:

Proof. By Lemma 23 and Lemma 20, we get \(a_i = \text{default-lazier}(G, b'_i) \rightarrow_{\mathcal{A}} \text{default-lazier}(G, b'_{i+1}) = a_{i+1}\) for all \(1 \leq i \leq n\). Further, \(s_i = \text{default-sym}(G, a_i) \rightarrow_{S} \text{default-sym}(G, a_{i+1})\). Since \(s_i \in \gamma_a(a_i)\) and \(a_i \in \gamma_b(b'_i) \subseteq \gamma_b(b_i)\), we get \(s_i \in \gamma_b(\gamma_a(a_i))\) for all \(1 \leq i \leq n + 1\). Since \(b'_i \Rightarrow b'_{i+1}\) does not involve lazy initialization, \(s_i \Rightarrow s_s i_{i+1}\) does not involve lazy initialization. By Lemma 9, we have \(\text{default-concr}(T, E, s_1) \rightarrow_{C} \text{default-concr}(T, E, s_2) \cdots \rightarrow_{C} \text{default-concr}(T, E, s_{n+1})\) for some \(T, E\) satisfying the path condition of \(s_{n+1}\). \(\square\)

Since \(\text{default-concr}(T, E, s_i) \in \gamma_s(s_i)\) and \(s_i \in \gamma_a(a_i), \text{default-concr}(T, E, s_i) \in \bigcup_{a \in \gamma_a(b_i)} \bigcup_{s \in \gamma_s(a)} \gamma(s).\) Therefore, for any lazier# symbolic trace \(b_1 \rightarrow_{b} b_2 \cdots \rightarrow_{b} b_n\), we generate a corresponding concrete trace \(c_1 \rightarrow_{C} c_2 \cdots \rightarrow_{C} c_n\) and \(c_i \in \gamma_s(\gamma_a(\gamma_b(b_i)))\) for all \(1 \leq i \leq n\). Using the soundness result of symbolic execution with lazier# initialization, we get the complete path coverage.