Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs*

Xianghua Deng, Robby, John Hatcliff
Department of Computing and Information Sciences, Kansas State University
{deng, robb, hatcliff}@cis.ksu.edu

Abstract

Recent work has demonstrated that symbolic execution techniques can serve as a basis for formal analysis capable of automatically checking heap-manipulating software components against strong interface specifications. In this paper, we present an enhancement to existing symbolic execution algorithms for object-oriented programs that significantly improves upon the algorithms currently implemented in Bogor/Kiasan and JPF. To motivate and justify the new strategy for handling heap data in our enhanced approach, we present a significant empirical study of the performance of related algorithms and an interesting case counting analysis of the heap shapes that can appear in several widely used Java data structure packages.

1 Introduction

In development contexts that emphasize reusable components, it is important for development methodologies and processes to be supported by the following capabilities: (1) specification notations such as those used in the Design-by-Contract (DBC) [15] paradigm that provide “software contracts” to specify the assumptions that a component makes about its context and the behavior/functionality of the services that the component guarantees to provide to clients, and (2) tools for automatically checking that clients conform to component contract assumptions and that a component’s implementation provides functionality that satisfies what its contract guarantees. These capabilities can be difficult to provide in OOP languages due to the extensive use of dynamically created heap objects, where one has to be able to precisely reason about objects, their data, and the relationships between them. We refer to invariants and functional behavioral specifications on complex heap data structures as strong properties [17] as they are hard to analyze due to aliasing issues (e.g., equivalence of object structures); lightweight properties such as simple relationships between scalar values and variable null-ness are the counterpart of strong properties that are also important to specify and enforce as an integral part of the development process.

Tools such as ESC/Java [9] that are founded on theorem-proving techniques have made significant contributions in the area of automated contract checking. However, ESC/Java and related tools have significant difficulties in supporting checking of strong properties of heap-manipulating programs; they provide weak support for generation of informative counter-examples, and they lack integration with existing quality assurance mechanisms such as testing.

Recent efforts such as JPF’s “lazy initialization” approach to symbolic execution [11] and others (e.g., [19, 6]) have demonstrated that symbolic execution can serve as a basis for checking strong contract properties and invariants of complex heap-based data structures and for supporting automated test case generation. However, because symbolic execution is typically implemented as a path-sensitive analysis, it can require significant computational resources.

In previous work, we have introduced Kiasan [6] – a symbolic execution framework for object-oriented languages (including Java) built on top of the Bogor model-checking framework [16]. A significant part of our effort has focused on using: (a) empirical studies, and (b) a careful case analysis of heap states necessary for verifying invariants of several complex data structure examples to drive improvements in Kiasan (in particular, its treatment of heap data) that provide performance levels that enable Kiasan to be incorporated into realistic development contexts. In this paper, we present the results of those empirical studies and analyses and the resulting algorithmic improvements. Specifically, the main contributions of this paper are:

- a new algorithm that significantly improves upon the performance of related algorithms currently implemented in Kiasan and JPF,
- a rigorous case analysis of all possible heap shapes generated by several complex data structure implementations that establishes the optimality of our new algorithm on these data structures,
- an empirical evaluation (using twenty three different data structure packages) of JPF’s lazy initialization algorithm, Kiasan’s original lazier initialization algorithm, and the new lazier# initialization algorithm proposed that shows that the lazier# algorithm significantly improves upon the lazier algorithm which in turn significantly improves upon JPF’s lazy algorithm.

*This work was supported in part by the US National Science Foundation (NSF) awards 0454348, 0429141, and CAREER award 0644288, the US Air Force Office of Scientific Research (AFOSR), and Lockheed Martin Advanced Technology Laboratories.
In an accompanying tech report [8], we provide a proof of the relative soundness and completeness of the lazyer# algorithm along with additional performance data and discussion. It is important to note the scope of the empirical study that we present here goes far beyond previous work on this topic which only considered anywhere from 1-7 examples.

2 Background

Symbolic Execution Basics: King proposed symbolic execution [12] (SymEx) 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). Figure 1 illustrates the symbolic computation tree of an example method max (each tree node is a symbolic state (a, b, z, \( \phi \)) that associates a symbol/concrete value to a, b, z, and predicate \( \phi \) that constrains the symbols). When symbolically executing max with no initial information about its argument, the initial state for the max method has a symbol \( \alpha \) for a, symbol \( \beta \) for b, 0 for z, and the constraint \( \phi \) set to true (no constraints imposed yet). When executing line 3, the symbolic execution does not have sufficient information to decide which branch to take because \( \phi \rightarrow (b > a) \) and \( \phi \rightarrow -(b > a) \) are all satisfiable under the current symbolic state – thus, both branches are explored. As each branch is traversed, the predicate is augmented with a constraint corresponding to the logical condition that would have caused the particular branch to be followed. Thus, the predicate \( \phi \) is often referred to as the path condition because it represents the conditions on variables that would be necessary for execution to flow down the current path. An assignment of concrete values to the symbols that satisfy the path condition can be used to form a test case that drives execution along the current path. If the path condition becomes false, the path is infeasible hence abandoned. The example shows that the true branch of line 5 is always infeasible.

As we can observe from this example, SymEx is a path-sensitive analysis, and is typically phrased as a depth-first search of a method’s execution paths. SymEx does not attempt to discover loop invariants, so symbolically executing a loop or cycle of recursive methods could result in an unbounded search – which is typically terminated by imposing some sort of bounding. SymEx also relies on underlying decision procedures (e.g., linear arithmetic) and constraint solvers to make deductive inferences and to find concrete solutions of constraint sets. If data/operations are encountered that lie outside the scope of the decision procedure’s theory, some other course of action must be taken.

Handling Objects: While the basic approach to SymEx of programs with simple scalar data was proposed decades ago and provides an intriguing approach to program checking and test generation, it has gained a reputation across the years as a technique that does poorly when applied to real systems and languages like Java with complex features. However, recent work on SymEx for object-oriented (OO) languages [11, 10, 6] has introduced significant innovations and moved the technology forward to the point where it can now be applied to complex Java data structures such as those in the Java Collection Framework. While some OO SymEx frameworks use a logical representation of the heap [19, 10], Kiasan leverages advances in explicit-state model checking to represent heap data directly using heap graphs. This approach provides the most precise alias information on heap objects and allows Kiasan to decide strong heap-oriented properties such as graph isomorphism without calling decision procedures [17].

Lazy Initialization: To handle unknown heap structures, Kiasan uses an enhancement of the lazy initialization algorithm originally introduced in [11]. The lazy initialization algorithm starts with no or partial knowledge of object values (i.e., symbolic objects whose fields are uninitialized) 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. Otherwise, for an unmaterialized reference field, the algorithm systematically (safely) explores all possible points-to relationships by non-deterministically choosing among the following values for the field: (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).

To illustrate lazy initialization, consider the following swap method for Node in Figure 2. The top part of Figure 3 illustrates the symbolic execution computation tree built using lazy initialization. To save space in the display of the
tree, we represent each tree node (system state) by a unique label corresponding to the path through the tree to the current code. The bottom part of Figure 3 shows heap configurations for some of the states in the computation tree. To generate the computation tree of Figure 3, the symbolic execution begins with a non-deterministic choice of possible aliasing between the method parameter n and the this reference (i.e., States 1, 2, and 3). Note that both the next and the data fields of this and n are unknown (unmaterialized). Out of the three cases, State 1 does not satisfy the `NonNull` precondition for n, thus it is not considered further. Now, consider the sub-tree starting from State 3. Upon executing `swap`'s first statement, the `this`.data field is now materialized according to the lazy algorithm described above; it non-deterministically chooses the value of `this`.data to be: `null` (31), equal to `this` – `n0` (32), `n1` (33), or a fresh symbolic object `e0` (34). Let us continue on with the sub-tree starting from State 33. Upon executing `swap`'s second statement, the algorithm non-deterministically chooses the null value, `n0`, `n1`, or a fresh symbolic object `e1` for the n’s data field, thus, resulting in the States 331, 332, 333, and 334, respectively. Executing `swap`’s last statement from 334 produces 3341 (the trace 3-33-334-3341 is highlighted in Figure 3). Note that the symbolic computation tree characterizes all possible concrete executions of `swap`. Kiasan’s lazy initialization algorithm has been formalized and its relative soundness and completeness have been proven [6]. In addition, all `swap`’s post-states in Figure 3 satisfy `swap`’s post-condition, thus, we conclude that the postcondition always holds (checking the postcondition requires reconstructing the effective pre-state of each post-state [7]).

**Lazier Initialization:** As we can observe, the lazy algorithm produces a rather large state-space even for `swap`. In [6], we introduced an optimized algorithm called **lazier initialization** based on the observation that when an uninitialized reference type variable is first read, it is not necessary to resolve the aliasing/object value at that particular point; only when the object referenced by the variable is accessed, that is when it is necessary to resolve the value. Basically, the lazier algorithm divides the lazy initialization into two steps as follows. Step 1, when an uninitialized reference type variable is read, it is lazier-ly initialized with the `null` value or a fresh **symbolic reference** value (whose type is the same as the variable’s type); in essence, the symbolic reference represents all possible objects that may be referenced by the variable (i.e., it abstracts such a set of objects). Non-reference-type variables are handled similarly to the lazy algorithm. Step 2, when a field of a symbolic reference is accessed (read/write), (a) the symbolic reference is then replaced by non-deterministically choosing any existing object or a fresh symbolic object (with compatible types); if the access is a read access and the field is unmaterialized, (b) the field is then initialized (with a `null` value or a fresh symbolic reference). The effects of these two steps are: (1) delaying the non-deterministic choice of objects in the lazy algorithm, and (2) the second step may not be needed in some cases. Thus, it produces a (significantly) smaller state-space (see our experiment data in Section 5).

To illustrate the lazier algorithm, let us reconsider the `swap` example in Figure 2. The left hand side of Figure 4 illustrates the symbolic computation tree using lazier initialization; the highlighted path in the (lazier) computation tree corresponds to the highlighted path in the (lazy) computation tree (i.e., it simulates the lazy path). Symbolic references are annotated with ∙. Similar to the lazy algorithm, the initialization algorithm starts with a non-deterministic choice. However, there are only two choices instead of three in the beginning. State 1 in Figure 3 is abstracted into State
1 in Figure 4, and State 2 and 3 in Figure 3 are abstracted into State 2 in Figure 4 (i.e., both $n_0$ and $n_1$ may actually be $n_0$ or $n_1$). When $n_0$’s data field is read at swap’s first statement, $n_0$ is replaced with $n_0$ (there is no existing symbolic objects for the non-deterministic choice, thus, it uses a fresh symbolic object), and $n_0$’s data field is initialized with either a NULL value (21) or a fresh symbolic reference $f_0$ (22). From State 22, there are three possible choices when executing swap’s second statement. We first replace $n_1$ by nondeterministically choosing the existing object $n_0$ or a fresh symbolic object $n_1$. In the former case, the data field has been initialized, thus no special treatment is needed (221). In the latter case, $n_1$’s data field is “lazier-ly” initialized with either NULL (222) or a fresh symbolic reference $e_1$ (223). Executing swap’s last statement from 223 produces 2231. Note that 2231 in Figure 4 safely approximates 3341 in Figure 3.

As we can observe, the computation tree in Figure 4 is much smaller than the one in Figure 3, because the nondeterministic choices for this, $n$, and the data fields are delayed, and the second steps of the lazier initialization for data never happen. Moreover, all swap’s post-states in Figure 4 still satisfy swap’s postcondition, thus, we conclude that the postcondition always holds. Note that we do not need to replace $f_0$ and $e_1$ with symbolic objects when checking the postcondition, as they will be compared against themselves (i.e., data==old(n.data) if $f_1 = e_1$). Kiasan’s lazier algorithm has been formalized and proved that it simulates the lazy algorithm (i.e., it is relatively sound and complete) [6].

$k$-bounding: There are two main challenges when using symbolic execution: (1) the termination of and (2) the scalability of the algorithm. To address these issues, Kiasan [6] incorporates a different bounding technique to help manage symbolic execution’s complexity, while providing fine-grained control over parts of the heap that one is interested in. In essence, we bound the sequence of lazy initializations originating from each initial symbolic object up to a user-supplied value $k$. This user-adjustable bounding provides an effective and controllable trade-off between analysis cost and behavioral coverage. When using a bound $k$, the analysis can guarantee the correctness of a program on any heap object configuration with reference chains whose lengths are at most $k$. In the case where the analysis does not exhaust $k$, a complete behavior coverage is guaranteed.

To handle diverging loops (or recursions), we limit the number of loop iterations that do not (lazily) initialize any heap object. That is, we prefer exhausting the $k$-bound first to try to achieve the advertised heap configuration coverage. Kiasan’s path-sensitivity allows it to know exactly the paths and the states under which its bounds are exhausted. This provides us with better control (methodologically) on how we increase the coverage on paths that exhaust the $k$-bound without exhausting the loop bound. In addition, for each path traversed, Kiasan can generate a JUnit test as well as a graphical visualization of data values flowing in and out of the method [7] – which can be very useful for developers when trying to diagnose the cause of a fault (the supplementary data on our website include these JUnit tests and visualizations for all our examples). The bounding strategy also allows Kiasan to terminate even without storing its state-spaces (i.e., following a stateless search as in model checking). Thus, it can be easily parallelized/distributed (by forking the search at state-space branches).

3 Case-Optimality Analysis

**Motivation:** As described previously, the lazier initialization algorithm significantly reduces the state-space for symbolic execution of object-oriented programs while still preserving strong heap-oriented properties. However, one might wonder whether it can still be improved. More specifically, we are interested in investigating whether the algorithm is case-optimal – it considers the minimum number of behavior cases (i.e., pairs of pre/post-states) when analyzing a given property and example (e.g., it considers only non-isomorphic heap shapes). Clearly, the answer is problem-dependent (and size-dependent for programs working with possibly unbounded number of objects).

For example, consider an insertion method of a binary search tree data structure. Intuitively, given an element to insert, the method should maintain the invariant of a binary
search tree. That is, given that the pre-state of the method satisfies the invariant, the tree still satisfies the invariant after insertion (Kiasan can additionally check stronger properties such as ensuring that the set of elements at post-states includes the set of elements at the prestate and the newly inserted elements, and only those elements). More concretely, for any binary search tree with $m$ nodes, after the insertion, the inserted element could be located in one of the $m$ internal nodes or the $m + 1$ NULL leaves. Thus, there are $2m + 1$ possible non-isomorphic cases for each input binary search tree with $m$ nodes.

**Analysis Method:** To generalize, we want to calculate the possible number of cases for the insertion method of the binary search tree for any $k$-bound. Since we limit the longest reference chain to $k$, the heights of the input trees are less than $k$. Thus, we need to consider the number of non-isomorphic binary search trees with $m$ nodes and heights less than $k$. Let such a number be $b(m, k)$; the total number of non-isomorphic input trees is $\sum_{k=0}^{m} b(m, k)$. Thus, the optimal number of cases for all non-isomorphic input tree heights less than $k$ is $c_k = \sum_{i=0}^{m} (2m + 1) b(m, k)$.

In order to compute $c_k$, we need to first calculate $b(m, k)$. A non-empty tree with height less than $k$ and $m > 0$ nodes can have $i$ nodes in the left subtree with height less than $k - 1$ and $m - 1 - i$ nodes in the right subtree with height less than $k - 1$ for any $0 \leq i \leq m - 1$. Thus,

$$b(m, k) = \sum_{i+j=m-1} b(i, k-1) b(j, k-1) \quad m > 0, k \geq 1,$$

with boundary condition, $b(0, 0) = 1$.

**Generating Function:** The recurrence relation (1) is very complex to work with. We will use a standard combinatorics technique called generating functions to simplify the calculation. A generating function [21] of a sequence of numbers $(a_n)$ is $G(x) = \sum_{n=0}^{\infty} a_n x^n$. This definition of $G(x)$ is sometimes called the ordinary generating function of $(a_n)$. For example, the generating function for $1, 1, 1, 1, \ldots$ is $G(x) = \sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$. Let us define a generating function for $b(m, n)$ as:

$$T_k(x) = \sum_{m=0}^{\infty} b(m, k) x^m \quad k \geq 0.$$

Now, we proceed to simplify $T_k(x)$. The terms in (1) can be simplified using the following technique: $G(x)H(x) = \sum_{k=0}^{\infty} \sum_{i+j=k} a_i b_j x^k$ for $G(x) = \sum_{i=0}^{m} a_i x^i$ and $H(x) = \sum_{j=0}^{n} b_j x^j$. After multiplying $x^m$ to both sides of (1) and summing over $1 \leq m \leq \infty$, we have:

$$T_k(x) = x[T_{k-1}(x)]^2 + 1 \quad k \geq 1.$$

Using recurrence (3) and $T_0(x) = 1$, we have $T_1(x) = 1 + x$, $T_2(x) = 1 + x + 2x^2 + x^3$, etc. Finally, we can use $T_k(x)$ to compute the number of cases, $c_k = \sum_{m=0}^{\infty} (2m + 1) b(m, k)$.

$$c_k = \sum_{m=0}^{\infty} (2m + 1) b(m, k) = \sum_{m=0}^{\infty} mb(m, k) + \sum_{m=0}^{\infty} b(m, k).$$

**Figure 5. A Binary Search Tree Insertion**

In order to calculate $\sum_{m=0}^{\infty} mb(m, k) = T_k'(1)$ and get $T_k'(x) = \sum_{m=0}^{\infty} mb(m, k)x^{m-1}$. Hence, $\sum_{m=0}^{\infty} mb(m, k) = T_k'(1)$. Therefore, $c_k = 2T_k'(1) + T_k(1)$. We use this formula to calculate $c_k = 2T_k'(1)+T_k(1) = 4$.

**Non-optimality of The Lazier Initialization Algorithm:** From Table 1, we know that the lazier algorithm considers 12 cases when $k = 1$ for the insert helper method shown in Figure 5. Since $12 > c_1$, we conclude that there is an inefficiency in the lazier algorithm. We have used this technique to calculate the minimum number of cases for different $k$-bounds for the binary search tree, AVL tree, and red-black tree (interested readers are referred to [8] for details). We are not aware of any other work that uses generating functions to compute complex data structure configurations based on maximal length of reference chains.

### 4 The Lazier# Initialization Algorithm

To address the inefficiency of the lazier initialization algorithm, we have developed an even lazier algorithm which we named the **lazier# initialization** algorithm. We observed that one source of efficiency in the lazier algorithm is due to the fact that it is optimized for non-NULL variables; it optimistically assumes most variables are non-NULL. That is, it eagerly initializes an uninitialized (reference type) variable as NULL or a fresh symbolic reference upon access. For example, consider the source code of `insert` shown in Figure 5. The lazier algorithm non-deterministically chooses between NULL and a fresh symbolic reference for the field `t.element` at line 4. However, the `t.element` is only used when comparing with the inserted element by `comparator.compare`, and the Java Comparator interface does not require `compare`’s parameters to be non-NULL. Thus, whether the value is NULL or non-NULL is irrelevant (i.e., processing the `compare` interface following a compositional checking approach to check the `insert` implementation will produce either a negative value, zero, or a positive value regardless). Therefore, the non-deterministic choice is too early at line 4 in the sense that it unnecessarily exposes details about the heap objects.

In the `lazier# initialization` algorithm, we introduce an intermediate step by initializing such variables with a new

---

1 This is in line with the current Java Modeling Language (JML) [13] default invariant for reference type variables.
flavor of symbolic value that abstracts null as well as any object of the appropriate type. For the rest of the paper, we use the general term “symbolic values” (with annotation \( \gamma \)) for abstract values that abstract both null and any object of the appropriate type, and we use the term “symbolic references” (with annotation \( \gamma \) as used previously) to refer to non-null values (i.e., we now have three abstraction levels for objects: (1) symbolic objects, (2) symbolic references, and (3) symbolic values). 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 (i.e., there is no non-deterministic choice). Step 2, when a field of a symbolic value is accessed, the symbolic value is replaced with null (which results in raising a null dereference exception), or a fresh symbolic reference. In the case of the latter, the algorithm proceeds similarly to the lazier algorithm (but, an uninitialized field is lazier\#-ly initialized instead lazierly initialized). The first step can be further optimized by directly using a fresh symbolic reference if the variable is known to be non-null; for the rest of the paper, we refer to this as the Non-null Variable (NV) optimization.

To illustrate the lazier\# initialization algorithm, let us revisit the swap example. Figure 6 illustrates the symbolic execution tree using the lazier\# algorithm and a trace (along with its states and their sibling states) that simulates the highlighted trace in Figure 4 (and thus, it simulates the trace highlighted in Figure 3). The algorithm starts with one state (State 1). Notice that \( n \) refers to \( \tilde{h}_1 \) instead of a symbolic value because of the NV optimization mentioned above (without the optimization, we use a fresh symbolic value \( h_1 \)). When executing swap’s first statement, \( h_0 \) is replaced with a fresh symbolic object \( n_0 \), and its data field is initialized with a fresh symbolic value \( \tilde{e}_0 \), thus resulting in State 11. Continue on with executing swap’s second statement, \( h_1 \) is replaced with either the existing symbolic object \( n_0 \) or a fresh symbolic object \( n_1 \). In the former case, \( n_0 \)’s data field has been initialized, thus no special treatment is needed (111). In the latter case, \( n_1 \)’s data field is initialized with a fresh symbolic value \( \tilde{e}_1 \) (112). From 112, it produces 1121. As we can observe, the lazier\# computation tree in Figure 6 realizes a correct abstraction of both the lazy (Figure 3) and lazier (Figure 4) computation trees while still exposing enough information to establish swap’s post-condition.

**Formalization:** Our description of the lazier\# initialization algorithm above is informal. To be precise, we present the formalization of \( k \)-bounded symbolic execution using the lazier\# algorithm in Figure 7 (along with the lazy and the lazier algorithms for reference). We have proved that the lazier\# algorithm simulates the lazier algorithm, and thus, simulates the lazy algorithm; formal simulation proofs as well as relative soundness and completeness proofs of Kisan’s lazy, lazier, and lazier\# initialization algorithms can be found at [22].

First, we introduce the semantic domains shown in the upper portion of Figure 7. There are two kinds of data in Java: primitive and non-primitive (including objects and arrays); symbolic execution treats input parameters or globals as symbols which can be either primitive or non-primitive. Thus, we have total four kinds of data: concrete primitive which is modeled by the Consts domain; symbolic primitive is modeled by the Symbols\(_{prim}\) domain; concrete and symbolic non-primitives are all modeled by the domain Symbols\(_{non-prim}\). Each symbol in Symbols (the union of Symbols\(_{prim}\) and Symbols\(_{non-prim}\)) is modeled as a partial mapping from fields/indexes to values. For symbols in Symbols\(_{prim}\) created by new-prim-sym function, the mappings are empty. While concrete non-primitive symbols (created by new-obj and new-arr functions) have the fields/indexes defined; symbolic objects (created by new-sym function) have all fields undefined and are initialized by lazy/lazier/lazier\# algorithms when used. Each symbol \( \alpha^\gamma_{n,a} \) has three properties: type \( \tau \), object bound \( m \), array bound \( n \). The object bounds are used to enforce the \( k \) bounds discussed in Section 2 (and similarly for arrays). We define the symbolic execution state as a hextuple which consists of globals, program counter, locals, operand stack, heap, and the path condition.

Second, we introduce the transition rules shown in the bottom portion of Figure 7. Each transition is in the form of \( s \mapsto_{\text{S}} \text{instr} \Rightarrow s_1[\text{stmt}_1] \mid \ldots \mid s_n[\text{stmt}_n] \mid \text{exception,} s_0[\text{stmt}_0] \mid \text{Error,} s'[\text{stmt}'] \) which intuitively means that instr non-deterministically transforms state \( s \) into: \( s_1 \) to \( s_n \) under condition \( \text{stmt}_1 \) to \( \text{stmt}_n \), an exception and a resulting state \( s_0 \) under condition \( \text{stmt}_0 \), or an error and a resulting state \( s' \) under condition \( \text{stmt}' \). As previously mentioned, states with inconsistent path condition are ignored. The essence of lazy, lazier, and lazier\# algorithms is captured in the semantics of getField. If the field is defined, all three algorithms are the same; it just returns the
value mapped by the field. Also, all three algorithms behave the same if the field value is undefined and the field is of primitive type; that is, the field is initialized to a new primitive symbolic. When the field is undefined and the type of the field is non-primitive type, the three algorithms differ as follows.

For the lazy algorithm, the field is initialized to null, an existing heap object with a compatible type (using the collect function), or a fresh new symbol with object bound decreased by one if the object bound is greater than 0. For the lazier algorithm, the field is initialized to null or a fresh symbolic reference with object bound decreased by one. Since the lazier algorithm introduces symbolic references, the getfield rule needs to consider the case that a field from a symbolic reference is accessed. The lazier semantics defines a 2-step semantics for this case: (1) the symbolic reference is initialized to an existing heap object with compatible type or a new symbolic object shown in init-sym-ref function (note that the program counter is not changed), and (2) regular (without symbolic) getfield rule is
applied. For the \texttt{lazier\# algorithm}, the field is initialized to a symbolic value. If there is field access from a symbolic value, a 3-step semantics rule is defined: (1) the symbolic value is initialized to either \texttt{null} or a fresh symbolic reference by \texttt{init-sym-val} function; steps (2) and (3) correspond to the (1) and (2) steps in lazier initialization algorithm. Note that the rule does not use the NV optimization (it is simple to include it, but it makes the rule a bit complicated). Below are the relative soundness and completeness propositions relating the lazier\# and lazier algorithms (detailed proofs are available at [22]). Note that \( \mathbb{R}' \) is a binary relation on lazier symbolic states and lazier\# symbolic states; \( a \mathbb{R}' b \) iff the concretization of lazier\# state \( b \) to a set of lazier symbolic states includes state \( a \).

**Proposition 1** (Soundness). Given any lazier symbolic trace \( a_1 \rightarrow_{s} a_2 \rightarrow_{s} \cdots \rightarrow_{s} a_n \), there is a corresponding lazier\# symbolic trace \( b_1 \rightarrow_{s} b_2 \rightarrow_{s} \cdots \rightarrow_{s} b_n \) such that \( a_k \mathbb{R}' b_k \) for all \( 1 \leq k \leq n \).

**Proposition 2** (Completeness). Given any lazier\# symbolic trace \( b_1 \rightarrow_{s} b_2 \rightarrow_{s} \cdots \rightarrow_{s} b_n \), there is a corresponding lazier symbolic trace \( a_1 \rightarrow_{s} a_2 \rightarrow_{s} \cdots \rightarrow_{s} a_n \) such that \( a_k \mathbb{R}' b_k \) for all \( 1 \leq k \leq n \).

**Optimality:** Our experiment data in the next section confirms that the lazier\# algorithm is significantly faster than the lazier algorithm when analyzing complex data structures. Furthermore, the counting arguments of the previous sections show that, for several complex data structures, the abstract heap characterization generated by the lazier\# algorithm is optimal in the sense that it does not generate heap shapes that are overly concrete – the cases of heap configurations that the algorithm generates match exactly the number of cases produced by using the generating function technique described in the previous section.

5 Evaluation

To evaluate the effectiveness of lazier\# algorithm, we have performed a comparative study on twenty three examples. Most examples are data structure and algorithm examples taken from the \texttt{java.util} package and the data structure book [20]. Table 1 shows the excerpts of the experiment results (we have included data for the most complex examples – complete results including statement and branch coverage information can be found on the Bogor website [22]). All the experiments are conducted in a 2.4GHz Opteron Linux workstation with 512MB Java heap. Recall that Kiasan performs a per-method analysis (similar to ESC/Java), and moreover, a bound of \( k = 2 \) is almost always sufficient for achieving 100% multiple condition coverage (MCC), so the results indicate the feasibility of Kiasan in actual development for code similar to these examples.

We present comparison among lazy/lazier/lazier\# algorithms on number of states, number of cases, total running time, and theorem prover time. The number of “cases” corresponds to the number of paths (number of post-states) in the symbolic computation tree. In general, lazier\# is better than lazier which in turn is better than lazy in terms of smaller numbers of states, smaller numbers of cases, and shorter total running/theorem prover times. However, there are some anomalies in the running time and theorem prover time comparison. For example, for all AvlTree methods with \( k = 3 \), the lazier\# takes more total running time and theorem prover time than lazier. This is because (for reasons unknown to us) the underlying theorem prover, CVC Lite, takes more time under the lazier\# initialization algorithm for this example (even though lazier\# invokes the theorem prover fewer times). If we examine the difference between total time and theorem prover time which is actual running time of the algorithms, it follows the general trend: lazier\# takes less time than lazier.

Given our case analysis in Section 3, we are most interested in the number of cases explored. We have three observations about the numbers of cases:

**Observation 1:** for some examples, such as AVL tree, the numbers of cases are the same for lazier and lazier\#. This is because in the example, \texttt{null} is not allowed for tree elements and this confirms our previous observation that lazier initialization is optimal for non-\texttt{null} data.

**Observation 2:** for lazier\#, the numbers of cases of binary search tree, AVL tree, and red-black tree insertion match exactly with the numbers calculated by the combinatorics technique discussed in Section 3 – thus establishing that the algorithm is case-optimal for these examples (which are the most complicated ones in our example pool).

**Observation 3:** all the numbers of cases for search/insertremove operations are the same for each tree (binary search tree, AVL tree, red-black tree) under the lazier\# algorithm. This is because the search/inserteremove operations that involve finding the position for the element first and the rest operations (inserting or removing tree node and then rebalance the tree) are deterministic. So the calculations for insertion are applicable to the search and remove operations.

**Conclusion:** The omitted data follows the same trends as presented in Table 1. For the most complicated examples, the lazier/lazier\# algorithms have been able to produce dramatic performance improvements over JPF’s lazy algorithm (e.g., moving from 5-7hrs down to 1-2mins for \texttt{TreeMap}).

6 Related Work

Throughout the paper we have contrasted our approach to others, thus we limit ourselves to a concise discussion
<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>States</th>
<th>Time(s)</th>
<th>Cases</th>
<th>Time(s)</th>
<th>Total Time</th>
<th>Theorem Prover Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lazy-Tree/Map</td>
<td>find</td>
<td>0.929</td>
<td>2099</td>
<td>12</td>
<td>6</td>
<td>6.0</td>
<td>1.4</td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>0.245</td>
<td>755</td>
<td>3</td>
<td>4</td>
<td>5.1</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>push</td>
<td>0.021</td>
<td>201</td>
<td>4</td>
<td>12</td>
<td>1.4</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>pop</td>
<td>0.027</td>
<td>345</td>
<td>4</td>
<td>4</td>
<td>3.1</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>get</td>
<td>0.021</td>
<td>1199</td>
<td>6</td>
<td>4</td>
<td>2.5</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>remove</td>
<td>0.021</td>
<td>1199</td>
<td>6</td>
<td>4</td>
<td>2.5</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>lastKey</td>
<td>0.021</td>
<td>755</td>
<td>3</td>
<td>3</td>
<td>3.1</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>add</td>
<td>0.021</td>
<td>755</td>
<td>3</td>
<td>3</td>
<td>3.1</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>indexO</td>
<td>0.021</td>
<td>755</td>
<td>3</td>
<td>3</td>
<td>3.1</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>removeElementAt</td>
<td>0.021</td>
<td>755</td>
<td>3</td>
<td>3</td>
<td>3.1</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>java.util.List</td>
<td>find</td>
<td>0.021</td>
<td>755</td>
<td>3</td>
<td>3</td>
<td>3.1</td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>0.021</td>
<td>755</td>
<td>3</td>
<td>3</td>
<td>3.1</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>push</td>
<td>0.021</td>
<td>755</td>
<td>3</td>
<td>3</td>
<td>3.1</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>pop</td>
<td>0.021</td>
<td>755</td>
<td>3</td>
<td>3</td>
<td>3.1</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>get</td>
<td>0.021</td>
<td>755</td>
<td>3</td>
<td>3</td>
<td>3.1</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>remove</td>
<td>0.021</td>
<td>755</td>
<td>3</td>
<td>3</td>
<td>3.1</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>lastKey</td>
<td>0.021</td>
<td>755</td>
<td>3</td>
<td>3</td>
<td>3.1</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>add</td>
<td>0.021</td>
<td>755</td>
<td>3</td>
<td>3</td>
<td>3.1</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>indexO</td>
<td>0.021</td>
<td>755</td>
<td>3</td>
<td>3</td>
<td>3.1</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>removeElementAt</td>
<td>0.021</td>
<td>755</td>
<td>3</td>
<td>3</td>
<td>3.1</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Table 1. Experiment Data (excerpts); s – seconds; m – minutes; h – hours

here. TestEra [14] and Korat [3] generate non-isomorphic complex heap structures a priori instead of using lazy initialization algorithm (thus, they are less efficient). In addition, they focus on generating heap structure configurations without scalar data. The closest work to ours is the symbolic execution engine of JPF [11] which originally inspired our work (i.e., the lazy initialization algorithm). We have demonstrated that lazier# algorithm significantly reduces analysis cost compared to the lazy algorithm, and even compared to our lazier algorithm introduced in [6]. Other approaches such as XRT [10] and CUTE [19] use a logical heap representation, and they depend on theorem provers to decide assertions. Recent work on the KeY system uses symbolic execution and dynamic logic [2]. While using a logical representation does not introduce explicit non-deterministic choices when considering aliasing cases similar to Kiasan, such non-deterministic choices are still done by the underlying theorem prover. Currently, there is no conclusive rigorous empirical studies that compare the two main approaches (explicit vs. logical heap representation). In contrast, Kiasan focuses on automatic reasoning of strong heap-oriented properties similar to [17]. In addition, we mentioned before that Kiasan’s stateless search is easily parallelizable, thus, it can leverage the recent trend in multi-core processor architectures. Furthermore, we focus more on establishing strong heap coverage as described in Section 3, while they focus on branch coverage; based on our empirical studies, k = 2 is often enough to reach 100% feasible multiple condition coverage (MCC) [7].

In contrast to all of the work cited above except [2], Kiasan fully supports the DBC methodology. In this respect, ESC/Java-like frameworks [9, 4, 1] are the most popular contract-based checking tools for object-oriented programs based on weakest precondition calculi. One limitation of this approach is that it is difficult to generate counterexamples for contract violations. Recent work on [5] 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 alarms (if not, the tool has found a test case illustrating the error). 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). It is much simpler in our case to generate counter-examples (or even test cases), because the lazy initialization algorithms generate almost concrete graphs that can be directly leveraged to generate...
concrete pre-/post-states illustrating different computation paths (including error paths). We believe that Kiasan provides an alternative solution for contract-based static checking framework that is able to reason about strong heap-oriented properties, while the work presented in this paper takes us further in term of reducing the analysis cost.

7 Conclusion and Future Work

Symbolic execution techniques provide a promising formal foundation for automatically checking interface contracts and class invariants that state strong properties over heap data as required in modern software engineering approaches. To continue to push these techniques toward practical tools that can be integrated software development contexts, we have presented a case counting method to quantify heap coverage for their evaluation. We illustrated the lazier initialization algorithm we introduced in [6] (as well as early lazy initialization algorithm [11]) are sub-optimal on some complex data structures such as the red-black tree implementation in java.util.TreeMap. We described the lazier# initialization algorithm that addressed the inefficiency of the lazier algorithm, and demonstrated that it is optimal on those data structures with respect to the counting method. We also presented empirical case studies to demonstrate the effectiveness of lazier# compared to the lazy and the lazier initialization algorithms.

Moving forwards, we plan to investigate more efficient algorithms for symbolic execution. While the lazier# algorithm is optimal for complex data structures that we used for experiments, we have yet to show that it is optimal in general. In addition, we have presented a case counting method for several complex tree structures; we plan to investigate how it can be applied to arbitrary (cyclic) heap shapes. We also plan to conduct empirical case studies to compare graph-based and logical-based heap representations, as well as experimenting with hybrid graph and logical representations.

References

Bibliography


Appendix A

Data Table and Swap States
<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>States</th>
<th>Cases</th>
<th>Total Time</th>
<th>Theorem Prover Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Lazy</td>
<td>Lazier</td>
<td>Lazy</td>
<td>Lazier#</td>
</tr>
<tr>
<td>AP</td>
<td>partition</td>
<td>1</td>
<td>0.2s</td>
<td>0.2s</td>
<td>0.2s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0.9s</td>
<td>0.5s</td>
<td>0.5s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.0s</td>
<td>2.0s</td>
<td>1.2s</td>
</tr>
<tr>
<td>GC</td>
<td>Mark</td>
<td>1</td>
<td>32.2s</td>
<td>30.8s</td>
<td>32.6s</td>
</tr>
<tr>
<td>LL</td>
<td>merge</td>
<td>1</td>
<td>0.8s</td>
<td>0.5s</td>
<td>0.4s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0.9s</td>
<td>0.8s</td>
<td>0.6s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.2s</td>
<td>2.2s</td>
<td>2.4s</td>
</tr>
<tr>
<td>Sort</td>
<td>insertSort</td>
<td>1</td>
<td>0.4s</td>
<td>0.2s</td>
<td>0.2s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0.9s</td>
<td>0.8s</td>
<td>0.6s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.2s</td>
<td>2.2s</td>
<td>2.4s</td>
</tr>
<tr>
<td>StackA</td>
<td>push</td>
<td>1</td>
<td>0.6s</td>
<td>0.2s</td>
<td>0.2s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0.4s</td>
<td>0.2s</td>
<td>0.3s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>0.8s</td>
<td>0.2s</td>
<td>0.2s</td>
</tr>
<tr>
<td>StackL</td>
<td>pop</td>
<td>1</td>
<td>0.3s</td>
<td>0.2s</td>
<td>0.2s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0.3s</td>
<td>0.2s</td>
<td>0.2s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>0.5s</td>
<td>0.2s</td>
<td>0.2s</td>
</tr>
<tr>
<td>TreeMap</td>
<td>get</td>
<td>1</td>
<td>4.2s</td>
<td>1.5s</td>
<td>1.0s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>16.0s</td>
<td>7.6s</td>
<td>7.4s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>7.0s</td>
<td>3.1m</td>
<td>2.0m</td>
</tr>
<tr>
<td></td>
<td>remove</td>
<td>1</td>
<td>1.4s</td>
<td>1.1s</td>
<td>0.7s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>16.0s</td>
<td>9.9s</td>
<td>5.7s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>5.1s</td>
<td>7.4m</td>
<td>1.9m</td>
</tr>
<tr>
<td></td>
<td>lastKey</td>
<td>1</td>
<td>0.7s</td>
<td>0.4s</td>
<td>0.3s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>7.7s</td>
<td>2.1s</td>
<td>2.4s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>27.0m</td>
<td>28.1s</td>
<td>26.4s</td>
</tr>
<tr>
<td></td>
<td>put</td>
<td>1</td>
<td>3.4s</td>
<td>1.6s</td>
<td>0.9s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>34.7s</td>
<td>14.2s</td>
<td>6.8s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>&gt;24h</td>
<td>11.0m</td>
<td>1.9m</td>
</tr>
<tr>
<td>TCF</td>
<td>classify</td>
<td>1</td>
<td>0.9s</td>
<td>0.8s</td>
<td>0.6s</td>
</tr>
<tr>
<td></td>
<td>add</td>
<td>1</td>
<td>2.0s</td>
<td>1.3s</td>
<td>0.6s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>6.5s</td>
<td>3.7m</td>
<td>1.8s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>21.0s</td>
<td>6.0s</td>
<td>1.3s</td>
</tr>
<tr>
<td></td>
<td>ensureCapacity</td>
<td>1</td>
<td>2.3s</td>
<td>2.0s</td>
<td>0.8s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>5.5s</td>
<td>4.0s</td>
<td>1.6s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>14.2s</td>
<td>5.3s</td>
<td>2.8s</td>
</tr>
<tr>
<td>Vector</td>
<td>insertElementAt</td>
<td>1</td>
<td>3.4s</td>
<td>1.8s</td>
<td>0.9s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>13.2s</td>
<td>5.1s</td>
<td>0.9s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.5m</td>
<td>15.9s</td>
<td>3.2s</td>
</tr>
<tr>
<td></td>
<td>lastIndexOf</td>
<td>1</td>
<td>0.6s</td>
<td>0.7s</td>
<td>0.8s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1.9s</td>
<td>1.5m</td>
<td>1.3s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>6.9s</td>
<td>3.7s</td>
<td>3.4s</td>
</tr>
<tr>
<td></td>
<td>indexOf</td>
<td>1</td>
<td>0.9s</td>
<td>1.6s</td>
<td>1.0s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2.1s</td>
<td>2.0s</td>
<td>1.1s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>4.5s</td>
<td>4.1s</td>
<td>0.5s</td>
</tr>
<tr>
<td></td>
<td>removeElementAt</td>
<td>1</td>
<td>0.8s</td>
<td>0.5s</td>
<td>0.6s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1.0s</td>
<td>0.5s</td>
<td>0.7s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1.2s</td>
<td>0.7s</td>
<td>0.7s</td>
</tr>
<tr>
<td>Class Method</td>
<td></td>
<td></td>
<td>abs</td>
<td>lazy</td>
<td>Lazier</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>ABS</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>find</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>insert</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>findMax</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>remove</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>insert</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>find</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>insert</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>remove</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>remove</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>addBefore</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>indexOf</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>clear</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>lastIndexOf</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>removeLast</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>toArray</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

| | | | seconds | | minutes | | hours |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| | | | 0.3 | | 0.1 | | 0.1 | | 0.1 | | 0.0 | | 0.0 | |
| | | | 1.7 | | 2.4 | | 3.4 | | 1.1 | | 2.0 | | 3.0 | |
| | | | 5.5 | | 6.1 | | 6.0 | | 4.2 | | 1.4 | | 2.2 | |
| | | | 3.3 | | 2.0 | | 3.2 | | 0.6 | | 0.4 | | 0.8 | |
| | | | 8.0 | | 18.1 | | 54.2 | | 5.5 | | 10.6 | | 46.7 | |
| | | | 0.4 | | 0.4 | | 0.4 | | 0.4 | | 0.4 | | 0.4 | |
| | | | 1.2 | | 0.5 | | 0.7 | | 0.7 | | 0.7 | | 0.8 | |
| | | | 3.3 | | 1.5 | | 1.2 | | 2.0 | | 0.6 | | 0.7 | |
| | | | 0.8 | | 0.5 | | 0.7 | | 0.5 | | 0.1 | | 0.1 | |
| | | | 0.7 | | 0.1 | | 0.2 | | 0.2 | | 0.2 | | 0.2 | |
| | | | 2.3 | | 1.9 | | 1.9 | | 1.6 | | 0.9 | | 0.9 | |
| | | | 5.0 | | 15.0 | | 5.3 | | 9.5 | | 1.7 | | 1.7 | |
| | | | 2.4 | | 1.9 | | 0.7 | | 0.5 | | 0.1 | | 0.0 | |
| | | | 1.6 | | 0.9 | | 1.1 | | 0.9 | | 0.9 | | 0.4 | |
| | | | 3.0 | | 2.5 | | 2.4 | | 2.5 | | 0.8 | | 1.0 | |
| | | | 1.0 | | 0.9 | | 0.5 | | 0.5 | | 0.1 | | 0.0 | |
| | | | 0.2 | | 0.2 | | 0.2 | | 0.2 | | 0.2 | | 0.2 | |
| | | | 1.4 | | 1.9 | | 2.0 | | 1.8 | | 2.2 | | 1.3 | |
| | | | 0.0 | | 0.0 | | 0.0 | | 0.0 | | 0.0 | | 0.0 | |
| | | | 1.1 | | 1.5 | | 1.0 | | 0.9 | | 0.3 | | 0.5 | |
| | | | 0.0 | | 0.1 | | 0.0 | | 0.1 | | 0.1 | | 0.1 | |
| | | | 0.2 | | 0.2 | | 0.4 | | 0.2 | | 0.0 | | 0.0 | |
| | | | 1.9 | | 2.0 | | 1.6 | | 0.4 | | 0.6 | | 0.4 | |
| | | | 2.0 | | 0.4 | | 0.1 | | 1.4 | | 1.2 | | 0.2 | |
| | | | 0.2 | | 0.1 | | 0.0 | | 0.0 | | 0.0 | | 0.0 | |
| | | | 2.8 | | 2.3 | | 2.5 | | 1.0 | | 0.5 | | 0.0 | |
| | | | 2.8 | | 2.3 | | 2.5 | | 1.0 | | 0.5 | | 0.0 | |
| | | | 2.8 | | 2.3 | | 2.5 | | 1.0 | | 0.5 | | 0.0 | |
| | | | 2.8 | | 2.3 | | 2.5 | | 1.0 | | 0.5 | | 0.0 | |
| | | | 2.8 | | 2.3 | | 2.5 | | 1.0 | | 0.5 | | 0.0 | |
| | | | 2.8 | | 2.3 | | 2.5 | | 1.0 | | 0.5 | | 0.0 | |
| | | | 2.8 | | 2.3 | | 2.5 | | 1.0 | | 0.5 | | 0.0 | |
| | | | 2.8 | | 2.3 | | 2.5 | | 1.0 | | 0.5 | | 0.0 | |
| | | | 2.8 | | 2.3 | | 2.5 | | 1.0 | | 0.5 | | 0.0 | |
| | | | 2.8 | | 2.3 | | 2.5 | | 1.0 | | 0.5 | | 0.0 | |
| | | | 2.8 | | 2.3 | | 2.5 | | 1.0 | | 0.5 | | 0.0 | |
| | | | 2.8 | | 2.3 | | 2.5 | | 1.0 | | 0.5 | | 0.0 | |

| Table A.1: Experiment Data (2); s – seconds; m – minutes; h – hours |
A.1 Lazy and Lazier Swap States

A.1.1 Lazy Swap States

A.1.2 Lazier Swap States

A.1.3 Lazier# States
Figure A.2: Swap–Lazy States (2)
Figure A.3: Swap–Lazier States

Figure A.4: Swap–Lazier# States
Appendix B

Counting Trees

In this chapter, we count numbers of different binary search trees (including red-black trees, AVL trees), and the numbers of outcomes after certain operations like insert/remove/search one element to the trees. We use a technique called generating functions [8, 2] to facilitate the counting.

Without loss of generality, we assume that the elements contained in tree nodes are integers (\(\mathbb{Z}\)). First we define BST to be the set of all binary search trees and \(BST_n = \{ t \in BST \mid \text{height}(t) < n \}\) for all \(n \in \mathbb{N}\). Then we define two relations.

- \(R : BST \times BST\) as
  
  \[ t_1 R t_2 \iff \exists f : \mathbb{Z} \to \mathbb{Z}, \text{dom } f \supseteq \text{elements}(t_1) \wedge f \text{ is strictly increasing } \wedge f(t_1) = t_2. \text{ (B.1)} \]

  where \(\text{elements}(t)\) returns all the elements of tree \(t\) and \(f(t)\) substitutes elements of \(t\) using \(f\) and keeps the structure of \(t\). The relation \(R\) is an equivalence relation:

  1. reflexivity, let \(f\) be the identity map then we get \(t R t\) for all \(t \in BST\).
  2. symmetry, if we have \(t_1 R t_2\) for some \(f\), we need to show \(t_2 R t_1\). By the property of \(f\) is strictly increasing, \(f\) must be injective. Then we know that \(f^{-1}\) is a partial function and strictly increasing. So we get \(t_2 R t_1\) by \(f^{-1}\).
  3. transitivity, if we have \(t_1 R t_2\) and \(t_2 R t_3\), we need to show \(t_1 R t_3\). Suppose \(f_1\) maps \(t_1\) to \(t_2\) and \(f_2\) maps \(t_2\) to \(t_3\). We can define a function \(f' : \mathbb{Z} \to \mathbb{Z}\) as \(f' = f_2 \circ f_1\) and clearly \(f'\) is strictly increasing. Thus we conclude that \(t_1 R t_3\) by \(f'\).

- \(R' : (BST \times \mathbb{Z}) \times (BST \times \mathbb{Z})\)
  
  \[ (t_1, x_1) R' (t_2, x_2) \iff \exists f : \mathbb{Z} \to \mathbb{Z}, \text{dom } f \supseteq \text{elements}(t_1) \cup \{x_1\} \wedge f \text{ is strictly increasing } \wedge f(x_1) = x_2 \wedge f(t_1) = t_2. \text{ (B.2)} \]

  Similarly, \(R'\) is also an equivalence relation.

So we want to count two things:
1. \(|BST_n/R|\), the number of partitions of binary search trees with heights less than \(n\). Essentially, we count the number of unlabeled binary trees.

2. \(|(BST_n \times \mathbb{Z})/R'|\), the number of partitions of pairs of binary search trees with heights less than \(n\) and integers. Obviously,

\[
(BST_n \times \mathbb{Z})/R' = \bigcup_{T \in BST_n/R} (T \times \mathbb{Z})/R'.
\]

Now we proceed to count \(|(T \times \mathbb{Z})/R'|\) for \(T \in BST_n/R\). We claim that

\[
|(T \times \mathbb{Z})/R'| = 2 \times \text{nodes}(T) + 1,
\]

where \(\text{nodes}(T)\) is the number of nodes of any tree in \(T\). Clearly, the all the trees in \(T\) have the same shape, \(\text{nodes}(T)\) is well-defined. Suppose \(\text{nodes}(T) = k\) and we define a tree \(t \in T\) which has elements: 2, 4, \ldots, 2k. Then define \(P = \{(t, i) \mid 1 \leq i \leq 2k + 1\}\). Clearly, \(P \subset (T \times \mathbb{Z})\). If we can show for all \((t', x) \in (T \times \mathbb{Z}), (t', x) R' p\) for some \(p \in P\), then we can conclude \(|(T \times \mathbb{Z})/R'| = |P| = 2 \times \text{nodes}(T) + 1\). Given any \((t', x) \in (T \times \mathbb{Z})\) and the elements are \(e_1, e_2, \ldots, e_k\) (in increasing order), since \(t', t \in T\), then \(t' R t\), that is, there exists a strictly increasing function \(f\) such that \(f(t') = t\). Then we know \(f(e_i) = 2i\) for all \(1 \leq i \leq k\). If \(x = e_i\) for some \(1 \leq i \leq k\), we get \((t', x) R' (t, i)\) by \(f\). Otherwise, WLOG, suppose \(e_1 < x < e_2\), we define a new function \(f'\) as follows:

\[
f'(y) = \begin{cases} f(y) & \text{if } f(y) \text{ is defined and } y \geq e_2 \text{ or } e_1 \leq y \\ 3 & \text{if } y = x \\ \text{undefined} & \text{otherwise} \end{cases}
\]

Clearly, \(f'\) is strictly increasing. Therefore, we get \((t', x) R' (t, 3)\) by \(f'\). We conclude that

\[
|(BST_n \times \mathbb{Z})/R'| = \sum_{T \in BST_n/R} 2 \times \text{nodes}(T) + 1.
\]

This number is used to count number of outcomes after the search/insert/remove operations. The search/insert/remove operations are similar:

- these operations take in a tree \(t\) and an integer \(x\);
- the most important part of these operations is to find/search the suitable position for \(x\) in \(t\). Suppose \(t\) has \(n\) nodes, there are total \(2n + 1\) positions that include \(n\) nodes and \(n + 1\) nulls. That is, for any binary search tree \(t\) with \(n\) nodes,

\[
|(\{(t, x)\}_{R'} \mid x \in \mathbb{Z})| = 2n + 1.
\]
### B.1 Counting Binary Search Trees

#### B.1.1 Counting Numbers of Binary Search Trees $BST_n/R$

Since we only consider tree shapes but not the labels (elements) of tree nodes, the number of binary search trees with height less than $n$ is the same as the number of binary trees with height less than $n$. Let $a_n$ be the number of binary trees whose heights less than $n$ for $n \geq 0$. We admit empty tree as a legal binary tree with height $-1$. Clearly we have $a_0 = 1$ for only empty tree with height less than $0$. Let consider $a_n$ for $n \geq 1$. Then for each tree of height less than $n$, either it is empty or it non-empty. For a non-empty binary tree, it has a root. The left and right subtrees of the root have heights less than $n - 1$. Therefore, we get

$$a_n = 1 + a_{n-1}^2 \quad n \geq 1 \quad (a_0 = 1). \quad (B.3)$$

We get

- $a_1 = 1 + a_0^2 = 2$
- $a_2 = 1 + a_1^2 = 1 + 2^2 = 5$
- $a_3 = 1 + a_2^2 = 1 + 5^2 = 26$
- $a_4 = 1 + a_3^2 = 1 + 26^2 = 677$

This sequence grows double exponentially. In fact, Aho\cite{Aho} showed $a_n = [k^{2^n}]$ where $[]$ is the nearest integer function and $k = 1.502837 \ldots$

#### B.1.2 Counting $(BST_n \times \mathbb{Z})/R'$

Let $b(m, n)$ be $|\{t \in BST_n \mid t \text{ has } m \text{ nodes }\}|$, binary trees with $m$ nodes and height less than $n$. Clearly $b(0, n) = 1$ for all $n \geq 0$ and $b(m, 0) = 0$ for all $m \geq 1$. Let $c_n = (BST_n \times \mathbb{Z})/R'$. We have $c_n = \sum_{0 \leq i} (2i + 1)b(i, n)$.\footnote{This result allows the duplicated resulting trees and corresponds to the stateless case.} Define a generating function for $b(m, n)$ as

$$T_n(x) = \sum_{m \geq 0} b(m, n)x^m \quad n \geq 0. \quad (B.4)$$

Note from the definition of $b(m, n)$, we can see clearly that $a_n = T_n(1)$ where $a_n$ is the number of binary trees whose heights less than $n$. A non-empty tree with height less than $n$ and $m > 0$ nodes can have $i$ nodes left subtree with height less than $n - 1$ and $m - 1 - i$ nodes right subtree with height less than $n - 1$ for any $0 \leq i \leq m - 1$. Thus we get

$$b(m, n) = \sum_{i+j=m-1} b(i, n-1)b(j, n-1) \quad m > 0, n \geq 1 \quad (B.5)$$
After multiplying \( x^m \) to both sides of (B.5) and summing over \( 1 \leq m \leq \infty \), we get

\[
T_n(x) = x[T_{n-1}(x)]^2 + 1 \quad n \geq 1. \tag{B.6}
\]

Since \( b(0, 0) = 1 \) and \( b(m, 0) = 0 \) for \( m > 0 \), we have \( T_0(x) = 1 \). Using recurrence (B.6), we can get \( T_1(x) = 1 + x \), \( T_2(x) = 1 + x + 2x^2 + x^3 \), etc. From the definition of \( a_n \) and \( b(m, n) \), we know \( a_n = \sum_{m \geq 0} b(m, n) \). Thus \( a_n = T_n(1) \) and for \( x = 1 \) (B.6) becomes (B.3) as expected.

Next, define generating function

\[
G_n(x) = \sum_{m \geq 0} (2m + 1)b(m, n)x^m \quad n \geq 0. \tag{B.7}
\]

Then

\[
G_n(x) = \sum_{m \geq 0} (2m + 1)b(m, n)x^m = 2 \sum_{m \geq 0} mb(m, n)x^m + \sum_{m \geq 0} b(m, n)x^m = 2xT_n'(x) + T_n(x).
\]

Clearly, \( c_n = G_n(1) \). In order to get \( G_n(x) \), we need to calculate \( T_n'(x) \),

\[
T_n'(x) = (x(T_{n-1}(x))^2)' = 2xT_{n-1}(x)T_{n-1}'(x) + (T_{n-1}(x))^2 \quad n \geq 1, T_0'(x) = 0.
\]

We can get

\[
T_1'(1) = 1 \\
T_2'(1) = 2 \times T_1(1)T_1'(1) + (T_1(1))^2 = 4 + 4 = 8 \\
T_3'(1) = 2 \times T_2(1)T_2'(1) + (T_2(1))^2 = 2 \times 5 \times 8 + 5^2 = 105 \\
T_4'(1) = 2 \times T_3(1)T_3'(1) + (T_3(1))^2 = 2 \times 26 \times 105 + 26^2 = 6136 \\
\vdots
\]

Finally, we can calculate \( c_n = G_n(1) \):

\[
c_0 = G_0(1) = 2T_0'(1) + T_0(1) = 1, \\
c_1 = G_1(1) = 2T_1'(1) + T_1(1) = 2 \times 1 + 2 = 4, \\
c_2 = G_2(1) = 2T_2'(1) + T_2(1) = 2 \times 8 + 5 = 21, \\
c_3 = G_3(1) = 2T_3'(1) + T_3(1) = 2 \times 105 + 26 = 236, \\
c_4 = G_4(1) = 2T_4'(1) + T_4(1) = 2 \times 6136 + 667 = 12939, \\
\vdots
\]

11
Numbers of non-isomorphic binary search trees after insert operation  Now we only consider the insert operation and want to find out the number of resulting trees,

\[ f_n = ||\text{insert}(t, x)/R | t \in BST_n \land x \in \mathbb{Z}||, \]  

(B.8)

where \( \text{insert}(t, x) \) is the resulting tree after inserting \( x \) into tree \( t \). The above calculation of \( c_n \) trees contain a lot of duplications. For example, an empty tree is inserted with element 1 will have the same resulting tree as a tree with a single node 1 and inserted with element 1. Clearly \( f_n \) consists of all the input binary tree except the empty tree and the set of trees with one node of depth \( n \). Let \( e_h \) be the total number of new trees (after insertion) with depth \( h \). Since after insertion, the resulting tree can not be empty, we have

\[ f_h = a_h + e_h - 1 \quad h > 1, \]  

(B.9)

and \( f_0 = 1 \).

In order to calculate \( e_h \), we need to count the number binary trees with one node of depth \( n \). Define \( d(h, l) \) as the number of binary trees with height less than \( h \) and having \( l \) nodes with depth \( h - 1 \) for \( h \geq 0, l \geq 0 \). Then \( d(0, 0) = 1, d(0, n) = 0, \) for \( n > 0, d(1, 1) = 1, \) and \( d(h, 0) = a_{h-1} \). Similar to (B.5), Since each node of depth \( h - 1 \) can have a left or right new child, \( e_h = \sum_{l \geq 0} 2l \cdot d(h, l) \) for \( h > 1 \) and \( e_0 = 1, e_1 = 2 \). Then for \( h > 1 \), we have

\[ d(h, l) = \sum_{i+j=l} d(h-1, i)d(h-1, j) \quad h > 1. \]

Define a generating function for \( d(h, l) \)

\[ F_h(x) = \sum_{i \geq 0} d(h, i)x^i. \]

Then we get \( F_h(x) = (F_{h-1}(x))^2 \) for \( h > 1 \) and \( F_0(x) = 1, F_1(x) = 1 + x \). Since each node of depth \( h - 1 \) can have a left or right new child, \( e_h = \sum_{l \geq 0} 2l \cdot d(h, l) \) for \( h > 1 \) and \( e_0 = 1, e_1 = 2 \). Then for \( h > 1, e_h = 2F_h'(1) = 2 \left((1 + x)^{2h-1}\right)'(1) = 2h(1 + 1) = 2^{h+1}. \) Thus

\[ f_0 = 1 + 2 - 1 = 1, \]
\[ f_1 = 2 + 2 - 1 = 3, \]
\[ f_2 = 5 + 8 - 1 = 12, \]
\[ f_3 = 26 + 16 - 1 = 41, \]
\[ f_4 = 677 + 2^5 - 1 = 708, \]
\[ \vdots \]

B.2 Counting Number of Red-black Trees

Red-black tree is a special kind of binary search tree. We will denote \( RBT \) as the set of red-black trees. Clearly, \( RBT \subset BST \). Similarly, we define \( RBT_n \) as the set of red-black trees with heights
less than or equal to $n$. In this section, we consider the kind of red-black trees whose leaf nodes have no element and are treated the same as the empty tree null. We admit the empty tree(null) as a legal red black tree with height 0 and black height 0.

### B.2.1 Counting Number of Red-black Trees $RST_n/R$

Define

$$a(n, k) = |\{ t \mid t \in RBT_n \land blackheight(t) = k \} | / R|$$  \hspace{1cm} (B.10)

as the number of unlabeled red-black trees with height at most $n$ and black height is $k$. Clearly we have $a(0, 0) = 1$ for only the empty tree with height 0 and black height 0 and $a(n, k) = 0$ for $k > n$. If $k = 0$, the only legal red black tree is the empty tree null. Thus $a(n, 0) = 1$ for all $n \geq 0$. Let us consider $a(n, k)$ for $k \geq 1$ and $n \geq k$. By the property of red-black tree, the root of any non-empty red-black tree has to be black. There are four cases according to the colors of the children of the root as shown in Figure B.1:

1. both the left and right children of the root node are black as shown in Figure B.1(a). Then two subtrees have height less than $n - 1$ and black height $k - 1$.

2. left child is black but right child is red as shown in Figure B.1(d). Then two subtrees of left child have height less than or equal to $n - 2$ and black height $k - 1$. Right child of the root has height less than $n - 1$ and black height $k - 1$.

3. right child is red but left child is black as shown in Figure B.1(c). It is symmetric to the black-red case.

4. both left and right children are red as shown in Figure B.1(b). Four grand children of the root have height less than or equal to $n - 2$ and black height $k - 1$.

Therefore, we get

$$a(n, k) = a(n - 1, k - 1)^2 + 2a(n - 1, k - 1)a(n - 2, k - 1)^2 + a(n - 2, k - 1)^4$$

$$= [a(n - 1, k - 1) + a(n - 2, k - 1)^2]^2, \quad n \geq 1, k \geq 1.$$
Figure B.1: Red Black Counting Cases
Here we let \( a(n, k) = 0 \) for \( n < 0 \). Then we get \( a(1, 1) = 1. \)

\[
\begin{align*}
  a(2, 1) &= a(1, 0) + a(0, 0)^2 = (1 + 1^2)^2 = 4. \\
  a(2, 2) &= a(1, 1) + a(0, 1)^2 = (1 + 0)^2 = 1. \\
  a(3, 1) &= a(2, 0) + a(1, 0)^2 = (1 + 1^2)^2 = 4. \\
  a(3, 2) &= a(2, 1) + a(1, 1)^2 = (4 + 1^2)^2 = 25. \\
  a(3, 3) &= a(2, 2) + a(1, 2)^2 = (1 + 0^2)^2 = 1. \\
  a(4, 1) &= a(3, 0) + a(2, 0)^2 = (1 + 1^2)^2 = 4. \\
  a(4, 2) &= a(3, 1) + a(2, 1)^2 = (4 + 4^2)^2 = 400. \\
  a(4, 3) &= a(3, 2) + a(2, 2)^2 = (25 + 1^2)^2 = 676. \\
  a(4, 4) &= a(3, 3) + a(2, 3)^2 = (1 + 0^2)^2 = 1.
\end{align*}
\]

\[ \vdots \]

Let \( b_n = | \text{RBT}_n / R | \) be the number of red-black trees with heights less than or equal to \( n \). Clearly \( b_n = \sum_{k=0}^{n} a(n, k) \). Thus we get

\[
\begin{align*}
  b_0 &= a(0, 0) = 1, \\
  b_1 &= a(1, 0) + a(1, 1) = 2, \\
  b_2 &= a(2, 0) + a(2, 1) + a(2, 2) = 1 + 4 + 1 = 6, \\
  b_3 &= a(3, 0) + a(3, 1) + a(3, 2) + a(3, 3) = 1 + 4 + 25 + 1 = 31, \\
  b_4 &= a(4, 0) + a(4, 1) + a(4, 2) + a(4, 3) + a(4, 4) = 1 + 4 + 400 + 676 + 1 = 1082, \\
  \vdots 
\end{align*}
\]

**B.2.2 Counting \( \text{RBT}_n \times \mathbb{Z} / R' \)**

We will first count the numbers of red-black trees indexed by Height. Define

\[ f(n, h, k) = | \{ t \in \text{RBT}_n \mid \text{blackheight}(t) = k \land \text{leaf}(t) = n \} | / R, \] (B.11)

the number of red-black trees with \( n \) leaf nodes (nulls) and heights less than or equal to \( h \) and black heights equal to \( k \). So we have

\[
\begin{align*}
  f(n, h, k) &= \sum_{i+j=n} f(i, h-1, b-1)a(j, h-1, b-1) + \sum_{i+j+k=n} f(i, h-2, b-1)f(j, h-2, b-1)f(k, h-1, b-1) \\
  &\quad + \sum_{i+j+k=n} f(i, h - 1, b - 1)f(j, h - 2, b - 1)f(k, h - 2, b - 1) \\
  &\quad + \sum_{i+j+k+l=n} a(i, h - 2, b - 1)a(j, h - 2, b - 1)a(k, h - 2, b - 1)a(l, h - 2, b - 1),
\end{align*}
\]

15
for \( n > 0 \). Clearly, we have \( f(1, 0, 0) = 1 \) and \( f(1, h, k) = 0 \) for \((h, k) \neq (0, 0)\) and \( f(n, h, k) = 0 \) for \( h < 0 \) or \( k < 0 \). Define generating function \( F_{h,k}(x) = \sum_{n=1}^{\infty} a(n, h, b)x^n \) for \( k \geq 0 \). Then we get

\[
F_{h,k}(x) = F_{h-1,k-1}^2(x) + 2F_{h-1,k-1}^4(x) = (F_{h-1,k-1}(x) + F_{h-2,k-1}(x))^2.
\]

The boundary condition is

\[
F_{h,0}(x) = \begin{cases} 
  x, & \text{if } h \geq 0; \\
  0, & \text{otherwise.}
\end{cases}
\]

We can get \( F_{1,1}(x) = x^2, F_{2,1}(x) = (x + x^2)^2, F_{3,1}(x) = x^4, F_{3,2}(x) = (x + x^2)^2 + x^4, F_{3,3}(x) = x^8, F_{4,1}(x) + (x + x^2)^2 = x^4 + 2x^3 + x^2, \)

\[
F_{4,2}(x) = (F_{3,1}(x) + F_{2,1}(x)^2)^2 = [(x + x^2)^2 + (x + x^4) + x^4] = x^4 + 4x^5 + 8x^6 + 16x^7 + \\
32x^8 + 48x^9 + 58x^{10} + 68x^{11} + 72x^{12} + 56x^{13} + 28x^{14} + 8x^{15} + x^{16},
\]

\[
F_{4,3}(x) = (F_{3,2}(x) + F_{2,2}(x)^2)^2 = [(x + x^2)^2 + x^4]^2 + x^8 = x^8 + 8x^9 + 32x^{10} + \\
80x^{11} + 138x^{12} + 168x^{13} + 144x^{14} + 80x^{15} + 25x^{16},
\]

and \( F_{4,4}(x) = x^{16} \). Define \( G_h(x) = \sum_{i=0}^{h} F_{h,i}(x) \). So \( [x^n]G_h(x) \) is the number of red-black trees with \( n-1 \) nodes \(^2\) and heights less than or equal to \( h \). Let compute \( G_h(x) = g(h, 0) + g(h, 1)x + g(h, 2)x^2 + \cdots : \)

\[
G_1(x) = x + x^2, \quad G_2(x) = 2x^2 + 2x^3 + x^2 + x, \\
G_3(x) = x^8 + ((x + x^2)^2 + x^4)^2 + (x + x^2)^2 + x \\
= 5x^8 + 8x^7 + 8x^6 + 4x^5 + 2x^4 + 2x^3 + x^2 + x, \\
G_4(x) = 27x^{16} + 88x^{15} + 172x^{14} + 224x^{13} + 210x^{12} + 148x^{11} + \\
90x^{10} + 56x^9 + 33x^8 + 16x^7 + 8x^6 + 4x^5 + 2x^4 + 2x^3 + x^2 + x \\
\vdots
\]

Now we will compute number \( p_h = |(RBT_h \times \mathbb{Z})/R'| \), the total number of outcomes after insert operation for red-black trees with height less than or equal to \( h \) is

\[
p_h = \sum_{i=0}^{\infty} (2i - 1)g(h, i).
\]

\(^2\)This is because for a \( n \) node binary tree, it has \( n + 1 \) null leaves \([5]\).
Then we have \( p_h = 2G'(1) - G(1) \).

\[
\begin{align*}
p_1 &= 2 \times 3 - 2 = 4, \\
p_2 &= 2 \times 17 - 6 = 28, \\
p_3 &= 2 \times (40 + 56 + 48 + 20 + 8 + 6 + 2 + 1) - 31 = 331, \\
P_4 &= 2 \times (27 \cdot 16 + 88 \cdot 15 + 172 \cdot 14 + 224 \cdot 13 + 210 \cdot 12 + 148 \cdot 11 + 90 \cdot 10 + 56 \cdot 9 + 33 \cdot 8 + 16 \cdot 7 \\
&\quad \quad \quad + 8 \cdot 6 + 4 \cdot 5 + 2 \cdot 4 + 2 \cdot 3 + 2 + 1) - 1082 = 2 \cdot 13085 - 1082 = 25088,
\end{align*}
\]

::

### B.3 Counting AVL Trees

AVL Tree [7] is another balanced binary search tree. The structure constraint is that for any node in the tree, the heights of its left subtree and right subtree differ at most by 1. Similar to red-black tree, we treat nulls as legal nodes.

We define \( AVL \) be the set of all AVL tree and \( AVL_n = \{ t \in AVL \mid height(t) = n \} \) for \( n \in \mathbb{N} \). So \( AVL_0 \) is a singleton that only contains the empty tree null.

#### B.3.1 Counting Numbers of AVL Trees \( AVL_n/R \)

Define \( a_n = |AVL_n/R| \), the number of unlabeled AVL tree. For a tree with height \( h \) greater than 0, there are three cases according to heights of its left and right subtrees:

- the heights of the left and right subtrees are the same. So the heights of left and right subtrees must be \( h - 1 \).

- the height of the left subtree is one larger than the height of the right subtree. So the height of left subtree is \( h - 1 \) and right subtree is \( h - 2 \).

- the height of the left subtree is one smaller than the height of the right subtree. So the height of left subtree is \( h - 2 \) and right subtree is \( h - 1 \).

So we get

\[
a_h = a_{h-1}^2 + 2a_{h-1}a_{h-2} \quad h > 0. \tag{B.12}
\]

The boundary condition is \( a_0 = 1 \). So we get

\[
\begin{align*}
a_1 &= a_0^2 = 1 \\
a_2 &= a_1^2 + 2a_0a_1 = 3 \\
a_3 &= a_2^2 + 2a_2a_1 = 15 \\
a_4 &= a_3^2 + 2a_3a_2 = 15^2 + 2 \times 15 \times 3 = 315
\end{align*}
\]
B.3.2 Counting \((AVL_n \times \mathbb{Z})/R'\)

Define \(b(h, n)\) be \(|\{ t \in AVL_n \mid t \text{ has } n \text{ nodes}\}|\) (not counting leaf nodes \textit{null}), AVL trees with \(n\) nodes and height \(h\).

\[
b(h, n) = \sum_{i+j=n-1} b(h-1, i)b(h-1, j) + 2 \sum_{i+j=n-1} b(h-1, i)b(h-2, j) \quad h > 0, n \geq 1. \tag{B.13}
\]

Clearly, we have \(b(0, 0) = 1\) and \(b(0, n) = 0\) for \(n > 0\). Define generating functions

\[
H_h(x) = \sum_{i=0} b(h, i)x^i. \tag{B.14}
\]

We have \(H_0(x) = 1\). We can multiply (B.13) by \(x^n\) and summing over \(1 \leq n \leq \infty\) and get

\[
H_h(x) = xH^2_{h-1}(x) + 2xH_{h-1}(x)H_{h-2}(x). \tag{B.15}
\]

So we get

\[
\begin{align*}
H_1(x) &= x \\
H_2(x) &= x \times x^2 + 2x \times x = x^3 + 2x^2 \\
H_3(x) &= 4x^4 + 6x^5 + 4x^6 + x^7 \\
&\vdots
\end{align*}
\]

Now we will compute number \(c_h = |(AVL_h \times \mathbb{Z})/R'|\), the total number of outcomes after \textit{insert} operation for AVL trees with height equal to \(h\) is

\[
c_h = \sum_{i=0} (2i + 1)b(h, i).
\]

Then we have \(c_h = 2H'_h(1) + H_h(1)\). From (B.15), we get

\[
H'_h(1) = H^2_{h-1}(1) + H'_{h-1}(1)H_{h-1}(1) + 2H'_h(1)H_{h-2}(1) + 2H_{h-1}(1)H'_{h-2}(1).
\]

Then we get \(H'_0(1) = 0, H'_1(1) = 1, H'_2(1) = 7, H'_3(1) = 77\),… Therefore,

\[
\begin{align*}
c_0 &= 2 \times 0 + 1 = 1, \\
c_1 &= 2 \times 1 + 1 = 3, \\
c_2 &= 2 \times 7 + 3 = 17, \\
c_3 &= 2 \times 77 + 15 = 169, \\
&\vdots
\end{align*}
\]

18
Appendix C

Formalization of Kiasan Symbolic Execution

C.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: $\text{sub} : D \times (D \rightarrow D) \rightarrow D$ 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 $\text{sub-fun} : (D' \rightarrow D) \times (D \rightarrow D) \rightarrow (D' \rightarrow D)$ 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 $\text{sub-fun}_1 : (D' \rightarrow D) \times D \times D \rightarrow (D' \rightarrow D)$ as $\text{sub-fun}_1(f, d, d') = \text{sub-fun}(f, (d, d'))$.

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

- the functional substitution function $\text{sub-fun2} : (D' \rightarrow D' \rightarrow D) \times (D \rightarrow D) \rightarrow (D'' \rightarrow D' \rightarrow D)$ 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 $\text{sub-fun2}_1 : (D'' \rightarrow D' \rightarrow D) \times D \times D \rightarrow (D' \rightarrow D' \rightarrow D)$ 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 \( f^{\text{ho}} : D'' \rightarrow D' \rightarrow D \), we have \( \text{sub-fun}(f, g) = \text{sub-fun}(\text{sub-fun}_1(f, d, d'), g), \text{sub-seq}(q, g) = \text{sub-seq}(\text{sub-seq}_1(q, d, d'), g), \text{sub-fun}(f^{\text{ho}}, g) = \text{sub-fun}(\text{sub-fun}_1(f^{\text{ho}}, d, d'), g) \).

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 \( f^{\text{ho}} : D'' \rightarrow D' \rightarrow D \), then for any \( g : D \rightarrow D \), \( \text{sub-fun}(f, g) = \text{sub-fun}(f, g |_{R \cap \text{dom} \ g}), \text{sub-seq}(q, g) = \text{sub-seq}(q, g |_{R \cap \text{dom} \ g}), \text{sub-fun}(f^{\text{ho}}, g) = \text{sub-fun}(f^{\text{ho}}, g |_{R \cap \text{dom} \ g}) \).

C.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.

C.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, \texttt{assume} and \texttt{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 \texttt{INT}, \texttt{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, \( \text{PCs} \)
- the set of boolean expressions, \( \Phi \),
- the set of locations, \( \text{Locs} \),

\(^1\cup\) denotes disjoint union
• the set of natural numbers, \( \mathbb{N} \),
• the set of constants, _Consts_, including \( \mathbb{N} \), _True_, _False_, _null_, etc.,
• the set of fields, _Fields_, including _len_, _def_, _conc_, etc.,
• the set of integer symbols, _Symbols_{\text{INT}}_,
• the set of primitive symbols, _Symbols_{\text{prim}}_, including _Symbols_{\text{INT}}_,
• the set of values, _Values_ = _Consts_ \( \cup \) _Locs_ \( \cup \) _Symbols_{\text{prim}}_,
• the set of indexes, _Indexes_ = _Fields_ \( \cup \) \( \mathbb{N} \) \( \cup \) _Symbols_{\text{INT}}_,
• the set of non-primitive symbols, _Symbols_{\text{non-prim}}_ = \{ X^{m,n}_r | X^{m,n}_r : _Indexes_ \rightarrow _Values_ \},
• the set of symbols, _Symbols_ = _Symbols_{\text{prim}}_ \( \cup \) _Symbols_{\text{non-prim}}_,
• the set of globals, _Globals_ = \{ g | g : _Fields_ \rightarrow _Values_ \},
• the set of operand stacks, _Stacks_ = \{ \omega | \omega : \text{Seq}(_Values_) \}, all sequences of values,
• the set of locals, _Locals_ = \{ l | l : \mathbb{N} \rightarrow _Values_ \},
• the set of heaps, _Heaps_ = \{ h | h : _Locals_ \rightarrow _Symbols_{\text{non-prim}}_ \},
• the set of bytecode instruction with additional _assert_ and _assume_ instructions, _Instrs_,

We follow Java type system in the semantic domains: we use _Types_{\text{prim}}_ to model the primitive types and _Types_{\text{non-prim}}_ for the reference types which are divided into object types (_Types_{\text{record}}_), array types (_Types_{\text{array}}_), and symbolic types (_SymTypes_). _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, _eor_, is introduced to indicate that the end of code array is reached and execution stops. Similar to types, _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 _Consts_ domain. For simplicity, we unify concrete objects and all symbolic values into the _Symbols_ domain. Each member of _Symbols_{\text{non-prim}}_ domain, _X^{m,n}_r, 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^{m,n}_r, is modeled as a partial mapping from its fields to values. Each primitive symbol _X_ or field _f_ also has a property of its type \( \tau \). Since arrays are also modeled by _Symbols_, the domain (_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 _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 new-sym function). Fields of the array include indexes and length, len, (which is always defined). Symbolic arrays and concrete arrays are created using the new-sarr and new-arr functions respectively. Locs represents the set of addresses in the heap.

**State** 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 [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, Types,
- \( pc \) ranges over program counters, PCs,
- \( \phi \) ranges over boolean expressions, \( \Phi \),
- \( i \) and \( j \) range over locations, Locs,
- \( m, n, \) and \( k \) range over natural numbers, \( \mathbb{N} \),
- \( c \) and \( d \) range over constants, Consts,
- \( f \) ranges over fields, Fields,
- \( X, Y, \) and \( Z \) range over symbols, Symbols,
- \( v \) ranges over values, Values,
- \( i \) ranges over indexes, Indexes,
- \( g \) ranges over globals, Globals,
- \( \omega \) ranges over operand stacks, Stacks,
- \( l \) ranges over locals, 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, $\text{default} : \text{Types} \rightarrow \text{Values}$ as $\lambda \tau.v$, where $v$ is the default value of $\tau$;
- fields of a type function, $\text{fields} : \text{Types} \rightarrow \mathcal{P}(\text{Fields})$ as $\lambda \tau.\{f_r \mid f_r$ 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, $\text{acc-idx} : \text{Symbols}_{\text{non-prim}} \rightarrow \mathcal{P}(\mathbb{N} \cup \text{Symbols}_{\text{int}})$ as $\lambda X.\{i \in \mathbb{N} \cup \text{Symbols}_{\text{int}} \mid X(i) \downarrow\}$;
- locations that map to symbolic objects function, $\text{collect} : \text{Heaps} \rightarrow \mathcal{P}(\text{Locs})$ as $\lambda h.\{i \mid h(i)(\text{conc}) \uparrow\}$;
- the set of all symbols in a state function, $\text{symbols} : \Sigma_s \rightarrow \mathcal{P}(\text{Symbols})$ as $\lambda \sigma.\{X \mid X$ appears in $\sigma\}$;
- new primitive symbol function, $\text{new-prim-sym} : \text{Types}_{\text{prim}} \times \mathcal{P}(\text{Symbols}) \rightarrow \text{Symbols}_{\text{prim}}$ as $\lambda (\tau, ss).X_\tau, X \not\in ss$;
- new symbolic type function, $\text{new-sym-type} : \mathcal{P}(\text{Symbols}) \rightarrow \text{SymTypes}$ as $\lambda ss.\tau$ s.t. $\tau \in \text{SymTypes}$ and $\tau$ does not appear in $ss$;
- new array type function, $\text{array-type} : \text{Types} \rightarrow \text{Types}_{\text{array}}$ as $\lambda \tau.\tau'$, where $\tau'$ is the array type of element type $\tau$;
- new symbolic record function, $\text{new-sym} : \mathcal{P}(\text{Symbols}) \times \mathbb{N} \times \mathbb{N} \rightarrow \text{Symbols}_{\text{non-prim}}$ as $\lambda (ss, m, n).X^{m,n}_\tau$, 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, $\text{new-sarr} : \mathcal{P}(\text{Symbols}) \times \mathbb{N} \times \mathbb{N} \rightarrow \text{Symbols}$ as $\lambda (ss, m, n).\text{new-sym}(ss \cup \{X\}, m, n)[\text{LEN} \mapsto X]$ where $X = \text{new-prim-sym}($int, ss$)$;
- new concrete object function, $\text{new-obj} : \mathcal{P}(\text{Symbols}) \times \text{Types}_{\text{record}} \rightarrow \text{Symbols}$ as $\lambda (ss, \tau).X^{0,0}_\tau$, s.t. $X \not\in ss \land \forall f_r \in \text{fields}(\tau).X(f_r) = \text{default}(\tau')$;
- new concrete array function, $\text{new-arr} : \mathcal{P}(\text{Symbols}) \times \text{Types} \times (\mathbb{N} \cup \text{Symbols}_{\text{int}}) \times \mathbb{N} \rightarrow \text{Symbols}$ as $\lambda (ss, \tau, v, n).X^{0,0}_\tau, X \not\in ss \land \tau' = \text{array-type}(\tau) \land \text{dom } X = \{\text{DEF}, \text{LEN}, \text{CONC}\} \land X(\text{DEF}) = \text{default}(\tau) \land X(\text{LEN}) = v$.

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

$$
\frac{\text{pre}}{\sigma \Rightarrow_S \sigma_1[\parallel \sigma_2] \mid \text{Exception}, \sigma'\text{Error}, \sigma''}
$$

23
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 $pre$ 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 xxx#-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 `iadd` adds two integers from the top of the stack and puts result back into the stack. `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**: `new`, `getfield`, `putfield`, `instanceof`, `tau`, and `checkcast` $\tau$ are presented. Accesses to symbolic objects (e.g., `getfield`) 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, `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 `new` $\tau$ instruction during the execution) are reachable through object expansion; concretely, this only happens through assignments.

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

  - Instruction `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 getfield. Rules GETFIELD1-S and GETFIELD7-S are the default behavior of the `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 `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 `null`. GETFIELD4-S lazily initializes a non-primitive field by nondeterministically choosing from existing symbolic objects (with `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 def 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 if_icmplt and if_acmpeq.
  - Instruction if_icmplt compares the top two integral values on the stack. Since the two compared values may be symbolic and thus can not 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 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 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 FALSE, while assert signals an error and terminates the execution.

We use the binding, \( \sigma = (g, pc, l, \omega, h, \phi) \), for all the rules. And \( k \) is used as both the object bound and the array bound.
\[
\text{code}(pc) = \text{getfield} f_\tau \quad \omega = i::\omega'
\]

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

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

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

**PUTFIELD2-S**
\[
\sigma \Rightarrow_S \text{NullPointerException}, (g, pc, l, \omega', h, \phi)
\]
\[
\text{code}(pc) = \text{anewarray} \tau \quad \omega = m::\omega' \quad i \notin \text{dom} h
\]

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

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

**IASTORE1-S**
\[
\sigma \Rightarrow_S \text{ArrayIndexOutOfBoundsException}, (g, pc, l, \omega', h, \phi \cup \{i < 0 \lor i \geq h(i)(\text{LEN})\})
\]
\[
\text{code}(pc) = \text{iastore} \quad \omega = m::\omega' \quad Z = h(i) \quad i' \in \text{acc-idx}(Z)
\]

**IASTORE2-S**
\[
\sigma \Rightarrow_S (g, \text{next}(pc), l, \omega', h[i \mapsto Z[i' \mapsto v]], \phi \cup \{i = i'\})
\]
\[
\text{code}(pc) = \text{iastore} \quad \omega = v::i::\omega'
\]

**IASTORE3-S**
\[
\sigma \Rightarrow_S (g, \text{next}(pc), l, \omega', h[i \mapsto Z^{m,n}[i \mapsto v]], \phi \cup \{i \neq i' \mid i' \in I\})
\]
\[
\cup \{0 \leq i, i < Z(\text{LEN}), |l| < Z(\text{LEN})\}
\]
\[
\text{code}(pc) = \text{iastore} \quad \omega = v::i::\omega'
\]

**IASTORE4-S**
\[
\sigma \Rightarrow_S \text{NullPointerException}, (g, pc, l, \omega', h, \phi)
\]
\[
\text{code}(pc) = \text{iaload} \quad \omega = i::\omega'
\]

**IALOAD1-S**
\[
\sigma \Rightarrow_S \text{ArrayIndexOutOfBoundsException}, (g, pc, l, \omega', h, \phi \cup \{i < 0 \lor h(i)(\text{LEN}) \leq i\})
\]
\[
\text{code}(pc) = \text{iaload} \quad \omega = i::\omega' \quad Z = h(i) \quad i' \in \text{acc-idx}(Z)
\]

**IALOAD2-S**
\[
\sigma \Rightarrow_S (g, \text{next}(pc), l, Z(i')::\omega', h, \phi \cup \{i = i'\})
\]
\[
\text{code}(pc) = \text{iaload} \quad \omega = i::\omega' \quad Z^{m,n} = h(i) \quad I = \text{acc-idx}(Z^{m,n})
\]
\[
\sigma \Rightarrow_S (g, \text{next}(pc), l, v::\omega', h[i \mapsto Z^{m,n}[i \mapsto v]], \phi \cup \{i' \neq i \mid i' \in I\})
\]
\[
\cup \{0 \leq i, i < Z^{m,n}(\text{LEN}), |l| < Z^{m,n}(\text{LEN}), n > 0\}
\]
where \(v = \begin{cases} Z^{m,n}(\text{DEF}) & \text{if } Z^{m,n}(\text{DEF}) \downarrow \\ \text{new-prim-sym}(\text{INT}, \text{symbols}(\sigma)) & \text{if } Z^{m,n}(\text{DEF}) \uparrow \end{cases}\)

**IALOAD4-S**
\[
\sigma \Rightarrow_S \text{NullPointerException}, (g, pc, l, \omega', h, \phi)
\]
\[
\text{code}(pc) = \text{iaload} \quad \omega = i::\omega'
\]
C.2.2 Operational Semantics of Symbolic Execution with Lazier Initialization

First we introduce a new semantic domain: the set of symbolic locations, SymLocs, to model unknown non-null references. We let δ ranges over symbolic locations and each δ_{n}m 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{prim}} \cup \textbf{SymLocs}. We use Σ_{u} 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 Σ_{u} ⊃ Σ_{v}.

3Subscript \( a \) 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.

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

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

- **the *init-loc-heap* function:** \(\textsc{init-loc-heap}(h_a, ss, \delta^m_{\tau}, i) = (h'_a, \phi')\) where
  - if \(i \in \text{dom} h_a\): \(h'_a = \text{sub-fun}_2(h_a, \delta, i)\) and
    \[
    \phi' = \{\tau' <: \tau\} \text{ where } h_a(i) = X_{\tau'}.
    \]
  - if \(i \notin \text{dom} h_a\):
    \[
    \text{dom } h'_a = \text{dom } h_a \cup \{i\}
    \]
    and
    \[
    \forall j \in \text{dom } h_a, h'_a(j) = \text{sub-fun}_1(h_a(j), \delta, i)
    \]
    and \(h'_a(i) = X_{\tau'}\) where
    \[
    \begin{cases}
    \text{new-sarr}(ss, m, k) & \text{if } \tau \in \text{Types}_{\text{array}} \\
    \text{new-sym}(ss, m, k) & \text{if } \tau \in \text{Types}_{\text{record}}
    \end{cases}
    \]
    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,**

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

- **init-sym-loc** function,

\[
\textsc{init-sym-loc}^* = \lambda(\sigma_a, \delta^m_{\tau}).\{\text{init-sym-loc}(\sigma_a, \delta^m_{\tau}, i) \mid i \in \text{collect}(h) \\
\text{or } i \in (\text{Locs} \setminus \text{dom } h) \text{ if } m \geq 0\}.
\]

29
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.

\[
\begin{align*}
\text{IF\_ACMPEQ1-A} & : \quad \text{code}(pc) = \text{if\_acmpeq pc'} \quad \omega = \delta_{r}^{m,n} : \delta_{r}^{m,n} : \omega' \\
& \quad \sigma \Rightarrow_{\mathcal{A}} (g, pc', \omega', h, \phi) \\
\text{IF\_ACMPEQ2-A} & : \quad \text{code}(pc) = \text{if\_acmpeq pc'} \quad \omega = \delta_{r}^{m,n} : \omega' \\
& \quad \sigma \Rightarrow_{\mathcal{A}} \sigma' \quad \text{where } \sigma' \in \text{init-sym-loc}^*(\sigma, \delta_{r}^{m,n}) \\
\text{IF\_ACMPEQ3-A} & : \quad \text{code}(pc) = \text{if\_acmpeq pc'} \quad \omega = \delta_{r}^{m,n} : \omega' \\
& \quad \sigma \Rightarrow_{\mathcal{A}} \sigma' \quad \text{where } \sigma' \in \text{init-sym-loc}^*(\sigma, \delta_{r}^{m,n}) \\
\text{IFNULL-A} & : \quad \text{code}(pc) = \text{ifnull pc'} \quad \omega = \delta : \omega' \\
& \quad \sigma \Rightarrow_{\mathcal{A}} (g, \text{next}(pc), l, \omega', h, \phi) \\
\text{IFNONNULL-A} & : \quad \text{code}(pc) = \text{ifnonnull pc'} \quad \omega = \delta : \omega' \\
& \quad \sigma \Rightarrow_{\mathcal{A}} (g, pc', l, \omega', h, \phi) \\
\text{GETFIELD1-A} & : \quad \text{code}(pc) = \text{getfield } f_{r} \quad \omega = \delta_{r}^{m,n} : \omega' \\
& \quad \sigma \Rightarrow_{\mathcal{A}} \sigma' \quad \text{where } \sigma' \in \text{init-sym-loc}^*(\sigma, \delta_{r}^{m,n}) \\
& \quad \text{code}(pc) = \text{getfield } f_{r} \\
\text{GETFIELD2-A} & : \quad \omega = i : \omega' \quad Y^{m,n} = h(i) \quad Y(f_{r}) \quad \tau \in \text{Types}_{\text{non-prim}} \quad \delta \text{ is fresh} \\
& \quad \sigma \Rightarrow_{\mathcal{A}} (g, \text{next}(pc), l, \delta_{r}^{m-1,k} : \omega', h[i \mapsto Y_{m,n}[f_{r} \mapsto \delta_{r}^{m-1,k}]], \phi)
\end{align*}
\]

C.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 \( \delta_{r}^{m,n} \) 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

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

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:

\[
\text{init-sym-ref} : \Sigma_b \times \text{SymRefs} \times (\text{SymLocs} \cup \{\text{null}\}) \rightarrow \Sigma_b \\
\text{init-sym-ref\#} : \Sigma_b \times \text{SymRefs} \rightarrow \mathcal{P}(\Sigma_b).
\]

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

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

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

and

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

- **init-sym-ref\#** function,

\[
\text{init-sym-ref\#}(\sigma_b, \hat{\delta}) = \{\text{init-sym-ref}(\sigma_b, \hat{\delta}, \delta) | \delta \notin \text{collect-sym-locs}(\sigma_b)\} \\
\cup \{\text{init-sym-ref}(\sigma_b, \hat{\delta}, \text{null})\}.
\]

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 \texttt{if \_acmp eq} and \texttt{getfield} below. Compared to lazier initialization, there is difference in the operational semantics for lazier\# initialization. For instruction \texttt{getfield}, instead of returns a non-deterministic choice between \texttt{null} and a symbolic location, rule GETFIELD2-B just returns a fresh symbolic reference. So, for \texttt{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.

---

\(^4\text{Subscript} \ b \ \text{denotes that the component is a part of lazier\# state.}\)
\[ \text{IF\_ACMPEQ1-B} \quad \begin{array}{l} \text{code}(pc) = \text{if\_acmpeq} \; pc' \quad \omega = \tilde{\delta}^{m,n}_{\tau} :: \hat{\delta}^{m,n}_{\tau} :: \omega' \\ \sigma \Rightarrow \beta \; (g, pc', \omega', h, \phi) \end{array} \]

\[ \text{IF\_ACMPEQ2-B} \quad \begin{array}{l} \text{code}(pc) = \text{if\_acmpeq} \; pc' \quad \omega = v :: \hat{\delta}^{m,n}_{\tau} :: \omega' \\ \sigma \Rightarrow \beta \; \sigma' \quad \text{where } \sigma' \in \text{init-sym-ref}^a(\sigma, \tilde{\delta}^{m,n}_{\tau}) \end{array} \]

\[ \text{IF\_ACMPEQ3-B} \quad \begin{array}{l} \text{code}(pc) = \text{if\_acmpeq} \; pc' \quad \omega = \tilde{\delta}^{m,n}_{\tau} :: \omega' \\ \sigma \Rightarrow \beta \; \sigma' \quad \text{where } \sigma' \in \text{init-sym-ref}^a(\sigma, \tilde{\delta}^{m,n}_{\tau}) \end{array} \]

\[ \text{GETFIELD1-B} \quad \begin{array}{l} \text{code}(pc) = \text{getfield} \; f_{\tau} \quad \omega = \tilde{\delta}^{m,n}_{\tau} :: \omega' \\ \sigma \Rightarrow \beta \; \sigma' \quad \text{where } \sigma' \in \text{init-sym-ref}^a(\sigma, \tilde{\delta}^{m,n}_{\tau}) \end{array} \]

\[ \text{GETFIELD2-B} \quad \begin{array}{l} \text{code}(pc) = \text{getfield} \; f_{\tau} \quad \omega = i :: \omega' \quad Y^{m,n} = h(i) \\ \tau \in \text{Types}_{\text{non-prim}} \quad \hat{\delta} \text{ is fresh} \\ \sigma \Rightarrow \beta \; (g, \text{next}(pc), l, \tilde{\delta}^{m-1,k}_{\tau} :: \omega', h[i \mapsto Y^{m,n}[f_{\tau} \mapsto \hat{\delta}^{m-1,k}_{\tau}]], \phi) \end{array} \]

### C.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 \( \text{new-arr}_c : \mathcal{P}(\text{Symbols}) \times \text{Types} \times \mathbb{N} \to \text{Symbols}_{\text{non-prim}} = \)

\[ \lambda (ss, \tau, m).X_{\tau}, \text{ s.t. } X \notin ss \land \tau' = \text{array-type}(\tau) \land \forall 0 \leq j < m.X_{\tau}(j) = \text{default}(\tau) \land X_{\tau}(\text{LEN}) = m. \]

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.

\[ \text{IADD-C} \quad \begin{array}{l} \text{code}(pc) = \text{iadd} \quad \omega = c :: d :: \omega' \\ \sigma \Rightarrow \beta \; (g, \text{next}(pc), l, (c + d) :: \omega', h, \text{TRUE}) \end{array} \]

\[ \text{IF\_ICMPLT1-C} \quad \begin{array}{l} \text{code}(pc) = \text{if\_icmplt} \; pc' \quad \omega = d :: c :: \omega' \\ \sigma \Rightarrow \beta \; (g, pc', l, \omega', h, \text{TRUE}) \quad c < d \end{array} \]
\[
\begin{align*}
\text{IF\_ICMPLT\_2\_C} & \quad \text{code}(pc) = \text{if\_icmplt\ } pc' \quad \omega = d::c::\omega' \quad c \neq d \\
\text{IF\_ACMPEQ\_1\_C} & \quad \text{code}(pc) = \text{if\_acmpeq\ } pc' \quad \omega = i::j::\omega' \quad i \neq j \\
\text{IF\_ACMPEQ\_2\_C} & \quad \text{code}(pc) = \text{if\_acmpeq\ } pc' \quad \omega = i::j::\omega' \quad i = j \\
\text{IF\_NULL\_1\_C} & \quad \text{code}(pc) = \text{ifnull\ } pc' \quad \omega = i::\omega' \\
\text{IF\_NULL\_2\_C} & \quad \text{code}(pc) = \text{ifnull\ } pc' \quad \omega = \text{NULL}::\omega' \\
\text{IF\_NON\_NULL\_1\_C} & \quad \text{code}(pc) = \text{ifnonnull\ } pc' \quad \omega = i::\omega' \\
\text{IF\_NON\_NULL\_2\_C} & \quad \text{code}(pc) = \text{ifnonnull\ } pc' \quad \omega = \text{NULL}::\omega' \\
\text{ANE\_ARRAY\_1\_C} & \quad \text{code}(pc) = \text{anewarray\ } \tau \quad \omega = c::\omega' \quad c \geq 0 \quad i \notin dom\ h \\
\text{ANE\_ARRAY\_2\_C} & \quad \text{code}(pc) = \text{anewarray\ } \tau \quad \omega = c::\omega' \quad c < 0 \\
\text{NEW\_C} & \quad \sigma \Rightarrow (g, \text{next}(pc), l, i::\omega, \text{h}[i \mapsto \text{new\_obj}(\text{symbols}(\sigma), \tau)], \text{TRUE}) \\
\text{IA\_STORE\_1\_C} & \quad \text{code}(pc) = \text{iastore} \quad \omega = d::c::i::\omega' \quad c < 0 \lor c \geq h(i)(\text{LEN}) \\
\text{IA\_STORE\_2\_C} & \quad \text{code}(pc) = \text{iastore} \quad \omega = d::c::\text{NULL}::\omega' \\
\text{IA\_STORE\_3\_C} & \quad \sigma \Rightarrow \text{NullPointerException}, (g, pc, l, \omega', h, \text{TRUE}) \\
\text{IA\_LOAD\_1\_C} & \quad \text{code}(pc) = \text{iaload} \quad \omega = c::i::\omega' \quad c < 0 \lor c \geq h(i)(\text{LEN}) \\
\text{IA\_LOAD\_2\_C} & \quad \text{code}(pc) = \text{iaload} \quad \omega = c::\text{NULL}::\omega' \\
\text{IA\_LOAD\_3\_C} & \quad \sigma \Rightarrow \text{NullPointerException}, (g, pc, l, \omega', h, \text{TRUE}) \\
\text{GET\_FIELD\_1\_C} & \quad \text{code}(pc) = \text{getfield\ } f \quad \omega = i::\omega' \\
\text{GET\_FIELD\_2\_C} & \quad \text{code}(pc) = \text{getfield\ } f \quad \omega = \text{NULL}::\omega' \\
\end{align*}
\]
C.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.

C.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 $\gamma_s$ 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.
Definition of $\gamma_s$

Let us start with some definitions:

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

Then we introduce some semantic functions\(^6\) to facilitate the definition of $\gamma_s$.

$$
\begin{align*}
\mathcal{V}_s : \text{Values} &\rightarrow (Env \times \text{Sym}(\text{Locs})) \rightarrow \text{Values}_s \\
\mathcal{O}_s : \text{Symbols}_{\text{non-prim}} &\rightarrow (\Gamma \times Env \times \text{Sym}(\text{Locs})) \rightarrow \mathcal{P}(\text{Symbols}_{\text{non-prim}}) \\
\mathcal{H}_s : \text{Heaps} &\rightarrow ((\Gamma \times Env \times \text{Sym}(\text{Locs})) \rightarrow \mathcal{P}(\text{Heaps}_s)) \\
\mathcal{ST}_s : \Sigma_s &\rightarrow ((\Gamma \times Env \times \text{Sym}(\text{Locs})) \rightarrow \mathcal{P}(\Sigma_c)).
\end{align*}
$$

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

- the $\mathcal{V}_s$ function:
  $$
  \mathcal{V}_s[v][E, \rho] = \text{sub}(\text{sub}(v, E), \rho)
  $$

- the $\mathcal{O}_s$ function:
  $$
  \mathcal{O}_s[X_{\tau}][T, E, \rho] = \{ X'_{\tau} \mid \tau' = \text{sub}(\tau, T) \land \text{mapfields}(X, X'_{\tau}, E, \rho) \},
  $$

  where

  \[
  \text{mapfields}(X, X'_{\tau}, E, \rho) \overset{\text{def}}{=} \forall \tau. X(\tau) \downarrow \implies X'(\tau) = \mathcal{V}_s[X(\tau)][E, \rho], \text{ if } \tau' \in \text{Types}_{\text{record}}
  \]

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

  \[
  X'(\mathcal{V}_s[t][E, \rho]) = \mathcal{V}_s[X(t)][E, \rho] \land (X(\text{DEF}) \downarrow \implies \forall 0 \leq m < X'(\text{LEN})
  \]

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

- the $\mathcal{H}_s$ function\(^7\):

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

\(^6\)From this point on, we use subscript $s$ to denote symbolic state components/domains and $c$ for concrete state components/domains.

\(^7\)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
\[
\forall (i, X) \in h_s. \exists Y \in O_s[X] \langle T, E, \rho \rangle . (\rho(i), Y) \in h_c.
\]

\( \text{well-typed}(h_c) \) 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 \((j, X_i)\) in \( h_s \) \((i = \rho(j)\) and \(X_c \in O_s[X_i] \langle T, E, \rho \rangle \)), and if any field \( f\) of \( X_i \) 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:
\[
ST_s[(g, pc, l, \omega, h, \phi)] \langle T, E, \rho \rangle = \{ (\text{sub-fun}(\text{sub-fun}(g, E), \rho), pc, \text{sub-fun}(\text{sub-fun}(l, E), \rho), \text{sub-seq}(\text{sub-seq}(\omega, E), \rho), h', \text{TRUE}) \mid h' \in H_s[[h]] \langle T, E, \rho \rangle \}.
\]

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

**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 D.1 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

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

where $pc_{init}$ is the start program counter of the method.

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

and each local and global is initialized as follows: if it is primitive type, a symbolic primitive symbol is created; otherwise, it is nondeterministically initialized as a symbolic object with all its fields undefined or 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

$$\gamma_s^*(s) = \begin{cases} 
\gamma_s(\sigma_s), & \text{if } s = \sigma_s \text{ for some } \sigma_s \in \Sigma_s; \\
\{ (\text{Exception}, \sigma_e) \mid \sigma_e \in \gamma_s(\sigma_s) \}, & \text{if } s = (\text{Exception}, \sigma_s) \text{ for some } \sigma_s \in \Sigma_s; \\
\{ (\text{Error}, \sigma_e) \mid \sigma_e \in \gamma_s(\sigma_s) \}, & \text{if } s = (\text{Error}, \sigma_s) \text{ for some } \sigma_s \in \Sigma_s.
\end{cases}$$

### Simulation Relations

To show the relationship between $C$ and $S$, we define a relation.

**Definition 1.** $R \subseteq \Sigma_C \times \Sigma_S$, 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.** $C \triangleleft_R S$.

**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 $\sigma_s' \in S$ 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 $\sigma_c = (g_c, pc, l_c, d :: c :: \omega_c, h_e, \text{True})$, 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 $T, E \nvdash \phi$, $\forall \delta[v_1](E, \rho) = 37$
c, $V_s[v_2](E, \rho) = d$, sub-fun(sub-fun(g, E), \rho) = g_c, sub-fun(sub-fun(l, E), \rho) = l_c, sub-seq(sub-seq(\omega, E), \rho) = \omega_c$, and $h_c \in H_s[h](T, E, \rho)$. Using the rule IADD-S, we get $\sigma_s \rightarrow_S \sigma'_s$ with $\sigma'_s = (g_s, next(pc), l_s, Y :: \omega_s, h_s, \phi \cup \{Y = v_1 + v_2\})$ where $Y$ is fresh. We only need to show $\sigma'_s \in \gamma_s(\sigma_s)$, that is, to find $T', E', \rho'$ such that $\sigma'_s \in ST_s[\gamma_s](T', E', \rho')$. We claim that $T' = T, E' = E[Y \mapsto c + d]$, and $\rho' = \rho$ are the right choice. Since $Y$ is fresh, sub-fun(sub-fun(g, E'), \rho') = sub-fun(sub-fun(g, E), \rho) = g_c, sub-fun(sub-fun(l, E'), \rho') = sub-fun(sub-fun(l, E), \rho) = l_c, sub-seq(sub-seq(\omega, E'), \rho') = sub-seq(sub-seq(\omega, E), \rho) = \omega_c$, and $h_c \in H_s[h](T', E', \rho') = H_s[h](T, E, \rho)$. Furthermore, since $V_s[Y](E, \rho) = c + d = V_s[v_1](E, \rho) + V_s[v_2](E, \rho)$, we get $T, E \models (\phi \cup \{Y = v_1 + v_2\})$. Therefore, $\sigma'_s \in ST_s[\gamma_s](T, E, \rho) \subseteq \gamma_s(\sigma_s)$.

- Rule IF_ICMPLT2-C: Let $\sigma_c = (g_c, pc, l_c, d :: \omega_c, h_c, \text{true})$, then $c \geq d$ and $\sigma'_c = (g_c, next(pc), l_c, \omega_c, h_c, \text{true})$. Suppose $\sigma_c \mathcal{R} \sigma_s$. We need to show that there exists $\sigma'_s \in S$ such that $\sigma_s \rightarrow_S \sigma'_s$ and $\sigma'_s \mathcal{R} \sigma'_c$. 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 $\langle g_c, pc, l_c, \omega_c, h_c, \phi \rangle$ for some $T, E, \rho$ with $T, E \models \phi$. $V_s[v_1](E, \rho) = c$, $V_s[v_2](E, \rho) = d$, sub-fun(sub-fun(g, E), \rho) = g_c, sub-fun(sub-fun(l, E), \rho) = l_c, sub-seq(sub-seq(\omega, E), \rho) = \omega_c$, and $h_c \in H_s[h](T, E, \rho)$. Using the IF_ICMPLT-S, we get $\sigma_s \rightarrow_S \sigma'_s$ with $\sigma'_s = (g_c, next(pc), l_c, \omega_c, h_c, \phi \cup \{v \geq 0\})$ (the first end state). We only need to show $\sigma'_s \in \gamma_s(\sigma'_s)$. 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 $\sigma_c \mathcal{R} \sigma_s$, $\rho^{-1}(i)$ must be fresh in $\sigma_s$ (not in dom $h_c$) too. Thus we get sub-fun(sub-fun(g, E), \rho') = g_c, sub-fun(sub-fun(l, E), \rho') = l_c, and sub-seq(sub-seq(\omega, E), \rho') = \omega_c$. From $c \geq 0$, $V_s[v](E, \rho') \geq 0$, that is, $T, E \models \phi \cup \{v \geq 0\}$. It remains to show $h'_c \in H_s[h](T, E, \rho')$. Clearly well-typed(h'_c) because $i$ is fresh in $h_c$. Then we show that 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 new-arr(c)(\text{symbols}(\sigma_c), \tau, c) \in O_s(new-arr(\text{symbols}(\sigma_c), \tau, X, k)](T, E, \rho')$, we can get contains(h'_c, h'_c, T, E, \rho'). Next we need to show well-formed(h'_c, h'_c, T, E, \rho'). Since new-arr(\text{symbols}(\sigma_c), \tau, X, k)](conc) \downarrow$, symbol new-arr(c)(\text{symbols}(\sigma_c), \tau, c) of entry (i, new-arr(c)(\text{symbols}(\sigma_c), \tau, c)) 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 $f$ whose corresponding field is not defined in $h_c$, 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 $h_c$. 

38
and i is fresh in h_c. Therefore, f can not point to i, that is, symbol new-arr.(symbols(σ_c), τ, c) is well-formed. We get well-formed(h'_c, h'_s, T, E, ρ'). Thus h'_c ∈ H_s[h'_c](T, E, ρ'). Finally, σ'_c ∈ ST_s[σ'_c](T, E, ρ') ⊆ γ_s(σ'_s).

- Rule GETFIELD1-C: Suppose σ_c = (g_c, pc, l_c, i :: ω_c, h_c), then σ'_c = (g_c, next(pc), l_c, v :: ω_c, h_c) where X = h_c(i), v = X(f). Let τ_i be the real type of symbol h_c(i). Suppose σ_c R σ_s. We need to show that exists σ'_s ∈ Σ_s such that σ_s →_S σ'_s and σ'_c R σ'_s. Since σ_c R σ_s, we have σ_c ∈ γ_s(σ_s). The symbolic state σ_s must have the form of (g_s, pc, l_s, l :: ω_s, h_s(φ)) for some T, E, ρ with T, E ⊨ φ, ρ(τ_i) = i, sub-fun(sub-fun(g_s, E), ρ) = g_c, sub-fun(sub-fun(l_s, E), ρ) = l_c, sub-seq(sub-seq(ω_s, E), ρ) = ω_c, and h_c ∈ H_s[h_s](T, E, ρ).

WLOG, assume that the type of f, τ, 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, ρ):

- case v = NULL. We will apply the GETFIELD3-S rule and get σ'_s = (g, next(pc), l, NULL :: ω, h'_s, φ), where h'_s = h_s[i → h_s(i)[f → NULL]]. It suffices to show contains(h_c, h'_s, T, E, ρ) and well-formed(h'_s, T, E, ρ). Since ρ(τ_i) = i and σ_c R σ_s, X ∈ O threw Y(T, E, ρ). Furthermore, since h_c ∈ H_s[h_s](T, E, ρ) and X ∈ O[Z](T, E, ρ) by X(f) = NULL = h_s(i)(f), we get contains(h_c, h'_s, T, E, ρ) and well-formed(h'_s, T, E, ρ) hold. We get h_c ∈ H_s[h'_s](T, E, ρ). Then σ'_c ∈ γ_s(σ'_s).

- case v ∈ ρ(dom h_s) ∧ h_s(ρ^{-1}(v))(conc) ↑. We will apply the rule GETFIELD4-S and get σ'_s = (g, next(pc), l, v' :: ω, h'_s, φ ⊨ {τ' <: τ}) where h'_s = h_s[i → h_s(i)[f → j]] and Z_{τ'} = h_s(v'). We also have v' = ρ^{-1}(v) (v' ∈ collect(h_s)) because v ∈ ρ(dom h_s) and h_s(ρ^{-1}(v))(conc) ↑. Since well-typed(h_c), the type of h_s(v), τ_v, is a subtype of τ. Furthermore, since h_s(v) ∈ O[Z]p(T, E, ρ), we arrive at T ⊨ τ' <: τ. Thus T, E ⊨ φ ⊨ {τ' <: τ}. The rest of the proof is similar to the NULL case.

- case v ∈ Locs ∧ v ∉ ρ(dom h_s). We will apply the rule GETFIELD6-S (because we assume that bound k is sufficient large and m > 0) and get σ'_s = (g, next(pc), l, v :: ω, h'_s, φ ⊨ {τ' <: τ}) where h'_s = h_s[i → h_s(i)[f → j]] and Z_{τ'} = new-sym(symbols(σ), m−1, k). Define ρ' = ρ(j → v) and T' = T[τ' ↔ τ_v]. Since well-typed(h_c), we get τ_v <: τ. Furthermore, since ρ' = ρ(j → v) and T' = T[τ' ↔ τ_v], T' ⊨ τ' <: τ. Thus T', E ⊨ φ ⊨ {τ' <: τ}. Since j is fresh in h_s, sub-fun(sub-fun(g_s, E), ρ') = sub-fun(sub-fun(g_s, E), ρ), sub-fun(sub-fun(l_s, E), ρ') = sub-fun(sub-fun(l_s, E), ρ), and sub-seq(sub-seq(ω_s, E), ρ') = sub-seq(sub-seq(ω_s, E), ρ) hold. It remains to show contains(h_c, h'_s, T, E, ρ) and well-formed(h_c, h'_s, T, E, ρ). Since X ∈ O[Zc](T', E, ρ') and h_c(v) ∈ O[Zc](T', E, ρ'), contains(h_c, h'_s, T, E, ρ) holds. Since v ∉ ρ(dom h_c), h_c(v) is well-formed. Since the new symbol Z_{τ'} 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, ρ) and further, h_c ∈ H_s[h'_s](T, E, ρ'). Therefore, σ'_c ∈ ST_s[σ'_c](T, E, ρ') ⊆ γ_s(σ'_s).

□

Definition 2. ρ_s ⊆ Σ_s × P(Σ_c), as σ_s ρ_s, S_c ⟷ γ_s(σ_s) = S_c.
The relation $R_s$ is left total by definition. Also it is clear that for $\sigma_s \in \Sigma_s, S_c, S_c$ 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 $\sigma_s \in R_s$ it is clear that $S_c \subseteq I_C$ by the definition of $\gamma_s$ function.

**Proposition 2.** $S \prec_{R_s} P(C)$.

**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 $\sigma_s \rightarrow_S \sigma'_s$, $\sigma_s \prec_{R_s} S_c$, and $\sigma'_s \prec_{R_s} S'_c$ 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 $\sigma'_s = (g_s, \text{next}(pc), l_s, Z : \omega_s, h_s, \phi \cup \{Z = v_1 + v_2\})$ where $Z$ is fresh. Suppose $\sigma_s \prec_{R_s} S_c$ and $\sigma'_s \prec_{R_s} S'_c$. 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 $(g'_c, \text{next}(pc), l'_c, c : \omega'_c, h'_c, \text{TRUE})$ with some $T, E, \rho$ such that $T, E \not\vdash \phi \cup \{Z = v_1 + v_2\}$; $V_s[Z][T, E, \rho] = c$, $\text{sub-fun}(\text{sub-fun}(g'_c, E), \rho) = g'_c$, $\text{sub-fun}(\text{fun}(l'_c, E), \rho) = l'_c$, $\text{sub-seq}(\text{seq}(\omega_s, E), \rho) = \omega'_s$, and $h'_c \in \mathcal{H}_s[h_s](T, E, \rho)$. Take $\sigma_c = (g'_c, pc, l'_c, E(X) : E(Y) : \omega'_c, h'_c, \text{TRUE})$. Clearly $\sigma_c \rightarrow_{C} \sigma'_c$. We only need to show that $\sigma_c \in \gamma_s(s_c)$. Since $Z$ is fresh, $T, E \not\vdash \phi$. Thus $\sigma_c \in ST_s[\gamma_s](T, E, \rho) \subseteq \gamma_s(s_c)$.

- **Rule IF-JCOMPLT-S:** $\sigma_s = (g_s, pc, l_s, v_{1} : v_{2} : \omega_s, h_s, \phi)$ and $\sigma'_s = (g_s, \text{next}(pc), l_s, \omega_s, h_s, \phi \cup \{v_2 \geq v_1\})$. (We only consider one end state, the other end state is symmetric.) Suppose $\sigma_s \prec_{R_s} S_c$ and $\sigma'_s \prec_{R_s} S'_c$. 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 $(g'_c, \text{next}(pc), l'_c, c : \omega'_c, h'_c, \text{TRUE})$ with some $T, E, \rho$ such that $T, E \not\vdash \phi \cup \{v_2 \geq v_1\}$; $\text{sub-fun}(\text{sub-fun}(g'_c, E), \rho) = g'_c$, $\text{sub-fun}(\text{fun}(l'_c, E), \rho) = l'_c$, $\text{sub-seq}(\text{seq}(\omega_s, E), \rho) = \omega'_s$, and $h'_c \in \mathcal{H}_s[h_s](T, E, \rho)$. Take $\sigma_c = (g'_c, pc, l'_c, \mathcal{V}_s[v_1][T, E, \rho] : V_s[v_2][T, E, \rho] : \omega'_c, h'_c)$. Clearly $\sigma_c \rightarrow_{C} \sigma'_c$. We conclude that $\sigma_c \in ST_s[\gamma_s](T, E, \rho) \subseteq \gamma_s(s_c)$.

- **Rule ANEWARRAY2-S:** Suppose $\sigma_s = (g_s, pc, l_s, X : \omega_s, h_s, \phi)$, we only consider that non-exceptional end state here. Then $\sigma'_s = (g_s, \text{next}(pc), l_s, i : \omega_s, h_s[i \mapsto \text{new-varr}(\text{symbols}(\sigma_s), \tau, X, k)], \phi \cup \{X \geq 0\})$ where $i$ is fresh. Suppose $\sigma_s \prec_{R_s} S_c$ and $\sigma'_s \prec_{R_s} S'_c$. 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 $(g'_c, \text{next}(pc), l'_c, j : \omega'_c, h'_c, \text{TRUE})$ with some $T, E, \rho$ such that $T, E \not\vdash \phi \cup \{X \geq 0\}$; $\rho(i) = j$, $\text{sub-fun}(\text{fun}(g'_c, E), \rho) = g'_c$, $\text{sub-fun}(\text{fun}(l'_c, E), \rho) = l'_c$, $\text{sub-seq}(\text{seq}(\omega_s, E), \rho) = \omega'_s$, and $h'_c \in \mathcal{H}_s[h_s[i \mapsto \text{new-varr}(\text{symbols}(\sigma_s), \tau, X, k)]](T, E, \rho)$. We need to find a $\sigma_c \in \Sigma_s$ such that $\sigma_c \in \gamma_s(s_c)$ 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 $h'_c \in h'_c \setminus \{j, h'_c(j)\}$ satisfies the above two conditions. Since $T, E \not\vdash \phi \cup \{X \geq 0\}$, $T, E \not\vdash \phi$. To show $\sigma_c \in \gamma_s(s_c)$, it suffices to show that $h'_c \in \mathcal{H}_s[h_s](T, E, \rho)$. Since $\text{new-varr}(\text{symbols}(\sigma_s), \tau, X, k)$ will return a symbol with conc field defined and $h'_c \in \mathcal{H}_s[h_s[i \mapsto \text{new-varr}(\text{symbols}(\sigma_s), \tau, X, k)]](T, E, \rho)$, symbols in $h'_c$ such that their corresponding symbols in $h_s[i \mapsto \text{new-varr}(\text{symbols}(\sigma_s), \tau, X, k)]$ have conc fields not defined or do not have corresponding symbols can not contains $j(\rho(i))$. 40
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 $\sigma'_c = (g_s, \mathsf{next}(pc), l_s, \mathsf{null} :: \omega_s, h'_s, \phi')$ where $h'_s = h_s[i \mapsto h_s(i)[f_i \mapsto \mathsf{null}]]$. 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, $\sigma'_c \in \gamma_s(s'_c)$. Then $\sigma_c = (g'_c, pc, l'_c, \rho(i) :: \omega'_c, h'_c, \phi).$ From $h'_c \in \mathcal{H}_s[h'_c](T, E, \rho)$, it is clear that $\sigma_c \rightarrow_C^* \sigma'_c$. Then it suffices to show $h'_c \in \mathcal{H}_s[h'_c](T, E, \rho)$. Since $h'_c \in \mathcal{H}_s[h'_c](T, E, \rho)$, well-typed($h'_c$), contains($h'_c, h_s, T, E, \rho$), and well-formed($h'_c, h_s, T, E, \rho$) hold. Finally, $\sigma_c \in ST_s[\sigma_s](T, E, \rho) \subseteq \gamma_s(\sigma_s)$.

- **Rule GETFIELD6-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 $\sigma'_c = (g_s, \mathsf{next}(pc), l_s, j :: \omega_s, h'_s, \phi')$ where $h_s(i) = Y^m,n, h'_s = h_s[i \mapsto Y^m,n][f_j \mapsto \mathsf{null}][j \mapsto \mathsf{null}]$. Suppose $\sigma_c \mathcal{R}_c S_c$ 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, $\sigma'_c \in \gamma_s(s'_c)$. Then $\sigma_c = (g'_c, pc, l'_c, v :: \omega'_c, h'_c, \phi)$. From $h'_c \in \mathcal{H}_s[h'_c](T, E, \rho)$, it is clear that $\sigma_c \rightarrow_C^* \sigma'_c$. Suppose $\sigma'_c \in S'_c$, that is, $\sigma'_c \in \gamma_s(s'_c)$. Define $\rho' = \rho[j \mapsto v]$ and $\sigma_c = (g'_c, pc, l'_c, \rho'(i) :: \omega'_c, h'_c, \phi)$. From $h'_c \in \mathcal{H}_s[h'_c](T, E, \rho)$, it is clear that $\sigma_c \rightarrow_C^* \sigma'_c$. Since $T, E \vdash \phi \cup \{\tau' < \tau\}$, $T, E \vdash \phi$. Then it suffices to show $h'_c \in \mathcal{H}_s[h'_c](T, E, \rho')$. Since $h'_c \in \mathcal{H}_s[h'_c](T, E, \rho')$, well-typed($h'_c$) and contains($h'_c, h_s, T, E, \rho$) hold. Since $h'_c(j) = Z$ and $\text{con} \notin \text{dom} Z$, well-formed($h'_c, h_s, T, E, \rho$) 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 $c_k \mathcal{R}_S s_k$ for all $1 \leq k \leq n$. 

41
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 $s_1 \to_S s_2 \to_S \cdots \to_S s_n$ with
$s_1 \in I_S$, there is a corresponding concrete trace $c_1 \to_C c_2 \to_C \cdots \to_C c_n$ 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, $C_1 = \gamma'_1(s_1) \neq \emptyset$. Then we show there exists a trace in $\mathcal{P}(C)$,
$C_1 \to_C C_2 \to_C \cdots \to_C C_n$ such that $s_k \mathcal{R} C_k$ 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 $\to_C$, we get the corresponding concrete trace $c_1 \to_C c_2 \to_C \cdots \to_C c_n$. □

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

Following the outline of Section C.3.1, we relate the lazier initialization symbolic execution in
Section C.2.2 and symbolic execution in Section C.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-procedurely.

Definition of $\gamma_a$

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

$$\Pi = \{ F \mid F : \text{SymLocs} \to \text{Locs} \}.$$  \hspace{1cm} (C.1)

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

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

$$\mathcal{S}_T : a : \Sigma_a \to \Pi \to \mathcal{P}(\Sigma_a).$$

The definitions \footnote{Subscript $a$ is frequently used to indicate a component in the lazier symbolic states.} are listed as follows ($\forall F \in \Pi$).

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

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

$$\land pc(\phi', \phi, h_s, F) \land \phi' \text{ is satisfiable}\},$$

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

$$\forall \delta \in \Delta(h_a(F(\delta))(\text{CONC}) \uparrow);$$
heap : \mathcal{P}(\text{Symbols}) \times \mathcal{P}(\text{SymLocs}) \times \text{Heaps}_a \times \text{Heaps} \times \Pi \rightarrow \text{BOOLEAN}

if and only if

\text{dom } h_s = \text{dom } h_a \cup F(\Delta) \land \forall i \in \text{dom } h_a, h_s(i) = \text{sub-fun}(h_a(i), F)

\land \forall i \in (\text{dom } h_s - \text{dom } h_a). h_s(i) = X_r,

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

\text{pc} : \Phi \times \Phi \times \text{Heaps}_s \times \Pi \times \mathcal{P}(\text{SymLocs}) \rightarrow \text{BOOLEAN}

if and only if \( \phi' \) is the least set of predicates that satisfies following conditions:

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

Note: similar the property of substitution, Lemma 2,

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

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 \( \text{ST}_a \) function (we use binding \( \sigma_a = (g, \text{pc}, l, \omega, h, \phi) \)):

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

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, \( \text{ST}_a \) function return \( \emptyset \) or a singleton too.

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

\( \gamma_a(\sigma_a) = \bigcup_{F \in \Pi} \text{ST}_a[\, \sigma_a \,]|_{F} \).

Properties of \( \gamma_a \)

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

1. \( \delta \in \text{collect-sym-locs}(\sigma_a) \);
2. \( i \notin \text{dom } h \) 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 \( \sigma'_a \in init-sym-loc(\sigma_a, \delta, i) \) 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 \( init-sym-loc \), \( \sigma'_a = (sub-fun_1(g_a, \delta, i), pc, sub-fun_1(l_a, \delta, i), sub-seq_1(\omega_a, \delta, i), h'_a, \phi') \), where \( (h'_a, \phi') = init-loc-heap(h_a, \phi, \text{symbols}(\sigma_a), \delta, i) \). Since \( \sigma_s \in ST_a[\sigma_a](F) \), we have \( g_s = sub-fun(sub-fun_1(g_a, \delta, i), F) \), \( l_s = sub-fun(sub-fun_1(l_a, \delta, i), F) \), and \( \omega_s = sub-seq(sub-seq_1(\omega_a, \delta, i), F) \) 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 \( \text{pc}(\phi_s, \phi', h_s, F, \text{collect-sym-locs}(\sigma'_a)) \) 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 \( \sigma'_a \in init-sym-loc(\sigma_a, \delta, i) \) 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 \( \text{pc}(\phi_s, \phi, h_s, F, \text{collect-sym-locs}(\sigma_a)) \) 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 NULL.

- **transition relation**, \( a \rightarrow_\mathcal{A} a' \iff a \Rightarrow_\mathcal{A} a_2, a_2 \Rightarrow_\mathcal{A} a_3, \ldots, a_n \Rightarrow_\mathcal{A} a' \) 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.

44
• labels, we do not use this part and thus they are ignored.

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

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

**Simulation Relations**

We introduce a relation \( \mathcal{R}' \) between lazier symbolic states \( \Sigma_A \) 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 \( a_0 \in I_A \) such that \( s_0 \mathcal{R}' a_0 \).

**Proposition 5.** \( S \trianglelefteq_{\mathcal{R}'} A \).

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

- Rule IF_ACMPEQ1-S: Let \( \sigma_s = (g_s, pc, l_s, i : \text{j :: } \omega_s, h_s, \phi) \). 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 \( \sigma_a \rightarrow_A \sigma'_a \) and \( \sigma'_s \mathcal{R}' \sigma'_a \). Since \( \sigma_s \mathcal{R}' \sigma_a \), we have \( \sigma_s \in \gamma_a(\sigma_a) \). By Lemma 3, \( \sigma_s \in ST_A[\sigma_a](F) \). After taking the IF_ACMPEQ2-A rule, we get an invisible state \( t_1 = (g'_a, pc, l'_a, i : \text{d_l} :: \omega_a, h_a, \phi') \) with \( t_1 \in \text{init-sym-loc}(\sigma_a, \delta, i) \). By Lemma 1, \( \sigma_a \in ST_A[\sigma_a](F) \). After taking the IF_ACMPEQ1-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, \( \sigma_s \in ST_A[\sigma_a](F) \). 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 \( \sigma'_s \in \gamma_a(\sigma_a) \). Clearly \( \text{sub-fun}(g'_a, F) = \text{sub-fun}(g, F) = g_s, \text{sub-fun}(l'_a, F) = \text{sub-fun}(l, F) = l_s, \) and \( \text{sub-seq}(\omega'_a, F) = \text{sub-seq}(\omega, F) = \omega_a \). By applying Lemma 1 twice. It remains to show that \( (h_s, \phi) \in H_A[(\omega''_a, \phi'')](\text{symbols}(\sigma_a), \text{collect-sym-loc}(\sigma_a), F) \). Since \( \text{symbols}(\sigma_a) = \text{symbols}(t_2) \) and \( \text{collect-sym-loc}(\sigma_a) = \text{collect-sym-loc}(t_2) = \text{collect-sym-loc}(\sigma_a) \setminus \{\delta, \delta'\} \), we get \( (h_s, \phi) \in H_A[(\omega''_a, \phi'')](\text{symbols}(\sigma'_a), \text{collect-sym-loc}(\sigma'_a), F) \). Therefore, \( \sigma'_s \in \gamma_a(\sigma_a) \).

- Rule GETFIELD3-S: Suppose \( \sigma_s = (g_s, pc, l_s, i : \text{v :: } \omega_s, h_s, \phi) \). Then \( \sigma_s' = (g_s, \text{next}(pc), l_s, \text{null :: } \omega_s, h_s, \phi) \) where \( h_s(i) = Y \) and \( h'_s = h_s[i \mapsto Y[i \mapsto \text{null}]] \). Suppose \( \sigma_s \mathcal{R}' \sigma_a \), we need to show there exists \( \sigma'_a \in A \) such that \( \sigma_a \rightarrow_A \sigma'_a \) and \( \sigma'_s \mathcal{R}' \sigma'_a \). Since \( \sigma_s \mathcal{R}' \sigma_a \), we have \( \sigma_s \in \gamma_a(\sigma_a) \). By Lemma 3, \( \sigma_s \in ST_A[\sigma_a](F) \). After taking the GETFIELD1-A rule, we get another invisible state \( t = (g'_a, pc, l'_a, i : \text{v :: } \omega'_a, h'_a, \phi') \) with \( t \in \text{init-sym-loc}(\sigma_a, \delta, i) \). By Lemma 1, \( \sigma_a \in ST_A[\sigma_a](F) \). Finally, we take the rule GETFIELD3-S and get \( \sigma'_a = \)}
(g′, next(pc), l′, a, NULL :: ω′, h′[i ↦ h′(i)[f, ↦ NULL]], φ′). We need to show σ′ ∈ γa(σ′).
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. It is sufficient to show that (h[i ↦ h(i)[f, ↦ NULL]], φ) ∈ H_a(φ, h·[i ↦ h(i)[f − > NULL]], F′) (symbols(σ′), collect-sym-locss(σ′), F).
Since symbols(σ′) = symbols(σ) and σ′ = σ, we need to show that σ′ ∈ γa(σ′).
WLOG, suppose that σ has the form of (g, pc, l, a, NULL :: ω, h, φ) for some F with Ψ(δ) = i and σ ∈ σT_a(σ′, F). After taking the GETFIELD1-A rule, we get an invisible state t = (g′, pc′, l′, a, NULL :: ω′, h′, φ′) with t ∈ init-sym-locss(σ′, δ, i). By Lemma 3, we get σ ∈ σT_a(t′, F). Finally, we can take GETFIELD3-S transition rule and get σ′ = (g′, next(pc), l′, a, NULL :: ω′, h′([i ↦ h′(i)[f, ↦ δ]], φ′)) where δ′ ∈ collect-sym-locss(t′). Let F′ = F(δ′, i − > j). Since δ′ is fresh in t, sub-fun(g, F) = sub-fun(g′, F) = g, 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], φ) ⊆ (h′([i ↦ h′(i)[f, ↦ δ]], φ′)) (symbols(σ′), collect-sym-locss(σ′)). Since we already have h′([i ↦ h′(i)[f, ↦ δ]], F′) belongs to get-well-mapped(collect-sym-locss(t) ∪ {δ′}, h′[i ↦ h′(i)[f, ↦ δ]], F′). Since Z = new-sym(symbols(σ′), m − 1, k) and F′(δ′) = j, we have heap(collect-sym-locss(t) ∪ {δ′}, h′[i ↦ h′(i)[f, ↦ δ]], h[i ↦ h(i)[f, ↦ j], j ↦ Z, φ) ∈ get-well-mapped(collect-sym-locss(t) ∪ {δ′}, h′[i ↦ h′(i)[f, ↦ δ]], F′).
We introduce a new entry (δ′, j) then pc(φ′, φ) ⊆ (δ′, j, h[i ↦ h(i)[f, ↦ j], j ↦ Z, φ) ∈ get-well-mapped(collect-sym-locss(t) ∪ {δ′}, h′[i ↦ h′(i)[f, ↦ δ]], F′).

Next we define a relation.

Definition 5. R′ ⊆ ΣR × P(S), as follows:

σ ∈ R′ S ⇔ γa(σ) = S

Clearly, R′ is left total. Since R′ is right total, then for all σ, if σ ∈ R′ S, then S = 0. Furthermore, for any σ, S ∈ I and σ ∈ R′ S, it is clear that S ⊆ I by the definition of γa function.

Proposition 6. A ≡ R′ P(S).

Proof. It is sufficient to show that for all σ ∈ ΣR, S ∈ P(S) if σ → R′ σ and σ ∈ R′ S, then σ′ R′ S′ then S′ ⊆ S by the definition of γa function.
• Rule if_acmpeq: Suppose, WLOG, $\sigma_a = (g_a, pc, l_a, \delta_r : \sigma_{r'} : \omega_a, h_a, \phi')$. Then by the definition of $\rightarrow_{\mathcal{A}}$, the rule consists of three lazier symbolic transitions rules: IF_ACMPSEQ2-A, IF_ACMPSEQ1-A, and IF_ACMPSEQ1-S or IF_ACMPSEQ2-S. After taking IF_ACMPSEQ2-A rule, we get an invisible state $t_1 = (g_a', pc, l_a', i : \sigma_{r'} : \omega_a, h_a', \phi'')$ for some $i \in \text{Locs}$ and $t_1 \in \text{init-sym-loc}(\sigma_a, \delta, i)$. Then after taking IF_ACMPSEQ1-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_ACMPSEQ1-S rule and get $\sigma_a' = (g_a''', \text{next}(pc), l_a''', \omega_a', h_a', \phi''')$. Suppose $\sigma_a, S_s$ and $\sigma_a' S_s'$. We need to show that $S_s \rightarrow_S S_s'$, which 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, $\sigma_s' \in \gamma_a(s_s')$. Then $\sigma_a'$ must be in the form of $(g_a', \text{next}(pc), l_a', \omega_a', h_a', \phi)$ for some $F$ and $\sigma_a' \in \mathcal{ST}_{\mathcal{A}}[\sigma_a](F)$. 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 $\sigma_s \in \mathcal{ST}_{\mathcal{A}}[\sigma_a](F')$. Since $\delta$ and $\delta'$ do not appear in $\sigma_a'$, thus $t_2$, we have $\sigma_s \in \mathcal{ST}_{\mathcal{A}}[t_2](F')$ by Lemma 2 and property of $\mathcal{H}_a$. By applying Lemma 4 twice, we get $\sigma_s \in \mathcal{ST}_{\mathcal{A}}[\sigma_a](F')$.

• Rule getfield $f_i$: Suppose, WLOG, $\sigma_a = (g_a, pc, l_a, \delta_r : \omega_a, h_a, \phi')$ and $\tau \in \text{Types}_{\text{record}}$. By the definition of $\rightarrow_{\mathcal{A}}$, the transition consists of two lazier symbolic 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 $f$ field is undefined in $h_a'(i)$. We take the GETFIELD2-A rule and get $\sigma_a' = (g_a''', \text{next}(pc), l_a''', \omega_a', h_a', \phi''')$. Where $\delta'$ is fresh in $t$.

Suppose $\sigma_a, S_s$ and $\sigma_a' S_s'$. 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, $\sigma_s' \in \gamma_a(s_s')$. 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 $\sigma_s' \in \mathcal{ST}_{\mathcal{A}}[\sigma_a](F)$. Define $h_s$ as $h$ 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 $\sigma_s \in \mathcal{ST}_{\mathcal{A}}[t](F)$ (then we can apply Lemma 4 with $F[\delta \mapsto i]$). Now we use the definition of $\mathcal{H}_a$ to show $(h_s, \phi_s) \in \mathcal{H}_a[\mathcal{H}_a[h_s', \phi']][\text{symbols}(t), \text{collect-sym-locs}(t), F)$. 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 $h_s'[i \mapsto h_s'(i)[f_t \mapsto \delta']]$, 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 $F(\delta')$:

- $F(\delta') \notin F(\text{collect-sym-locs}(t)) \cup \text{dom} h_s'$: then the entry $(F(\delta'), h_s'(F(\delta'))) is removed from $h_s$. Since $\text{heap}(\text{collect-sym-locs}(\sigma_a), h_s'[i \mapsto h_s'(i)[f_t \mapsto \delta']] \cup \text{dom} h_s', F)$ holds
collect-sym-locs(σ′ₚ) = collect-sym-locs(t) = {δ′}, we have heap(collect-sym-locs(t), hₚ', F) holds.

- otherwise: so the entry (F(δ′), hₚ'(F(δ′))) is not removed from hₚ by the definition of hₚ.

We are done because heap(collect-sym-locs(σ′ₚ), hₚ'[i ↦ hₚ'(i)]{fₓ ↦ δ′}], hₚ', F) holds.

So we have proved (hₚ, φₚ) ∈ ℋₚ[[hₚ, φₚ]](symbols(t), collect-sym-locs(t), F). Thus σₚ ∈ ℳₚ[σₚ](F) holds and further, σₚ′ ∈ Sₚ. It remains to show that σₚ →₂ₚ σₚ′. There are two cases:

- hₚ(F(δ′)) is not defined: Since the σ′ₚ has only δ′ that is not in σₚ, so hₚ'(F(δ′)) is a fresh symbol. We can take the GETFIELDₚ-S rule and get σₚ →₂ₚ σₚ′.

- hₚ(F(δ′)) is defined: By the wellmappedness of hₚ, hₚ(F(δ′))(conc) is not defined. So we can take the GETFIELDₚₚ-S rule and get σₚ →₂ₚ σₚ′.

 Soundness and Completeness

**Proposition 7** (Soundness). Given any symbolic trace s₁ →₂ₛ s₂ →₂ₛ ⋯ →₂ₛ sₙ with s₁ ∈ Iₛ, there is a corresponding lazier symbolic trace a₁ →₂ₐ a₂ →₂ₐ ⋯ →₂ₐ aₙ with a₁ ∈ Iₐ such that sₖ ∈ ℜₚ aₖ for all 1 ≤ k ≤ n.

**Proof.** We proceed by mathematical induction on n using Proposition 9. □

**Proposition 8** (Completeness). Given any lazier symbolic trace a₁ →₂ₐ a₂ →₂ₐ ⋯ →₂ₐ aₙ with a₁ ∈ Iₐ, there is a corresponding symbolic trace s₁ →₂ₛ s₂ →₂ₛ ⋯ →₂ₛ sₙ such that sₖ ∈ ℜₚ aₖ for all 1 ≤ k ≤ n and s₁ ∈ Iₛ.

**Proof.** It is easy to show that there exists a trace in ℙ(S), s₁ = F →₂ₛ s₂ →₂ₛ ⋯ →₂ₛ sₙ such that aₖ ∈ ℜₛ sₖ for all 1 ≤ k ≤ n by mathematical induction on n using Proposition 6. Since sₙ ≠ ∅, we can pick a sₙ ∈ sₙ and use the definition of →₂ₛ, then get the corresponding symbolic trace s₁ →₂ₛ s₂ →₂ₛ ⋯ →₂ₛ sₙ. □

### C.3.3 Relative Soundness and Completeness of Symbolic Execution with Lazier# Initialization

Following the outline of Section C.3.2, we relate the lazier# initialization symbolic execution in Section C.2.3 and lazier symbolic execution in Section C.2.2. First, we define a function γₚ 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 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-procedure.
Definition of $\gamma_b$

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

$$\Xi = \{ G \mid G : \text{SymRefs} \rightarrow (\text{SymLocs} \cup \{\text{null}\}) \}.$$  \hspace{1cm} (C.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 $(g_a, pc', l_a, \omega_a, h_a, \phi_a) = ST_b[\sigma'_b](G)$.

Proof. Suppose $\sigma_b = (g_b, pc, l_b, \omega_b, h_b, \phi_b)$. By the definition of $\text{init-sym-ref}$, $\sigma'_b = (\text{sub-fun}(g_b, \hat{\delta}, v), pc', \text{sub-fun}(l_b, \hat{\delta}, v), \text{sub-seq}(\omega_b, \hat{\delta}, v), \text{sub-fun}2(h_b, \hat{\delta}, v), \phi)$. Since $\sigma_a = ST_b[\sigma_b](G)$, we have $g_a = \text{sub-fun}(\text{sub-fun}(g_b, \hat{\delta}, v), G), l_a = \text{sub-fun}(\text{sub-fun}(l_b, \hat{\delta}, v), G), \omega_a = \text{sub-seq}(\text{sub-seq}(\omega_b, \hat{\delta}, v), G)$, 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. \hfill \Box

Lemma 6. Let $\sigma_b = (g_b, pc, l_b, \omega_b, \phi_b) \in \Sigma_b$ 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. \hfill \Box

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 $\mathcal{B} = (\Sigma_G, I_B, \rightarrow_B, L_B)$ to model the state-space from the lazier# initialization symbolic executions. The components are defined as follows:

- states, $\Sigma_G = \Sigma_b \cup (\text{Exception} \times \Sigma_b) \cup (\text{Error} \times \Sigma_b)$.}

49
• initial states,
\[ I_B = \{ (g_b, p c_{init}, l_b, n i l, h_b, \{T R U E\}) \mid \text{dom}(g_b) = \text{Globals} \land \text{dom}(l_b) = \text{Locals} \}, \]

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, \( b \xrightarrow{B} b' \iff b \xrightarrow{B} b_2, b_2 \xrightarrow{B} b_3, \ldots, b_n \xrightarrow{B} b' \) 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 \( \gamma^*_b : \Sigma_B \rightarrow \mathcal{P}(\Sigma_A) \) as

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

**Simulation Relations**

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

**Definition 6.** \( \sigma_a \mathcal{R}' \sigma_b \iff \sigma_a \in \gamma^*_b(\sigma_b). \)

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

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

**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 \( \sigma_a \mathcal{R}' \sigma_b \) then there exists \( \sigma_b' \in \Sigma_B \) such that \( \sigma_b \xrightarrow{B} \sigma_b' \) and \( \sigma_a \mathcal{R}' \sigma_b' \). We will proceed with the rule induction on \( \xrightarrow{A}. \)

- **Rule if_acmpeq:** Suppose, WLOG, \( \sigma_a = (g_a, p c, l_a, \delta, \delta' :: \delta'' :: \omega_a, h_a, \phi') \). Then by the definition of \( \xrightarrow{A} \), the rule consists of three lazier symbolic transitions rules: IF_ACMEQ3-A, IF_ACMEQ2-A, and IF_ACMEQ1-S or IF_ACMEQ2-S. After taking IF_ACMEQ3-A rule, we get an invisible state \( t_1 = (g'_a, p c, l'_a, i :: \delta'' :: \omega'_a, h'_a, \phi''') \) for some \( i \in \text{Locs} \) and \( t_1 \in \text{init-sym-loc}(\sigma_a, \delta, i) \). Then after taking IF_ACMEQ2-A rule, we get another invisible state \( t_2 = (g''_a, p c, l''_a, i :: j :: \omega''_a, h''_a, \phi'''' \) 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_ACMEQ1-S rule and get \( \sigma''_a = (g''', \text{next}(p c), l''', \omega''', h''', \phi''') \). Suppose \( \sigma_a \mathcal{R}' \sigma_b \). We need to show that there exists any \( \sigma''_b \in \Sigma_B \) such that \( \sigma_b \xrightarrow{B} \sigma''_b \). WLOG, suppose that \( \sigma_b = (g_b, p c, l_b, \hat{\delta} :: \delta' :: \omega_b, h_b, \phi) \). Since \( \sigma_a \mathcal{R}' \sigma_b \), 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_ACMEQ3-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 \( \sigma_a \mathcal{R}' t'_0 \). Then we take
Definition 7. \( R'' \subseteq \Sigma_{\beta} \times \mathcal{P}(\Sigma_{\mathcal{A}}) \), as follows:

\[
\sigma_b R'' S_a \iff \gamma_b(\sigma_b) = S_a
\]

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

Proposition 10. \( \beta \prec_{R''} \mathcal{P}(\mathcal{A}) \).

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

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

- Rule if_acmpeq: Suppose, WLOG, \( \sigma_b = (g_b, pc, l_b, i : \omega_b, h_b, \phi) \). Then by the definition of \( \rightarrow_{\beta} \), the rule consists of five transitions rules: if ACMPEQ3-B, ACMPEQ2-B, ACMPEQ3-A, ACMPEQ2-A, and ACMPEQ1-S or ACMPEQ2-S. After taking ACMPEQ3-B rule, we get an invisible state \( t_1 = init-sym-ref(\sigma_b, \delta_1, \delta_1) \) and then ACMPEQ2-B rule, we get \( t_2 = init-sym-ref(t_1, \delta_2, \delta_2) \). Then by ACMPEQ3-A rule, we get to \( t_3 \) and by ACMPEQ2-A rule, we arrive at \( t_4 \) where \( \delta_1 \) and \( \delta_2 \) are fresh. WLOG, suppose we take ACMPEQ1-S rule and get \( \sigma'_b = (sub-fun_1(sub-fun_1(g_b, \hat{\delta_1}, \hat{\delta_1}), \hat{\delta_2}, \delta_2), next(pc), sub-fun_1(sub-fun_1(l_b, \hat{\delta_1}, \hat{\delta_1}), sub-seq_1(sub-seq_1(\omega, \hat{\delta_1}, \hat{\delta_1}), \delta_2, \hat{\delta_2}), sub-fun_2_1(sub-fun_2_1(h_b, \hat{\delta_1}, \hat{\delta_1}), \hat{\delta_2}, \delta_2), \phi') \). Suppose \( \sigma_b R' S_a \) and \( \sigma_b R' S'_a \). 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, \( \sigma'_a \in \gamma_b(\sigma'_b) \). Then \( \sigma'_a \) must be in the form of \( (g_a', next(pc), l_a', \omega_a', h_a', \phi) \) for some \( G \) and \( \sigma_a' \in ST_b[\sigma_b'](G) \). Define \( G' = G[\hat{\delta_1} \mapsto \delta_1][\hat{\delta_2} \mapsto \delta_2] \). Clearly \( G' \in legal-env(\sigma_b) \). Define \( \sigma_a = ST_b[\sigma_b](G') \). 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, $\sigma_b = (g_b, pc, l_p, \hat{\delta}, \delta', \omega_b, h_b, \phi')$ and $\tau \in \text{Types}_{\text{record}}$. 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 $t_1 = (\text{sub-fun}_1(g_b, \hat{\delta}, \delta), pc, \text{sub-fun}_1(l_p, \hat{\delta}, \delta), \text{sub-seq}_1(\omega_b, \hat{\delta}, \delta), \text{sub-fun}_2(h_b, \hat{\delta}, \delta), \phi)$ for some fresh $\delta$. Then rule GETFIELD1-A is taken and get an invisible state $t_2 = (g_2, pc, l_2, \omega_2, h_2, \phi_2) = \text{init-sym-loc}(t, \delta, i)$ for some $i \in \text{Locs}$. WLOG, assume that $f$ field is undefined in $h_2(i)$. We take the GETFIELD2-B rule and get $\sigma_b' = (g_2, \text{next}(pc), l_2, \hat{\delta}', \omega_2, h_2[i \mapsto h_2(i)[f_t \mapsto \hat{\delta}']], \phi_2)$), where $\hat{\delta}'$ is fresh in $t_2$.

Suppose $\sigma_b R'' S_a$ and $\sigma_b' R'' S_a'$. 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, $\sigma'_a \in \gamma_b(\sigma'_b)$. Then $\sigma_a$ must be in the form of $(g_a, \text{next}(pc), l'_a, \hat{\delta}', \omega'_a, h'_a, \phi)$ for some $G$ such that $G(\hat{\delta}') = \delta'$ and $\sigma'_a \in ST_b[\sigma_b'][G']$. Define $G' = G[\hat{\delta} \mapsto \delta]$. Clearly $G' \in \text{legal-env}(\sigma_b)$. Let $\sigma_a = ST_b[\sigma_b](G')$. 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 $s_a'$. Since $\hat{\delta}'$ is fresh in $t_2$ and $G(\hat{\delta}') = \delta'$, $\delta'$ is fresh in $s_a'$. Therefore, we can apply GETFIELD2-A and get $\sigma_a'$. We conclude that $\sigma_a \rightarrow_\mathcal{A} \sigma_a'$.

\[\square\]

**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 $a_k R' b_k$ for all $1 \leq k \leq n$.

**Proof.** We proceed by mathematical induction on $n$ using Proposition 9.

\[\square\]

**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 $a_k R' b_k$ for all $1 \leq k \leq n$ and $a_1 \in I_\mathcal{A}$.

**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 $b_k R' S_k$ 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 $a_1 \rightarrow_\mathcal{A} a_2 \rightarrow_\mathcal{A} \cdots \rightarrow_\mathcal{A} a_n$.

\[\square\]
Appendix D

Kripke Structures

D.1 Simulation on Kripke Structures

The presentation in this section is adapted from [6], and it is provided here for a quick reference.

Definition 8 (Kripke Structure). A Kripke structure is a triple, \( \mathcal{K} = (\Sigma_K, I_K, \rightarrow_K, L_K) \), where \( \Sigma_K \) is a set of states, \( I_K \) is a set of initial states that \( I_K \subseteq \Sigma_K \), \( \rightarrow_K \subseteq \Sigma_K \times \Sigma_K \) is the transition relation (finite image), and \( L_K : \Sigma_K \rightarrow \mathcal{P}(\text{Atom}) \) associates a set of atomic properties, \( \forall s \in \Sigma_K . L_K(s) \subseteq \text{Atom} \).

Definition 9 (Simulation Relation on Kripke Structures). For Kripke structures \( C = (\Sigma_C, I_C, \rightarrow_C, L_C) \) and \( S = (\Sigma_S, I_S, \rightarrow_S, L_S) \), a binary relation, \( \mathcal{R} \subseteq \Sigma_C \times \Sigma_S \), is a simulation of \( C \) by \( S \), written \( C \triangleleft_{\mathcal{R}} S \), if \( \forall c \in \Sigma_C, s \in \Sigma_S . c \mathcal{R} s \Longleftrightarrow \exists s' \in \Sigma_S . s \rightarrow s' \wedge c' \mathcal{R} s' \) and \( \forall s_0 \in I_C . \exists s_0 \in I_S . c_0 \mathcal{R} s_0 \).

Definition 10 (Left-/Right-total Simulation Relations). A binary relation, \( \mathcal{R} \subseteq S \times T \), is left total if \( \forall s \in S . \exists t \in T . s \mathcal{R} t \). The relation is right total if \( \forall t \in T . \exists s \in S . s \mathcal{R} t \).

Definition 11 (Power Kripke Structure). For a Kripke structure, \( \mathcal{K} = (\Sigma_K, I_K, \rightarrow_K, L_K) \), the power kripke structure \( \mathcal{P}(\mathcal{K}) = (\mathcal{P}(\Sigma_K), \mathcal{P}(I_K), \rightarrow_K, L_{\mathcal{P}(\mathcal{K})}) \), where \( \forall S, S' \subseteq \Sigma_K . S \rightarrow_K S' \) if and only if for every \( s' \in S' \), there exists some \( s \in S \) such that \( s \rightarrow_K s' \) and \( L_{\mathcal{P}(\mathcal{K})}(S) = \cap \{ L_{\mathcal{K}}(s) \mid s \in S \} \).