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,robby,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.

*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.

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 on 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.

```
1 int max(int a, int b) {
2   int z = a;
3   if (b > a) z = b;
4   if (z < a || z < b)
5     assert false;
6   return z;
7 }
```

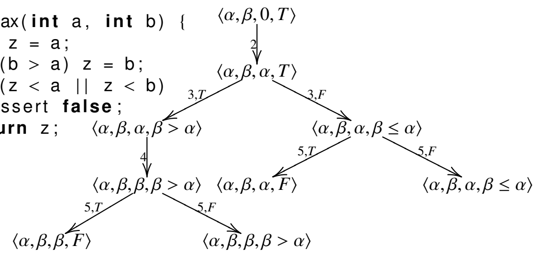


Figure 1. A Symbolic Execution Example

In an accompanying tech report [8], we provide a proof of the relative soundness and completeness of the lazier# 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] (SymExe) 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 $\langle \mathsf{a}, \mathsf{b}, \mathsf{z}, \phi \rangle$ that associates a symbol/concrete value to a, b, z, and predicate ϕ 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 α for a, symbol β for b, 0 for z, and the constraint ϕ 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 $\phi \implies (\mathsf{b} > \mathsf{a})$ and $\phi \implies \neg(\mathsf{b} > \mathsf{a})$ 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 ϕ 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, SymExe is a path-sensitive analysis, and is typically phrased as a depth-first search of a method's execution paths. SymExe 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. SymExe 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.

```
public class Node<E> {
  //@ ensures data == \old(n.data) && n.data == \old(data);
  public void swap(@NonNull Node<E> n)
  { E e = data; data = n.data; n.data = e; }
  private Node<E> next; E data;
}
```

Figure 2. A Swap Example

Handling Objects: While the basic approach to SymExe 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 SymExe 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 SymExe 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

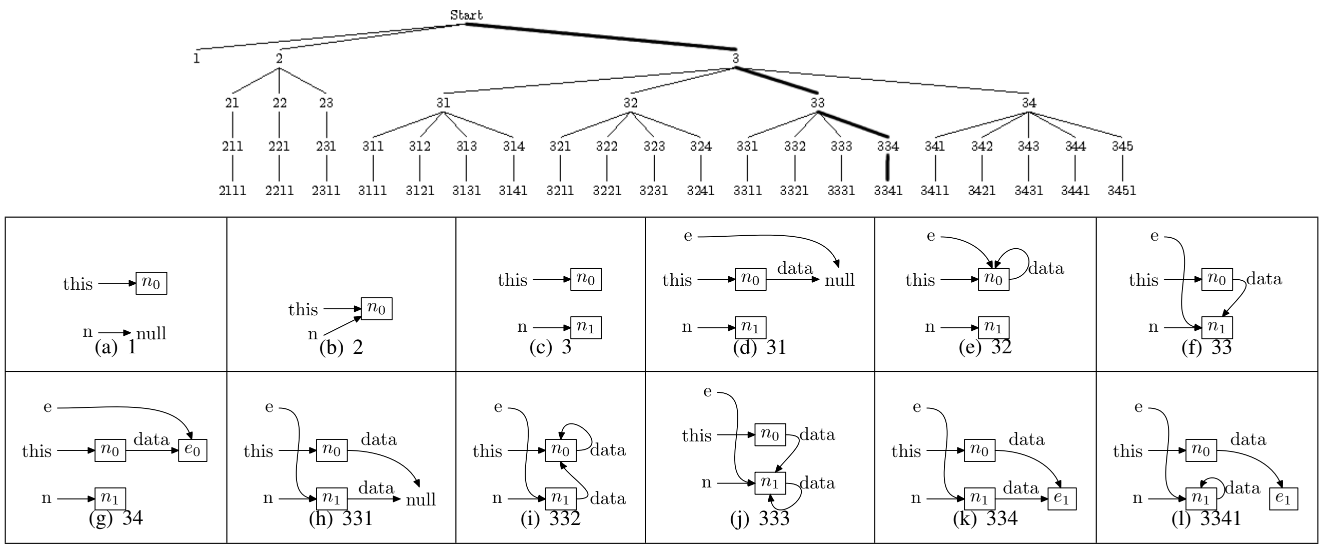


Figure 3. Lazy Symbolic Execution Tree and An Example Trace (3-33-334-3341 and Sibling States)

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 – n_0 (32), n_1 (33), or a fresh symbolic object e_0 (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, n_0, n_1, or a fresh symbolic object e_1 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 postcondition, 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 ˆ. 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

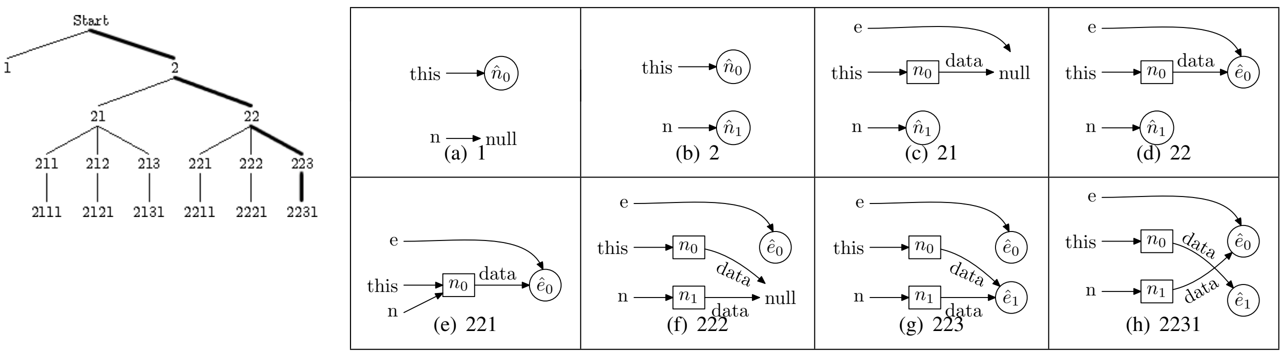


Figure 4. Lazier Symbolic Execution Tree and An Example Trace (2-22-223-2231 and Sibling States)

1 in Figure 4, and State 2 and 3 in Figure 3 are abstracted into State 2 in Figure 4 (i.e., both $\hat{n}_0$ and $\hat{n}_1$ may actually be n_0 or n_1). When $\hat{n}_0$'s data field is read at swap's first statement, $\hat{n}_0$ 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 $\hat{e}_0$ (22). From State 22, there are three possible choices when executing swap's second statement. We first replace $\hat{n}_1$ by non-deterministically 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 non-deterministic 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 $\hat{e}_0$ and $\hat{e}_1$ with symbolic objects when checking the postcondition, as they will be compared against themselves (i.e., data==\old(n.data) iff $\hat{e}_1 = \hat{e}_1$). 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 [18]. 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 $\sum_{m\geq 0} b(m, k)$. Thus, the optimal number of cases for all non-isomorphic input tree heights less than k is $c_k = \sum_{m\geq 0}(2m + 1)b(m, k)$.

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,

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

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 $\langle a_n \rangle$ is $G(x) = \sum_{n=0}^{\infty} a_n x^n$. This definition of $G(x)$ is sometimes called the ordinary generating function of $\langle a_n \rangle$. For example, the generating function for $1, 1, \ldots, 1, \ldots$ is $G(x) = \sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$. Let us define a generating function for $b(m, n)$ as:

$$T_k(x) = \sum_{m=0}^{\infty} b(m, k)x^m \quad k \geq 0. \quad (2)$$

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

$$T_k(x) = x[T_{k-1}(x)]^2 + 1 \quad k \geq 1. \quad (3)$$

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 $T(x)$ to compute the number of cases, $c_k = \sum_{m\geq 0}(2m + 1)b(m, k)$.

$$c_k = \sum_{m\geq 0}(2m+1)b(m, k) = 2\sum_{m\geq 0} mb(m, k)+\sum_{m\geq 0} b(m, k). \quad (4)$$

```
 1 BinaryNode<T> insert(T x, BinaryNode<T> t) {
 2   if (t == null)
 3     t = new BinaryNode<T>(x, null, null);
 4   else if (comparator.compare(x, t.element) < 0)
 5     t.left = insert(x, t.left);
 6   else if (comparator.compare(x, t.element) > 0)
 7     t.right = insert(x, t.right);
 8   else
 9     ; // Duplicate; do nothing
10   return t;
11 }
```

Figure 5. A Binary Search Tree Insertion

In order to calculate $\sum_{m\geq 0} mb(m, k)$, we take the derivative of $T_k(x)$ and get $T_k'(x) = \sum_{m\geq 0} mb(m, k)x^{m-1}$. Hence, $\sum_{m\geq 0} mb(m, k) = T_k'(1)$. Therefore, $c_k = 2T_k'(1) + T_k(1)$. We use this formula to calculate $c_1 = 2T_1'(1) + T_1(1) = 4$.

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[1]. 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.

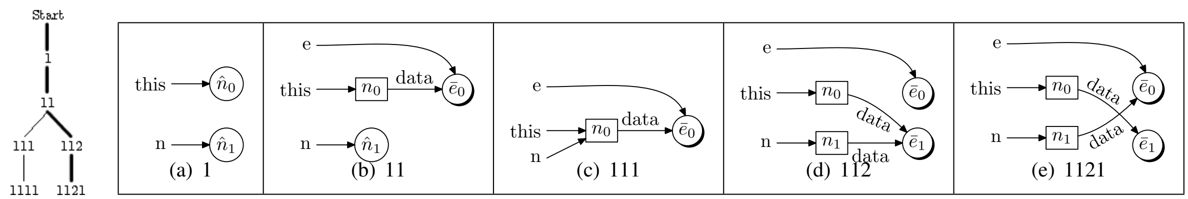


Figure 6. Lazier# Symbolic Execution Tree and An Example Trace (1-11-112-1121 and Sibling States)

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 $\bar{\cdot}$) for abstract values that abstract both NULL and any object of the appropriate type, and we use the term "symbolic references" (with annotation $\hat{\cdot}$ 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 $\hat{n}_1$ instead of a symbolic value because of the *NV* optimization mentioned above (without the optimization, we use a fresh symbolic value $\bar{n}_1$). When executing swap's first statement, $\hat{n}_0$ is replaced with a fresh symbolic object n_0, and its data field is initialized with a fresh symbolic value $\bar{e}_0$, thus resulting in State 11. Continue on with executing swap's second statement, $\hat{n}_1$ 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 $\bar{e}_1$ (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 Kiasan'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 $\alpha_\tau^{m,n}$ has three properties: type τ, 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 $s \vdash_{\mathcal{S}} instr \Rightarrow s_1[stmt_1] \mid \ldots \mid s_n[stmt_n] \mid exception, s_0[stmt_0] \mid Error, s'[stmt']$ which intuitively means that *instr* non-deterministically transforms state s into: s_1 to s_n under condition $stmt_1$ to $stmt_n$, an exception and a resulting state s_0 under condition $stmt_0$, or an error and a resulting state s' under condition $stmt'$. 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

Semantic Domains

INT	$\in$	$Types_{prim}$	=	primitive types: int,char,float,etc.		LEN, DEF, CONC, f_τ	$\in$	*Fields*	=	the set of fields
		$Types_{non-prim}$	=	$Types_{record} \uplus Types_{array} \uplus SymTypes$		ι	$\in$	*Indices*	=	$Fields \cup \mathbb{N} \cup Symbols_{\mathrm{INT}}$
τ	$\in$	*Types*	=	$Types_{prim} \uplus Types_{non-prim}$		$\alpha_\tau^{m,n}, \beta, \gamma$	$\in$	*Symbols*	=	$\{\alpha_\tau^{m,n} \mid \alpha_\tau^{m,n} : Indices \rightharpoonup Values\}$
pc	$\in$	*PCs*	=	the set of program counters					=	$Symbols_{prim} \uplus Symbols_{non-prim}$
ϕ	$\in$	Φ	=	$\mathcal{P}$(the set of boolean expressions)		v	$\in$	*Values*	=	$Consts \cup Locs \cup Symbols_{prim} \cup SymRefs \cup SymVals$
i, j	$\in$	*Locs*	=	the set of locations		g	$\in$	*Globals*	=	$\{g \mid g : Fields \rightharpoonup Values\}$
$\hat{\delta}_\tau^{m,n}$	$\in$	*SymRefs*	=	the set of symbolic references		σ	$\in$	*Stacks*	=	$\{\sigma \mid \sigma$ is a sequence of values$\}$
$\bar{\delta}_\tau^{m,n}$	$\in$	*SymVals*	=	the set of symbolic values		l	$\in$	*Locals*	=	$\{l \mid l : \mathbb{N} \rightharpoonup Values\}$
m, n, k	$\in$	$\mathbb{N}$	=	the set of natural numbers		h	$\in$	*Heaps*	=	$\left\{h \mid h : Locs \rightharpoonup Symbols_{non-prim}\right\}$
NULL, c, d	$\in$	*Consts*	=	the set of constants including $\mathbb{N}$		s	$\in$	*States*	=	$Globals \times PCs \times Locals \times Stacks \times Heaps \times \Phi$

Auxiliary Functions ($\downarrow$ = defined, $\uparrow$ = undefined)

default	=	$\lambda\tau.v$, where v is τ's default value	*symbols*	=	$\lambda s.\{\alpha \mid \alpha$ appears in $s\}$
fields	=	$\lambda\tau.\{f_{\tau'} \mid f_{\tau'}$ is a field in $\tau\}$	*new-prim-sym*	=	$\lambda(\tau, ps).\alpha_\tau, \alpha \notin ps$
$\tau' <: \tau$	=	τ' is a subtype of τ (reflexive)	*new-sym-type*	=	$\lambda ps.\tau$ s.t. $\tau \in SymTypes \wedge \tau$ does not appear in ps
acc-idx	=	$\lambda\alpha.\{k \in \mathbb{N} \cup Symbols_{\mathrm{INT}} \mid \alpha(k)\downarrow\}$	*array-type*	=	$\lambda\tau.\tau'$, where τ' is a array type of element type τ
collect	=	$\lambda h.\{i \mid h(i) = \alpha \wedge \alpha(\mathrm{CONC})\uparrow\}$	*subst*	=	$\lambda(s, \hat{\delta}, i).s'$, s' is the resulting state of substituting $\hat{\delta}$ with i in s.

new-sym $= \lambda(ps, m, n).\alpha_\tau^{m,n}$, s.t. $\alpha \notin ps \wedge \tau = \textit{new-sym-type}(ps) \wedge \forall\iota \in Indices.\alpha(\iota)\uparrow$

new-sarr $= \lambda(ps, m, n).\textit{new-sym}(ps \cup \{\alpha\}, m, n)[\mathrm{LEN} \mapsto \alpha]$ where $\alpha = \textit{new-prim-sym}(\mathrm{INT}, ps)$

new-obj $= \lambda(ps, \tau).\alpha_\tau^{0,0}$, s.t. $\alpha \notin ps \wedge \forall f_{\tau'} \in \textit{fields}(\tau).\alpha(f_{\tau'}) = \textit{default}(\tau')$

new-arr $= \lambda(ps, \tau, v, n).\alpha_{\tau'}^{0,n}, \alpha \notin ps \wedge \tau' = \textit{array-type}(\tau) \wedge \operatorname{dom}\alpha = \{\mathrm{DEF}, \mathrm{LEN}, \mathrm{CONC}\} \wedge \alpha(\mathrm{DEF}) = \textit{default}(\tau) \wedge \alpha(\mathrm{LEN}) = v$

init-sym-val $= \lambda(s, \bar{\delta}_\tau^{m,n}).\{\textit{subst}(s, \bar{\delta}, \mathrm{NULL}), \textit{subst}(s, \bar{\delta}, \hat{\delta}_\tau^{m,n})\}$ where $\hat{\delta}$ is fresh

init-sym-ref $= \lambda((g, pc, l, \sigma, h, \phi), \hat{\delta}_\tau^{m,n}).\{\textit{subst}((g, pc, l, \sigma, h', \phi'), \hat{\delta}, i) \mid i \in \textit{collect}(h), h' = h, \phi' = \phi \cup \{\tau' <: \tau\}$ where $h(i) = \alpha_{\tau'}$;
$i \notin \operatorname{dom} h, h' = h[i \mapsto \gamma_{\tau'}^{m,n}], \phi' = \phi \cup \{\tau' <: \tau\}$ if $\tau \in Types_{record} \wedge m \geq 0$
where $\gamma_{\tau'} = \textit{new-sym}(\textit{symbols}(g, pc, l, \sigma, h, \phi), m, n)$; (the array case is similar)$\}$

Transitions (Non-deterministic, k-Bounded): $s \vdash_S instr \Rightarrow s_1[stmt_1] \mid \ldots \mid s_n[stmt_n] \mid exception, s_0[stmt_0] \mid Error, s'[stmt']$

Lazy

$(g, pc, l, i :: \sigma, h, \phi) \vdash_S \texttt{getfield}\ f_\tau \Rightarrow (g, nxt(pc), l, v :: \sigma, h, \phi)$ where $v = \beta^{m,n}(f_\tau), \beta^{m,n} = h(i)$ if $\beta^{m,n}(f_\tau)\downarrow$
$\mid (g, nxt(pc), l, v :: \sigma, h', \phi')$ when $\beta^{m,n} = h(i)$, j is fresh, $\tau'' = \textit{new-sym-type}((g, pc, l, i :: \sigma, h, \phi))$,
$v = \textit{new-prim-sym}(\tau, (g, pc, l, i :: \sigma, h, \phi)), h' = h[i \mapsto \beta^{m,n}[f_\tau \mapsto v]], \phi' = \phi$ if $\beta^{m,n}(f_\tau)\uparrow \wedge \tau \in Types_{prim}$;
$v = \mathrm{NULL}, h' = h[i \mapsto \beta^{m,n}[f_\tau \mapsto v]], \phi' = \phi$ if $\tau \in Types_{non-prim}$;
$v \in (\textit{collect}(h) \cup \{j\})_{\tau''}, h' = h[i \mapsto \beta^{m,n}[f_\tau \mapsto v]][j \mapsto \gamma_{\tau'}], \phi' = \phi \cup \{\tau' <: \tau'', \tau'' <: \tau\}$
where $\gamma_{\tau'} = \textit{new-sym}((g, pc, l, i :: \sigma, h, \phi), m - 1, k)$ if $\beta^{m,n}(f_\tau)\uparrow \wedge \tau \in Types_{record} \wedge m > 0$;
$v \in (\textit{collect}(h) \cup \{j\})_{\tau''}, h' = h[i \mapsto \beta^{m,n}[f_\tau \mapsto v]][j \mapsto \gamma_{\tau'}], \phi' = \phi \cup \{\gamma(\mathrm{LEN}) \geq 0, \tau' <: \tau'', \tau'' <: \tau\}$
where $\gamma_{\tau'} = \textit{new-sarr}((g, pc, l, i :: \sigma, h, \phi), m - 1, k)$ if $\beta^{m,n}(f_\tau)\uparrow \wedge \tau \in Types_{array} \wedge m > 0$;
$v \in \textit{collect}(h)_{\tau''}, h' = h[i \mapsto \beta^{m,n}[f_\tau \mapsto v]], \phi' = \phi \cup \{\tau'' <: \tau\}$ if $\beta^{m,n}(f_\tau)\uparrow \wedge \tau \in Types_{non-prim} \wedge m = 0$

$(g, pc, l, \mathrm{NULL} :: \sigma, h, \phi) \vdash_S \texttt{getfield}\ f \Rightarrow$ NullPointerException, $(g, pc, l, \sigma, h, \phi)$

$(g, pc, l, j :: i :: \sigma, h, \phi) \vdash_S \texttt{if\_acmpeq}\ pc' \Rightarrow (g, nxt(pc), l, \sigma, h, \phi)$ if $i \neq j \mid (g, pc', l, \sigma, h, \phi)$ if $i = j$

Lazier

$(g, pc, l, i :: \sigma, h, \phi) \vdash_S \texttt{getfield}\ f_\tau \Rightarrow (g, nxt(pc), l, v :: \sigma, h, \phi)$ where $v = \beta^{m,n}(f_\tau), \beta^{m,n} = h(i)$ if $\beta^{m,n}(f_\tau)\downarrow$;
$(g, nxt(pc), l, v :: \sigma, h[i \mapsto \beta^{m,n}[f_\tau \mapsto v]], \phi)$ when $\beta^{m,n} = h(i)$,
$v = \textit{new-prim-sym}(\textit{symbols}(g, pc, l, i :: \sigma, h, \phi), \tau)$ if $\beta^{m,n}(f_\tau)\uparrow \wedge \tau \in Types_{prim}$;
$v = \mathrm{NULL}$, if $\beta^{m,n}(f_\tau)\uparrow \wedge \tau \in Types_{non-prim}$;
$v = \hat{\delta}_\tau^{m-1,k}$, where $\hat{\delta}$ is fresh if $\beta^{m,n}(f_\tau)\uparrow \wedge \tau \in Types_{non-prim}$;

$(g, pc, l, \hat{\delta}_\tau^{m,n} :: \sigma, h, \phi) \vdash_S \texttt{getfield}\ f_\tau \Rightarrow s'$ where $s' \in \textit{init-sym-ref}((g, pc, l, \hat{\delta}_\tau^{m,n} :: \sigma, h, \phi), \hat{\delta})$

$(g, pc, l, \hat{\delta}_\tau^{m,n} :: \hat{\delta}_\tau^{m,n} :: \sigma, h, \phi) \vdash_S \texttt{if\_acmpeq}\ pc' \Rightarrow (g, pc', l, \sigma, h, \phi)$

$(g, pc, l, \hat{\delta}_\tau^{m,n} :: v :: \sigma, h, \phi) \vdash_S \texttt{if\_acmpeq}\ pc' \Rightarrow s'$ where $s' \in \textit{init-sym-ref}((g, pc, l, \hat{\delta}_\tau^{m,n} :: v :: \sigma, h, \phi), \hat{\delta})$

Lazier#

$(g, pc, l, i :: \sigma, h, \phi) \vdash_S \texttt{getfield}\ f_\tau \Rightarrow (g, nxt(pc), l, v :: \sigma, h, \phi)$ where $v = \beta^{m,n}(f_\tau), \beta^{m,n} = h(i)$ if $\beta^{m,n}(f_\tau)\downarrow$;
$(g, nxt(pc), l, v :: \sigma, h[i \mapsto \beta^{m,n}[f_\tau \mapsto v]], \phi)$ when $\beta^{m,n} = h(i)$,
$v = \textit{new-prim-sym}(\textit{symbols}(g, pc, l, i :: \sigma, h, \phi), \tau)$ if $\beta^{m,n}(f_\tau)\uparrow \wedge \tau \in Types_{prim}$;
$v = \bar{\delta}_\tau^{m-1,k}$, where $\bar{\delta}$ is fresh if $\beta^{m,n}(f_\tau)\uparrow \wedge \tau \in Types_{non-prim}$;

$(g, pc, l, \bar{\delta}_\tau^{m,n} :: \sigma, h, \phi) \vdash_S \texttt{getfield}\ f_\tau \Rightarrow s'$ where $s' \in \textit{init-sym-val}((g, pc, l, \bar{\delta}_\tau^{m,n} :: \sigma, h, \phi), \bar{\delta})$

$(g, pc, l, \bar{\delta}_\tau^{m,n} :: \bar{\delta}_\tau^{m,n} :: \sigma, h, \phi) \vdash_S \texttt{if\_acmpeq}\ pc' \Rightarrow (g, pc', l, \sigma, h, \phi)$

$(g, pc, l, \bar{\delta}_\tau^{m,n} :: v :: \sigma, h, \phi) \vdash_S \texttt{if\_acmpeq}\ pc' \Rightarrow s'$ where $s' \in \textit{init-sym-val}((g, pc, l, \hat{\delta}_\tau^{m,n} :: v :: \sigma, h, \phi), \hat{\delta})$

Note: implicit universal quantifications on free variables; mid-bar (|) indicates a non-deterministic choice, and semi-colon (;) indicates different cases

Figure 7. Bytecode-level Symbolic Execution Operational Semantics (excerpts)

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 symbol. 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 reference) `getfield` rule is applied. For the *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 NULL or a fresh symbolic reference by *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 $\mathcal{R}''$ is a binary relation on lazier symbolic states and lazier# symbolic states; $a\ \mathcal{R}''\ b$ 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* $a_1 \longrightarrow_{\mathcal{S}} a_2 \longrightarrow_{\mathcal{S}} \cdots \longrightarrow_{\mathcal{S}} a_n$*, there is a corresponding lazier# symbolic trace* $b_1 \longrightarrow_{\mathcal{S}} b_2 \longrightarrow_{\mathcal{S}} \cdots \longrightarrow_{\mathcal{S}} b_n$ *such that* $a_k\ \mathcal{R}''\ b_k$ *for all* $1 \leq k \leq n$.[2]

Proposition 2 (Completeness). *Given any lazier# symbolic trace* $b_1 \longrightarrow_{\mathcal{S}} b_2 \longrightarrow_{\mathcal{S}} \cdots \longrightarrow_{\mathcal{S}} b_n$*, there is a corresponding lazier symbolic trace* $a_1 \longrightarrow_{\mathcal{S}} a_2 \longrightarrow_{\mathcal{S}} \cdots \longrightarrow_{\mathcal{S}} a_n$ *such that* $a_k\ \mathcal{R}''\ b_k$ *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 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, NULL is not allowed for tree elements and this confirms our previous observation that lazier initialization is optimal for non-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/insert/remove operations are the same for each tree (binary search tree, AVL tree, red-black tree) under the lazier# algorithm. This is because the search/insert/remove 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 TreeMap).

6 Related Work

Throughout the paper we have contrasted our approach to others, thus we limit ourselves to a concise discussion

[2] Note that if $s_1 \longrightarrow_{\mathcal{S}} s_2$, then the program counters of s_1 and s_2 have to be different (each $\longrightarrow_{\mathcal{S}}$ may correspond to one or more $\Rightarrow_{\mathcal{S}}$). In essence, we model 2/3-steps semantics of lazier/lazier# into 1-step semantics.

Class	Method		States			Cases			Total Time			Theorem Prover Time		
		k	Lazy	Lazier	Lazier#	Lazy	Lazier	Lazier#	Lazy	Lazier	Lazier#	Lazy	Lazier	Lazier#
AvlTree	find	1	3271	2420	1864	5	4	4	1.2s	1.5s	0.8s	0.4s	0.5s	0.1s
		2	48244	23807	18800	29	21	21	8.9s	7.2s	6.9s	2.8s	3.9s	2.5s
		3	10944306	459718	351798	275	190	190	1.7h	2.4m	3.7m	1.1h	1.9m	3.2m
	insert	1	4719	3841	3053	5	4	4	2.1s	2.6s	1.5s	0.3s	1.6s	0.7s
		2	56832	31905	25702	29	21	21	10.3s	7.0s	7.5s	4.5s	3.1s	4.0s
		3	11036507	542929	422049	275	190	190	2.1h	2.8m	3.9m	1.4h	2.2m	3.5m
BinarySearchTree	insert	1	6097	5521	1621	13	12	4	3.5s	2.1s	1.1s	0.9s	0.5s	0.1s
		2	91691	63931	12551	112	94	21	22.7s	17.1s	5.0s	9.5s	7.9s	1.7s
		3	3349343	1855571	234595	2161	1668	236	50.1m	16.4m	1.5m	39.1m	12.6m	1.1m
	remove	1	4146	3693	1001	13	12	4	2.4s	2.5s	1.3s	1.2s	0.7s	0.5s
		2	74896	49422	9254	112	94	21	22.4s	14.5s	5.1s	12.8s	5.6s	1.9s
		3	3031511	1599087	197738	2161	1668	236	43.0m	13.8m	1.3m	35.1m	10.9m	1.0m
	find	1	4890	4301	1162	13	12	4	2.1s	2.9s	0.8s	0.3s	1.0s	0.2s
		2	89819	57292	10443	126	98	21	23.1s	16.3s	4.9s	10.9s	8.4s	1.8s
		3	3822839	1808683	212296	2873	1788	236	55.5m	15.7m	1.4m	42.5m	12.4m	1.1m
StackList	push	1	758	758	374	4	4	2	0.7s	0.7s	0.6s	0.0s	0.0s	0.0s
		2	1466	1390	687	6	6	3	0.9s	0.8s	0.5s	0.0s	0.1s	0.1s
		3	2450	2260	1119	8	8	4	2.1s	1.7s	0.7s	0.4s	0.1s	0.0s
	pop	1	196	196	189	2	2	2	0.2s	0.2s	0.2s	0.0s	0.0s	0.1s
		2	425	387	377	3	3	3	0.4s	0.3s	0.5s	0.0s	0.0s	0.1s
		3	770	675	662	4	4	4	0.4s	0.6s	0.5s	0.0s	0.2s	0.0s
java.util.TreeMap	get	1	4309	2009	1199	8	6	4	4.2s	2.1s	1.6s	3.0s	1.2s	1.0s
		2	85601	27489	17440	62	40	28	16.0s	10.3s	7.7s	8.5s	4.3s	4.4s
		3	20707094	774545	470913	782	482	331	7.0h	3.1m	2.0m	5.0h	2.3m	1.4m
	remove	1	2247	1721	1110	7	5	4	1.4s	1.4s	1.4s	0.1s	0.4s	0.9s
		2	74892	37832	17081	73	43	28	16.0s	12.2s	5.8s	7.9s	6.2s	2.3s
		3	17631620	1166311	472985	1075	579	331	5.1h	7.6m	1.9m	3.7h	6.4m	1.4m
	lastKey	1	1219	664	657	2	2	2	0.7s	0.4s	0.6s	0.1s	0.0s	0.3s
		2	15680	7658	7614	6	6	6	7.7s	2.5s	3.6s	3.5s	0.5s	1.2s
		3	3524450	205430	204738	31	31	31	27.0m	27.9s	30.3s	21.5m	17.3s	19.3s
java.util.Vector	add	1	986	818	354	6	6	3	2.0s	1.4s	0.7s	1.0s	0.8s	0.5s
		2	2932	1514	472	20	14	5	6.5s	2.8s	1.2s	4.2s	1.4s	0.7s
		3	10990	2906	590	74	30	7	21.0s	5.6s	0.9s	15.8s	3.3s	0.4s
	indexOf	1	644	588	438	7	6	6	0.9s	1.6s	0.9s	0.2s	0.3s	0.4s
		2	1195	1135	486	17	16	7	2.1s	2.0s	1.1s	0.5s	1.0s	0.7s
		3	2686	2339	486	44	38	7	4.5s	4.1s	0.5s	2.5s	1.7s	0.1s
	removeElementAt	1	202	200	197	3	3	3	0.8s	0.3s	0.3s	0.6s	0.1s	0.1s
		2	382	320	257	6	5	4	1.0s	0.7s	0.6s	0.2s	0.1s	0.4s
		3	999	566	318	16	9	5	2.0s	0.9s	0.6s	0.7s	0.5s	0.2s

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 counter-examples 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

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, 3362:49–69, 2004.

[2] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proceedings of the international symposium on Software testing and analysis (ISSTA)*, pages 123–133. ACM Press, 2002.

[4] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, pages 108–128, 2004.

[5] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proc. 27th International Conference on Software Engineering*, pages 422–431. ACM Press, May 2005.

[6] X. Deng, J. Lee, and Robby. Bogor/Kiasan: A *k*-bounded symbolic execution for checking strong heap properties of open systems. In *21st International Conference on Automated Software Engineering (ASE)*, pages 157–166, 2006.

[7] X. Deng, Robby, and J. Hatcliff. Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems. In *Testing: Academic and Industrial Conference – Practice and Research Techniques*, 2007. To appear.

[8] X. Deng, Robby, and J. Hatcliff. Towards a case-optimal symbolic execution algorithm for analyzing strong properties of object-oriented programs. Technical Report SAnToS-TR2007-3, CIS Department, Kansas State University, 2007.

[9] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 234–245, 2002.

[10] W. Grieskamp, N. Tillmann, and W. Schulte. XRT - exploring runtime for .NET - architecture and applications. *Workshop on Software Model Checking*, 2005.

[11] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. *Tools and Algorithms for Construction and Analysis of Systems*, pages 553–568, 2003.

[12] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[13] G. T. Leavens, A. L. Baker, and C. Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop*, oct 1998.

[14] D. Marinov and S. Khurshid. Testera: A novel framework for automated testing of java programs. In *16th IEEE Conference on Automated Software Engineering (ASE 2001)*, 2001.

[15] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[16] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.

[17] Robby, E. Rodríguez, M. B. Dwyer, and J. Hatcliff. Checking JML specifications using an extensible software model checking framework. *International Journal on Software Tools for Technology Transfer*, 2006.

[18] K. H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill Science/Engineering/Math, 4th edition, 1998.

[19] K. Sen and G. Agha. CUTE: A concolic unit testing engine for C. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2005.

[20] M. A. Weiss. *Data Structures and Algorithm Analysis in Java*. Addison Wesley, 1998.

[21] H. S. Wilf. *Generatingfunctionology*. Academic Press, 1990.

[22] http://bogor.projects.cis.ksu.edu.

Bibliography

[1] A. V. Aho and N. J. A. Sloane. Some doubly exponential sequences. *Fibonacci Quarterly*, 11:429–437, 1970.

[2] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Reading, Massachusetts: Addison-Wesley, 2 edition, 1994.

[3] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. *Tools and Algorithms for Construction and Analysis of Systems*, pages 553–568, 2003.

[4] Tim Lindholm and Frank Yellin. The Java virtual machine specification (2nd edition). http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html.

[5] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill Science/Engineering/-Math, 4 edition, 1998.

[6] David Schmidt. Binary relations for abstraction and refinement. Technical report, Kansas State University, November 2000.

[7] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison Wesley, 1998.

[8] Herbert S. Wilf. *Generatingfunctionology*. Academic Press, 1990.

Appendix A

Data Table and Swap States

Class	Method		States			Cases			Total Time			Theorem Prover Time		
		k	Lazy	Lazier	Lazier#	Lazy	Lazier	Lazier#	Lazy	Lazier	Lazier#	Lazy	Lazier	Lazier#
AP	partition	1	152	152	150	1	1	1	0.2s	0.2s	0.2s	0.0s	0.0s	0.0s
		2	533	392	389	4	3	3	0.9s	0.5s	0.5s	0.4s	0.2s	0.3s
		3	2340	1010	1006	15	7	7	3.4s	2.0s	1.2s	1.4s	0.8s	0.6s
GC	Mark	1	226510	222575	222149	306	306	306	32.2s	30.8s	32.6s	2.1s	2.0s	2.2s
LL	merge	1	1007	1007	732	1	1	1	0.8s	0.5s	0.4s	0.0s	0.0s	0.1s
		2	4707	3931	3631	6	6	5	2.3s	1.9s	1.8s	0.6s	0.1s	0.0s
		3	63932	18122	17754	63	30	19	13.0s	6.2s	5.7s	3.5s	2.5s	1.1s
Sort	insertionSort	1	190	190	178	2	2	2	0.4s	0.2s	0.2s	0.1s	0.0s	0.0s
		2	506	389	376	5	4	4	0.9s	0.8s	0.6s	0.3s	0.1s	0.3s
		3	2427	1102	1088	19	10	10	4.1s	2.2s	2.4s	3.0s	1.2s	1.7s
	shellsort	1	192	192	180	2	2	2	0.3s	0.3s	0.2s	0.1s	0.1s	0.0s
		2	553	419	406	5	4	4	1.2s	0.9s	0.7s	0.6s	0.3s	0.2s
		3	2674	1206	1192	19	10	10	5.5s	2.6s	2.8s	3.3s	1.1s	1.4s
StackAr	push	1	250	238	120	4	4	2	0.6s	0.2s	0.2s	0.4s	0.1s	0.1s
		2	250	238	120	4	4	2	0.4s	0.2s	0.3s	0.1s	0.1s	0.0s
		3	250	238	120	4	4	2	0.5s	0.8s	0.2s	0.1s	0.4s	0.0s
	pop	1	105	105	104	2	2	2	0.3s	0.2s	0.2s	0.0s	0.1s	0.0s
		2	105	105	104	2	2	2	0.2s	0.3s	0.2s	0.0s	0.1s	0.1s
		3	105	105	104	2	2	2	0.3s	0.2s	0.2s	0.1s	0.0s	0.0s
StackLi	push	1	758	758	374	4	4	2	0.7s	0.7s	0.3s	0.0s	0.0s	0.0s
		2	1466	1390	687	6	6	3	0.9s	1.2s	0.7s	0.0s	0.0s	0.0s
		3	2450	2260	1119	8	8	4	2.1s	1.6s	0.9s	0.4s	0.0s	0.0s
	pop	1	196	196	189	2	2	2	0.2s	0.1s	0.2s	0.0s	0.0s	0.0s
		2	425	387	377	3	3	3	0.5s	0.2s	0.3s	0.0s	0.0s	0.0s
		3	770	675	662	4	4	4	0.5s	0.7s	0.5s	0.0s	0.0s	0.1s
TreeMap	get	1	4309	2009	1199	8	6	4	4.2s	1.5s	1.0s	3.0s	0.3s	0.6s
		2	85601	27489	17440	62	40	28	16.0s	7.6s	7.4s	8.5s	3.4s	3.3s
		3	20707094	774545	470913	782	482	331	7.0h	3.1m	2.0m	5.0h	2.3m	1.5m
	remove	1	2247	1721	1110	7	5	4	1.4s	1.1s	0.7s	0.1s	0.1s	0.2s
		2	74892	37832	17081	73	43	28	16.0s	9.9s	5.7s	7.9s	5.7s	2.6s
		3	17631620	1166311	472985	1075	579	331	5.1h	7.4m	1.9m	3.7h	6.2m	1.4m
	lastKey	1	1219	664	657	2	2	2	0.7s	0.4s	0.3s	0.1s	0.1s	0.1s
		2	15680	7658	7614	6	6	6	7.7s	2.1s	2.4s	3.5s	0.5s	0.5s
		3	3524450	205430	204738	31	31	31	27.0m	28.1s	26.4s	21.5m	17.0s	16.5s
	put	1	9951	4125	1871	16	10	4	3.4s	1.6s	0.9s	1.1s	0.1s	0.2s
		2	181493	54221	22872	144	78	28	34.7s	14.2s	6.8s	18.7s	9.2s	2.7s
		3	N/A	2676863	530005	N/A	1978	331	>24h	11.0m	1.9m	N/A	7.8m	1.3m
TCF	classify	1	404	404	404	15	15	15	0.9s	0.8s	0.6s	0.6s	0.3s	0.2s
Vector	add	1	986	818	354	6	6	3	2.0s	1.3s	0.6s	1.0s	0.6s	0.3s
		2	2932	1514	472	20	14	5	6.5s	3.7s	1.0s	4.2s	2.7s	0.6s
		3	10990	2906	590	74	30	7	21.0s	6.0s	1.3s	15.8s	4.3s	0.7s
	ensureCapacity	1	1051	831	610	17	13	9	2.3s	2.0s	0.8s	1.1s	1.1s	0.2s
		2	3239	1703	826	57	29	13	5.5s	4.0s	1.6s	2.3s	2.0s	0.7s
		3	11339	3447	1042	205	61	17	14.2s	5.3s	2.8s	9.5s	2.8s	2.1s
	insertElementAt	1	1425	1131	469	6	6	3	3.4s	1.8s	0.9s	2.0s	0.8s	0.3s
		2	6874	3175	836	41	26	8	13.2s	5.1s	0.9s	10.1s	3.3s	0.5s
		3	41077	9235	1389	210	78	15	1.5m	15.9s	3.2s	1.3m	12.6s	1.8s
	lastIndexOf	1	662	608	445	9	8	7	0.6s	0.7s	0.8s	0.2s	0.2s	0.5s
		2	1199	1141	975	19	18	17	1.9s	1.5s	1.3s	0.8s	0.5s	0.6s
		3	2650	2313	2141	46	40	39	6.9s	3.7s	3.4s	3.6s	1.9s	1.7s
	indexOf	1	644	588	438	7	6	6	0.9s	1.6s	1.0s	0.2s	0.3s	0.5s
		2	1195	1135	486	17	16	7	2.1s	2.0s	1.1s	0.5s	1.0s	0.7s
		3	2686	2339	486	44	38	7	4.5s	4.1s	0.5s	2.5s	1.7s	0.2s
	removeElementAt	1	202	200	197	3	3	3	0.8s	0.5s	0.6s	0.6s	0.1s	0.3s
		2	382	320	257	6	5	4	1.0s	0.5s	0.7s	0.2s	0.2s	0.2s
		3	999	566	318	16	9	5	2.0s	1.2s	0.7s	0.7s	0.6s	0.2s

Class	Method	k	States			Cases			Total Time			Theorem Prover Time		
		k	Lazy	Lazier	Lazier#	Lazy	Lazier	Lazier#	Lazy	Lazier	Lazier#	Lazy	Lazier	Lazier#
ABS	abs	1	60	60	60	2	2	2	0.3s	0.1s	0.1s	0.1s	0.0s	0.0s
AvlTree	find	1	3271	2420	1864	5	4	4	1.2s	0.8s	0.9s	0.4s	0.1s	0.0s
		2	48244	23807	18800	29	21	21	8.9s	5.9s	7.0s	2.8s	2.3s	2.5s
		3	10944306	459718	351798	275	190	190	1.7h	2.4m	3.4m	1.1h	2.0m	3.0m
	insert	1	4719	3841	3053	5	4	4	2.1s	1.4s	1.0s	0.3s	0.1s	0.1s
		2	56832	31905	25702	29	21	21	10.3s	5.6s	7.1s	4.5s	2.0s	2.5s
		3	11036507	542929	422049	275	190	190	2.1h	2.7m	3.7m	1.4h	2.2m	3.2m
	findMax	1	1457	1457	1132	2	2	2	1.2s	0.5s	0.7s	0.7s	0.0s	0.2s
		2	12738	8733	7127	5	5	5	3.3s	2.0s	3.2s	0.6s	0.4s	0.8s
		3	2395408	107574	85445	20	20	20	8.0m	18.1s	54.2s	5.5m	10.6s	46.7s
BinaryHeap	deleteMin	1	290	290	288	2	2	2	0.4s	0.4s	0.4s	0.1s	0.1s	0.1s
		2	670	491	488	4	3	3	0.9s	0.7s	0.9s	0.4s	0.4s	0.1s
		3	2327	894	890	11	5	5	3.3s	1.5s	1.2s	2.0s	0.6s	0.7s
	insert	1	458	349	336	4	3	3	0.8s	0.5s	0.7s	0.5s	0.1s	0.1s
		2	1470	664	650	12	6	6	2.0s	1.5s	1.3s	0.9s	0.4s	0.4s
		3	5388	1113	1098	34	9	9	5.9s	2.1s	1.6s	3.2s	0.9s	1.0s
BinarySearchTree	insert	1	6097	5521	1621	13	12	4	3.5s	2.4s	1.4s	0.9s	0.2s	0.1s
		2	91691	63931	12551	112	94	21	22.7s	15.0s	5.1s	9.5s	8.2s	1.7s
		3	3349343	1855571	234595	2161	1668	236	50.1m	16.6m	1.5m	39.1m	13.2m	1.2m
	remove	1	4146	3693	1001	13	12	4	2.4s	1.9s	0.7s	1.2s	0.1s	0.0s
		2	74896	49422	9254	112	94	21	22.4s	14.8s	6.0s	12.8s	7.7s	3.4s
		3	3031511	1599087	197738	2161	1668	236	43.0m	13.6m	1.3m	35.1m	11.2m	1.0m
	find	1	4890	4301	1162	13	12	4	2.1s	2.6s	0.6s	0.3s	0.9s	0.0s
		2	89819	57292	10443	126	98	21	23.1s	15.1s	4.8s	10.9s	7.4s	1.0s
		3	3822839	1808683	212296	2873	1788	236	55.5m	15.8m	1.5m	42.5m	12.8m	1.1m
	findMax	1	1430	1430	689	3	3	2	0.9s	0.8s	0.5s	0.1s	0.0s	0.0s
		2	14851	12821	3719	13	13	5	4.3s	5.3s	1.9s	1.6s	0.9s	0.4s
		3	331374	258323	43215	131	131	26	3.3m	2.7m	25.4s	3.0m	2.4m	20.2s
	findMin	1	1428	1428	688	3	3	2	0.8s	0.9s	0.5s	0.1s	0.0s	0.0s
		2	14888	12858	3729	13	13	5	4.3s	3.8s	1.8s	1.7s	1.0s	0.3s
		3	332455	259404	43400	131	131	26	3.6m	2.7m	25.1s	3.1m	2.4m	19.4s
DisjSets	Find	1	267	267	266	1	1	1	0.3s	0.3s	0.5s	0.1s	0.1s	0.3s
		2	1224	1224	1223	7	7	7	1.4s	1.6s	1.5s	0.6s	0.8s	0.7s
		3	6302	6302	6301	55	55	55	7.9s	7.8s	9.5s	5.2s	4.6s	7.2s
	union	2	1295	1295	1294	2	2	2	1.4s	1.4s	1.5s	0.7s	0.8s	0.6s
		3	6939	6939	6938	20	20	20	7.7s	6.8s	7.0s	5.5s	4.7s	4.6s
DisjSetsFast	Find	1	282	282	281	1	1	1	0.6s	0.3s	0.3s	0.1s	0.1s	0.1s
		2	1269	1269	1268	7	7	7	2.5s	1.7s	1.9s	1.5s	1.1s	1.1s
		3	6486	6486	6485	55	55	55	10.4s	8.3s	10.6s	7.6s	5.8s	6.7s
	union	2	1644	1644	1643	6	6	6	2.3s	2.5s	2.1s	1.6s	1.1s	1.2s
		3	10097	10097	10096	60	60	60	14.8s	15.2s	14.6s	11.1s	12.4s	12.2s
DoubleLinkedList	remove	1	664	664	371	2	2	2	0.7s	0.7s	0.5s	0.1s	0.1s	0.0s
		2	5105	1616	1100	19	6	6	3.5s	1.6s	1.3s	0.3s	0.1s	0.4s
		3	15422	3831	3039	51	16	16	6.8s	2.7s	1.5s	1.5s	0.2s	0.2s
	addBefore	1	6178	1010	364	8	2	1	2.5s	0.6s	0.2s	0.6s	0.0s	0.0s
		2	34918	2922	608	24	6	1	8.5s	1.6s	0.3s	4.2s	0.2s	0.0s
		3	99182	5846	914	40	12	1	14.9s	2.2s	0.5s	7.5s	0.8s	0.1s
	indexOf	1	668	668	373	2	2	2	0.5s	0.6s	0.2s	0.0s	0.3s	0.0s
		2	5214	1650	1130	19	6	6	2.6s	1.4s	1.0s	0.6s	0.3s	0.5s
		3	15709	3937	3139	51	16	16	6.9s	2.2s	3.4s	3.4s	0.2s	1.3s
	clear	1	226	226	223	1	1	1	0.2s	0.1s	0.2s	0.0s	0.0s	0.0s
		2	1194	513	507	3	2	2	0.8s	0.5s	0.5s	0.1s	0.2s	0.0s
		3	2798	841	832	4	3	3	1.3s	0.6s	0.7s	0.2s	0.0s	0.1s
	lastIndexOf	1	670	670	374	2	2	2	0.7s	0.3s	0.3s	0.4s	0.0s	0.0s
		2	5214	1654	1132	19	6	6	2.0s	1.5s	0.8s	0.6s	0.1s	0.2s
		3	15693	3583	2783	51	14	14	5.6s	3.6s	1.8s	2.2s	1.3s	0.7s
	removeLast	1	278	180	152	2	2	1	0.5s	0.2s	0.3s	0.1s	0.0s	0.1s
		2	1677	597	453	6	4	2	0.9s	0.6s	0.5s	0.1s	0.1s	0.1s
		3	3759	1143	839	8	6	3	2.5s	0.8s	0.7s	1.1s	0.1s	0.1s
	toArray	1	165	165	162	1	1	1	0.2s	0.2s	0.1s	0.0s	0.0s	0.0s
		2	1973	437	399	30	3	2	3.3s	0.3s	0.4s	2.0s	0.0s	0.1s
		3	14484	842	691	351	7	3	19.2s	0.7s	0.7s	6.4s	0.1s	0.2s

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

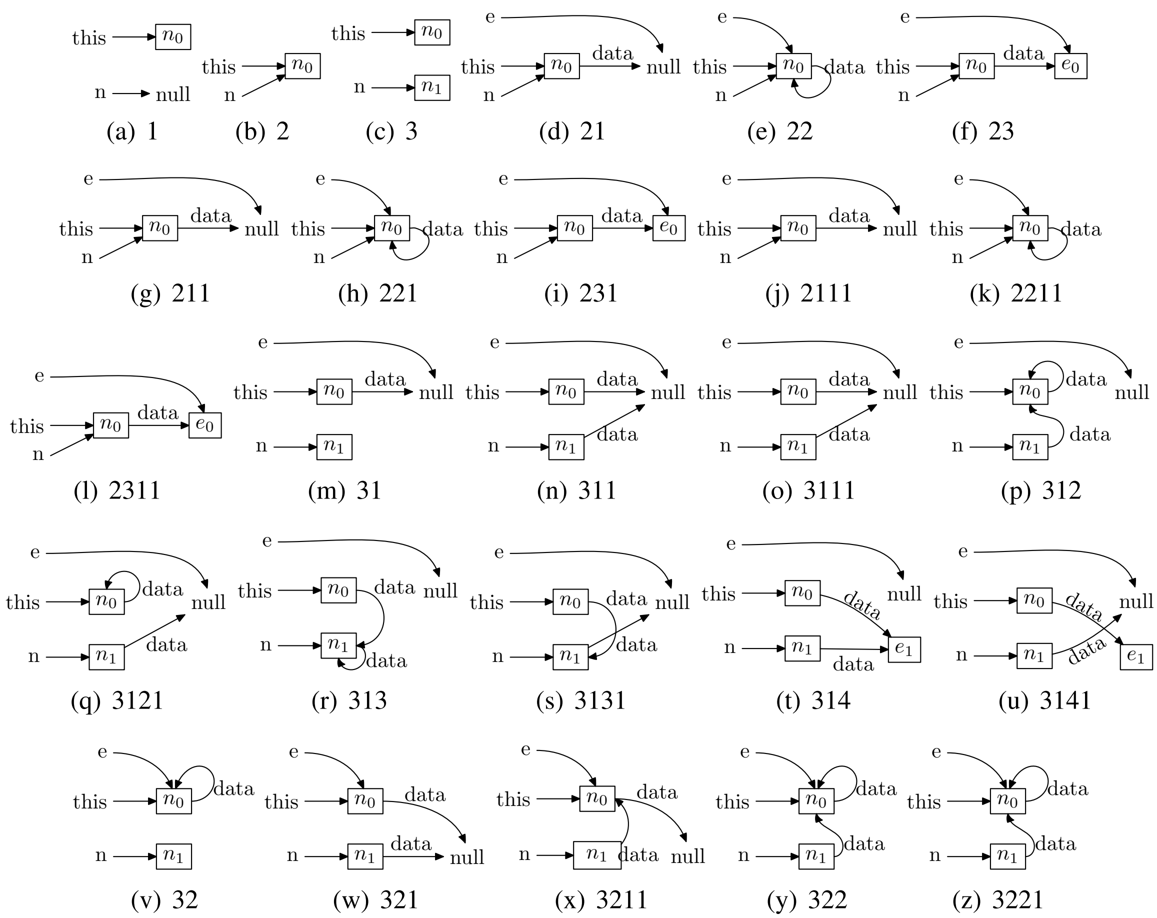
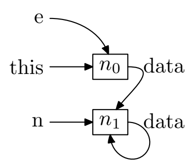


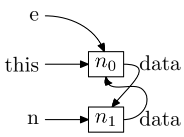
Figure A.1: *Swap–Lazy States (1)*

A.1.2 Lazier Swap States

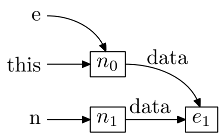
A.1.3 Lazier# States



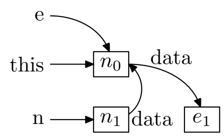
(a) 323



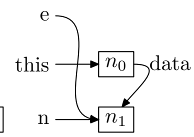
(b) 3231



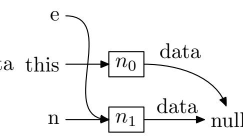
(c) 324



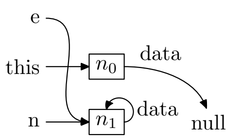
(d) 3241



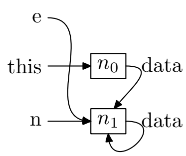
(e) 33



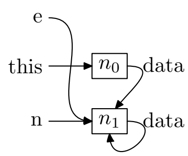
(f) 331



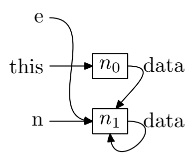
(g) 3311



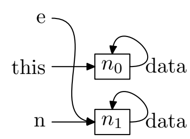
(h) 332



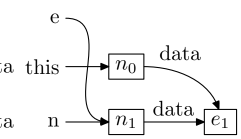
(i) 3321



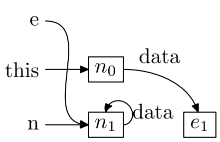
(j) 333



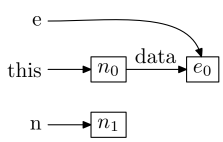
(k) 3331



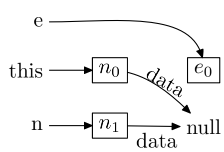
(l) 334



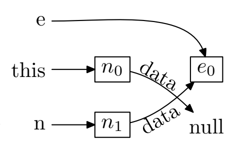
(m) 3341



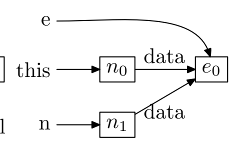
(n) 34



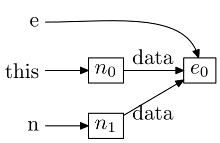
(o) 341



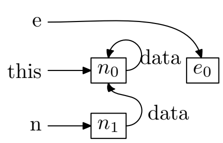
(p) 3411



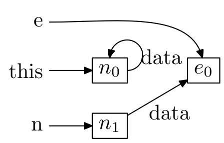
(q) 342



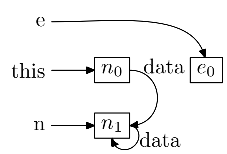
(r) 3421



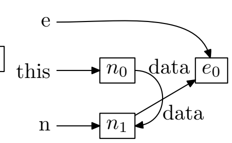
(s) 343



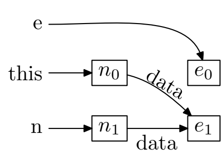
(t) 3431



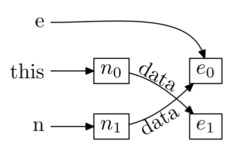
(u) 344



(v) 3441



(w) 345



(x) 3451

Figure A.2: *Swap–Lazy States (2)*

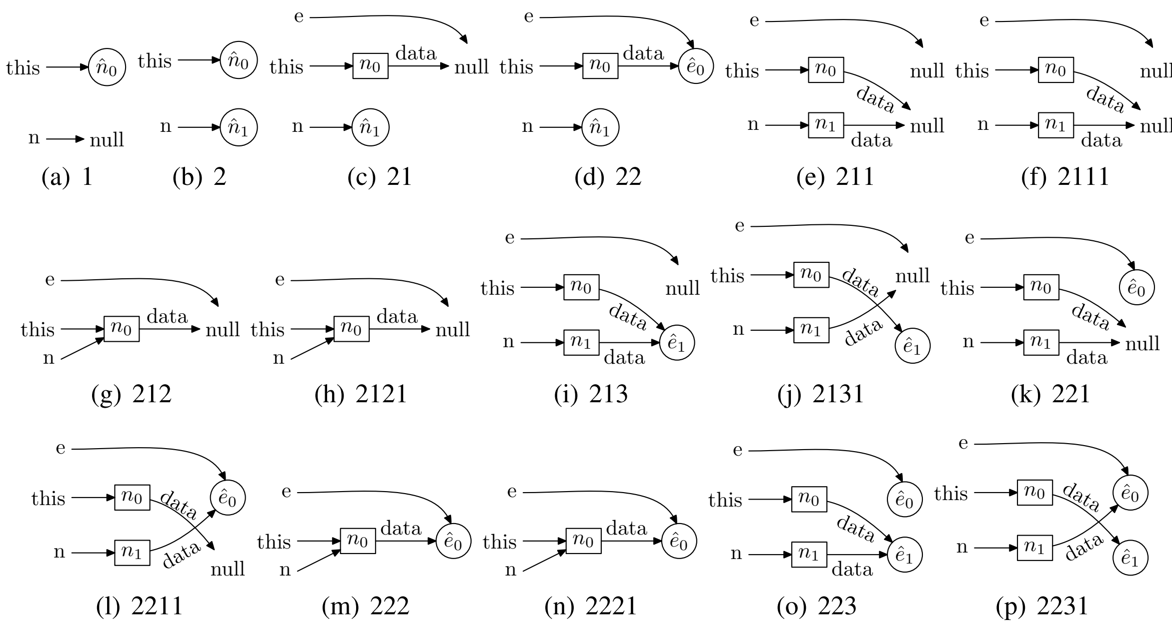


Figure A.3: *Swap–Lazier States*

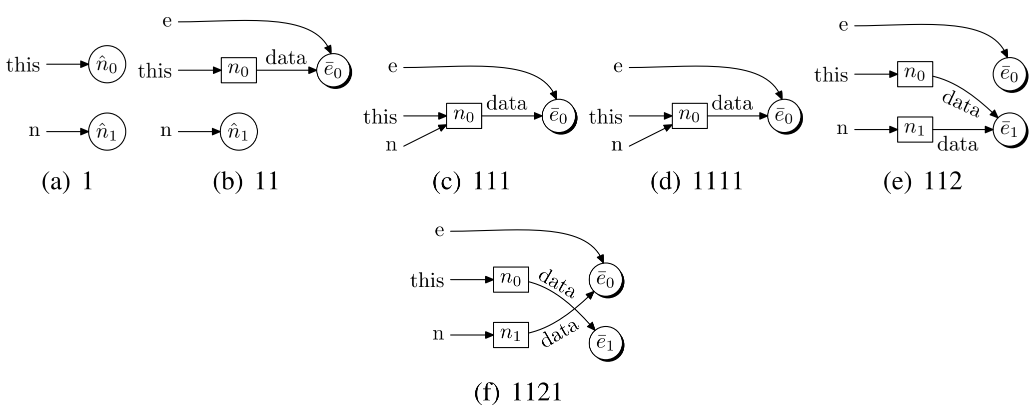


Figure A.4: *Swap–Lazier# States*

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 $BST_n = \{ t \in BST \mid height(t) < n \}$ for all $n \in \mathbb{N}$. Then we define two relations.

- $R : BST \times BST$ as

$$t_1 \, R \, t_2 \iff \exists f : \mathbb{Z} \rightharpoonup \mathbb{Z}. \operatorname{dom} f \supseteq elements(t_1) \land f \text{ is strictly increasing } \land f(t_1) = t_2. \quad \text{(B.1)}$$

  where $elements(t)$ 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'.

- $R' : (BST \times \mathbb{Z}) \times (BST \times \mathbb{Z})$

$$(t_1, x_1) \, R' \, (t_2, x_2) \iff \exists f : \mathbb{Z} \rightharpoonup \mathbb{Z}. \operatorname{dom} f \supset elements(t_1) \cup \{x_1\} \land f \text{ is strictly increasing} \land f(x_1) = x_2 \land f(t_1) = t_2. \quad \text{(B.2)}$$

  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,

$$(BST_n \times \mathbb{Z})/R' = \biguplus_{T \in BST_n/R} (T \times \mathbb{Z})/R'.$$

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

$$|(T \times \mathbb{Z})/R'| = 2 \times \#nodes(T) + 1,$$

where $\#nodes(T)$ is the number of nodes of any tree in T. Clearly, the all the trees in T have the same shape, $\#nodes(T)$ is well-defined. Suppose $\#nodes(T) = k$ and we define a tree $t \in T$ which has elements: $2, 4, \ldots, 2k$. Then define $P = \{ (t, i) \mid 1 \le i \le 2k + 1 \}$. 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 $|(T \times \mathbb{Z})/R'| = |P| = 2 \times \#nodes(T) + 1$. 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 than $f(t') = t$. Then we know $f(e_i) = 2i$ for all $1 \le i \le k$. If $x = e_i$ for some $1 \le i \le k$, 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:

$$f'(y) = \begin{cases} f(y) & \text{if } f(y) \text{ is defined and } y \ge e_2 \text{ or } e_1 \le y \\ 3 & \text{if } y = x \\ undefined & \text{otherwise} \end{cases}.$$

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

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

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

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

We get

$$\begin{aligned}
a_1 &= 1 + a_0^2 = 2 \\
a_2 &= 1 + a_1^2 = 1 + 2^2 = 5 \\
a_3 &= 1 + a_2^2 = 1 + 5^2 = 26 \\
a_4 &= 1 + a_3^2 = 1 + 26^2 = 677 \\
&\vdots
\end{aligned}$$

This sequence grows double exponentially. In fact, Aho[1] 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 $|\{ [t]_R \mid t \in BST_n \wedge t \text{ has } m \text{ nodes} \}|$, 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 $c_n = \sum_{0 \leq i}(2i+1)b(i,n)$. [1] Define a generating function for $b(m,n)$ as

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

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

$$b(m,n) = \sum_{i+j=m-1} b(i,n-1)b(j,n-1) \quad m > 0, n \geq 1 \tag{B.5}$$

[1]This result allows the duplicated resulting trees and corresponds to the stateless case.

After multiplying x^m to both sides of (B.5) and summing over $1 \le m \le \infty$, we get

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

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 $a_n = \sum_{m\ge0} b(m,n)$. Thus $a_n = T_n(1)$ and for $x = 1$ (B.6) becomes (B.3) as expected.

Next, define generating function

$$G_n(x) = \sum_{m\ge0}(2m+1)b(m,n)x^m \quad n \ge 0. \tag{B.7}$$

Then

$$\begin{aligned} G_n(x) &= \sum_{m\ge0}(2m+1)b(m,n)x^m \\ &= 2\sum_{m\ge0} mb(m,n)x^m + \sum_{m\ge0} b(m,n)x^m \\ &= 2xT_n'(x) + T_n(x). \end{aligned}$$

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

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

We can get

$$\begin{aligned} &T_1'(1) = 1 \\ &T_2'(1) = 2 \times T_1(1)T_1'(1) + (T_1(1))^2 = 4 + 4 = 8 \\ &T_3'(1) = 2 \times T_2(1)T_2'(1) + (T_2(1))^2 = 2 \times 5 \times 8 + 5^2 = 105 \\ &T_4'(1) = 2 \times T_3(1)T_3'(1) + (T_3(1))^2 = 2 \times 26 \times 105 + 26^2 = 6136 \\ &\qquad \vdots \end{aligned}$$

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

$$\begin{aligned} c_0 &= G_0(1) = 2T_0'(1) + T_0(1) = 1, \\ c_1 &= G_1(1) = 2T_1'(1) + T_1(1) = 2 \times 1 + 2 = 4, \\ c_2 &= G_2(1) = 2T_2'(1) + T_2(1) = 2 \times 8 + 5 = 21, \\ c_3 &= G_3(1) = 2T_3'(1) + T_3(1) = 2 \times 105 + 26 = 236, \\ c_4 &= G_4(1) = 2T_4'(1) + T_4(1) = 2 \times 6136 + 667 = 12939, \\ &\qquad \vdots \end{aligned}$$

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,

$$f_n = |\{\, \mathtt{insert}(t, x)/R \mid t \in BST_n \wedge x \in \mathbb{Z}\,\}|, \tag{B.8}$$

where $\mathtt{insert}(t, x)$ 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

$$f_h = a_h + e_h - 1 \quad h > 1, \tag{B.9}$$

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 $h > 1$, $e_h = 2F_h'(1) = 2\left((1+x)^{2^{h-1}}\right)'(1) = 2^h(1+1) = 2^{h+1}$. Thus

$$\begin{aligned}
f_0 &= 1 + 2 - 1 = 1, \\
f_1 &= 2 + 2 - 1 = 3, \\
f_2 &= 5 + 8 - 1 = 12, \\
f_3 &= 26 + 16 - 1 = 41, \\
f_4 &= 677 + 2^5 - 1 = 708, \\
&\vdots
\end{aligned}$$

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

$$a(n,k) = |\{ t \mid t \in RBT_n \wedge blackheight(t) = k \} / R| \tag{B.10}$$

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 $k-1$.

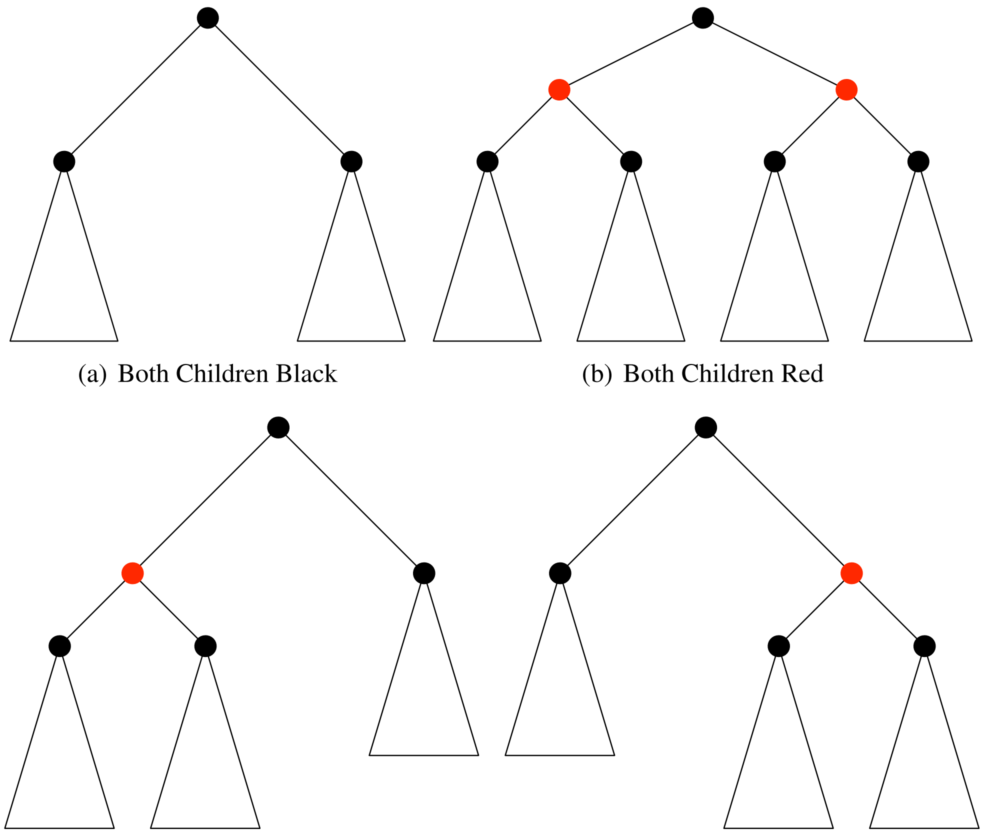
2. left child is black but right child is red as shown in Figure B.1(d). Then two subtrees of left child have height less than or equal to $n-2$ and black height $k-1$. Right child of the root has height less than $n-1$ and black height $k-1$.

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

$$\begin{aligned} a(n,k) = a(n-1,k-1)^2 + 2a(n-1,k-1)a(n-2,k-1)^2 + a(n-2,k-1)^4 \\ = [a(n-1,k-1) + a(n-2,k-1)^2]^2, \quad n \geq 1, k \geq 1. \end{aligned}$$



(a) Both Children Black

(b) Both Children Red

(c) Left Child Red and Right Child Black

(d) Left Child Black and Right Child Red

Figure B.1: *Red Black Counting Cases*

Here we let $a(n,k) = 0$ for $n < 0$. Then we get $a(1,1) = 1$.

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

Let $b_n = |RBT_n/R|$ 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

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

B.2.2 Counting $(RBT_n \times \mathbb{Z})/R'$

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

$$f(n,h,k) = |\{\, t \in RBT_h \mid blackheight(t) = k \wedge leaf(t) = n \,\}/R|, \tag{B.11}$$

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

$$
\begin{aligned}
f(n,h,k) = \sum_{i+j=n} f(i,h{-}1,b{-}1)a(j,h{-}1,b{-}1) &+ \sum_{i+j+k=n} f(i,h{-}2,b{-}1)f(j,h{-}2,b{-}1)f(k,h{-}1,b{-}1)\\
&+ \sum_{i+j+k=n} f(i,h-1,b-1)f(j,h-2,b-1)f(k,h-2,b-1)\\
&+ \sum_{i+j+k+l=n} a(i,h-2,b-1)a(j,h-2,b-1)a(k,h-2,b-1)a(l,h-2,b-1),
\end{aligned}
$$

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 $F_{h,b}(x) = \sum_{n=1}^{\infty} a(n,h,b)x^n$ for $k \geq 0$. Then we get

$$F_{h,b}(x) = F^2_{h-1,b-1}(x) + 2F^2_{h-2,b-1}(x)F_{h-1,b-1}(x) + F^4_{h-2,b-1}(x) = (F_{h-1,b-1}(x) + F^2_{h-2,b-1}(x))^2.$$

The boundary condition is

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

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

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

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

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 $G_h(x) = g(h,0) + g(h,1)x + g(h,2)x^2 + \cdots$:

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

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

$$p_h = \sum_{i=0} (2i - 1)g(h,i).$$

[2] This is because for a n node binary tree, it has $n + 1$ NULL leaves [5].

Then we have $p_h = 2G'(1) - G(1)$.

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

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

$$
\begin{aligned}
a_1 &= a_0^2 = 1\\
a_2 &= a_1^2 + 2a_0a_1 = 3\\
a_3 &= a_2^2 + 2a_2a_1 = 15\\
a_4 &= a_3^2 + 2a_3a_2 = 15^2 + 2 \times 15 \times 3 = 315\\
&\vdots
\end{aligned}
$$

B.3.2 Counting $(AVL_n \times \mathbb{Z})/R'$

Define $b(h,n)$ be $|\{\, [t]_R \mid t \in AVL_n \wedge t$ has n nodes $\}|$ (not counting leaf nodes 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

$$H_h(x) = xH_{h-1}^2(x) + 2xH_{i-1}(x)H_{i-2}(x). \tag{B.15}$$

So we get

$$\begin{aligned}
H_1(x) &= x \\
H_2(x) &= x \times x^2 + 2x \times x = x^3 + 2x^2 \\
H_3(x) &= 4x^4 + 6x^5 + 4x^6 + x^7 \\
&\vdots
\end{aligned}$$

Now we will compute number $c_h = |(AVL_h \times \mathbb{Z})/R'|$, the total number of outcomes after `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

$$H_h'(1) = H_{h-1}^2(1) + H_{h-1}'(1)H_{h-1}(1) + 2H_{h-1}'(1)H_{h-2}(1) + 2H_{h-1}(1)H_{h-2}'(1).$$

Then we get $H_0'(1) = 0, H_1'(1) = 1, H_2'(1) = 7, H_3'(1) = 77, \ldots$. Therefore,

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

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 $\mathrm{Seq}(D)$ is the set of all sequences of elements in D:

- the substitution function: $\mathit{sub} : D \times (D \rightharpoonup D) \to D$ as

$$\mathit{sub}(d, g) = \begin{cases} g(d) & \text{if } d \in \operatorname{dom} g; \\ d & \text{otherwise.} \end{cases}$$

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

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

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

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

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

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

Then we introduce some simple properties of the substitution functions:

Lemma 1. *Suppose partial function* $g : D \rightharpoonup D$ *for some domain* D *satisfies* $\operatorname{dom} g \cap \operatorname{ran} g = \emptyset$*. Then for any* $(d, d') \in g$ *and function* $f : D' \rightharpoonup D$*, sequence* $q : Seq(D)$*, and funtional* $f^{ho} : D'' \rightharpoonup D' \rightharpoonup D$*, we have* $\textit{sub-fun}(f, g) = \textit{sub-fun}(\textit{sub-fun}_1(f, d, d'), g)$*,* $\textit{sub-seq}(q, g) = \textit{sub-seq}(\textit{sub-seq}_1(q, d, d'), g)$*,* $\textit{sub-fun2}(f^{ho}, g) = \textit{sub-seq}(\textit{sub-fun2}_1(f^{go}, d, d'), g)$*.*

Lemma 2. *if* R *be the range of* $f : D' \rightharpoonup D$*, the set of elements in a sequence* $q : Seq(D)$ *or the second range of* $f^{ho} : D'' \rightharpoonup D' \rightharpoonup D$*, then for any* $g : D \rightharpoonup D$*,* $\textit{sub-fun}(f, g) = \textit{sub-fun}(f, g\,|_{R \cap \operatorname{dom} g})$*,* $\textit{sub-seq}(q, g) = \textit{sub-seq}(q, g\,|_{R \cap \operatorname{dom} g})$*,* $\textit{sub-fun2}(f^{ho}, g) = \textit{sub-fun2}(f^{ho}, g\,|_{R \cap \operatorname{dom} g})$*.*

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, `assume` and `assert`. First, the semantics domains are introduced. Then some auxiliary functions that facilitate the definition of semantic rules are defined. Finally the semantic rules for bytecode instructions and `assume/assert` are presented.

Semantic Domains

The semantic/syntactical domains are listed as following:

- the set of primitive types, $\textbf{Types}_{prim}$, consisting of INT, CHAR, etc.,
- the set of array types, $\textbf{Types}_{array}$,
- the set of record types, $\textbf{Types}_{record}$,
- the set of symbolic types, **SymTypes**,
- the set of non-primitive types, $\textbf{Types}_{non-prim} = \textbf{Types}_{record} \uplus$ [1] $\textbf{Types}_{array} \uplus \textbf{SymTypes}$,
- the set of all types, $\textbf{Types} = \textbf{Types}_{prim} \uplus \textbf{Types}_{non-prim}$,
- the set of program counters, **PCs**
- the set of boolean expressions, Φ,
- the set of locations, **Locs**,

[1] $\uplus$ 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, $\mathbf{Symbols}_{\mathrm{INT}}$,
- the set of primitive symbols, $\mathbf{Symbols}_{prim}$, including $\mathbf{Symbols}_{\mathrm{INT}}$,
- the set of values, $\mathbf{Values} = \mathbf{Consts} \uplus \mathbf{Locs} \uplus \mathbf{Symbols}_{prim}$,
- the set of indexes, $\mathbf{Indexes} = \mathbf{Fields} \uplus \mathbb{N} \uplus \mathbf{Symbols}_{\mathrm{INT}}$,
- the set of non-primitive symbols, $\mathbf{Symbols}_{non-prim} = \{ X_\tau^{m,n} \mid X_\tau^{m,n} : \mathbf{Indexes} \rightharpoonup \mathbf{Values} \}$,
- the set of symbols, $\mathbf{Symbols} = \mathbf{Symbols}_{prim} \uplus \mathbf{Symbols}_{non-prim}$,
- the set of globals, $\mathbf{Globals} = \{ g \mid g : \mathbf{Fields} \rightharpoonup \mathbf{Values} \}$,
- the set of operand stacks, $\mathbf{Stacks} = \{ \omega \mid \omega : \mathrm{Seq}(\mathbf{Values}) \}$, all sequences of values,
- the set of locals, $\mathbf{Locals} = \{ l \mid l : \mathbb{N} \rightharpoonup \mathbf{Values} \}$,
- the set of heaps, $\mathbf{Heaps} = \left\{ h \mid h : \mathbf{Locs} \rightharpoonup \mathbf{Symbols}_{non-prim} \right\}$,
- the set of bytecode instruction with additional `assert` and `assume` instructions, **Instrs**,

We follow Java type system in the semantic domains: we use $\mathbf{Types}_{prim}$ to model the primitive types and $\mathbf{Types}_{non-prim}$ for the reference types which are divided into object types ($\mathbf{Types}_{record}$), array types ($\mathbf{Types}_{array}$), 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, eof, 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 $\mathbf{Symbols}_{non-prim}$ domain, $X_\tau^{m,n}$, has three properties (we often omit properties when they are not important/applicable): τ is the type of the symbol, m is the object field or array element expansion bound, and n is the number of array elements bound. (We will discuss the difference between m and n for arrays at the end of this section.) And each non-primitive symbol, $X_\tau^{m,n}$, is modeled as a partial mapping from its fields to values. Each primitive symbol X_τ or field f_τ also has a property of its type τ. 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 ϕ (as a conjunctive-set of formulas) as another state component. So the definition of the set of symbolic states is :

$$\Sigma_s = \mathbf{Globals} \times \mathbf{PCs} \times \mathbf{Locals} \times \mathbf{Stacks} \times \mathbf{Heaps} \times \Phi$$

and we let σ ranges over Σ_s.

We will follow the convention that

- τ ranges over types, **Types**,
- pc ranges over program counters, **PCs**,
- ϕ ranges over boolean expressions, Φ,
- 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**,
- ι ranges over indexes, **Indexes**,
- g ranges over globals, **Globals**,
- ω 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 auxilurary functions to facilitate the definition of operational semantics:

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

Semantics Rules

Given an array of instructions, we define a function $\mathit{code} : \mathbf{PCs} \rightharpoonup \mathbf{Instrs}$ 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

$$\frac{pre}{\sigma \Rightarrow_{\mathcal{S}} \sigma_1[\| \ \sigma_2] \mid \textsc{Exception}, \sigma' | \textsc{Error}, \sigma''}$$

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 σ, if *pre* is satisfied, after executing instruction pointed by the program counter component of σ, the resulting state is σ_1 or nondeterministically σ_1 or σ_2; or an *exception* thrown with a state σ'; or Error with a state σ''. Exceptions are handled the same way as JVM specification [4] does. If an error occurred, then the program stops. For simplicity, we assume that garbage collection is performed after each transition. Moreover, we stop exploring paths whose state's path condition is unsatisfiable.

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

- Arithmetic instructions: Instruction `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` τ, `getfield` f, `putfield` f, `instanceof` τ, and `checkcast` τ are presented. Accesses to symbolic objects (*e.g.*, `getfield` f) operate according to the *lazy initialization* algorithm described previously. Similar to [3], we limit the choosing range to symbolic objects/arrays by introducing an additional field, 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` τ instruction during the execution) are reachable through object expansion; concretely, this only happens through assignments.

  - Instruction `new` τ creates a fresh object of type τ 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` τ 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` τ: 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` τ tests whether an object is a type of τ. 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 τ, false otherwise. Rule INSTANCEOF1-S represents the NULL case and INSTANCEOF2-S does the non-NULL case.
- Instruction `checkcast` τ 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` τ, `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` τ 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.

$$\text{IADD-S}\ \frac{code(pc) = \texttt{iadd} \qquad \omega = v_1 :: v_2 :: \omega'}{\sigma \Rightarrow_S (g, next(pc), l, Y :: \omega', h, \phi \cup \{Y = v_1 + v_2\})}$$

where $Y = \textit{new-prim-sym}(\textsc{int}, symbols(\sigma))$

$$\text{IF\_ICMPLT-S}\ \frac{code(pc) = \texttt{if\_icmplt}\ pc' \qquad \omega = v_1 :: v_2 :: \omega'}{\sigma \Rightarrow_S (g, next(pc), l, \omega', h, \phi \cup \{v_2 \not< v_1\}) \parallel (g, pc', l, \omega', h, \phi \cup \{v_2 < v_1\})}$$

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

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

$$\text{GETFIELD2-S}\ \frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = i :: \omega' \qquad h(i)(f_\tau)\uparrow \qquad \tau \in \mathbf{Types}_{prim}}{\sigma \Rightarrow_S (g, next(pc), l, X :: \omega', h[i \mapsto h(i)[f \mapsto X]], \phi)}$$

where $X = \textit{new-prim-sym}(\tau, symbols(\sigma))$

$$\text{GETFIELD3-S}\ \frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = i :: \omega' \qquad h(i)(f_\tau)\uparrow \qquad \tau \in \mathbf{Types}_{non-prim}}{\sigma \Rightarrow_S (g, next(pc), l, \textsc{null} :: \omega', h[i \mapsto h(i)[f \mapsto \textsc{null}]], \phi)}$$

$$\text{GETFIELD4-S}\ \frac{\begin{array}{c} code(pc) = \texttt{getfield}\ f_\tau \\ \omega = i :: \omega' \qquad h(i)(f_\tau)\uparrow \qquad \tau \in \mathbf{Types}_{non-prim} \qquad j \in collect(h) \qquad Z_{\tau'} = h(j) \end{array}}{\sigma \Rightarrow_S (g, next(pc), l, j :: \omega', h[i \mapsto h(i)[f \mapsto j]], \phi \cup \{\tau' <: \tau\})}$$

$$\text{GETFIELD5-S}\ \frac{\begin{array}{c} code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = i :: \omega' \\ h(i)(f_\tau)\uparrow \qquad \tau \in \mathbf{Types}_{array} \qquad Y^{m,n} = h(i) \qquad m > 0 \qquad j \notin \operatorname{dom} h \end{array}}{\sigma \Rightarrow_S (g, next(pc), l, j :: \omega', h[i \mapsto h(i)[f \mapsto j]][j \mapsto Z_{\tau'}], \phi \cup \{\tau' <: \tau, Z(\textsc{len}) \geq 0\})}$$

where $Z_{\tau'} = \textit{new-sarr}(symbols(\sigma), m - 1, k)$

$$\text{GETFIELD6-S}\ \frac{\begin{array}{c} code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = i :: \omega' \\ h(i)(f_\tau)\uparrow \qquad \tau \in \textbf{Types}_{record} \qquad Y^{m,n} = h(i) \qquad m > 0 \qquad j \notin \operatorname{dom} h \end{array}}{\sigma \Rightarrow_S (g, next(pc), l, j :: \omega', h[i \mapsto h(i)[f \mapsto j]][j \mapsto Z_{\tau'}], \phi \cup \{\tau' <: \tau\})}$$

$$\text{where } Z_{\tau'} = new\text{-}sym(symbols(\sigma), m-1, k)$$

$$\text{GETFIELD7-S}\ \frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = \text{NULL} :: \omega'}{\sigma \Rightarrow_S \text{NullPointerException}, (g, pc, l, \omega', h, \phi)}$$

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

$$\text{PUTFIELD2-S}\ \frac{code(pc) = \texttt{putfield}\ f \qquad \omega = v :: \text{NULL} :: \omega'}{\sigma \Rightarrow_S \text{NullPointerException}, (g, pc, l, \omega', h, \phi)}$$

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

$$\text{ANEWARRAY2-S}\ \frac{code(pc) = \texttt{anewarray}\ \tau \qquad \omega = X :: \omega' \qquad i \notin \operatorname{dom} h}{\begin{array}{c}\sigma \Rightarrow_S (g, next(pc), l, \omega', h[i \mapsto new\text{-}arr(symbols(\sigma), \tau, X, k)], \phi \cup \{X \geq 0\}) \parallel \\ \text{NegativeArraySizeException}, (g, pc, l, \omega', h, \phi \cup \{X < 0\})\end{array}}$$

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

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

$$\text{IASTORE3-S}\ \frac{\begin{array}{c} code(pc) = \texttt{iastore} \\ \omega = v :: \iota :: i :: \omega' \qquad Z^{m,n} = h(i) \qquad n > 0 \qquad I = acc\text{-}idx(Z) \end{array}}{\begin{array}{c}\sigma \Rightarrow_S (g, next(pc), l, \omega', h[i \mapsto Z^{m,n-1}[\iota \mapsto v]], \phi \cup \{\, \iota \neq \iota' \mid \iota' \in I\,\} \\ \cup \{0 \leq \iota, \iota < Z(\textsc{len}), |I| < Z(\textsc{len})\})\end{array}}$$

$$\text{IASTORE4-S}\ \frac{code(pc) = \texttt{iastore} \qquad \omega = v :: \iota :: \text{NULL} :: \omega'}{\sigma \Rightarrow_S \text{NullPointerException}, (g, pc, l, \omega', h, \phi)}$$

$$\text{IALOAD1-S}\ \frac{code(pc) = \texttt{iaload} \qquad \omega = \iota :: i :: \omega'}{\sigma \Rightarrow_S \text{ArrayIndexOutOfBoundsException}, (g, pc, l, \omega', h, \phi \cup \{\iota < 0 \vee h(i)(\textsc{len}) \leq \iota\})}$$

$$\text{IALOAD2-S}\ \frac{code(pc) = \texttt{iaload} \qquad \omega = \iota :: i :: \omega' \qquad Z = h(i) \qquad \iota' \in acc\text{-}idx(Z)}{\sigma \Rightarrow_S (g, next(pc), l, Z(\iota') :: \omega', h, \phi \cup \{\iota = \iota'\})}$$

$$\text{IALOAD3-S}\ \frac{code(pc) = \texttt{iaload} \qquad \omega = \iota :: i :: \omega' \qquad Z^{m,n} = h(i) \qquad I = acc\text{-}idx(Z^{m,n})}{\begin{array}{c}\sigma \Rightarrow_S (g, next(pc), l, v :: \omega', h[i \mapsto Z^{m,n-1}[\iota \mapsto v]], \phi \cup \{\, \iota' \neq \iota \mid \iota' \in I\,\} \\ \cup \{0 \leq \iota, \iota < Z^{m,n}(\textsc{len}), |I| < Z^{m,n}(\textsc{len}), n > 0\})\end{array}}$$

$$\text{where } v = \begin{cases} Z^{m,n}(\textsc{def}) & \text{if } Z^{m,n}(\textsc{def})\downarrow \\ new\text{-}prim\text{-}sym(\text{INT}, symbols(\sigma)) & \text{if } Z^{m,n}(\textsc{def})\uparrow \end{cases}$$

$$\text{IALOAD4-S}\ \frac{code(pc) = \texttt{iaload} \qquad \omega = \iota :: \text{NULL} :: \omega'}{\sigma \Rightarrow_S \text{NullPointerException}, (g, pc, l, \omega', h, \phi)}$$

$$\text{IF\_ACMPEQ1-S}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = j :: i :: \omega' \qquad i \neq j}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \omega', h, \phi)}$$

$$\text{IF\_ACMPEQ2-S}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = j :: i :: \omega' \qquad i = j}{\sigma \Rightarrow_{\mathcal{S}} (g, pc', l, \omega', h, \phi)}$$

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

$$\text{IFNULL2-S}\ \frac{code(pc) = \texttt{ifnull}\ pc' \qquad \omega = \text{NULL} :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} (g, pc', l, \omega', h, \phi)}$$

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

$$\text{IFNONNULL2-S}\ \frac{code(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = \text{NULL} :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \omega', h, \phi)}$$

$$\text{INSTANCEOF1-S}\ \frac{code(pc) = \texttt{instanceof}\ \tau \qquad \omega = \text{NULL} :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, 1 :: \omega', h, \phi)}$$

$$\text{INSTANCEOF2-S}\ \frac{code(pc) = \texttt{instanceof}\ \tau \qquad \omega = i :: \omega' \qquad X_{\tau'} = h(i)}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, 1 :: \omega', h, \phi \cup \{\tau' <: \tau\}) \parallel (g, next(pc), l, 0 :: \omega', h, \phi \cup \{\tau' \not<: \tau\})}$$

$$\text{CHECKCAST1-S}\ \frac{code(pc) = \texttt{checkcast}\ \tau \qquad \omega = \text{NULL} :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \text{NULL} :: \omega', h, \phi)}$$

$$\text{CHECKCAST2-S}\ \frac{code(pc) = \texttt{checkcast}\ \tau \qquad \omega = i :: \omega' \qquad X_{\tau'} = h(i)}{\begin{array}{c}\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, i :: \omega', h, \phi \cup \{\tau' <: \tau\}) \parallel \\ \text{ClassCastException}, (g, pc, l, i :: \omega', h, \phi \cup \{\tau' \not<: \tau\})\end{array}}$$

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

$$\text{ASSERT-S}\ \frac{code(pc) = \texttt{assert} \qquad \omega = v :: \omega'}{\sigma \Rightarrow_{\mathcal{S}} (g, next(pc), l, \omega', h, \phi \cup \{v = 1\}) \parallel \text{ERROR}, (g, pc, l, \omega', h, \phi \cup \{v = 0\})}$$

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 $\delta_\tau^{m,n}$ has the same three properties as non-primitive symbols do. Clearly, we need to add the symbolic locations into values. So we have **Values** = **Consts** $\cup$ **Locs** $\cup$ **Symbols**$_{prim}$ $\cup$ **SymLocs**. We use Σ_a[3] to denote the set of lazier states. The only difference between lazier and symbolic states is that the lazier states can have symbolic location. Thus $\Sigma_a \supset \Sigma_s$.

[3]Subscript a denotes that the component is a part of lazier state.

Auxiliary Functions

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

$$\begin{aligned}
&\textit{init-loc-heap} : (\mathbf{Heaps}_a \times \mathcal{P}(\mathbf{Symbols}) \times \mathbf{SymLocs} \times \mathbf{Locs}) \to (\mathbf{Heaps}_a \times \Phi))\\
&\textit{init-sym-loc} : \Sigma_a \times \mathbf{SymLocs} \times \mathbf{Locs} \to \Sigma_a\\
&\textit{init-sym-loc}^* : \Sigma_a \times \mathbf{SymLocs} \to \mathcal{P}(\Sigma_a).
\end{aligned}$$

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

- the *init-loc-heap* function: $\textit{init-loc-heap}(h_a, ss, \delta_\tau^{m,n}, i) = (h'_a, \phi')$ where
  - if $i \in \operatorname{dom} h_a$: $h'_a = \textit{sub-fun2}_1(h_a, \delta, i)$ and
  $$\phi' = \{\tau' <: \tau\} \text{ where } h_a(i) = X_{\tau'}.$$
  - if $i \notin \operatorname{dom} h_a$:
  $$\operatorname{dom} h'_a = \operatorname{dom} h_a \cup \{i\}$$
  and
  $$\forall j \in \operatorname{dom} h_a . h'_a(j) = \textit{sub-fun}_1(h_a(j), \delta_\tau, i)$$
  and $h'_a(i) = X_{\tau'}$ where
  $$X_{\tau'} = \begin{cases} \textit{new-sarr}(ss, m, k) & \text{if } \tau \in \mathbf{Types}_{array} \\ \textit{new-sym}(ss, m, k) & \text{if } \tau \in \mathbf{Types}_{record} \end{cases}$$
  and
  $$\phi' = \begin{cases} \{X(\textsc{len}) \geq 0, \tau <: \tau'\} & \text{if } \tau \in \mathbf{Types}_{array} \\ \{\tau' <: \tau\} & \text{if } \tau \in \mathbf{Types}_{record} \end{cases}.$$
- *init-sym-loc* function,
$$\begin{aligned}&\textit{init-sym-loc} = \lambda(\sigma_a, \delta_\tau^{m,n}, i).\{(\textit{sub-fun}_1(g, \delta, i), pc, \textit{sub-fun}_1(l, \delta, i), \textit{sub-seq}_1(\omega, \delta, i),\\ &\#1(\textit{init-loc-heap}(h, \textit{symbols}(\sigma_a), \delta_\tau^{m,n}, i)), \#2(\textit{init-loc-heap}(h, \textit{symbols}(\sigma_a), \delta_\tau^{m,n}, i)) \cup \phi)\}\end{aligned}$$
- *init-sym-loc** function,
$$\begin{aligned}\textit{init-sym-loc}^* = \lambda(\sigma_a, \delta_\tau^{m,n}).\{&\textit{init-sym-loc}(\sigma_a, \delta_\tau^{m,n}, i) \mid i \in \textit{collect}(h)\\ &\text{or } i \in (\mathbf{Locs} \setminus \operatorname{dom} h) \text{ if } m \geq 0\}.\end{aligned}$$

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.

$$\text{IF\_ACMPEQ1-A}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = \delta_\tau^{m,n} :: \delta_\tau^{m,n} :: \omega'}{\sigma \Rightarrow_{\mathcal{A}} (g, pc', \omega', h, \phi))}$$

$$\text{IF\_ACMPEQ2-A}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = v :: \delta_\tau^{m,n} :: \omega'}{\sigma \Rightarrow_{\mathcal{A}} \sigma' \qquad \text{where } \sigma' \in \textit{init-sym-loc}^*(\sigma, \delta_\tau^{m,n})}$$

$$\text{IF\_ACMPEQ3-A}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = \delta_\tau^{m,n} :: v :: \omega'}{\sigma \Rightarrow_{\mathcal{A}} \sigma' \qquad \text{where } \sigma' \in \textit{init-sym-loc}^*(\sigma, \delta_\tau^{m,n})}$$

$$\text{IFNULL-A}\ \frac{code(pc) = \texttt{ifnull}\ pc' \qquad \omega = \delta :: \omega'}{\sigma \Rightarrow_{\mathcal{A}} (g, next(pc), l, \omega', h, \phi)}$$

$$\text{IFNONNULL-A}\ \frac{code(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = \delta :: \omega'}{\sigma \Rightarrow_{\mathcal{A}} (g, pc', l, \omega', h, \phi)}$$

$$\text{GETFIELD1-A}\ \frac{code(pc) = \texttt{getfield}\ f_\tau \qquad \omega = \delta_\tau^{m,n} :: \omega'}{\sigma \Rightarrow_{\mathcal{A}} \sigma' \qquad \text{where } \sigma' \in \textit{init-sym-loc}^*(\sigma, \delta_\tau^{m,n})}$$

$$\text{GETFIELD2-A}\ \frac{\begin{array}{c} code(pc) = \texttt{getfield}\ f_\tau \\ \omega = i :: \omega' \qquad Y^{m,n} = h(i) \qquad Y(f_\tau)\uparrow \qquad \tau \in \mathbf{Types}_{non-prim} \qquad \delta \text{ is fresh} \end{array}}{\sigma \Rightarrow_{\mathcal{A}} (g, next(pc), l, \delta_\tau^{m-1,k} :: \omega', h[i \mapsto Y^{m,n}[f_\tau \mapsto \delta_\tau^{m-1,k}]], \phi)}$$

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 $\hat{\delta}$ ranges over **SymRefs** and each $\hat{\delta}_\tau^{m,n}$ just like $\delta_\tau^{m,n}$ except that it can be initialized to NULL. Clearly, we need to add the new domain into the domain

Values. So we have

$$\textbf{Values} = \textbf{Consts} \cup \textbf{Locs} \cup \textbf{Symbols}_{prim} \cup \textbf{SymLocs} \cup \textbf{SymRefs}.$$

We use Σ_b[4] to denote the set of lazier# states. Clearly $\Sigma_b \supset \Sigma_a$.

Auxiliary Functions

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

$$\begin{aligned}&\textit{init-sym-ref} : \Sigma_b \times \textbf{SymRefs} \times (\textbf{SymLocs} \cup \{\textsc{null}\}) \rightarrow \Sigma_b\\ &\textit{init-sym-ref}^* : \Sigma_b \times \textbf{SymRefs} \rightarrow \mathcal{P}(\Sigma_b).\end{aligned}$$

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

- *init-sym-ref* function,

$$\begin{aligned}\textit{init-sym-ref}(\sigma_b, \hat{\delta}, \textsc{null}) = \{(&\textit{sub-fun}_1(g, \hat{\delta}, \textsc{null}), pc, \textit{sub-fun}_1(l, \hat{\delta}, \textsc{null}),\\ &\textit{sub-seq}_1(\omega, \hat{\delta}, \textsc{null}), \textit{sub-fun2}_1(h, \hat{\delta}, \textsc{null}), \phi)\}\end{aligned}$$

  and

$$\begin{aligned}\textit{init-sym-ref}(\sigma_b, \hat{\delta}, \delta) = \{(&\textit{sub-fun}_1(g, \hat{\delta}, \delta), pc, \textit{sub-fun}_1(l, \hat{\delta}, \delta), \textit{sub-seq}_1(\omega, \hat{\delta}, \delta),\\ &\textit{sub-fun2}_1(h, \hat{\delta}, \delta), \phi)\}\end{aligned}$$

- *init-sym-ref** function,

$$\begin{aligned}\textit{init-sym-ref}^*(\sigma_b, \hat{\delta}) = &\{\textit{init-sym-ref}(\sigma_b, \hat{\delta}, \delta) \mid \delta \notin \textit{collect-sym-locs}(\sigma_b)\}\\ &\cup \{\textit{init-sym-ref}(\sigma_b, \hat{\delta}, \textsc{null})\}.\end{aligned}$$

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

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

[4] Subscript b denotes that the component is a part of lazier# state.

$$\text{IF\_ACMPEQ1-B}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = \hat{\delta}^{m,n}_{\tau} :: \hat{\delta}^{m,n}_{\tau} :: \omega'}{\sigma \Rightarrow_{\mathcal{B}} (g, pc', \omega', h, \phi)}$$

$$\text{IF\_ACMPEQ2-B}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = v :: \hat{\delta}^{m,n}_{\tau} :: \omega'}{\sigma \Rightarrow_{\mathcal{B}} \sigma' \qquad \text{where } \sigma' \in \textit{init-sym-ref}^{*}(\sigma, \hat{\delta}^{m,n}_{\tau})}$$

$$\text{IF\_ACMPEQ3-B}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = \hat{\delta}^{m,n}_{\tau} :: v :: \omega'}{\sigma \Rightarrow_{\mathcal{B}} \sigma' \qquad \text{where } \sigma' \in \textit{init-sym-ref}^{*}(\sigma, \hat{\delta}^{m,n}_{\tau})}$$

$$\text{GETFIELD1-B}\ \frac{code(pc) = \texttt{getfield}\ f_{\tau} \qquad \omega = \hat{\delta}^{m,n}_{\tau} :: \omega'}{\sigma \Rightarrow_{\mathcal{B}} \sigma' \qquad \text{where } \sigma' \in \textit{init-sym-ref}^{*}(\sigma, \hat{\delta}^{m,n}_{\tau})}$$

$$\text{GETFIELD2-B}\ \frac{\begin{array}{c} code(pc) = \texttt{getfield}\ f_{\tau} \\ \omega = i :: \omega' \qquad Y^{m,n} = h(i) \qquad Y(f_{\tau})\uparrow \qquad \tau \in \mathbf{Types}_{non-prim} \qquad \hat{\delta} \text{ is fresh} \end{array}}{\sigma \Rightarrow_{\mathcal{B}} (g, next(pc), l, \hat{\delta}^{m-1,k}_{\tau} :: \omega', h[i \mapsto Y^{m,n}[f_{\tau} \mapsto \hat{\delta}^{m-1,k}_{\tau}]], \phi)}$$

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:

$$\sigma_c \in \Sigma_c = \mathbf{Globals} \times \mathbf{PCs} \times \mathbf{Locals} \times \mathbf{Stacks} \times \mathbf{Heaps} \times \textsc{Boolean}.$$

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

We also need to change the definition of *new-arr* to $\textit{new-arr}_c : \mathcal{P}(\mathbf{Symbols}) \times \mathbf{Types} \times \mathbb{N} \rightarrow \mathbf{Symbols}_{non-prim} =$

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

The concrete JVM bytecode operational semantics is listed below. We use the binding $\sigma = (g, pc, l, \omega, h, \textsc{True})$ for all the rules. When the last component of the end state is $\textsc{False}$, 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.

$$\text{IADD-C}\ \frac{code(pc) = \texttt{iadd} \qquad \omega = c :: d :: \omega'}{\sigma \Rightarrow_{C} (g, next(pc), l, (c + d) :: \omega', h, \textsc{True})}$$

$$\text{IF\_ICMPLT1-C}\ \frac{code(pc) = \texttt{if\_icmplt}\ pc' \qquad \omega = d :: c :: \omega' \qquad c < d}{\sigma \Rightarrow_{C} (g, pc', l, \omega', h, \textsc{True})}$$

$$\text{IF\_ICMPLT2-C}\ \frac{code(pc) = \texttt{if\_icmplt}\ pc' \qquad \omega = d :: c :: \omega' \qquad c \nless d}{\sigma \Rightarrow_C (g, next(pc), l, \omega', h, \textsc{True})}$$

$$\text{IF\_ACMPEQ1-C}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = i :: j :: \omega' \qquad i \neq j}{\sigma \Rightarrow_C (g, next(pc), l, \omega', h, \textsc{True})}$$

$$\text{IF\_ACMPEQ2-C}\ \frac{code(pc) = \texttt{if\_acmpeq}\ pc' \qquad \omega = i :: j :: \omega' \qquad i = j}{\sigma \Rightarrow_C (g, pc', l, \omega', h, \textsc{True})}$$

$$\text{IFNULL1-C}\ \frac{code(pc) = \texttt{ifnull}\ pc' \qquad \omega = i :: \omega'}{\sigma \Rightarrow_C (g, next(pc), l, \omega', h, \phi)}$$

$$\text{IFNULL2-C}\ \frac{code(pc) = \texttt{ifnull}\ pc' \qquad \omega = \textsc{null} :: \omega'}{\sigma \Rightarrow_C (g, pc', l, \omega', h, \phi)}$$

$$\text{IFNONNULL1-C}\ \frac{code(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = i :: \omega'}{\sigma \Rightarrow_C (g, pc', l, \omega', h, \phi)}$$

$$\text{IFNONNULL2-C}\ \frac{code(pc) = \texttt{ifnonnull}\ pc' \qquad \omega = \textsc{null} :: \omega'}{\sigma \Rightarrow_C (g, next(pc), l, \omega', h, \phi)}$$

$$\text{ANEWARRAY1-C}\ \frac{code(pc) = \texttt{anewarray}\ \tau \qquad \omega = c :: \omega' \qquad c \geq 0 \qquad i \notin \mathrm{dom}\, h}{\sigma \Rightarrow_C (g, next(pc), l, i :: \omega', h[i \mapsto new\text{-}arr_c(symbols(\sigma), \tau, c)], \textsc{True})}$$

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

$$\text{NEW-C}\ \frac{code(pc) = \texttt{new}\ \tau \qquad i \notin \mathrm{dom}\, h}{\sigma \Rightarrow_C (g, next(pc), l, i :: \omega, h[i \mapsto new\text{-}obj(symbols(\sigma), \tau)], \textsc{True})}$$

$$\text{IASTORE1-C}\ \frac{code(pc) = \texttt{iastore} \qquad \omega = d :: c :: i :: \omega' \qquad c < 0 \lor c \geq h(i)(\textsc{len})}{\sigma \Rightarrow_C \text{ArrayIndexOutOfBoundsException}, (g, pc, l, \omega', h, \textsc{True})}$$

$$\text{IASTORE2-C}\ \frac{code(pc) = \texttt{iastore} \qquad \omega = d :: c :: i :: \omega' \qquad 0 \leq c < h(i)(\textsc{len})}{\sigma \Rightarrow_C (g, next(pc), l, \omega', h[i \mapsto h(i)[c \mapsto d]], \textsc{True})}$$

$$\text{IASTORE3-C}\ \frac{code(pc) = \texttt{iastore} \qquad \omega = d :: c :: \textsc{null} :: \omega'}{\sigma \Rightarrow_C \text{NullPointerException}, (g, pc, l, \omega', h, , \textsc{True})}$$

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

$$\text{IALOAD2-C}\ \frac{code(pc) = \texttt{iaload} \qquad \omega = c :: \textsc{null} :: \omega'}{\sigma \Rightarrow_C \text{NullPointerException}, (g, pc, l, \omega', h, \textsc{True})}$$

$$\text{IALOAD3-C}\ \frac{code(pc) = \texttt{iaload} \qquad \omega = c :: i :: \omega' \qquad 0 \leq c < h(i)(\textsc{len})}{\sigma \Rightarrow_C (g, next(pc), l, h(i)(c) :: \omega', h, \textsc{True})}$$

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

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

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

$$\text{PUTFIELD2-C}\ \frac{code(pc) = \texttt{putfield}\ f \qquad \omega = v :: \textsc{null} :: \omega'}{\sigma \Rightarrow_C \text{NullPointerException}, (g, pc, l, \omega', h, \textsc{True})}$$

$$\text{INSTANCEOF1-C}\ \frac{code(pc) = \texttt{instanceof}\ \tau \qquad \omega = \textsc{null} :: \omega'}{\sigma \Rightarrow_C (g, next(pc), l, 1 :: \omega', h, \textsc{True})}$$

$$\text{INSTANCEOF2-C}\ \frac{code(pc) = \texttt{instanceof}\ \tau \qquad \omega = i :: \omega' \qquad X_{\tau_1} = h(i) \qquad \tau_1 <: \tau}{\sigma \Rightarrow_C (g, next(pc), l, 1 :: \omega', h, \textsc{True})}$$

$$\text{INSTANCEOF2-C}\ \frac{code(pc) = \texttt{instanceof}\ \tau \qquad \omega = i :: \omega' \qquad X_{\tau_1} = h(i) \qquad \tau_1 \not<: \tau}{\sigma \Rightarrow_C (g, next(pc), l, 0 :: \omega', h, \textsc{True})}$$

$$\text{CHECKCAST1-C}\ \frac{code(pc) = \texttt{checkcast}\ \tau \qquad \omega = \textsc{null} :: \omega'}{\sigma \Rightarrow_C (g, next(pc), l, \textsc{null} :: \omega', h, \textsc{True})}$$

$$\text{CHECKCAST2-C}\ \frac{code(pc) = \texttt{checkcast}\ \tau \qquad \omega = i :: \omega' \qquad X_{\tau_1} = h(i) \qquad \tau_1 <: \tau}{\sigma \Rightarrow_C (g, next(pc), l, i :: \omega', h, \textsc{True})}$$

$$\text{CHECKCAST2-C}\ \frac{code(pc) = \texttt{checkcast}\ \tau \qquad \omega = i :: \omega' \qquad X_{\tau_1} = h(i) \qquad \tau_1 \not<: \tau}{\sigma \Rightarrow_C \text{ClassCastException}, (g, pc, l, \omega', h, \textsc{True})}$$

$$\text{ASSUME1-C}\ \frac{code(pc) = \texttt{assume} \qquad \omega = 0 :: \omega'}{\sigma \Rightarrow_C (g, next(pc), l, \omega', h, \textsc{False})}$$

$$\text{ASSUME2-C}\ \frac{code(pc) = \texttt{assume} \qquad \omega = 1 :: \omega'}{\sigma \Rightarrow_C (g, next(pc), l, \omega', h, \textsc{True})}$$

$$\text{ASSERT1-C}\ \frac{code(pc) = \texttt{assert} \qquad \omega = 0 :: \omega'}{\sigma \Rightarrow_C \textsc{Error}, (g, pc, l, \omega', h, \textsc{True})}$$

$$\text{ASSERT2-C}\ \frac{code(pc) = \texttt{assert} \qquad \omega = 1 :: \omega'}{\sigma \Rightarrow_C (g, next(pc), l, \omega', h, \textsc{True})}$$

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 γ_s [5] to relate symbolic states and concrete state. Second, we will introduce binary relations between

[5] Since we assume the ideal case: the object bound and array length bounds k are sufficient large, any symbol/array always has bounds greater than 0.

concrete states and symbolics and prove simulation between concrete state-space and symbolic state-space. Finally, we will prove the relative sound and completeness of symbolic execution regards to concrete execution within one method and no invocation in the body of the method.

Definition of γ_s

Let us start with some definitions:

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

Then we introduce some semantic functions[6] to facilitate the definition of γ_s.

$$\begin{aligned}
\mathcal{V}_s :& \mathbf{Values}_s \to ((Env \times \mathrm{Sym}(\mathbf{Locs})) \to \mathbf{Values}_c) \\
\mathcal{O}_s :& \mathbf{Symbols}_{non-prim} \to ((\Gamma \times Env \times \mathrm{Sym}(\mathbf{Locs})) \to \mathcal{P}(\mathbf{Symbols}_{non-prim})) \\
\mathcal{H}_s :& \mathbf{Heaps}_s \to ((\Gamma \times Env \times \mathrm{Sym}(\mathbf{Locs})) \to \mathcal{P}(\mathbf{Heaps}_c)) \\
\mathcal{ST}_s :& \Sigma_{\mathcal{S}} \to ((\Gamma \times Env \times \mathrm{Sym}(\mathbf{Locs})) \to \mathcal{P}(\Sigma_C)).
\end{aligned}$$

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

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

$$\mathcal{V}_s[\![v]\!](E,\rho) = sub(sub(v,E),\rho)$$

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

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

  where

$$\begin{aligned}
&mapfields(X, X'_{\tau'}, E, \rho) \stackrel{\text{def}}{=} \forall\iota. X(\iota)\!\downarrow \implies X'(\iota) = \mathcal{V}_s[\![X(\iota)]\!](E,\rho), \text{ if } \tau' \in \mathbf{Types}_{record} \\
&mapfields(X, X'_{\tau'}, E, \rho) \stackrel{\text{def}}{=} X'(\textsc{len}) = \mathcal{V}_s[\![X(\textsc{len})]\!](E,\rho) \land \forall\iota \in acc\text{-}idx(X). \\
&\qquad X'(\mathcal{V}_s[\![\iota]\!](E,\rho)) = \mathcal{V}_s[\![X(\iota)]\!](E,\rho) \land (X(\textsc{def})\!\downarrow \implies \forall (0 \le m < X'(\textsc{len}) \\
&\qquad \land m \notin \{\, \mathcal{V}_s[\![\iota]\!](E,\rho) \mid \iota \in acc\text{-}idx(X) \,\}). X'(m) = X(\textsc{def})), \text{if } \tau' \in \mathbf{Types}_{array}
\end{aligned}$$

- the $\mathcal{H}_s$ function [7]:

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

[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 $contains(h_c, h_s, T, E, \rho)$ if and only if

$$\forall (i, X) \in h_s . \exists Y \in O_s[\![X]\!](T, E, \rho).(\rho(i), Y) \in h_c.$$

$well\text{-}typed(h_c)$ 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 $\textsc{null}$ or a location in h_c which is mapped a non-primitive symbol of a compatible type.

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

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

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

$$\mathcal{ST}_s[\![(g, pc, l, \omega, h, \phi)]\!](T, E, \rho) = \{\, (sub\text{-}fun(sub\text{-}fun(g, E), \rho), pc, \\ sub\text{-}fun(sub\text{-}fun(l, E), \rho), sub\text{-}seq(sub\text{-}seq(\omega, E), \rho), h', \textsc{True}) \mid h' \in \mathcal{H}_s[\![h]\!](T, E, \rho) \,\}.$$

Finally, the definition of $\gamma_s : \Sigma_s \to \mathcal{P}(\Sigma_c)$ is

$$\gamma_s(\sigma_s) = \bigcup_{\forall E, T \models \phi, \forall \rho} \mathcal{ST}_s[\![\sigma_s]\!](T, E, \rho).$$

Concrete and Symbolic Kripke Structures

Given a method m, we have a set of global variables G and local variables L (ordered from $0..n$). We use Kripke structures [8] $C = (\Sigma_C, I_C, \longrightarrow_C, L_C)$ and $\mathcal{S} = (\Sigma_\mathcal{S}, I_\mathcal{S}, \longrightarrow_\mathcal{S}, L_\mathcal{S})$ to model the state-spaces from the concrete and the symbolic executions, respectively. Each component is defined as following

- states,

$$\Sigma_C = \Sigma_c \cup (\textsc{Exception} \times \Sigma_c) \cup (\textsc{Error} \times \Sigma_c).$$

$$\Sigma_\mathcal{S} = \Sigma_s \cup (\textsc{Exception} \times \Sigma_s) \cup (\textsc{Error} \times \Sigma_s).$$

  Furthermore, we require that all the Σ_C and $\Sigma_\mathcal{S}$ 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

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

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

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

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

- transition relations,

$$\begin{aligned} c_1 \longrightarrow_C c_2 &\iff c_1 \Rightarrow_C c_2 \wedge \text{ last component of } c_2 \text{ is TRUE.} \\ s_1 \longrightarrow_S s_2 &\iff s_1 \Rightarrow_S s_2 \wedge \text{ the path condition of } s_2 \text{ is satisfiable.} \end{aligned}$$

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

Function γ_s is trivially extended to $\gamma_s^* : \Sigma_S \to \mathcal{P}(\Sigma_C)$ as

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

Simulation Relations

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

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

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

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. *$C \lhd_{\mathcal{R}} S$.*

Proof. It is sufficient to show that for all $\sigma_c \in \Sigma_C, \sigma_s \in \Sigma_S$ if $\sigma_c \longrightarrow_C \sigma_c'$ and $\sigma_c \;\mathcal{R}\; \sigma_s$ then there exists $\sigma_s' \in S$ such that $\sigma_s \longrightarrow_S \sigma_s'$ and $\sigma_c' \;\mathcal{R}\; \sigma_s'$.

We will proceed with the rule induction on $\longrightarrow_C$.

- Rule IADD-C: Let $\sigma_c = (g_c, pc, l_c, d :: c :: \omega_c, h_c, \text{TRUE})$, then $\sigma_c' = (g_c, next(pc), l_c, (c + d) :: \omega_c, h_c, \text{TRUE})$. Suppose $\sigma_c \;\mathcal{R}\; \sigma_s$. We need to show that there exists $\sigma_s' \in \Sigma_S$ such that $\sigma_s \longrightarrow_S \sigma_s'$ 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 σ_s must have the form of $(g_s, pc, l_s, v_1 :: v_2 :: \omega_s, h_s, \phi)$ for some T, E, ρ with $T, E \vDash \phi$, $\mathcal{V}_s[\![v_1]\!](E, \rho) =$

c, $\mathcal{V}_s[\![v_2]\!](E,\rho) = d$, $\textit{sub-fun}(\textit{sub-fun}(g_s, E), \rho) = g_c$, $\textit{sub-fun}(\textit{sub-fun}(l_s, E), \rho) = l_c$, $\textit{sub-seq}(\textit{sub-seq}(\omega_s, E), \rho) = \omega_c$, and $h_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Using the rule IADD-S, we get $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$ with $\sigma'_s = (g_s, \textit{next}(pc), l_s, Y :: \omega_s, h_s, \phi \cup \{Y = v_1 + v_2\})$ where Y is fresh. We only need to show $\sigma'_c \in \gamma_s(\sigma'_s)$, that is, to find T', E', ρ' such that $\sigma'_c \in \mathcal{ST}_s[\![\sigma'_s]\!](T', E', \rho')$. We claim that $T' = T$, $E' = E[Y \mapsto c + d]$, and $\rho' = \rho$ are the right choice. Since Y is fresh, $\textit{sub-fun}(\textit{sub-fun}(g_s, E'), \rho') = \textit{sub-fun}(\textit{sub-fun}(g_s, E), \rho) = g_c$, $\textit{sub-fun}(\textit{sub-fun}(l_s, E'), \rho') = \textit{sub-fun}(\textit{sub-fun}(l_s, E), \rho) = l_c$, $\textit{sub-seq}(\textit{sub-seq}(\omega_s, E'), \rho') = \textit{sub-seq}(\textit{sub-seq}(\omega_s, E), \rho) = \omega_c$, and $h_c \in \mathcal{H}_s[\![h_s]\!](T', E', \rho') = \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Furthermore, since $\mathcal{V}_s[\![Y]\!](E', \rho) = c + d = \mathcal{V}_s[\![v_1]\!](E, \rho) + \mathcal{V}_s[\![v_2]\!](E, \rho)$, we get $T, E' \vDash (\phi \cup \{Y = v_1 + v_2\})$. Therefore, $\sigma'_c \in \mathcal{ST}_s[\![\sigma'_s]\!](T, E', \rho) \subseteq \gamma_s(\sigma'_s)$.

- Rule IF_ICMPLT2-C: Let $\sigma_c = (g_c, pc, l_c, d :: c :: \omega_c, h_c, \textsc{True})$, then $c \geq d$ and $\sigma'_c = (g_c, \textit{next}(pc), l_c, \omega_c, h_c, \textsc{True})$. Suppose $\sigma_c \ \mathcal{R} \ \sigma_s$. We need to show that there exists $\sigma'_s \in \Sigma_{\mathcal{S}}$ such that $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$ and $\sigma'_c \ \mathcal{R} \ \sigma'_s$. Since $\sigma_c \ \mathcal{R} \ \sigma_s$, we have $\sigma_c \in \gamma_{\mathcal{S}}(\sigma_s)$. The symbolic state σ_s must have the form of $(g_s, pc, l_s, v_2 :: v_1 :: \omega_s, h_s, \phi)$ for some T, E, ρ with $T, E \vDash \phi$, $\mathcal{V}_s[\![v_1]\!](E, \rho) = c$, $\mathcal{V}_s[\![v_2]\!](E, \rho) = d$, $\textit{sub-fun}(\textit{sub-fun}(g_s, E), \rho) = g_c$, $\textit{sub-fun}(\textit{sub-fun}(l_s, E), \rho) = l_c$, $\textit{sub-seq}(\textit{sub-seq}(\omega_s, E), \rho) = \omega_c$, and $h_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Using the IF_ICMPLT-S, we get $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$ with $\sigma'_s = (g_s, \textit{next}(pc), l_s, \omega_s, h_s, \phi \cup \{v_1 \geq v_2\})$ (the first end state). We only need to show $\sigma'_c \in \gamma_s(\sigma'_s)$. Since $\mathcal{V}_s[\![v_1]\!](E, \rho) = c$, $\mathcal{V}_s[\![v_2]\!](E, \rho) = d$, and $c \geq d$, we get $T, E \vDash \phi \cup \{v_1 \geq v_2\}$. Therefore, $\sigma'_c \in \mathcal{ST}_s[\![\sigma'_s]\!](T, E, \rho) \subseteq \gamma_s(\sigma'_s)$.

- Rule ANEWARRAY1-C: Suppose $\sigma_c = (g_c, pc, l_c, c :: \omega_c, h_c, \textsc{True})$. Then $c \geq 0$ and $\sigma'_c = (g_c, \textit{next}(pc), l_c, \omega_c, h'_c, \textsc{True})$ where i is fresh and $h'_c = h_c[i \mapsto \textit{new-arr}_c(\textit{symbols}(\sigma_c), \tau, c)]$. Suppose $\sigma_c \ \mathcal{R} \ \sigma_s$. We need to show that exists $\sigma'_s \in \Sigma_{\mathcal{S}}$ such that $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$ 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 σ_s has the form of $(g_s, pc, l_s, v :: \omega_s, h_s, \phi)$ for some T, E, ρ with $T, E \vDash \phi$, $\mathcal{V}_s[\![v]\!](E, \rho) = c$, $\textit{sub-fun}(\textit{sub-fun}(g_s, E), \rho) = g_c$, $\textit{sub-fun}(\textit{sub-fun}(l_s, E), \rho) = l_c$, $\textit{sub-seq}(\textit{sub-seq}(\omega_s, E), \rho) = \omega_c$, and $h_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Using the ANEWARRAY2-S rule, we get $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$ with $\sigma'_s = (g_s, \textit{next}(pc), l_s, \omega_s, h'_s, \phi \cup \{v \geq 0\})$ where $h'_s = h_s[j \mapsto \textit{new-arr}(\textit{symbols}(\sigma_s), \tau, X, k)]$ and j is fresh (the first end state). We only need to show $\sigma'_c \in \gamma_s(\sigma'_s)$. Define $\rho' = \rho[j \mapsto i][\rho^{-1}(i) \mapsto \rho(j)]$. It is clear that $\rho' \in S$ and for location $i' \notin \{j, \rho^{-1}(i)\}$, $\rho'(i') = \rho(i')$. Since i is fresh in σ_c and $\sigma_c \ \mathcal{R} \ \sigma_s$, $\rho^{-1}(i)$ must be fresh in σ_s (not in $\operatorname{dom} h_s$) too. Thus we get $\textit{sub-fun}(\textit{sub-fun}(g_s, E), \rho') = g_c$, $\textit{sub-fun}(\textit{sub-fun}(l_s, E), \rho') = l_c$, and $\textit{sub-seq}(\textit{sub-seq}(\omega_s, E), \rho') = \omega_c$. From $c \geq 0$, $\mathcal{V}_s[\![v]\!](E, \rho') \geq 0$, that is, $T, E \vDash \phi \cup \{v \geq 0\}$. It remains to show $h'_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho')$. Clearly $\textit{well-typed}(h'_c)$ because i is fresh in h_c. Then we show that $\textit{contains}(h'_c, h'_s, T, E, \rho')$. For any entry $(i', X') \in h_s$, since j and $\rho^{-1}(i)$ are fresh in h_s, we get $\mathcal{O}_s[\![X']\!](T, E, \rho') = \mathcal{O}_s[\![X']\!](T, E, \rho)$. Furthermore, since $\textit{new-arr}_c(\textit{symbols}(\sigma_c), \tau, c) \in \mathcal{O}_s[\![\textit{new-arr}(\textit{symbols}(\sigma_c), \tau, X, k)]\!](T, E, \rho')$, we can get $\textit{contains}(h'_c, h'_s, T, E, \rho')$. Next we need to show $\textit{well-formed}(h'_c, h_s, T, E, \rho')$. Since $\textit{new-arr}(\textit{symbols}(\sigma_s), \tau, X, k)(\textsc{conc}) \downarrow$, symbol $\textit{new-arr}_c(\textit{symbols}(\sigma_c), \tau, c)$ of entry $(i, \textit{new-arr}_c(\textit{symbols}(\sigma_c), \tau, c))$ in h'_c is well-formed under E and ρ'. 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_s, by the $\textit{well-formed}(h_c, h_s, T, E, \rho)$, f can not be any location that points to concrete object in h_c. But h'_c has only one extra concrete object at i than h_c

and i is fresh in h_c. Therefore, f can not point to i, that is, symbol $\textit{new-arr}_c(\textit{symbols}(\sigma_c), \tau, c)$ is well-formed. We get $\textit{well-formed}(h'_c, h'_s, T, E, \rho')$. Thus $h'_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho')$. Finally, $\sigma'_c \in \mathcal{ST}_s[\![\sigma'_s]\!](T, E, \rho') \subseteq \gamma_s(\sigma'_s)$.

- Rule GETFIELD1-C: Suppose $\sigma_c = (g_c, pc, l_c, i :: \omega_c, h_c)$, then $\sigma'_c = (g_c, \textit{next}(pc), l_c, v :: \omega_c, h_c)$ where $X = h_c(i), v = X(f)$. Let τ_v be the real type of symbol $h_c(v)$. Suppose $\sigma_c \ \mathcal{R}\ \sigma_s$. We need to show that exists $\sigma'_s \in \Sigma_{\mathcal{S}}$ such that $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$ 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 σ_s must have the form of $(g_s, pc, l_s, i' :: \omega_s, h_s, \phi)$ for some T, E, ρ with $T, E \vDash \phi$, $\rho(i') = i$, $\textit{sub-fun}(\textit{sub-fun}(g_s, E), \rho) = g_c$, $\textit{sub-fun}(\textit{sub-fun}(l_s, E), \rho) = l_c$, $\textit{sub-seq}(\textit{sub-seq}(\omega_s, E), \rho) = \omega_c$, and $h_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. WLOG, assume that the type of f, 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 $\textit{well-formed}(h_c, h_s, T, E, \rho)$:

  - case $v = \text{NULL}$. We will apply the GETFIELD3-S rule and get $\sigma'_s = (g, \textit{next}(pc), l, \text{NULL} :: \omega, h'_s, \phi)$, where $h'_s = h_s[i \mapsto h_s(i)[f \mapsto \text{NULL}]]$. It suffices to show $\textit{contains}(h_c, h'_s, T, E, \rho)$ and $\textit{well-formed}(h_c, h'_s, T, E, \rho)$. Since $\rho(i') = i$ and $\sigma_c \ \mathcal{R}\ \sigma_s$, $X \in \mathcal{O}_s[\![Y]\!](T, E, \rho)$. Furthermore, Since $h_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$ and $X \in \mathcal{O}_s[\![h_s(i)[f_\tau \mapsto \text{NULL}]]\!](T, E, \rho)$ by $X(f) = \text{NULL} = h_s(i)(f)$, we get $\textit{contains}(h_c, h'_s, T, E, \rho)$ and $\textit{well-formed}(h_c, h'_s, T, E, \rho)$ hold. We get $h_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho)$. Then $\sigma'_c \in \gamma_s(\sigma'_s)$.
  - case $v \in \rho(\operatorname{dom} h_s) \wedge h_s(\rho^{-1}(v))(\text{CONC}) \uparrow$. We will apply the rule GETFIELD4-S and get $\sigma'_s = (g, \textit{next}(pc), l, v' :: \omega, h'_s, \phi \cup \{\tau' <: \tau\})$ where $h'_s = h_s[i \mapsto h_s(i)[f \mapsto j]]$ and $Z_{\tau'} = h_s(v')$. We also have $v' = \rho^{-1}(v)$ ($v' \in \textit{collect}(h_s)$ because $v \in \rho(\operatorname{dom} h_s)$ and $h_s(\rho^{-1}(v))(\text{CONC}) \uparrow$). Since $\textit{well-typed}(h_c)$, the type of $h_c(v)$, τ_v, is a subtype of τ. Furthermore, since $h_c(v) \in \mathcal{O}_s[\![Z_{\tau'}]\!](T, E, \rho)$, we arrive at $T \vDash \tau' <: \tau$. Thus $T, E \vDash \phi \cup \{\tau' <: \tau\}$. The rest of the proof is similar to the NULL case.
  - case $v \in \mathbf{Locs} \wedge v \notin \rho(\operatorname{dom} h_s)$. We will apply the rule GETFIELD6-S (because we assume that bound k is sufficient large and $m > 0$) and get $\sigma'_s = (g, \textit{next}(pc), l, v :: \omega', h'_s, \phi \cup \{\tau' <: \tau\})$ where $h'_s = h_s[i \mapsto h_s(i)[f \mapsto j]]$ and $Z_{\tau'} = \textit{new-sym}(\textit{symbols}(\sigma), m-1, k)$. Define $\rho' = \rho[j \mapsto v]$ and $T' = T[\tau' \mapsto \tau_v]$. Since $\textit{well-typed}(h_c)$, we get $\tau_v <: \tau$. Furthermore, since $\rho' = \rho[j \mapsto v]$ and $T' = T[\tau' \mapsto \tau_v]$, $T' \vDash \tau' <: \tau$. Thus $T', E \vDash \phi \cup \{\tau' <: \tau\}$. Since j is fresh in h_s, $\textit{sub-fun}(\textit{sub-fun}(g_s, E), \rho') = \textit{sub-fun}(\textit{sub-fun}(g_s, E), \rho)$, $\textit{sub-fun}(\textit{sub-fun}(l_s, E), \rho') = \textit{sub-fun}(\textit{sub-fun}(l_s, E), \rho)$, and $\textit{sub-seq}(\textit{sub-seq}(\omega_s, E), \rho') = \textit{sub-seq}(\textit{sub-seq}(\omega_s, E), \rho)$ hold. It remains to show $\textit{contains}(h_c, h'_s, T, E, \rho)$ and $\textit{well-formed}(h_c, h'_s, T, E, \rho)$. Since $X \in \mathcal{O}_s[\![Y^{m,n}[f_\tau \mapsto v']]\!](T', E, \rho')$ and $h_c(v) \in \mathcal{O}_s[\![Z_{\tau'}]\!](T', E, \rho')$, $\textit{contains}(h_c, h'_s, T', E, \rho)$ holds. Since $v \notin \rho(\operatorname{dom} h_s)$, $h_c(v)$ is well-formed. Since the new symbol $Z_{\tau'}$ in h'_s has CONC field undefined, the rest of symbols in h_c are well-formed. Thus we get $\textit{well-formed}(h_c, h'_s, T, E, \rho)$ and further, $h_c \in \mathcal{H}_s[\![h'_s]\!](T', E, \rho')$. Therefore, $\sigma'_c \in \mathcal{ST}_s[\![\sigma'_s]\!](T', E, \rho') \subseteq \gamma_s(\sigma'_s)$.

□

Definition 2. $\mathcal{R}_\bullet \subseteq \Sigma_{\mathcal{S}} \times \mathcal{P}(\Sigma_C)$, *as* $\sigma_s\ \mathcal{R}_\bullet\ S_c \iff \gamma^*_s(\sigma_s) = S_c$.

The relation $\mathcal{R}_\bullet$ is left total by definition. Also it is clear that for $\sigma_s\ \mathcal{R}_\bullet\ S_c$, S_c is not empty, if and only if the path condition ϕ of σ_s is satisfiable. Furthermore, for any $\sigma_s \in I_{\mathcal{S}}$ and $\sigma_s\ \mathcal{R}_\bullet\ S_c$, it is clear that $S_c \subseteq I_C$ by the definition of γ_s function.

Proposition 2. $\mathcal{S} \lhd_{\mathcal{R}_\bullet} \mathcal{P}(C)$.

Proof. It is sufficient to show that for all $\sigma_s, \sigma'_s \in \Sigma_{\mathcal{S}}, S_c, S'_c \in \mathcal{P}(\Sigma_C)$, if $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$, $\sigma_s\ \mathcal{R}_\bullet\ S_c$, and $\sigma'_s\ \mathcal{R}_\bullet\ S'_c$ then $S_c \stackrel{\bullet}{\longrightarrow}_C S'_c$.

We will prove by rule induction on symbolic operational semantics transitions, $\longrightarrow_{\mathcal{S}}$.

- Rule IADD-S: $\sigma_s = (g_s, pc, l_s, v_1 :: v_2 :: \omega_s, h_s, \phi)$. Then $\sigma'_s = (g_s, \mathit{next}(pc), l_s, Z :: \omega_s, h_s, \phi \cup \{Z = v_1 + v_2\})$ where Z is fresh. Suppose $\sigma_s\ \mathcal{R}_\bullet\ S_c$ and $\sigma'_s\ \mathcal{R}_\bullet\ S'_c$. We need to show that $S_c \stackrel{\bullet}{\longrightarrow}_C S'_c$, that is, for any $\sigma'_c \in S'_c$, there exists some $\sigma_c \in S_c$ such that $\sigma_c \longrightarrow_C \sigma'_c$. Suppose $\sigma'_c \in S'_c$, that is, $\sigma'_c \in \gamma_s(s'_s)$. Then σ'_c must be in the form of $(g'_c, \mathit{next}(pc), l'_c, c :: \omega'_c, h'_c, \textsc{True})$ with some T, E, ρ such that $T, E \vDash \phi \cup \{Z = v_1 + v_2\}$, $\mathcal{V}_s[\![Z]\!](E, \rho) = c$, $\mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(g_s, E), \rho) = g'_c$, $\mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(l_s, E), \rho) = l'_c$, $\mathit{sub\text{-}seq}(\mathit{sub\text{-}seq}(\omega_s, E), \rho) = \omega'_c$, and $h'_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Take $\sigma_c = (g'_c, pc, l'_c, E(X) :: E(Y) :: \omega'_c, h'_c, \textsc{True})$. Clearly $\sigma_c \longrightarrow_C \sigma'_c$. We only need to show that $\sigma_c \in \gamma_s(\sigma_s)$. Since Z is fresh, $T, E \vDash \phi$. Thus $\sigma_c \in \mathcal{ST}_s[\![\sigma_s]\!](T, E, \rho) \subseteq \gamma_s(\sigma_s)$.

- Rule IF_ICMPLT-S: $\sigma_s = (g_s, pc, l_s, v_1 :: v_2 :: \omega_s, h_s, \phi)$ and $\sigma'_s = (g_s, \mathit{next}(pc), l_s, \omega_s, h_s, \phi \cup \{v_2 \geq v_1\})$. (we only consider one end state, the other end state is symmetric.) Suppose $\sigma_s\ \mathcal{R}_\bullet\ S_c$ and $\sigma'_s\ \mathcal{R}_\bullet\ S'_c$. We need to show that $S_c \stackrel{\bullet}{\longrightarrow}_C S'_c$, that is, for any $\sigma'_c \in S'_c$, there exists some $\sigma_c \in S_c$ such that $\sigma_c \longrightarrow_C \sigma'_c$. Suppose $\sigma'_c \in S'_c$, that is, $\sigma'_c \in \gamma_s(s'_s)$. Then σ'_c must be in the form of $(g'_c, \mathit{next}(pc), l'_c, \omega'_c, h'_c, \textsc{True})$ with some T, E, ρ such that $T, E \vDash \phi \cup \{v_2 \geq v_1\}$, $\mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(g_s, E), \rho) = g'_c$, $\mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(l_s, E), \rho) = l'_c$, $\mathit{sub\text{-}seq}(\mathit{sub\text{-}seq}(\omega_s, E), \rho) = \omega'_c$, and $h'_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Take $\sigma_c = (g'_c, pc, l'_c, \mathcal{V}_s[\![v_1]\!](E, \rho) :: \mathcal{V}_s[\![v_2]\!](E, \rho) :: \omega'_c, h'_c)$. Clearly $\sigma_c \longrightarrow_C \sigma'_c$. We conclude that $\sigma_c \in \mathcal{ST}_s[\![\sigma_s]\!](T, E, \rho) \subseteq \gamma_s(\sigma_s)$.

- Rule ANEWARRARY2-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 $\sigma'_s = (g_s, \mathit{next}(pc), l_s, i :: \omega_s, h_s[i \mapsto \mathit{new\text{-}arr}(\mathit{symbols}(\sigma_s), \tau, X, k)], \phi \cup \{X \geq 0\})$ where i is fresh. Suppose $\sigma_s\ \mathcal{R}_\bullet\ S_c$ and $\sigma'_s\ \mathcal{R}_\bullet\ S'_c$. We need to show that $S_c \stackrel{\bullet}{\longrightarrow}_C S'_c$, that is, for any $\sigma'_c \in S'_c$, there exists some $\sigma_c \in S_c$ such that $\sigma_c \longrightarrow_C \sigma'_c$. Suppose $\sigma'_c \in S'_c$, that is, $\sigma'_c \in \gamma_s(s'_s)$. Then σ'_c must be in the form of $(g'_c, \mathit{next}(pc), l'_c, j :: \omega'_c, h'_c, \textsc{True})$ with some T, E, ρ such that $T, E \vDash \phi \cup \{X \geq 0\}$, $\rho(i) = j$, $\mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(g_s, E), \rho) = g'_c$, $\mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(l_s, E), \rho) = l'_c$, $\mathit{sub\text{-}seq}(\mathit{sub\text{-}seq}(\omega_s, E), \rho) = \omega'_c$, and $h'_c \in \mathcal{H}_s[\![h_s[i \mapsto \mathit{new\text{-}arr}(\mathit{symbols}(\sigma_s), \tau, X, k)]]\!](T, E, \rho)$. We need to find a $\sigma_c \in \Sigma_C$ such that $\sigma_c \in \gamma_s(\sigma_s)$ and $\sigma_c \longrightarrow_C \sigma'_c$. We claim that $\sigma_c = (g'_c, pc, l'_c, E(\alpha) :: \omega'_c, h_c, \textsc{True})$ where $h_c = h'_c \setminus (j, h'_c(j))$ satisfies the above two conditions. Since $T, E \vDash \phi \cup \{X \geq 0\}$, $T, E \vDash \phi$. To show $\sigma_c \in \gamma_s(\sigma_s)$, it suffices to show that $h_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Since $\mathit{new\text{-}arr}(\mathit{symbols}(\sigma_s), \tau, X, k)$ will return a symbol with $\textsc{conc}$ field defined and $h'_c \in \mathcal{H}_s[\![h_s[i \mapsto \mathit{new\text{-}arr}(\mathit{symbols}(\sigma_s), \tau, X, k)]]\!](T, E, \rho)$, symbols in h'_c such that their corresponding symbols in $h_s[i \mapsto \mathit{new\text{-}arr}(\mathit{symbols}(\sigma_s), \tau, X, k)]$ have $\textsc{conc}$ fields not defined or do not have corresponding symbols can not contains j ($\rho(i)$).

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, ρ), and *well-formed*(h_c, h_s, T, E, ρ). We get $\sigma_c \in \mathcal{ST}_s[\![\sigma_s]\!](T, E, \rho) \subseteq \gamma_s(\sigma_s)$. Clearly $\sigma_c \longrightarrow_C \sigma'_c$.

- 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 $\sigma'_s = (g_s, \mathit{next}(pc), l_s, \textsc{null} :: \omega_s, h'_s, \phi)$ where $h'_s = h_s[i \mapsto h_s(i)[f_\tau \mapsto \textsc{null}]]$. Suppose $\sigma_s \,\mathcal{R}_\bullet\, S_c$ and $\sigma'_s \,\mathcal{R}_\bullet\, S'_c$. We need to show that $S_c \overset{\bullet}{\longrightarrow}_C S'_c$, that is, for any $\sigma'_c \in S'_c$, there exists some $\sigma_c \in S_c$ such that $\sigma_c \longrightarrow_C \sigma'_c$. Suppose $\sigma'_c \in S'_c$, that is, $\sigma'_c \in \gamma_s(s'_s)$. Then σ'_c must be in the form of $(g'_c, \mathit{next}(pc), l'_c, \textsc{null} :: \omega'_c, h'_c, \textsc{True})$ with some T, E, ρ such that $T, E \vDash \phi$, $\mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(g_s, E), \rho) = g'_c$, $\mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(l_s, E), \rho) = l'_c$, $\mathit{sub\text{-}seq}(\mathit{sub\text{-}seq}(\omega_s, E), \rho) = \omega'_c$, and $h'_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho)$. We need to find a σ_c such that $\sigma_c \longrightarrow_C \sigma'_c$ and $\sigma_c \in \gamma_s(\sigma_s)$. Take $\sigma_c = (g'_c, pc, l'_c, \rho(i) :: \omega'_c, h'_c, \textsc{True})$. From $h'_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho)$, it is clear that $\sigma_c \longrightarrow_C \sigma'_c$. Then it suffices to show $h'_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho)$. Since $h'_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho)$, *well-typed*($h'_c$), *contains*($h'_c, h_s, T, E, \rho$), and *well-formed*($h'_c, h_s, T, E, \rho$) hold. Finally, $\sigma_c \in \mathcal{ST}_s[\![\sigma_s]\!](T, E, \rho) \subseteq \gamma_s(\sigma_s)$.

- 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 $\sigma'_s = (g_s, \mathit{next}(pc), l_s, j :: \omega_s, h'_s, \phi')$ where $h_s(i) = Y^{m,n}$, $h'_s = h_s[i \mapsto Y^{m,n}[f_\tau \mapsto j]][j \mapsto Z_{\tau'}]$, $\phi' = \phi \cup \{\tau' <: \tau\}$ where $Z_{\tau'} = \mathit{new\text{-}sym}(\mathit{symbols}(\sigma_s), m-1, k)$ and $j \notin \operatorname{dom} h_s$. Suppose $\sigma_s \,\mathcal{R}_\bullet\, S_c$ and $\sigma'_s \,\mathcal{R}_\bullet\, S'_c$. We need to show that $S_c \overset{\bullet}{\longrightarrow}_C S'_c$, that is, for any $\sigma'_c \in S'_c$, there exists some $\sigma_c \in S_c$ such that $\sigma_c \longrightarrow_C \sigma'_c$. Suppose $\sigma'_c \in S'_c$, that is, $\sigma'_c \in \gamma_s(s'_s)$. Then σ'_c must be in the form of $(g'_c, \mathit{next}(pc), l'_c, v' :: \omega'_c, h'_c, \textsc{True})$ with some T, E, ρ such that $T, E \vDash \phi'$, $\mathcal{V}_s[\![v]\!](E, \rho) = v'$, $\mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(g_s, E), \rho) = g'_c$, $\mathit{sub\text{-}fun}(\mathit{sub\text{-}fun}(l_s, E), \rho) = l'_c$, $\mathit{sub\text{-}seq}(\mathit{sub\text{-}seq}(\omega_s, E), \rho) = \omega'_c$, and $h'_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho)$. We need to find a σ_c such that $\sigma_c \longrightarrow_C \sigma'_c$ and $\sigma_c \in \gamma_s(\sigma_s)$. Define $\rho' = \rho[j \mapsto v']$ and $\sigma_c = (g'_c, pc, l'_c, \rho'(i) :: \omega'_c, h'_c, \textsc{True})$. From $h'_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho)$, it is clear that $\sigma_c \longrightarrow_C \sigma'_c$. Since $T, E \vDash \phi \cup \{\tau' <: \tau\}$, $T, E \vDash \phi$. Then it suffices to show $h'_c \in \mathcal{H}_s[\![h_s]\!](T, E, \rho')$. Since $h'_c \in \mathcal{H}_s[\![h'_s]\!](T, E, \rho)$, *well-typed*($h'_c$) and *contains*($h'_c, h_s, T, E, \rho$) hold. Since $h'_s(j) = Z$ and $\textsc{conc} \notin \operatorname{dom} Z$, *well-formed*($h'_c, h_s, T, E, \rho$) hold. Finally, $\sigma_c \in \mathcal{ST}_s[\![\sigma_s]\!](T, E, \rho') \subseteq \gamma_s(\sigma_s)$.

□

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 $c_1 \longrightarrow_C c_2 \longrightarrow_C \cdots \longrightarrow_C c_n$ with $c_1 \in I_C$, there is a corresponding symbolic trace $s_1 \longrightarrow_S s_2 \longrightarrow_S \cdots \longrightarrow_S s_n$ with $s_1 \in I_S$ such that $c_k \,\mathcal{R}\, s_k$ for all $1 \le k \le n$.*

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

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

Proof. Since the ϕ of s_1 is not false, $C_1 = \gamma_s^*(s_1) \neq \emptyset$. Then we show there exists a trace in $\mathcal{P}(C)$, $C_1 \overset{\bullet}{\longrightarrow}_C C_2 \overset{\bullet}{\longrightarrow}_C \cdots \overset{\bullet}{\longrightarrow}_C C_n$ such that $s_k \mathcal{R}_\bullet C_k$ for all $1 \leq k \leq n$ by mathematical induction on n using Proposition 2. Since the ϕ of s_n is satisfiable, then $C_n \neq \emptyset$. Pick any $c_n \in C_n$ and use the definition of $\overset{\bullet}{\longrightarrow}_C$, we get the corresponding concrete trace $c_1 \longrightarrow_C c_2 \longrightarrow_C \cdots \longrightarrow_C c_n$. $\square$

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 γ_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 γ_a

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

$$\Pi = \{\, F \mid F : \textbf{SymLocs} \to \textbf{Locs} \,\}. \tag{C.1}$$

Then we define some semantics functions with subscript a denoting lazier symbolic domains/components:

$$\mathcal{H}_a : (\textbf{Heaps}_a \times \Phi) \to (\mathcal{P}(\textbf{Symbols}) \times \mathcal{P}(\textbf{SymLocs}) \times \Pi) \to \mathcal{P}(\textbf{Heaps}_s \times \Phi))$$
$$\mathcal{ST}_a : \Sigma_a \to \Pi \to \mathcal{P}(\Sigma_s).$$

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

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

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

  where *well-mapped* : $\mathcal{P}(\textbf{SymLocs}) \times \textbf{Heaps}_a \times \Pi \to$ Boolean with *well-mapped*(Δ, h_a, F) if and only if

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

[9] Subscript a is frequently used to indicate a component in the lazier symbolic states.

$heap : \mathcal{P}(\textbf{Symbols}) \times \mathcal{P}(\textbf{SymLocs}) \times \textbf{Heaps}_a \times \textbf{Heaps}_s \times \Pi \to \textsc{Boolean}$ with $heap(ss, \Delta, h_a, h_s, F)$ if and only if

$$\operatorname{dom} h_s = \operatorname{dom} h_a \cup F(\Delta) \wedge \forall i \in \operatorname{dom} h_a . h_s(i) = \textit{sub-fun}(h_a(i), F)$$
$$\wedge \forall i \in (\operatorname{dom} h_s - \operatorname{dom} h_a). h_s(i) = X_\tau,$$

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

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

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

Note: similar the property of substitution, Lemma 2,

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

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

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

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

  where *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, $\mathcal{ST}_a$ function return $\emptyset$ or a singleton too.

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

$$\gamma_a(\sigma_a) = \bigcup_{\forall F \in \Pi} \mathcal{ST}_a[\![\sigma_a]\!](F).$$

Properties of γ_a

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

1. $\delta \in$ *collect-sym-locs*(σ_a)*;*
2. $i \notin \operatorname{dom} h$ *or* $h(i)(\textsc{conc}) \uparrow$*;*
3. $(h', \phi') = \textit{init-loc-heap}(h, \textit{symbols}(\sigma_a), \delta, i)$ *with* ϕ' *is satisfiable.*

Lemma 3. *Let $\sigma_a \in \Sigma_a$ and $F \in \Pi$. Suppose $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F)$. For any $(\delta, i) \in F$, if $\sigma'_a \in$ init-sym-loc(σ_a, δ, i) and i is a legal value for δ regarding to σ_a, then $\sigma_s \in \mathcal{ST}_a[\![\sigma'_a]\!](F)$.*

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*, $\sigma'_a = (\textit{sub-fun}_1(g_a, \delta, i), pc, \textit{sub-fun}_1(l_a, \delta, i), \textit{sub-seq}_1(\omega_a, \delta, i), h'_a, \phi')$, where $(h'_a, \phi') =$ *init-loc-heap*$(h_a, \phi, \textit{symbols}(\sigma_a), \delta)i$. Since $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F)$, we have $g_s = \textit{sub-fun}(\textit{sub-fun}_1(g_a, \delta, i), F)$, $l_s = \textit{sub-fun}(\textit{sub-fun}_1(l_a, \delta, i), F)$, and $\omega_s = \textit{sub-seq}(\textit{sub-seq}_1(\omega_a, \delta, i), F)$ by Lemma 1. It remains to show that

$$(h_s, \phi_s) \in \mathcal{H}_a[\![(h'_a, \phi')]\!](\textit{symbols}(\sigma'_a), \textit{collect-sym-locs}(\sigma'_a), F).$$

We know that $(h_s, \phi_s) \in \mathcal{H}_a[\![(h_a, \phi)]\!](\textit{symbols}(\sigma_a), \textit{collect-sym-locs}(\sigma_a), F)$. We will proceed by the definition of $\mathcal{H}_a$. Since σ'_a has one fewer symbolic location (δ) than σ_a, the predicate *well-mapped*$(\textit{symbols}(\sigma'_a), \textit{collect-sym-locs}(\sigma'_a), F)$ holds. Also it is easy to see that both *heap*$(\textit{collect-sym-locs}(\sigma'_a), h'_a, h_s, F)$ and $pc(\phi_s, \phi', h_s, F, \textit{collect-sym-locs}(\sigma'_a))$ hold. We conclude that $\sigma_s \in \mathcal{ST}_a[\![\sigma'_a]\!](F)$ holds. □

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 δ regarding to σ_a, if $\sigma'_a \in$ init-sym-loc(σ_a, δ, i) and $\sigma_s \in \mathcal{ST}_a[\![\sigma'_a]\!](F)$, then $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F)$.*

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

$$(h_s, \phi_s) \in \mathcal{H}_a[\![(h_a, \phi)]\!](\textit{symbols}(\sigma_a), \textit{collect-sym-locs}(\sigma_a), F).$$

We know that $(h_s, \phi_s) \in \mathcal{H}_a[\![(h'_a, \phi')]\!](\textit{symbols}(\sigma'_a), \textit{collect-sym-locs}(\sigma'_a), F)$. We will proceed by the definition of $\mathcal{H}_a$. Since σ_a has one more symbolic location (δ) than σ'_a and by i is legal for δ regarding to σ_a, the predicate *well-mapped*$(\textit{symbols}(\sigma_a), \textit{collect-sym-locs}(\sigma_a), F)$ holds. Also it is easy to see that both *heap*$(\textit{collect-sym-locs}(\sigma_a), h_a, h_s, F)$ and $pc(\phi_s, \phi, h_s, F, \textit{collect-sym-locs}(\sigma_a))$ hold. We conclude that $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F)$ holds. □

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 $\mathcal{A} = (\Sigma_{\mathcal{A}}, I_{\mathcal{A}}, \longrightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ 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,

  $$I_{\mathcal{A}} = \{ (g_a, pc_{init}, l_a, nil, h_a, \{\text{True}\}) \mid \text{dom}(g_a) = G \wedge \text{dom}(l_a) = L \},$$

  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, $a \longrightarrow_{\mathcal{A}} a' \iff a \Rightarrow_{\mathcal{A}} a_2, a_2 \Rightarrow_{\mathcal{A}} a_3, \ldots, a_n \Rightarrow_{\mathcal{A}} a'$ 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.

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

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

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

Simulation Relations

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

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

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

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

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

- Rule IF_ACMPEQ1-S: Let $\sigma_s = (g_s, pc, l_s, i :: j :: \omega_s, h_s, \phi)$. Then $i \neq j$ and $\sigma'_s = (g_s, \mathit{next}(pc), l_s, \omega_s, h_s, \phi)$. Suppose $\sigma_s \; \mathcal{R}' \; \sigma_a$, we need to show there exists $\sigma'_a \in \mathcal{A}$ such that $\sigma_a \longrightarrow_{\mathcal{A}} \sigma'_a$ and $\sigma'_s \; \mathcal{R}' \; \sigma'_a$. Since $\sigma_s \; \mathcal{R}' \; \sigma_a$, we have $\sigma_s \in \gamma_a(\sigma_a)$. WLOG, suppose that σ_a has the form of $(g_a, pc, l_a, \delta_\tau :: \delta'_{\tau'} :: \omega_a, h_a, \phi')$ for some F with $\mathcal{V}'[\![\delta]\!](F) = i$, $\mathcal{V}'[\![\delta']\!](F) = j$, and $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F)$. After taking the IF_ACMPEQ2-A rule, we get an invisible state $t_1 = (g'_a, pc, l'_a, i :: \delta'_{\tau'} :: \omega'_a, h'_a, \phi'')$ with $t_1 \in \mathit{init\text{-}sym\text{-}loc}(\sigma_a, \delta, i)$. By Lemma 3, we have $\sigma_s \in \mathcal{ST}_a[\![t_1]\!](F)$. 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 $t_2 \in \mathit{init\text{-}sym\text{-}loc}(t_1, \delta', j)$. By Lemma 3, we have $\sigma_s \in \mathcal{ST}_a[\![t_2]\!](F)$. Finally, we take the IF_ACMPEQ1-S rule and get $\sigma'_a = (g''_a, \mathit{next}(pc), l''_a, \omega''_a, h''_a, \phi''')$. Now it is sufficient to show that $\sigma'_s \in \gamma_a(\sigma'_a)$. Clearly $\mathit{sub\text{-}fun}(g''_a, F) = \mathit{sub\text{-}fun}(g_a, F) = g_s, \mathit{sub\text{-}fun}(l''_a, F) = \mathit{sub\text{-}fun}(l_a, F) = l_s$, and $\mathit{sub\text{-}seq}(\omega''_a, F) = \mathit{sub\text{-}seq}(\omega_a, F) = \omega_s$ by applying Lemma 1 twice. It remains to show that
$(h_s, \phi) \in \mathcal{H}_a[\![(h''_a, \phi''')]\!](\mathit{symbols}(\sigma'_a), \mathit{collect\text{-}sym\text{-}locs}(\sigma'_a), F)$. Since $\mathit{symbols}(\sigma'_a) = \mathit{symbols}(t_2)$ and $\mathit{collect\text{-}sym\text{-}locs}(\sigma'_a) = \mathit{collect\text{-}sym\text{-}locs}(t_2) = \mathit{collect\text{-}sym\text{-}locs}(\sigma_a) \setminus \{\delta, \delta'\}$, we get $(h_s, \phi) \in \mathcal{H}_a[\![(h''_a, \phi''')]\!](\mathit{symbols}(\sigma'_a), \mathit{collect\text{-}sym\text{-}locs}(\sigma'_a), F)$. Therefore, $\sigma'_s \in \gamma_a(\sigma'_a)$.

- Rule GETFIELD3-S: Suppose $\sigma_s = (g_s, pc, l_s, i :: \omega_s, h_s, \phi)$. Then $\sigma'_s = (g_s, \mathit{next}(pc), l_s, \textsc{null} :: \omega_s, h'_s, \phi)$ where $h_s(i) = Y$ and $h'_s = h_s[i \mapsto Y[f_\tau \mapsto \textsc{null}]]$. Suppose $\sigma_s \; \mathcal{R}' \; \sigma_a$, we need to show there exists $\sigma'_a \in \mathcal{A}$ such that $\sigma_a \longrightarrow_{\mathcal{A}} \sigma'_a$ and $\sigma'_s \; \mathcal{R}' \; \sigma'_a$. Since $\sigma_s \; \mathcal{R}' \; \sigma_a$, we have $\sigma_s \in \gamma_a(\sigma_a)$. WLOG, suppose that σ_a has the form of $(g_a, pc, l_a, \delta_{\tau'} :: \omega_a, h_a, \phi')$ for some F with $\mathcal{V}'[\![\delta]\!](F) = i$ and $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F)$. After taking the GETFIELD1-A rule, we get an invisible state $t = (g'_a, pc, l'_a, i :: \omega'_a, h'_a, \phi'')$ with $t \in \mathit{init\text{-}sym\text{-}loc}(\sigma_a, \delta, i)$. By Lemma 3, we have $\sigma_s \in \mathcal{ST}_a[\![t]\!](F)$. Finally, we take the rule GETFIELD3-S and get $\sigma'_a =$

$(g'_a, next(pc), l'_a, \textsc{null} :: \omega'_a, h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \textsc{null}]], \phi'')$. We need to show $\sigma'_s \in \gamma_a(\sigma'_a)$. By Lemma 1, $sub\text{-}fun(g'_a, F) = sub\text{-}fun(g_a, F) = g_s$, $sub\text{-}fun(l'_a, F) = sub\text{-}fun(l_a, F) = l_s$, and $sub\text{-}seq(\omega'_a, F) = sub\text{-}seq(\omega_a, F) = \omega_s$ hold. It is sufficient to show that $(h_s[i \mapsto h_s(i)[f_\tau \mapsto \textsc{null}]], \phi) \in \mathcal{H}_a[\![(h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \textsc{null}]], \phi')]\!](symbols(\sigma'_a), collect\text{-}sym\text{-}locs(\sigma'_a), F)$. Since $symbols(\sigma'_a) = symbols(t)$ and $collect\text{-}sym\text{-}locs(\sigma'_a) = collect\text{-}sym\text{-}locs(t) = collect\text{-}sym\text{-}locs(\sigma_a \setminus \{\delta\})$, and $h'_a(i)(f) = h_s(i)(f) = \textsc{null}$, $(h_s[i \mapsto h_s(i)[f_\tau \mapsto \textsc{null}]], \phi) \in \mathcal{H}_a[\![(h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \textsc{null}]], \phi'')]\!](symbols(\sigma'_a), collect\text{-}sym\text{-}locs(\sigma'_a), F)$ by the definition of $\mathcal{H}_a$.

- Rule GETFIELD6-S Suppose $\sigma_s = (g_s, pc, l_s, i :: \omega_s, h_s, \phi)$. Then $\sigma'_s = (g_s, next(pc), l_s, j :: \omega_s, h'_s, \phi')$ where $h_s(i) = Y^{m,n}$ and $h'_s = h_s[i \mapsto Y^{m,n}[f_\tau \mapsto j]][j \mapsto Z_{\tau'}]$, $\phi' = \phi \cup \{\tau' <: \tau\}$ where $Z_{\tau'} = new\text{-}sym(symbols(\sigma_s), m-1, k)$ and $j \notin \operatorname{dom} h_s$. Suppose $\sigma_s\ \mathcal{R}'\ \sigma_a$, we need to show there exists $\sigma'_a \in \mathcal{A}$ such that $\sigma_a \longrightarrow_{\mathcal{A}} \sigma'_a$ and $\sigma'_s\ \mathcal{R}'\ \sigma'_a$. Since $\sigma_s\ \mathcal{R}'\ \sigma_a$, we have $\sigma_s \in \gamma_a(\sigma_a)$. WLOG, suppose that σ_a has the form of $(g_a, pc, l_a, \delta_{\tau'} :: \omega_a, h_a, \phi')$ for some F with $\mathcal{V}'[\![\delta]\!](F) = i$ and $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F)$. After taking the GETFIELD1-A rule, we get an invisible state $t = (g'_a, pc, l'_a, i :: \omega'_a, h'_a, \phi'')$ with $t \in init\text{-}sym\text{-}loc(\sigma_a, \delta, i)$. By Lemma 3, we get $\sigma_s \in \mathcal{ST}_a[\![t]\!](F)$. Finally, We can take GETFIELD3-S transition rule and get $\sigma'_a = (g'_a, next(pc), l'_a, \delta_\tau :: \omega'_a, h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \delta'']], \phi'')$ where $\delta'' \notin collect\text{-}sym\text{-}locs(t)$. Let $F' = F[\delta'' \mapsto j]$. Since δ'' is fresh in t, $sub\text{-}fun(g'_a, F') = sub\text{-}fun(g'_a, F) = g_s$, $sub\text{-}fun(l'_a, F') = sub\text{-}fun(l'_a, F) = l_s$, and $sub\text{-}seq(\omega'_a, F') = sub\text{-}seq(\omega'_a, F) = \omega_s$. It remains to show $(h_s[i \mapsto h_s(i)[f_\tau \mapsto j]][j \mapsto Z_{\tau'}], \phi \cup \{\tau' <: \tau\}) \in \mathcal{H}_a[\![(h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \delta'']], \phi'')]\!](symbols(\sigma'_a), collect\text{-}sym\text{-}locs(s'_a)$ Since we already have $(h_s, \phi) \in \mathcal{H}_a[\![(h'_a, \phi'')]\!](symbols(t), collect\text{-}sym\text{-}locs(t), F)$, according to the definition of $\mathcal{H}_a$ function, we only need to consider the extra elements: δ'', j, and Z. Since j is not in the domain of h_s, j is not in the domain of h'_a. So j is not in the domain of $h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \delta'']]$. We get $well\text{-}mapped(collect\text{-}sym\text{-}locs(t) \cup \{\delta''\}, h'_a[i \mapsto h'_a(i)[f \mapsto \delta'']], F')$. Since $Z = new\text{-}sym(symbols(\sigma_s), m-1, k)$ and $F'(\delta'') = j$, we have $heap(collect\text{-}sym\text{-}locs(t) \cup \{\delta''\}, h'_a[i \mapsto h'_a(i)[f \mapsto \delta]], h_s[i \mapsto h_s(i)[f_\tau \mapsto j]][j \mapsto Z_{\tau'}])$. Since F' introduce a new entry (δ'', j) then $pc(\phi'', \phi \cup \{\tau' <: \tau\}, h_s[i \mapsto h_s(i)[f_\tau \mapsto j]][j \mapsto Z_{\tau'}], F', collect\text{-}sym\text{-}locs(t) \cup \{\delta''\})$ holds. Thus $(h_s[i \mapsto h_s(i)[f_\tau \mapsto j]][j \mapsto Z_{\tau'}], \phi \cup \{\tau' <: \tau\}) \in \mathcal{H}_a[\![(h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \delta]], \phi'')]\!](symbols(\sigma'_a), collect\text{-}sym\text{-}locs(\sigma'_a), F')$ holds.

□

Next we define a relation.

Definition 5. *$\mathcal{R}'_\bullet \subseteq \Sigma_{\mathcal{A}} \times \mathcal{P}(\Sigma_{\mathcal{S}})$, as follows:*

$$\sigma_a\ \mathcal{R}'_\bullet\ S_s \iff \gamma^*_a(\sigma_a) = S_s$$

Clearly, $\mathcal{R}'_\bullet$ is left total. Since $\mathcal{R}'$ is right total, then for all σ_a, if $\sigma_a\ \mathcal{R}'_\bullet\ S_s$, then $S_s \neq \emptyset$. Furthermore, for any $\sigma_a \in I_{\mathcal{A}}$ and $\sigma_a\ \mathcal{R}'_\bullet\ S_s$, it is clear that $S_s \subseteq I_{\mathcal{S}}$ by the definition of γ_a function.

Proposition 6. *$\mathcal{A} \lhd_{\mathcal{R}'_\bullet} \mathcal{P}(\mathcal{S})$.*

Proof. It is sufficient to show that for all $\sigma_a \in \Sigma_{\mathcal{A}}, S_s \in \mathcal{P}(\Sigma_{\mathcal{S}})$ if $\sigma_a \longrightarrow_{\mathcal{A}} \sigma'_a$ and $\sigma_a\ \mathcal{R}'_\bullet\ S_s$ and $\sigma'_a\ \mathcal{R}'_\bullet\ S'_s$ then $S_s \xrightarrow{\bullet}_{\mathcal{S}} S'_s$.

We will prove by rule induction on transitions, $\longrightarrow_{\mathcal{A}}$.

- Rule `if_acmpeq`: Suppose, WLOG, $\sigma_a = (g_a, pc, l_a, \delta_\tau :: \delta'_{\tau'} :: \omega_a, h_a, \phi')$. Then by the definition of $\longrightarrow_{\mathcal{A}}$, the rule consists of three lazier symbolic transitions rules: IF_ACMPEQ2-A, IF_ACMPEQ1-A, and IF_ACMPEQ1-S or IF_ACMPEQ2-S. After taking IF_ACMPEQ2-A rule, we get an invisible state $t_1 = (g'_a, pc, l'_a, i :: \delta'_{\tau'} :: \omega'_a, h'_a, \phi'')$ for some $i \in \mathbf{Locs}$ and $t_1 \in$ *init-sym-loc*(σ_a, δ, i). Then after taking IF_ACMPEQ1-A rule, we get another invisible state $t_2 = (g''_a, pc, l''_a, i :: j :: \omega''_a, h''_a, \phi''')$ for some $j \in \mathbf{Locs}$ and $t_2 \in$ *init-sym-loc*(t_2, δ', j). WLOG, suppose $i \neq j$ (the $i = j$ case is symmetric). Finally, we take IF_ACMPEQ1-S rule and get $\sigma'_a = (g''_a, next(pc), l''_a, \omega''_a, h''_a, \phi''')$. Suppose $\sigma_a \mathrel{\mathcal{R}'_\bullet} S_s$ and $\sigma'_a \mathrel{\mathcal{R}'_\bullet} S'_s$. We need to show that $S_s \overset{\bullet}{\longrightarrow}_{\mathcal{S}} S'_s$, that is, for any $\sigma'_s \in S'_s$, there exists some $\sigma_s \in S_s$ such that $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$. Suppose $\sigma'_s \in S'_s$, that is, $\sigma'_s \in \gamma_a(s'_a)$. Then σ'_s must be in the form of $(g'_s, next(pc), l'_s, \omega'_s, h'_s, \phi)$ for some F and $\sigma'_s \in \mathcal{ST}_a[\![\sigma'_a]\!](F)$. Define $\sigma_s = (g'_s, next(pc), l'_s, i :: j :: \omega'_s, h'_s, \phi)$. It is clear that $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$. We only need to show $\sigma_s \in S_s$, that is, $\sigma_s \in \gamma_a(\sigma_a)$. Define $F' = F[\delta \mapsto i][\delta' \mapsto j]$. We will show $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F')$. Since δ and δ' do not appear in σ'_a, thus t_2, we have $\sigma_s \in \mathcal{ST}_a[\![t_2]\!](F')$ by Lemma 2 and property of $\mathcal{H}_a$. By applying Lemma 4 twice, we get $\sigma_s \in \mathcal{ST}_a[\![\sigma_a]\!](F')$.

- Rule `getfield` f_τ: Suppose, WLOG, $\sigma_a = (g_a, pc, l_a, \delta_{\tau'} :: \omega_a, h_a, \phi')$ and $\tau \in \mathbf{Types}_{record}$. By the definition of $\longrightarrow_{\mathcal{A}}$, the transition consists of two lazier rules: GETFIELD1-A and (GETFIELD2-A, GETFIELD3-A, or GETFIELD1-S). After taking the GETFIELD1-A rule, we get an invisible state $t = (g'_a, pc, l'_a, i :: \omega'_a, h'_a, \phi'')$ for some $i \in \mathbf{Locs}$ and $t =$ *init-sym-loc*(σ_a, δ, i). WLOG, assume that f field is undefined in $h'_a(i)$. We take the GETFIELD2-A rule and get $\sigma'_a = (g'_a, next(pc), l'_a, \delta'_\tau :: \omega'_a, h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \delta']], \phi'')$, where δ' is fresh in t.

  Suppose $\sigma_a \mathrel{\mathcal{R}'_\bullet} S_s$ and $\sigma'_a \mathrel{\mathcal{R}'_\bullet} S'_s$. We need to show that $S_s \overset{\bullet}{\longrightarrow}_{\mathcal{S}} S'_s$, that is, for any $\sigma'_s \in S'_s$, there exists some $\sigma_s \in S_s$ such that $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$. Suppose $\sigma'_s \in S'_s$, that is, $\sigma'_s \in \gamma_a(\sigma'_a)$. Then σ'_s must be in the form of $(g'_s, next(pc), l'_s, j :: \omega'_s, h'_s, \phi)$ for some F such that $F(\delta') = j$ and $\sigma'_s \in \mathcal{ST}_a[\![\sigma'_a]\!](F)$. Define h_s as h'_s after following two operations:

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

  - $F(\delta') \notin F(\textit{collect-sym-locs}(t) \cup \operatorname{dom} h'_a$: then the entry $(F(\delta'), h'_s(F(\delta'))$ is removed from h_s. Since *heap*$(\textit{collect-sym-locs}(\sigma'_a), h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \delta']], h'_s, F)$ holds and

$collect\text{-}sym\text{-}locs(\sigma'_a) - collect\text{-}sym\text{-}locs(t) = \{\delta'\}$, we have $heap(collect\text{-}sym\text{-}locs(t), h'_a, F)$ holds.

– otherwise: so the entry $(F(\delta'), h'_s(F(\delta')))$ is not removed from h_s by the definition of h_s. We are done because $heap(collect\text{-}sym\text{-}locs(\sigma'_a), h'_a[i \mapsto h'_a(i)[f_\tau \mapsto \delta']], h'_s, F)$ holds.

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

– $h_s(F(\delta'))$ is not defined: Since the σ'_a has only δ' that is not in σ_a, so $h'_s(F(\delta'))$ is a fresh symbol. We can take the GETFIELD6-S rule and get $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$.

– $h_s(F(\delta'))$ is defined: By the wellmappedness of h_a, $h_s(F(\delta'))(\textsc{conc})$ is not defined. So we can take the GETFIELD4-S rule and get $\sigma_s \longrightarrow_{\mathcal{S}} \sigma'_s$.

□

Soundness and Completeness

Proposition 7 (Soundness)**.** *Given any symbolic trace $s_1 \longrightarrow_{\mathcal{S}} s_2 \longrightarrow_{\mathcal{S}} \cdots \longrightarrow_{\mathcal{S}} s_n$ with $s_1 \in I_{\mathcal{S}}$, there is a corresponding lazier symbolic trace $a_1 \longrightarrow_{\mathcal{A}} a_2 \longrightarrow_{\mathcal{A}} \cdots \longrightarrow_{\mathcal{A}} a_n$ with $a_1 \in I_{\mathcal{A}}$ such that $s_k \mathcal{R}' a_k$ for all $1 \leq k \leq n$.*

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

Proposition 8 (Completeness)**.** *Given any lazier symbolic trace $a_1 \longrightarrow_{\mathcal{A}} a_2 \longrightarrow_{\mathcal{A}} \cdots \longrightarrow_{\mathcal{A}} a_n$ with $a_1 \in I_{\mathcal{A}}$, there is a corresponding symbolic trace $s_1 \longrightarrow_{\mathcal{S}} s_2 \longrightarrow_{\mathcal{S}} \cdots \longrightarrow_{\mathcal{S}} s_n$ such that $s_k \mathcal{R}' a_k$ for all $1 \leq k \leq n$ and $s_1 \in I_{\mathcal{S}}$.*

Proof. It is easy to show that there exists a trace in $\mathcal{P}(\mathcal{S})$, $S_1 \stackrel{\bullet}{\longrightarrow}_{\mathcal{S}} S_2 \stackrel{\bullet}{\longrightarrow}_{\mathcal{S}} \cdots \stackrel{\bullet}{\longrightarrow}_{\mathcal{S}} S_n$ such that $a_k \mathcal{R}_\bullet S_k$ for all $1 \leq k \leq n$ by mathematical induction on n using Proposition 6. Since $S_n \neq \emptyset$, we can pick a $s_n \in S_n$ and use the definition of $\stackrel{\bullet}{\longrightarrow}_{\mathcal{S}}$, then get the corresponding symbolic trace $s_1 \longrightarrow_{\mathcal{S}} s_2 \longrightarrow_{\mathcal{S}} \cdots \longrightarrow_{\mathcal{S}} s_n$. □

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 laizer symbolic execution in Section C.2.2. First, we define a function γ_b 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-procedurely.

Definition of γ_b

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

$$\Xi = \{\, G \mid G : \textbf{SymRefs} \to (\textbf{SymLocs} \cup \{\textsc{null}\})\,\}\,. \tag{C.2}$$

Then we define a function: *legal-env* : $\Sigma_b \to \mathcal{P}(\Xi)$ as

$$\begin{aligned} \textit{legal-env}(\sigma_b) = \{G \in \Xi \mid\ & G(\textit{collect-sym-refs}(\sigma_b)) \cap \textit{collect-sym-locs}(\sigma_b) = \emptyset \wedge \\ & \forall \hat{\delta}_1 \neq \hat{\delta}_2 \in \textit{collect-sym-refs}(\sigma_b).G(\hat{\delta}_1) = G(\hat{\delta}_2) \implies G(\hat{\delta}_1) = \textsc{null}\}, \end{aligned}$$

where *collect-sym-refs* collects all the symbolic references in a state.
And $\mathcal{ST}_b : \Sigma_b \times \Xi \to \Sigma_a$ as

$$\mathcal{ST}_b[\![\sigma_b]\!](G) = (\textit{sub-fun}(g, G), pc, \textit{sub-fun}(l, G), \textit{sub-seq}(\omega, G), \textit{sub-fun2}(h, G), \phi),$$

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

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

$$\gamma_b(\sigma_b) = \bigcup_{\forall G \in \textit{legal-env}(\sigma_b)} \mathcal{ST}_b[\![\sigma_b]\!](G).$$

Properties of γ_b

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

Proof. Suppose $\sigma_b = (g_b, pc, l_b, \omega_b, h_b, \phi)$. By the definition of *init-sym-ref*, $\sigma'_b = (\textit{sub-fun}_1(g_b, \hat{\delta}, v), pc'$, $\textit{sub-fun}_1(l_b, \hat{\delta}, v), \textit{sub-seq}_1(\omega_b, \hat{\delta}, v), \textit{sub-fun2}_1(h_b, \hat{\delta}, v), \phi)$. Since $\sigma_a = \mathcal{ST}_b[\![\sigma_b]\!](G)$, we have $g_a = \textit{sub-fun}(\textit{sub-fun}_1(g_b, \hat{\delta}, v), G)$, $l_a = \textit{sub-fun}(\textit{sub-fun}_1(l_b, \hat{\delta}, v), G)$, $\omega_a = \textit{sub-seq}(\textit{sub-seq}_1(\omega_b, \hat{\delta}, v), G)$, and $h_a = \textit{sub-fun2}(\textit{sub-fun2}_1(h_b, \hat{\delta}, v)), G)$, by Lemma 1. We conclude that $(g_a, pc', l_a, \omega_a, h_a, \phi_a) \in \mathcal{ST}_b[\![\sigma'_b]\!](G)$ holds. □

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

Proof. Proof is similar to Lemma 5. □

Improved Lazier Kripke Structure

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

- states, $\Sigma_{\mathcal{B}} = \Sigma_b \cup (\textsc{Exception} \times \Sigma_b) \cup (\textsc{Error} \times \Sigma_b)$.

- initial states,

$$I_{\mathcal{B}} = \{ (g_b, pc_{init}, l_b, nil, h_b, \{\textsc{True}\}) \mid \mathrm{dom}(g_b) = \mathit{Globals} \wedge \mathrm{dom}(l_b) = \mathit{Locals} \},$$

  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, $b \longrightarrow_{\mathcal{B}} b' \iff b \Rightarrow_{\mathcal{B}} b_2, b_2 \Rightarrow_{\mathcal{B}} b_3, \ldots, b_n \Rightarrow_{\mathcal{B}} b'$ 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 γ_a, function γ_b is trivially extended to $\gamma_b^* : \Sigma_{\mathcal{B}} \to \mathcal{P}(\Sigma_{\mathcal{A}})$ as

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

Simulation Relations

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

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

Clearly, for all $a_0 \in I_{\mathcal{A}}$, there exists a $b_0 \in I_{\mathcal{B}}$ such that $a_0 \ \mathcal{R}'' \ b_0$.

Proposition 9. $\mathcal{A} \lhd_{\mathcal{R}''} \mathcal{B}$.

Proof. It is sufficient to show that for all $\sigma_a, \sigma_a' \in \Sigma_{\mathcal{A}}, \sigma_b \in \Sigma_{\mathcal{B}}$ if $\sigma_a \longrightarrow_{\mathcal{A}} \sigma_a'$ and $\sigma_a \ \mathcal{R}'' \ \sigma_b$ then there exists $\sigma_b' \in \Sigma_{\mathcal{B}}$ such that $\sigma_b \longrightarrow_{\mathcal{B}} \sigma_b'$ and $\sigma_a' \ \mathcal{R}'' \ \sigma_b'$. We will proceed with the rule induction on $\longrightarrow_{\mathcal{A}}$.

- Rule `if_acmpeq`: Suppose, WLOG, $\sigma_a = (g_a, pc, l_a, \delta_\tau :: \delta'_{\tau'} :: \omega_a, h_a, \phi')$. Then by the definition of $\longrightarrow_{\mathcal{A}}$, the rule consists of three lazier symbolic transitions rules: IF_ACMPEQ3-A, IF_ACMPEQ2-A, and IF_ACMPEQ1-S or IF_ACMPEQ2-S. After taking IF_ACMPEQ3-A rule, we get an invisible state $t_1 = (g_a', pc, l_a', i :: \delta'_{\tau'} :: \omega_a', h_a', \phi'')$ for some $i \in \mathbf{Locs}$ and $t_1 \in \mathit{init\text{-}sym\text{-}loc}(\sigma_a, \delta, i)$. Then after taking IF_ACMPEQ2-A rule, we get another invisible state $t_2 = (g_a'', pc, l_a'', i :: j :: \omega_a'', h_a'', \phi''')$ for some $j \in \mathbf{Locs}$ and $t_2 \in \mathit{init\text{-}sym\text{-}loc}(t_2, \delta', j)$. WLOG, suppose $i \neq j$ (the $i = j$ case is symmetric). Finally, we take IF_ACMPEQ1-S rule and get $\sigma_a' = (g_a'', \mathit{next}(pc), l_a'', \omega_a'', h_a'', \phi''')$. Suppose $\sigma_a \ \mathcal{R}'' \ \sigma_b$. We need to show that there exists any $\sigma_b' \in \Sigma_{\mathcal{B}}$ such that $\sigma_b \longrightarrow_{\mathcal{B}} \sigma_b'$. WLOG, suppose that $\sigma_b = (g_b, pc, l_b, \hat{\delta} :: \delta' :: \omega_b, h_b, \phi)$. Since $\sigma_a \ \mathcal{R}'' \ \sigma_b$, there exists $G \in \mathit{legal\text{-}env}(\sigma_b)$ such that $\sigma_a = \mathcal{ST}_b[\![\sigma_b]\!](G)$. Clearly $G(\hat{\delta}) = \delta$. We take rule IF_ACMEQ3-B and get a state $t_0' = \mathit{init\text{-}sym\text{-}ref}(\sigma_b, \hat{\delta}, \delta)$ with stack $\delta :: \delta' :: \mathit{sub\text{-}seq}(\omega_b, \hat{\delta}, \delta)$. By Lemma 5, we get $\sigma_a \ \mathcal{R}'' \ t_0'$. Then we take

IF_ACMPEQ3-A, IF_ACMPEQ2-A, and IF_ACMPEQ1-S. We get $t'_1 = \textit{init-sym-loc}(t'_0, \delta, i)$ after rule IF_ACMPEQ3-A, $t'_2 = \textit{init-sym-loc}(t'_1, \delta', j)$ after rule IF_ACMPEQ2-A, and σ'_b after IF_ACMPEQ1-S. Since all the rules do not involve any symbolic references, it clearly that $t_1 \mathcal{R}'' t'_1$, and $t_2 \mathcal{R}'' t'_2$, and finally $\sigma'_a \mathcal{R}'' \sigma'_b$.

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

□

Next we define a relation.

Definition 7. $\mathcal{R}''_\bullet \subseteq \Sigma_\mathcal{B} \times \mathcal{P}(\Sigma_\mathcal{A})$*, as follows:*

$$\sigma_b \mathcal{R}''_\bullet S_a \iff \gamma_b^*(\sigma_b) = S_a$$

Clearly, $\mathcal{R}''_\bullet$ is left total. Since $\mathcal{R}''$ is right total, then for all σ_b, if $\sigma_b \mathcal{R}''_\bullet S_a$, then $S_a \neq \emptyset$. Furthermore, for any $\sigma_b \in I_\mathcal{B}$ and $\sigma_b \mathcal{R}''_\bullet S_a$, it is clear that $S_a \subseteq I_\mathcal{A}$ by the definition of γ_b function.

Proposition 10. $\mathcal{B} \lhd_{\mathcal{R}''_\bullet} \mathcal{P}(\mathcal{A})$.

Proof. It is sufficient to show that for all $\sigma_b \in \Sigma_\mathcal{B}, S_a \in \mathcal{P}(\Sigma_\mathcal{A})$ if $\sigma_b \longrightarrow_\mathcal{B} \sigma'_b$ and $\sigma_b \mathcal{R}''_\bullet S_a$ and $\sigma'_b \mathcal{R}'_\bullet S'_a$ then $S_a \stackrel{\bullet}{\longrightarrow}_\mathcal{A} S'_a$.

We will prove by rule induction on transitions, $\longrightarrow_\mathcal{B}$.

- Rule `if_acmpeq`: Suppose, WLOG, $\sigma_b = (g_b, pc, l_b, \hat{\delta}_1 :: \hat{\delta}_2 :: \omega_b, h_b, \phi)$. Then by the definition of $\longrightarrow_\mathcal{B}$, the rule consists of five transitions rules: IF_ACMPEQ3-B, IF_ACMEQ2-B, IF_ACMPEQ3-A, IF_ACMPEQ2-A, and IF_ACMPEQ1-S or IF_ACMPEQ2-S. After taking IF_ACMPEQ3-B rule, we get an invisible state $t_1 = \textit{init-sym-ref}(\sigma_b, \hat{\delta}_1, \delta_1)$ and then IF_ACMPEQ2-B rule, we get $t_2 = \textit{init-sym-ref}(t_1, \hat{\delta}_2, \delta_2)$. Then by IF_ACMPEQ3-A rule, we get to t_3 and by IF_ACMEQ2-A rule, we arrive at t_4 where δ_1 and δ_2 are fresh. WLOG, suppose we take IF_ACMPEQ1-S rule and get $\sigma'_b = (\textit{sub-fun}_1(\textit{sub-fun}_1(g_b, \hat{\delta}_1, \delta_1), \hat{\delta}_2, \delta_2), \textit{next}(pc), \textit{sub-fun}_1(\textit{sub-fun}_1(l_b, \hat{\delta}_1, \delta_1)$
$, \textit{sub-seq}_1(\textit{sub-seq}_1(\omega_b, \hat{\delta}_1, \delta_1), \hat{\delta}_2, \delta_2),$
$\textit{sub-fun2}_1(\textit{sub-fun2}_1(h_b, \hat{\delta}_1, \delta_1), \hat{\delta}_2, \delta_2), \phi')$. Suppose $\sigma_b \mathcal{R}'_\bullet S_a$ and $\sigma'_b \mathcal{R}'_\bullet S'_a$. We need to show that $S_a \stackrel{\bullet}{\longrightarrow}_\mathcal{S} S'_a$, that is, for any $\sigma'_a \in S'_a$, there exists some $\sigma_a \in S_a$ such that $\sigma_a \longrightarrow_\mathcal{A} \sigma'_a$. Suppose $\sigma'_a \in S'_a$, that is, $\sigma'_a \in \gamma_b(\sigma'_b)$. Then σ'_a must be in the form of $(g'_a, \textit{next}(pc), l'_a, \omega'_a, h'_a, \phi)$ for some G and $\sigma'_a \in \mathcal{ST}_b[\![\sigma'_b]\!](G)$. Define $G' = G[\hat{\delta}_1 \mapsto \delta_1][\hat{\delta}_2 \mapsto \delta_2]$. Clearly $G' \in \textit{legal-env}(\sigma_b)$. Define $\sigma_a = \mathcal{ST}_b[\![\sigma_b]\!](G')$. We need to show

that $\sigma_a \longrightarrow_{\mathcal{A}} \sigma'_a$. After applying Lemma 5 twice, we get $\sigma_a = \mathcal{ST}_b[\![t_2]\!](G')$. Since σ_a only differs from t_2 by some symbolic references which are not operands of the instruction, σ_a can takes exactly the same rules and get to σ'_a. We conclude that $\sigma_a \longrightarrow_{\mathcal{A}} \sigma'_a$.

- Rule `getfield` f_τ: Suppose, WLOG, $\sigma_b = (g_b, pc, l_b, \hat{\delta}_{\tau'} :: \omega_b, h_b, \phi')$ and $\tau \in \mathbf{Types}_{record}$. By the definition of $\longrightarrow_{\mathcal{B}}$, the transition multiple lazier# rules. The first one is GETFIELD1-B. WLOG, assume that the invisible state after GETFIELD1-B is $t_1 = (\textit{sub-fun}_1(g_b, \hat{\delta}, \delta)$, pc, $\textit{sub-fun}_1(l_b, \hat{\delta}, \delta)$, $\textit{sub-seq}_1(\omega_b, \hat{\delta}, \delta)$, $\textit{sub-fun2}_1(h_b, \hat{\delta}, \delta), \phi)$ for some fresh δ. Then rule GETFIELD1-A is taken and get an invisible state $t_2 = (g_2, pc, l_2, \omega_2, h_2, \phi_2) = \textit{init-sym-loc}(t, \delta, i)$ for some $i \in \mathbf{Locs}$. WLOG, assume that f field is undefined in $h_2(i)$. We take the GETFIELD2-B rule and get $\sigma'_b = (g_2, \textit{next}(pc), l_2, \hat{\delta}'_\tau :: \omega_2, h_2[i \mapsto h_2(i)[f_\tau \mapsto \hat{\delta}']], \phi_2)$, where $\hat{\delta}'$ is fresh in t_2.

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

□

Soundness and Completeness

Proposition 11 (Soundness)**.** *Given any lazier symbolic trace $a_1 \longrightarrow_{\mathcal{A}} a_2 \longrightarrow_{\mathcal{A}} \cdots \longrightarrow_{\mathcal{A}} a_n$ with $a_1 \in I_{\mathcal{A}}$, there is a corresponding lazier# symbolic trace $b_1 \longrightarrow_{\mathcal{B}} b_2 \longrightarrow_{\mathcal{B}} \cdots \longrightarrow_{\mathcal{B}} b_n$ with $b_1 \in I_{\mathcal{B}}$ such that $a_k\ \mathcal{R}''\ b_k$ for all $1 \leq k \leq n$.*

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

Proposition 12 (Completeness)**.** *Given any lazier# symbolic trace $b_1 \longrightarrow_{\mathcal{B}} b_2 \longrightarrow_{\mathcal{B}} \cdots \longrightarrow_{\mathcal{B}} b_n$ with $b_1 \in I_{\mathcal{B}}$, there is a corresponding symbolic trace $a_1 \longrightarrow_{\mathcal{A}} a_2 \longrightarrow_{\mathcal{A}} \cdots \longrightarrow_{\mathcal{A}} a_n$ such that $a_k\ \mathcal{R}''\ b_k$ for all $1 \leq k \leq n$ and $a_1 \in I_{\mathcal{A}}$.*

Proof. It is easy to show that there exists a trace in $\mathcal{P}(\mathcal{A})$, $S_1 \overset{\bullet}{\longrightarrow}_{\mathcal{A}} S_2 \overset{\bullet}{\longrightarrow}_{\mathcal{A}} \cdots \overset{\bullet}{\longrightarrow}_{\mathcal{A}} S_n$ such that $b_k\ \mathcal{R}''_\bullet\ S_k$ 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 $\overset{\bullet}{\longrightarrow}_{\mathcal{A}}$, then get the corresponding lazier symbolic trace $a_1 \longrightarrow_{\mathcal{A}} a_2 \longrightarrow_{\mathcal{A}} \cdots \longrightarrow_{\mathcal{A}} a_n$. □

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,* $\mathcal{K} = (\Sigma_\mathcal{K}, I_\mathcal{K}, \longrightarrow_\mathcal{K}, L_\mathcal{K})$*, where* $\Sigma_\mathcal{K}$ *is a set of states,* $I_\mathcal{K}$ *is a set of initial states that* $I_\mathcal{K} \subseteq \Sigma_\mathcal{K}$*,* $\longrightarrow_\mathcal{K} \subseteq \Sigma_\mathcal{K} \times \Sigma_\mathcal{K}$ *is the transition relation (finite image), and* $L_\mathcal{K} : \Sigma_\mathcal{K} \to \mathcal{P}(Atom)$ *associates a set of atomic properties,* $\forall s \in \Sigma_\mathcal{K}.L_\mathcal{K}(s) \subseteq Atom$*.*

Definition 9 (Simulation Relation on Kripke Structures)**.** *For Kripke structures* $C = (\Sigma_C, I_C, \longrightarrow_C, L_C)$ *and* $\mathcal{S} = (\Sigma_\mathcal{S}, I_\mathcal{S}, \longrightarrow_\mathcal{S}, L_\mathcal{S})$*, a binary relation,* $\mathcal{R} \subseteq \Sigma_C \times \Sigma_\mathcal{S}$*, is a simulation of* C *by* $\mathcal{S}$*, written* $C \lhd_\mathcal{R} \mathcal{S}$*, if* $\forall c \in \Sigma_C, s \in \Sigma_\mathcal{S}.c\ \mathcal{R}\ s \wedge c \longrightarrow c' \implies \exists s' \in \Sigma_\mathcal{S}.s \longrightarrow s' \wedge c'\ \mathcal{R}\ s'$ *and* $\forall c_0 \in I_C.\exists s_0 \in I_\mathcal{S}.c_0\ \mathcal{R}\ s_0$*.*

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,* $\mathcal{K} = (\Sigma_\mathcal{K}, I_\mathcal{K}, \longrightarrow_\mathcal{K}, L_\mathcal{K})$*, the power kripke structure* $\mathcal{P}(\mathcal{K}) = (\mathcal{P}(\Sigma_\mathcal{K}), \mathcal{P}(I_\mathcal{K}), \stackrel{\bullet}{\longrightarrow}_\mathcal{K}, L_{\mathcal{P}(\mathcal{K})})$*, where* $\forall S, S' \subseteq \Sigma_\mathcal{K}.S \stackrel{\bullet}{\longrightarrow}_\mathcal{K} S'$ *if and only if for every* $s' \in S'$*, there exists some* $s \in S$ *such that* $s \longrightarrow_\mathcal{K} s'$ *and* $L_{\mathcal{P}(\mathcal{K})}(S) = \cap \{ L_\mathcal{K}(s) \mid s \in S \}$*.*