

Precise Heap Summaries from Symbolic Execution

Anonymous

Abstract

One of the fundamental challenges of using symbolic execution to analyze software has been the treatment of dynamically allocated data. State-of-the-art symbolic execution techniques have addressed this challenge by constructing the heap *lazily*, materializing objects on the concrete heap “as needed” and using non-deterministic choice points to explore each feasible concrete heap configuration. Because analysis of the materialized heap locations relies on concrete program semantics, the lazy initialization approach exacerbates the state space explosion problem that limits the scalability of symbolic execution. In this work we present a novel approach for lazy symbolic execution of heap manipulating software which utilizes a fully symbolic heap constructed on-the-fly during symbolic execution. Our approach is 1) *scalable* – it does not create the additional points of non-determinism introduced by existing lazy initialization techniques and it explores each execution path only once for any given set of isomorphic heaps, 2) *precise* – at any given point during symbolic execution, the symbolic heap represents the exact set of feasible concrete heap structures for the program under analysis, and 3) *expressive* – the symbolic heap can represent recursive data structures and heaps resulting from loops and recursive control structures in the code. We report on a case-study of an implementation of our technique in the Symbolic PathFinder tool to illustrate its scalability, precision and expressiveness. We also discuss how test case generation – a common use for symbolic execution results – can benefit from symbolic execution which uses a fully symbolic heap.

Categories and Subject Descriptors CR-number [subcategory]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

In recent years symbolic execution – a program analysis technique for systematic exploration of program execution paths using symbolic input values – has provided the basis for various software testing and analysis techniques. For each execution path explored during symbolic execution, constraints on the symbolic inputs are collected to create a *path condition*. The set of path conditions computed by symbolic execution characterize the observed program ex-

ecution behaviours and can be used as an enabling technology for various applications, e.g., regression analysis [2, 8, 13–15, 17], data structure repair [10], dynamic discovery of invariants [4, 18], and debugging [12].

Initial work on symbolic execution largely focused on checking properties of programs with primitive types, such as integers and booleans [3, 11]. Despite recent advances in constraint solving technologies, improvements in raw computing power, and advances in reduction and abstraction techniques [1, 7] symbolic execution of programs of modest size containing only primitive types, remains challenging because of the large number of execution paths generated during symbolic analysis.

With the advent of object-oriented languages that manipulate dynamically allocated data, e.g., Java and C++, recent work has generalized the core ideas of symbolic execution to enable analysis of programs containing complex data structures with unbounded domains, i.e., data stored on the heap [5, 6, 9]. These techniques construct the heap in a lazy manner, deferring materialization of objects on the concrete heap until they are needed for the analysis to proceed. Treatment of heap allocated data then follows concrete program semantics once a heap location is materialized, resulting in a large number of feasible concrete heap configurations, and as a result, a large number of points of non-determinism to be analyzed, further exacerbating the state space explosion problem.

THIS PARA IS NOT QUITE RIGHT BUT THE IDEA IS STARTING TO COME OUT. Although lazy symbolic execution techniques have been instrumental in enabling analysis of heap manipulating programs, they miss an important opportunity to control the state space explosion problem by treating only inputs with primitive types symbolically and materializing a concrete heap. As we show in this work, the use of a fully *symbolic heap* during lazy symbolic execution, can improve the scalability of the analysis while maintaining precision and efficiency. Moreover, the number of path conditions computed by lazy symbolic execution when a symbolic heap is used produces considerably fewer path conditions – a valuable benefit for client analyses that use the results of symbolic execution, e.g., regression analyses.

The key advantages of our approach to lazy symbolic execution using a fully symbolic heap include:

- *Scalability*. Our approach constructs the symbolic heap on-the-fly during symbolic execution and avoids creating the additional points of non-determinism introduced by existing lazy initialization techniques. Moreover, it explores each execution path only once for any given set of isomorphic heaps.
- *Precision*. At any given point during symbolic execution, the symbolic heap represents the exact set of feasible concrete heap structures for the program under analysis
- *Expressiveness*. The symbolic heap can represent recursive data structures and heap structures resulting from loops and recursive control structures in the analyzed code.

This paper makes the following contributions:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CONF 'yy, Month d–d, 20yy, City, ST, Country.

Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm...\$15.00.

<http://dx.doi.org/10.1145/nnnnnnnn.nnnnnnn>

- We present a novel lazy symbolic execution technique for analyzing heap manipulating programs that constructs a fully symbolic representation of the heap on-the-fly during symbolic execution.
- We prove the soundness and completeness of our algorithm...
- We implement our approach in the Symbolic Pathfinder tool
- We demonstrate experimentally that our technique improves the scalability of symbolic execution of heap manipulating software over state-of-the-art techniques, while maintaining efficiency and precision.
- We discuss the benefits of using a symbolic heap that can be realized by the client analysis that uses the results of symbolic execution.

2. Background and Motivation

In this section we present the background on state of the art techniques that have been developed to handle data non-determinism arising from complex data structures. We present an overview of lazy initialization and lazier# initialization. We also present a brief description of the two bounding strategies used in symbolic execution in heap manipulating programs. Next we present a motivating examples where current concrete initialization of the heap structures struggle to scale to medium sized program due to non-determinism introduced in the symbolic execution tree. We use this example to motivate the need for a more truly symbolic and compact representation of the heap in a manner similar to that of primitive types.

Generalized symbolic execution technique generates a concrete representation of connected memory structures using only the implicit information from the program itself. In the original lazy initialization algorithm, symbolic execution explores different heap shapes by concretizing the heap at the first memory access (read) to an un-initialized symbolic object. At this point, a non-deterministic choice point of concrete heap locations is created that includes: (a) null, (b) an access to a new instance of the object, and (c) aliases to other type-compatible symbolic objects that have been concretized along the same execution path [?]. The number of choices explored in lazy initialization greatly increases the non-determinism and often makes the exploration of the program state space intractable.

The Lazier# algorithm is an improvement of the lazy initialization and it pushes the non-deterministic choices further into the execution tree. In the case of a memory access to an uninitialized reference location, by default, no choice point is created. Instead, the read returns a unique symbolic reference representing the contents of the location. The reference may assume any one of three states: uninitialized, non-null, or initialized. The reference is returned in an uninitialized state, and only in a subsequent memory access is the reference concretely initialized.

3. Generalized Symbolic Execution with Lazy Initialization

Generalized symbolic execution with lazy initialization is a technique to apply symbolic execution to non-primitive data types (i.e., objects) [9]. The approach initializes each symbolic object to be either null, a new instance of the object, or an alias to an object previously created as part of lazy initialization. Each of these outcomes is a possible choice, and a model checker is able to exhaustively enumerate these choices effectively creating all possible concrete heaps under the choice sets. The approach is first formalized in this section and then extended to describe the precise heap summaries.

Many of these concrete heaps explore redundant control flow paths in the program under test, but since symbolic model checking

```
public class LinkedList {

    /** assume the linked list is valid with no cycles */
    LLNode head;
    Data data0, data1, data2, data3, data4;

    private class Data { Integer val; }

    private class LLNode {
        protected Data elem;
        protected LLNode next; }

    public static boolean contains(LLNode root, Data val) {
        LLNode node = root;
        while (true) {
            if(node.val == val) return true;
            if(node.next == null) return false;
            node = node.next;
        }
    }

    public void run() {
        if(LinkedList.contains(head, data0) &&
           LinkedList.contains(head, data1) &&
           LinkedList.contains(head, data2) &&
           LinkedList.contains(head, data3) &&
           LinkedList.contains(head, data4)) return;
    }
}
```

Figure 1. Linked list

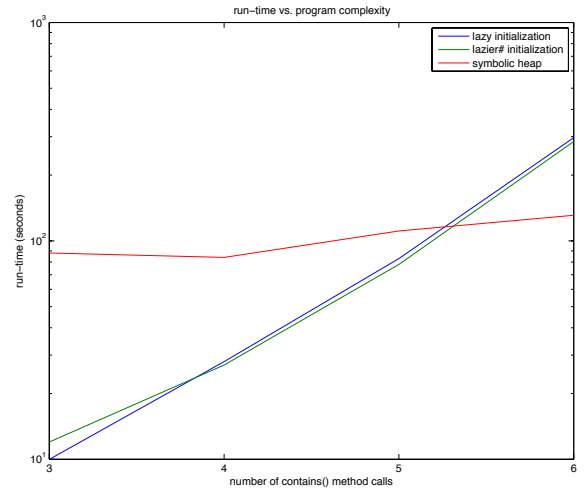


Figure 2. Time versus complexity for the linked list example

is stateless, subject to a bound to terminate the search, the model checker needlessly explores many redundant state leading to severe state explosion.

Figure 3 defines the surface syntax for the Javalite language [16]. Figure 4 is the machine syntax. Javalite is syntactic machine defined as rewrites on a string. The semantics use a CEKS model with a (C)ontrol string representing the expression being evaluated, an (E)nvironment for local variables, a (K)ontinuation for what is to be executed next, and a (S)tore for the heap.

The heap is a labeled bipartite graph consisting of references R and locations L in the store. The functions R and L are defined for convenience in manipulating the labeled bipartite graph.

- $R(l, f)$ maps location-field pairs from the store to a reference in R ; and

```

 $P ::= (\mu (C m))$ 
 $\mu ::= (CL \dots)$ 
 $T ::= \text{bool} \mid C$ 
 $CL ::= (\text{class } C ([T f] \dots) (M \dots))$ 
 $M ::= (T m [T x] e)$ 
 $e ::= x$ 
    |  $(\text{new } C)$ 
    |  $(e \$ f)$ 
    |  $(x \$ f := e)$ 
    |  $(e = e)$ 
    |  $(\text{if } e \text{ else } e)$ 
    |  $(\text{var } T x := e \text{ in } e)$ 
    |  $(e @ m e)$ 
    |  $(x := e)$ 
    |  $(\text{begin } e \dots)$ 
    |  $v$ 
 $x ::= \text{this} \mid id$ 
 $f ::= id$ 
 $m ::= id$ 
 $C ::= id$ 
 $v ::= r \mid \text{null} \mid \text{true} \mid \text{false}$ 
 $r ::= \text{number}$ 
 $id ::= \text{variable-not-otherwise-mentioned}$ 

```

Figure 3. The Javalite surface syntax.

```

 $\phi ::= (\phi) \mid \phi \bowtie \phi \mid \neg \phi \mid r = r \mid r \neq r$ 
 $l ::= \text{number}$ 
 $\eta ::= (mt (\eta [x \rightarrow v]))$ 
 $s ::= (\mu L R \phi_g \eta e k)$ 
 $k ::= \text{end}$ 
    |  $(* \$ f \rightarrow k)$ 
    |  $(x \$ f := * \rightarrow k)$ 
    |  $(* = e \rightarrow k)$ 
    |  $(v = * \rightarrow k)$ 
    |  $(\text{if } * e \text{ else } e \rightarrow k)$ 
    |  $(\text{var } T x := * \text{ in } e \rightarrow k)$ 
    |  $(* @ m e \rightarrow k)$ 
    |  $(v @ m * \rightarrow k)$ 
    |  $(x := * \rightarrow k)$ 
    |  $(\text{begin } * (e \dots) \rightarrow k)$ 
    |  $(\text{pop } \eta k)$ 

```

Figure 4. The machine syntax for Javalite with $\bowtie \in \{\wedge, \vee, \rightarrow\}$.

- $L(r)$ maps references to a set of location-constraint pairs in the store.

A reference is a node that gathers the possible store locations for an object during symbolic execution. Each store location is guarded by a constraint that determines the aliasing in the heap. Intuitively, the reference is a level of indirection between a variable and the store, and the reference is used to group a set of possible store locations each predicated on the possible aliasing in the associated constraint. For a variable (or field) to access any particular store location associated with its reference, the corresponding constraint must be satisfied.

Locations are boxes in the graphical representation and indicated with the letter l in the math. References are circles in the graphical representation and indicated with the letter r in the math. The special symbol \perp is an uninitialized reference. Edges from locations are labeled with field names f . Edges from the references are labeled with constraints $\phi \in \Phi$ (it is assumed that Φ is a power

set over individual constraints and ϕ is a set of constraints for the edge).

The function $\mathbb{V}\mathbb{S}(L, R, \phi_g, r, f)$ constructs the value-set given a heap, reference, and desired field:

$$\mathbb{V}\mathbb{S}(L, R, \phi_g, r, f) = \{(l' \phi \wedge \phi') \mid \exists l ((l \phi) \in L(r) \wedge \exists r' \in R(l, f) ((l' \phi') \in L(r') \wedge \mathbb{S}(\phi \wedge \phi' \wedge \phi_g)))\}$$

where $\mathbb{S}(\phi)$ returns true if ϕ is satisfiable.

The strengthen function $\mathbb{S}\mathbb{T}(L, r, \phi')$ strengthens every constraint from the reference r with ϕ' and keeps only location-constraint pairs that are satisfiable after this strengthening:

$$\mathbb{S}\mathbb{T}(L, r, \phi') = \{(l \phi \wedge \phi') \mid (l \phi) \in L(r) \wedge \mathbb{S}(\phi \wedge \phi')\}$$

4. Proofs

4.1 Definitions

Definition 1. A *state transition function* Φ is a mapping $\Phi : s \mapsto s$, which takes one machine state and transforms it into another machine state.

Definition 2. Given a sequence of states

$$\Sigma_n = s_0, s_1, \dots, s_n$$

where

$$s_i = (\mu_i L_i R_i \phi_i \eta_i e_i k_i)$$

the *control flow sequence* of Σ_n is the defined as the sequence of tuples

$$\sigma_n = \mathbb{C}\mathbb{F}(\Sigma_n) = (\eta_0, e_0, k_0), (\eta_1, e_1, k_1), \dots, (\eta_n, e_n, k_n)$$

Definition 3. A *feasible state sequence* is defined as a sequence of states resulting from repeated application of the state transition relation:

$$\Sigma_n = s_0, \Phi(s_0), \dots, \Phi_{n-1}(s_0)$$

Definition 4. Given a state transition function Φ , an initial state s_0 and a control flow sequence σ_n , the *feasible state set* $\zeta = \mathbb{F}\mathbb{S}(\Phi, s_0, \sigma_n)$ is defined as

$$\zeta = \{\forall s | \sigma_n = \mathbb{C}\mathbb{F}(\Sigma_n) \wedge s = \max_s(\Sigma) \wedge s = \Phi_{n-1}(s_0)\}$$

Definition 5. A symbolic system state represents a set of lazy system states. To denote the representation relationship, we use the expression $L \sqsubset S$ to say that lazy system state $L : (\mu_L L_L R_L \phi_L \eta_L e_L k_L)$ is represented by symbolic state $S : (\mu_S L_S R_S \phi_S \eta_S e_S k_S)$. The *representation relation* is defined as follows: $L \sqsubset S$ if and only if

$$\eta_L = \eta_S, e_L = e_S, k_L = k_S$$

and there exists functions $g : r \mapsto r$ and $h : l \mapsto l$ such that for any reference $r \in L$, location $l \in L$, and field f ,

$$l = L_L(r) \leftrightarrow h(l) \in L_S(g(r))$$

and

$$r = R_L(l, f) \leftrightarrow g(r) = R_S(h(l), f)$$

and the subheap of state S formed by selecting the image of g and h is feasible.

Definition 6. A symbolic state S is *exact* if and only if it represents the set of all feasible lazy states on the current execution path, and represents no infeasible state.

$$\forall (R_S^i L_S^i) \in \text{Heaps}(S_s) (\exists (R_L S_L) ((R_L S_L) = (R_S^i L_S^i)))$$

VARIABLE LOOKUP $(LR \phi_g \eta x k) \rightarrow$ $(LR \phi_g \eta \eta(x) k)$	NEW $r = \text{fresh}_r()$ $l = \text{fresh}_l(C)$ $R' = R[\forall f \in \text{fields}(C) \ (lf) \mapsto \text{fresh}_r())]$ $\rho = \{r' \mid \exists f \in \text{fields}(C) \ (r' = R'(l, f))\}$ $L' = L[r \mapsto \{(\phi_T l)\}][\forall r' \in \rho \ (r' \mapsto (\phi_T l_{\text{null}}))]$ <hr/> $(LR \phi_g \eta (\text{new } C) k) \rightarrow$ $(L' R' \phi_g \eta r k)$	FIELD ACCESS(EVAL) $(LR \phi_g \eta (e \$f) k) \rightarrow$ $(LR \phi_g \eta e (* \$f \rightarrow k))$
FIELD WRITE (EVAL) $(LR \phi_g \eta (x \$f := e) k) \rightarrow$ $(LR \phi_g \eta e (x \$f := * \rightarrow k))$	EQUALS (L-OPERAND EVAL) $(LR \phi_g \eta (e_0 = e) k) \rightarrow$ $(LR \phi_g \eta e_0 (* = e \rightarrow k))$	EQUALS (R-OPERAND EVAL) $(LR \phi_g \eta v (* = e \rightarrow k)) \rightarrow$ $(LR \phi_g \eta e (v = * \rightarrow k))$
EQUALS (BOOL) $v_0 \in \{\text{true}, \text{false}\} \quad v_1 \in \{\text{true}, \text{false}\}$ $v_r = \text{eq?}(v_0, v_1)$ <hr/> $(LR \phi_g \eta v_0 (v_1 = * \rightarrow k)) \rightarrow$ $(LR \phi_g \eta v_r k)$	IF-THEN-ELSE (EVAL) $(LR \phi_g \eta (\text{if } e_0 e_1 \text{ else } e_2) k) \rightarrow$ $(LR \phi_g \eta e_0 (\text{if } * e_1 \text{ else } e_2) \rightarrow k)$	IF-THEN-ELSE (TRUE) $(LR \phi_g \eta \text{true} (\text{if } * e_1 \text{ else } e_2) \rightarrow k) \rightarrow$ $(LR \phi_g \eta e_1 k)$
IF-THEN-ELSE (FALSE) $(LR \phi_g \eta \text{false} (\text{if } * e_1 \text{ else } e_2) \rightarrow k) \rightarrow$ $(LR \phi_g \eta e_2 k)$	VARIABLE DECLARATION (EVAL) $(LR \phi_g \eta (\text{var } T x := e_0 \text{ in } e_1) k) \rightarrow$ $(LR \phi_g \eta e_0 (\text{var } T x := * \text{ in } e_1 \rightarrow k))$	VARIABLE DECLARATION $(LR \phi_g \eta v (\text{var } T x * := \text{in } e_1 \rightarrow k)) \rightarrow$ $(LR \phi_g \eta [x \mapsto v] e_1 (\text{pop } \eta k))$
METHOD INVOCATION (OBJECT EVAL) $(LR \phi_g \eta (e_0 @ m e_1) k) \rightarrow$ $(LR \phi_g \eta e_0 (* @ m e_1 \rightarrow k))$	METHOD INVOCATION (ARG EVAL) $(LR \phi_g \eta v_0 (* @ m e_1 \rightarrow k)) \rightarrow$ $(LR \phi_g \eta e_1 (v_0 @ m * \rightarrow k))$	METHOD INVOCATION $(T m [T x] e_m) = \text{lookup}(m)$ $\eta_m = \eta[\text{this} \mapsto v_0][x \mapsto v_1]$ <hr/> $(LR \phi_g \eta v_1 (v_0 @ m * \rightarrow k)) \rightarrow$ $(LR \phi_g \eta_m e_m (\text{pop } \eta k))$
VARIABLE ASSIGNMENT (EVAL) $(LR \phi_g \eta (x := e) k) \rightarrow$ $(LR \phi_g \eta e (x := * \rightarrow k))$	VARIABLE ASSIGNMENT $(LR \phi_g \eta v (x := * \rightarrow k)) \rightarrow$ $(LR \phi_g \eta [x \mapsto v] v k)$	BEGIN (NO ARGS) $(LR \phi_g \eta (\text{begin}) k) \rightarrow$ $(LR \phi_g \eta k)$
BEGIN (ARG0 EVAL) $(LR \phi_g \eta (\text{begin } e_0 e_1 \dots) k) \rightarrow$ $(LR \phi_g \eta e_0 (\text{begin } * (e_1 \dots) \rightarrow k))$	BEGIN (ARGI EVAL) $(LR \phi_g \eta v (\text{begin } * (e_i e_{i+1} \dots) \rightarrow k)) \rightarrow$ $(LR \phi_g \eta e_i (\text{begin } * (e_{i+1} \dots) \rightarrow k))$	BEGIN (ARGN EVAL) $(LR \phi_g \eta v (\text{begin } * (e_n) \rightarrow k)) \rightarrow$ $(LR \phi_g \eta e_n (\text{begin } * () \rightarrow k))$
BEGIN $(LR \phi_g \eta v (\text{begin } * () \rightarrow k)) \rightarrow$ $(LR \phi_g \eta v k)$	NULL $(LR \phi_g \eta \text{null } k) \rightarrow$ $(LR \phi_g \eta r_{\text{null}} k)$	POP $(LR \phi_g \eta v (\text{pop } \eta_0 k)) \rightarrow$ $(LR \phi_g \eta_0 v k)$

Figure 5. Javalite rewrite rules that are common to generalized symbolic execution and precise heap summaries.

4.2 Theorems

Lemma 1. *If symbolic state \mathcal{S} is exact prior to executing the Field Access rule, then the state \mathcal{S}' is also exact.*

Proof. Suppose we have some symbolic state $\mathcal{S} = (L_{\mathcal{S}} R_{\mathcal{S}} \phi_g \eta r (* \$f \rightarrow k))$, and that all relevant fields are initialized. Take an arbitrary lazy state $\mathcal{L} \sqsubset \mathcal{S}$. Since \mathcal{S} is exact, $\mathcal{L} = (L_{\mathcal{L}} R_{\mathcal{L}} \phi_L \eta r (* \$f \rightarrow k))$. If we apply the lazy field access rule to \mathcal{L} , we achieve state $\mathcal{L}' = (L_{\mathcal{L}}[r' \mapsto (\phi' l')] R_{\mathcal{L}} \phi_L \eta r' k)$. Likewise, by applying the symbolic field access rule to \mathcal{S} , we obtain state $\mathcal{S}' = (L_{\mathcal{S}}[r' \mapsto \mathbb{V}\mathbb{S}(L_{\mathcal{S}}, R_{\mathcal{S}}, r, f, \phi_g)] R_{\mathcal{S}} \phi_g \eta r' k)$.

We now show that state \mathcal{S}' represents state \mathcal{L}' . Since η , e , and k are identical between \mathcal{S}' and \mathcal{L}' , the first condition is met by default. Constructing the reference and location mappings is only slightly less trivial:

$$g' = g$$

$$h' = h[r' \mapsto r']$$

It only remains to show that the image of \mathcal{L}' in \mathcal{S}' is feasible, WHICH WILL BE EASY ONCE WE DEFINE WHAT THAT MEANS.

Now, we show that no infeasible lazy states are allowed by \mathcal{S}' .
NEED TO FILL IN THIS BIT, TOO

$$\mathbb{S}(\bigwedge_{(\phi_l) \in L_s^i} \phi_l) \wedge \forall l \in L_L (\exists l \in L_s^i) \quad \square$$

5. Related Work

The related work goes here.

Acknowledgments

Acknowledgments, if needed.

References

- [1] S. Anand, C. S. Pasareanu, and W. Visser. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer (STTT)*, 11:53–67, January 2009.
- [2] J. Backes, S. Person, N. Rungta, and O. Tkachuk. Regression verification using impact summaries. In *Model Checking Software*, pages 99–116. Springer, 2013.
- [3] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, 1976.

<p>INITIALIZE (NULL)</p> $\Lambda = \{l \mid \exists \phi ((\phi l) \in L(r) \wedge R(l, f) = \perp)\} \quad \Lambda \neq \emptyset$ $r' = \text{fresh}_r() \quad \theta_{\text{null}} = \{(\phi_T l_{\text{null}})\} \quad l_x = \min_l(\Lambda)$ $\phi'_g = (\phi_g \wedge r' = r_{\text{null}})$ <hr/> $(LR \phi_g r f) \rightarrow_I (L[r' \mapsto \theta_{\text{null}}] R[(l_x, f) \mapsto r'] \phi'_g r f)$	<p>INITIALIZE (NEW)</p> $\Lambda = \{l \mid \exists \phi ((\phi l) \in L(r) \wedge R(l, f) = \perp)\} \quad \Lambda \neq \emptyset$ $C = \text{type}(f) \quad r_f = \text{init}_r() \quad l_f = \text{init}_l(C)$ $R' = R[\forall f \in \text{fields}(C) ((l_f f) \mapsto \perp)]$ $\rho = \{(r_a l_a) \mid \text{isInit}(r_a) \wedge \text{type}(l_a) = C \wedge \exists \phi_a ((\phi_a l_a) \in L(r_a))\}$ $\theta_{\text{new}} = \{(\phi_T l_f)\} \quad l_x = \min_l(\Lambda)$ $\phi'_g = (\phi_g \wedge r_f \neq r_{\text{null}} \wedge (\wedge_{(r_a l_a) \in \rho} r_f \neq r_a))$ <hr/> $(LR \phi_g r f) \rightarrow_I (L[r_f \mapsto \theta_{\text{new}}] R'[(l_x, f) \mapsto r_f] \phi'_g r f)$
<p>INITIALIZE (ALIAS)</p> $\Lambda = \{l \mid \exists \phi ((\phi l) \in L(r) \wedge R(l, f) = \perp)\} \quad \Lambda \neq \emptyset$ $C = \text{type}(f) \quad r' = \text{fresh}_r()$ $\rho = \{(r_a l_a) \mid \text{isInit}(r_a) \wedge \text{type}(l_a) = C \wedge \exists \phi_a ((\phi_a l_a) \in L(r_a))\}$ $(r_a l_a) \in \rho \quad \theta_{\text{alias}} = \{(\phi_T l_a)\} \quad l_x = \min_l(\Lambda)$ $\phi'_g = (\phi_g \wedge r' \neq r_{\text{null}} \wedge r' = r_a \wedge (\wedge_{(r'_a l_a) \in \rho} (r'_a \neq r_a) \wedge r' \neq r'_a))$ <hr/> $(LR \phi_g r f) \rightarrow_I (L[r' \mapsto \theta_{\text{alias}}] R[(l_x, f) \mapsto r'] \phi'_g r f)$	<p>INITIALIZE (END)</p> $\Lambda = \{l \mid \exists \phi ((\phi l) \in L(r) \wedge R(l, f) = \perp)\} \quad \Lambda = \emptyset$ <hr/> $(LR \phi_g r f) \rightarrow_I (LR \phi_g r f)$

Figure 6. The initialization machine, $s ::= (LR \phi_g r f)$, with $s \rightarrow_I^* s'$ indicating stepping the machine until the state does not change.

<p>FIELD ACCESS</p> $\{(\phi l)\} = L(r) \quad l \neq l_{\text{null}}$ $(LR \phi_g r f) \rightarrow_I^* (L' R' \phi'_g r f)$ $\{(\phi' l')\} = L'(R'(l, f)) \quad r' = \text{fresh}_r()$ <hr/> $(LR \phi_g \eta r (* \$f \rightarrow k)) \rightarrow$ $(L'[r' \mapsto (\phi' l')] R' \phi'_g \eta r' k)$	<p>FIELD WRITE</p> $r_x = \eta(x) \quad \{(\phi l)\} = L(r_x) \quad l \neq l_{\text{null}}$ $(LR \phi_g r_x f) \rightarrow_I^* (L' R' \phi'_g r_x f)$ $r' = \text{fresh}_r() \quad \theta = L'(r)$ <hr/> $(LR \phi_g \eta r (x \$f := * \rightarrow k)) \rightarrow$ $(L'[r' \mapsto \theta] R'[(l, f) \mapsto r'] \phi_g \eta r' k)$
<p>EQUALS (REFERENCE-TRUE)</p> $L(r_0) = L(r_1) \quad \phi'_g = (\phi_g \wedge r_0 = r_1)$ <hr/> $(LR \phi_g \eta r_0 (r_1 = * \rightarrow k)) \rightarrow$ $(LR \phi'_g \eta \text{true } k)$	<p>EQUALS (REFERENCE-FALSE)</p> $L(r_0) \neq L(r_1) \quad \phi'_g = (\phi_g \wedge r_0 \neq r_1)$ <hr/> $(LR \phi_g \eta r_0 (r_1 = * \rightarrow k)) \rightarrow$ $(LR \phi'_g \eta \text{false } k)$

Figure 7. Generalized symbolic execution with lazy initialization.

<p>SUMMARIZE</p> $\Lambda = \{l \mid \exists \phi ((\phi l) \in L(r) \wedge l \neq l_{\text{null}} \wedge R(l, f) = \perp)\} \quad \Lambda \neq \emptyset$ $C = \text{type}(f) \quad r_f = \text{init}_r() \quad l_f = \text{init}_l(C)$ $R' = R[\forall f \in \text{fields}(C) ((l_f f) \mapsto \perp)]$ $\rho = \{(r_a \phi_a l_a) \mid \text{isInit}(r_a) \wedge (\phi_a l_a) \in L(r_a) \wedge \text{type}(l_a) = C\}$ $\theta_{\text{null}} = \{(\phi l_{\text{null}}) \mid \phi = (r_f = r_{\text{null}})\}$ $\theta_{\text{new}} = \{(\phi l_f) \mid \phi = (r_f \neq r_{\text{null}} \wedge (\wedge_{(r'_a \phi'_a l'_a) \in \rho} r_f \neq r'_a))\}$ $\theta_{\text{alias}} = \{(\phi l_a) \mid \exists r_a (\exists \phi_a ((r_a \phi_a l_a) \in \rho \wedge \phi = (\phi_a \wedge r_f \neq r_{\text{null}} \wedge r_f = r_a \wedge (\wedge_{(r'_a \phi'_a l'_a) \in \rho} (r'_a \neq r_a) \wedge r_f \neq r'_a))))\}$ $\theta = \theta_{\text{alias}} \cup \theta_{\text{new}} \cup \theta_{\text{null}} \quad l_x = \min_l(\Lambda)$ <hr/> $(LR r f) \rightarrow_S (L[r_f \mapsto \theta] R'[(l_x, f) \mapsto r_f] r f)$	<p>SUMMARIZE (END)</p> $\Lambda = \{l \mid \exists \phi ((\phi l) \in L(r) \wedge R(l, f) = \perp)\} \quad \Lambda = \emptyset$ <hr/> $(LR r f) \rightarrow_I (LR r f)$
--	---

Figure 8. The summary machine, $s ::= (LR r f)$, with $s \rightarrow_I^* s'$ indicating stepping the machine until the state does not change.

- [4] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *ICSE*, pages 281–290, 2008.
- [5] X. Deng, J. Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 157–166, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2579-2.
- [6] X. Deng, Robby, and J. Hatcliff. Towards a case-optimal symbolic execution algorithm for analyzing strong properties of object-oriented programs. In *SEFM '07: Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods*, pages 273–282, Washington, DC, USA, 2007. IEEE Computer Society.
- [7] P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54, 2007.

$$\begin{array}{c}
\text{FIELD ACCESS} \\
\frac{\forall(\phi l) \in L(r) \ (l = l_{\text{null}} \rightarrow \neg \mathbb{S}(\phi \wedge \phi_g)) \quad (LR r f) \rightarrow_S^* (L' R' r f) \quad r' = \text{fresh}_r()}{(LR \phi_g \eta r (* \$f \rightarrow k)) \rightarrow (L'[r' \mapsto \mathbb{V}\mathbb{S}(L', R', r, f, \phi_g)] R' \phi_g \eta r' k)}
\end{array}$$

$$\begin{array}{c}
\text{FIELD WRITE} \\
\frac{r_x = \eta(x) \quad \forall(\phi l) \in L(r_x) \ (l = l_{\text{null}} \rightarrow \neg \mathbb{S}(\phi \wedge \phi_g)) \quad (LR r_x f) \rightarrow_S^* (L' R' r_x f) \quad \Psi_x = \{(r_{\text{cur}} \phi l) \mid (\phi l) \in L'(r_x) \wedge r_{\text{cur}} \in R(l, f)\} \quad X = \{(r_{\text{cur}} \theta l) \mid \exists \phi ((r_{\text{cur}} \phi l) \in \Psi_x \wedge \theta = \mathbb{ST}(L', r, \phi) \cup \mathbb{ST}(L', r_{\text{cur}}, \neg \phi))\} \quad R'' = R'[\forall(r_{\text{cur}} \theta l) \in X ((l f) \mapsto \text{fresh}_r())] \quad L'' = L'[\forall(r_{\text{cur}} \theta l) \in X (\exists r_{\text{targ}} (r_{\text{targ}} = R'(l, f) \wedge (r_{\text{targ}} \mapsto \theta)))]}{(LR \phi_g \eta r (x \$f := * \rightarrow k)) \rightarrow (L'' R'' \phi_g \eta k)}
\end{array}$$

$$\begin{array}{c}
\text{EQUALS (REFERENCES-TRUE)} \\
\frac{\theta_\alpha = \{(\phi_0 \wedge \phi_1) \mid \exists l ((\phi_0 l) \in L(r_0) \wedge (\phi_1 l) \in L(r_1))\} \quad \theta_0 = \{\phi_0 \mid \exists l_0 ((\phi_0 l_0) \in L(r_0) \wedge \forall(\phi_1 l_1) \in L(r_1) \ (l_0 \neq l_1))\} \quad \theta_1 = \{\phi_1 \mid \exists l_1 ((\phi_1 l_1) \in L(r_1) \wedge \forall(\phi_0 l_0) \in L(r_0) \ (l_0 \neq l_1))\} \quad \phi'_g = \phi_g \wedge (\bigvee_{\phi_\alpha \in \theta_\alpha} \phi_\alpha) \wedge (\bigwedge_{\phi_0 \in \theta_0} \neg \phi_0) \wedge (\bigwedge_{\phi_1 \in \theta_1} \neg \phi_1) \quad \mathbb{S}(\phi'_g)}{(LR \phi_g \eta r_0 (r_1 = * \rightarrow k)) \rightarrow (LR \phi'_g \eta \text{true } k)}
\end{array}$$

$$\begin{array}{c}
\text{EQUALS (REFERENCES-FALSE)} \\
\frac{\theta_\alpha = \{(\phi_0 \rightarrow \neg \phi_1) \mid \exists l ((\phi_0 l) \in L(r_0) \wedge (\phi_1 l) \in L(r_1))\} \quad \theta_0 = \{\phi_0 \mid \exists l_0 ((\phi_0 l_0) \in L(r_0) \wedge \forall(\phi_1 l_1) \in L(r_1) \ (l_0 \neq l_1))\} \quad \theta_1 = \{\phi_1 \mid \exists l_1 ((\phi_1 l_1) \in L(r_1) \wedge \forall(\phi_0 l_0) \in L(r_0) \ (l_0 \neq l_1))\} \quad \phi'_g = \phi_g \wedge (\bigwedge_{\phi_\alpha \in \theta_\alpha} \phi_\alpha) \vee ((\bigvee_{\phi_0 \in \theta_0} \phi_0) \vee (\bigvee_{\phi_1 \in \theta_1} \phi_1)) \quad \mathbb{S}(\phi'_g)}{(LR \phi_g \eta r_0 (r_1 = * \rightarrow k)) \rightarrow (LR \phi'_g \eta \text{false } k)}
\end{array}$$

Figure 9. Precise symbolic heap summaries from symbolic execution.

- [8] P. Godefroid, S. K. Lahiri, and C. Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *SAS*, pages 112–128, 2011.
- [9] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.
- [10] S. Khurshid, I. García, and Y. L. Suen. Repairing structurally complex data. In *SPIN*, pages 123–138, 2005.
- [11] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. ISSN 0001-0782. .
- [12] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *SAS*, pages 95–111, 2011.
- [13] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *FSE*, pages 226–237, 2008.
- [14] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI*, pages 504–515, 2011.
- [15] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *CAV*, pages 669–685, 2011.
- [16] S. O. Wesonga. Javalite - an operational semantics for modeling Java programs. Master’s thesis, Brigham Young University, Provo UT, 2012.
- [17] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *ISSTA*, pages 144–154, 2012.
- [18] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Feedback-driven dynamic invariant discovery. In *ISSTA*, pages 362–372, 2014.