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 execution 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, .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 onstruct 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:

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/nnnnnnn.nnnnnnn

- We present a novel lazy symbolic execution technique for analyzing heap manipulating programs that constructs a fully symbolic representatino of the heap on-the-fly durig 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 maninpulating 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 concreate 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. Pseudo-code

Figure 3 defines the surface syntax for the Javalite language [16]. The Figure 4 is the machine syntax. The semantics of Javalite is syntax based and defined as rewrites on a string. The semantics use a CEKS machine 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. This paper only defines salient features of the language and machine relevant to understanding the new algorithm.

This paper uses a standard definition of constraints $\phi \in \Phi$ assuming all the usual relational operators and connectives. 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.

```
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

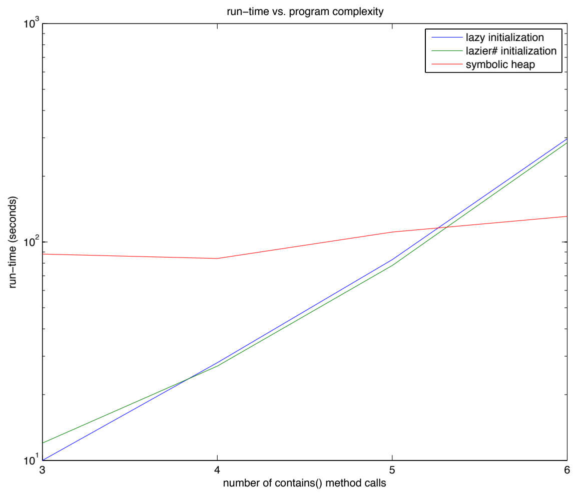

Figure 2. Time versus complexity for the linked list example

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

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

Figure 3. The Javalite surface syntax.

$$
\begin{array}{rcl}
\phi & ::= & \textit{constraint} \\
l & ::= & \textbf{number} \\
\eta & ::= & (mt\ (\eta\ [x \to v])) \\
s & ::= & (\mu\ L\ R\ \phi_g\ \eta\ e\ k) \\
k & ::= & \textbf{end} \\
 & \mid & (*\ \$\ f \to k) \\
 & \mid & (x\ \$\ f\ \textbf{:=}\ * \to k) \\
 & \mid & (*\ \textbf{=}\ e \to k) \\
 & \mid & (v\ \textbf{=}\ * \to k) \\
 & \mid & (\textbf{if}\ *\ e\ \textbf{else}\ e \to k\ ) \\
 & \mid & (\textbf{var}\ T\ x\ \textbf{:=}\ *\ \textbf{in}\ e \to k\ ) \\
 & \mid & (*\ @\ m\ e \to k\ ) \\
 & \mid & (v\ @\ m\ * \to k\ ) \\
 & \mid & (x\ \textbf{:=}\ * \to k) \\
 & \mid & (\textbf{begin}\ *\ (e\ ...) \to k\ ) \\
 & \mid & (\textbf{pop}\ \eta\ k)
\end{array}
$$

Figure 4. The machine syntax for Javalite.

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

$$
\begin{array}{rcl}
\mathbb{VS}(L, R, \phi_g, r, f) & = & \{(l'\ \phi \wedge \phi') \mid \\
 & & \quad \exists l\ ((l\ \phi) \in L(r) \wedge \\
 & & \quad\quad \exists r' \in R(l, f)(\\
 & & \quad\quad\quad (l'\ \phi') \in L(r') \wedge \\
 & & \quad\quad\quad \mathbb{S}(\phi \wedge \phi' \wedge \phi_g)))\}
\end{array}
$$

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

The strengthen function $\mathbb{ST}(L, r, \phi')$ strengthens every constraint from the reference r with ϕ' and keeps only location-constraint pairs that are satisfiable after this strengthening:

$$
\mathbb{ST}(L, r, \phi') \quad = \quad \{(l\ \phi \wedge \phi') \mid (l\ \phi) \in L(r) \wedge \mathbb{S}(\phi \wedge \phi')\}
$$

4. Proofs

4.1 Definitions

Definition A symbolic system state represents a set of lazy system states. To denote the representation relationship, we use the expression $\mathcal{L} \sqsubset \mathcal{S}$ to say that lazy system state $\mathcal{L}$: $(\mu\ L_{\mathcal{L}}\ R_{\mathcal{L}}\ \phi_{\mathcal{L}}\ \eta\ e\ k)$ is represented by symbolic state $\mathcal{S}$: $(\mu\ L_{\mathcal{S}}\ R_{\mathcal{S}}\ \phi_{\mathcal{S}}\ \eta\ e\ k)$. The representation relationship is defined as follows: $\mathcal{L} \sqsubseteq \mathcal{S}$ if and only if there exists functions $g : r \mapsto r$ and $h : l \mapsto l$ such that for any reference $r \in \mathcal{L}$, location $l \in \mathcal{L}$, and field f,

$$
l = L_{\mathcal{L}}(r) \leftrightarrow h(l) \in L_{\mathcal{S}}(g(r))
$$

and

$$
r = R_{\mathcal{L}}(l, f) \leftrightarrow g(r) = R_{\mathcal{S}}(h(l), f)
$$

4.2 Theorems

The following theorem demonstrates the use of the theorem environment.

THEOREM 1. *Let f be continuous on $[a, b]$. If G is an antiderivative for f on $[a, b]$, then*

$$
\int_a^b f(t)dt = G(b) - G(a).
$$

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.

[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-oreinted 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.

[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.

Variable lookup

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\,x\,k) \to \\ (L\,R\,\phi_g\,\eta\,\eta(x)\ k)\end{array}$$

New

$$\frac{\begin{array}{l} r = \text{fresh}_r() \qquad l = \text{fresh}_l(C) \\ R' = R[\forall f \in \mathit{fields}(\mathrm{C})\ ((l\,f) \mapsto \text{fresh}_r())] \\ \rho = \{r' \mid \exists f \in \mathit{fields}(\mathrm{C})\ (r' = R'(l,f))\} \\ L' = L[r \mapsto \{(\phi_T\ l)\}][\forall r' \in \rho\ (r' \mapsto (\phi_T\ l_{null}))] \end{array}}{\begin{array}{l}(L\,R\,\phi_g\,\eta\ (\mathbf{new}\ C)\ k) \to \\ (L'\,R'\,\phi_g\,\eta\,r\,k)\end{array}}$$

Field Access(eval)

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\ (e\ \$f)\ k) \to \\ (L\,R\,\phi_g\,\eta\,e\ (*\ \$f \to k))\end{array}$$

Field Write (eval)

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\ (x\ \$f := e)\ k) \to \\ (L\,R\,\phi_g\,\eta\,e\ (x\ \$f := * \ \to\ k))\end{array}$$

Equals (l-operand eval)

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\ (e_0 = e)\ k) \to \\ (L\,R\,\phi_g\,\eta\,e_0\ (* = e \to k))\end{array}$$

Equals (r-operand eval)

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\,v\ (* = e \to k)) \to \\ (L\,R\,\phi_g\,\eta\,e\ (v = * \to k))\end{array}$$

Equals (bool)

$$\frac{\begin{array}{l} v_0 \in \{\mathbf{true}, \mathbf{false}\} \qquad v_1 \in \{\mathbf{true}, \mathbf{false}\} \\ v_r = \text{eq?}(v_0, v_1)\end{array}}{\begin{array}{l}(L\,R\,\phi_g\,\eta\,v_0\ (v_1 = * \to k)) \to \\ (L\,R\,\phi_g\,\eta\,v_r\ k)\end{array}}$$

If-then-else (eval)

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\ (\mathbf{if}\ e_0\ e_1\ \mathbf{else}\ e_2)\ k) \to \\ (L\,R\,\phi_g\,\eta\,e_0\ (\mathbf{if}\ *\ e_1\ \mathbf{else}\ e_2) \to k)\end{array}$$

If-then-else (true)

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\ \mathbf{true}\ (\mathbf{if}\ *\ e_1\ \mathbf{else}\ e_2) \to k) \to \\ (L\,R\,\phi_g\,\eta\,e_1\ k)\end{array}$$

If-then-else (false)

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\ \mathbf{false}\ (\mathbf{if}\ *\ e_1\ \mathbf{else}\ e_2) \to k) \to \\ (L\,R\,\phi_g\,\eta\,e_2\ k)\end{array}$$

Variable Declaration (eval)

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\ (\mathbf{var}\ T\,x := e_0\ \mathbf{in}\ e_1)\ k) \to \\ (L\,R\,\phi_g\,\eta\,e_0\ (\mathbf{var}\ T\,x := *\ \mathbf{in}\ e_1 \to k))\end{array}$$

Variable Declaration

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\,v\ (\mathbf{var}\ T\,x\ * := \mathbf{in}\ e_1 \to k)) \to \\ (L\,R\,\phi_g\,\eta[x \mapsto v]\ e_1\ (\mathbf{pop}\ \eta\ k))\end{array}$$

Method Invocation (object eval)

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\ (e_0\ @\ m\ e_1)\ k) \to \\ (L\,R\,\phi_g\,\eta\,e_0\ (*\ @\ m\ e_1\ \ \to k))\end{array}$$

Method Invocation (arg eval)

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\,v_0\ (*\ @\ m\ e_1\ \ \to k)) \to \\ (L\,R\,\phi_g\,\eta\,e_1\ (v_0\ @\ m\ *\ \ \to k))\end{array}$$

Method Invocation

$$\frac{\begin{array}{l}(T\,m\ [T\,x]\ \ e_m) = \text{lookup}(m) \\ \eta_m = \eta[\mathbf{this} \mapsto v_0][x \mapsto v_1]\end{array}}{\begin{array}{l}(L\,R\,\phi_g\,\eta\,v_1\ (v_0\ @\ m\ *\ \ \to k)) \to \\ (L\,R\,\phi_g\,\eta_m\,e_m\ (\mathbf{pop}\ \eta\ k))\end{array}}$$

Variable Assignment (eval)

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\ (x := e)\ k) \to \\ (L\,R\,\phi_g\,\eta\,e\ (x := * \to\ k))\end{array}$$

Variable Assignment

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\,v\ (x := * \to\ k)) \to \\ (L\,R\,\phi_g\,\eta[x \mapsto v]\ v\,k)\end{array}$$

Begin (no args)

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\ (\mathbf{begin})\ k) \to \\ (L\,R\,\phi_g\,\eta\,k)\end{array}$$

Begin (arg0 eval)

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\ (\mathbf{begin}\ e_0\ e_1\ \ldots)\ k) \to \\ (L\,R\,\phi_g\,\eta\,e_0\ (\mathbf{begin}\ *\ (e_1\ \ldots) \to k))\end{array}$$

Begin (argi eval)

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\,v\ (\mathbf{begin}\ *\ (e_i\ e_{i+1}\ \ldots) \to k)) \to \\ (L\,R\,\phi_g\,\eta\,e_i\ (\mathbf{begin}\ *\ (e_{i+1}\ \ldots) \to k))\end{array}$$

Begin (argN eval)

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\,v\ (\mathbf{begin}\ *\ (e_n) \to k)) \to \\ (L\,R\,\phi_g\,\eta\,e_n\ (\mathbf{begin}\ *\ () \to k))\end{array}$$

Begin

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\,v\ (\mathbf{begin}\ *\ () \to k)) \to \\ (L\,R\,\phi_g\,\eta\,v\,k)\end{array}$$

NULL

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\ \mathbf{null}\ k) \to \\ (L\,R\,\phi_g\,\eta\,r_{null}\ k)\end{array}$$

Pop

$$\begin{array}{l}(L\,R\,\phi_g\,\eta\,v\ (\mathbf{pop}\ \eta_0\ k)) \to \\ (L\,R\,\phi_g\,\eta_0\,v\,k)\end{array}$$

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

[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.

Initialize (null)

$$\frac{\begin{array}{l}\Lambda = \{l \mid \exists\phi\ ((\phi\ l) \in L(r) \wedge R(l,f) = \bot\ )\} \qquad \Lambda \neq \emptyset \\ r' = \text{fresh}_r() \qquad \theta_{null} = \{(\phi_T\ l_{null})\} \qquad l_x = \text{min}_l(\Lambda) \\ \phi'_g = (\phi_g \wedge r' = r_{null})\end{array}}{(L\,R\,\phi_g\,r\,f) \to_I (L[r' \mapsto \theta_{null}]\ R[(l_x,f) \mapsto r']\ \phi'_g\ r\,f)}$$

Initialize (new)

$$\frac{\begin{array}{l}\Lambda = \{l \mid \exists\phi\ ((\phi\ l) \in L(r) \wedge R(l,f) = \bot)\} \qquad \Lambda \neq \emptyset \\ \text{C} = \text{type}(f) \qquad r_f = \text{init}_r() \qquad l_f = \text{init}_l(\text{C}) \\ \rho = \{(r_a\ l_a) \mid \text{isInit}(r_a) \wedge \text{type}(l_a) = \text{C} \wedge \exists\phi_a\ ((\phi_a\ l_a) \in L(r_a))\} \\ \theta_{new} = \{(\phi_T\ l_f)\} \qquad l_x = \text{min}_l(\Lambda) \\ \phi'_g = (\phi_g \wedge r_f \neq r_{null} \wedge (\ \wedge_{(r_a\ l_a)\in\rho}\, r_f \neq r_a))\end{array}}{(L\,R\,\phi_g\,r\,f) \to_I (L[r_f \mapsto \theta_{new}]\ R[(l_x,f) \mapsto r_f]\ \phi'_g\ r\,f)}$$

Initialize (alias)

$$\frac{\begin{array}{l}\Lambda = \{l \mid \exists\phi\ ((\phi\ l) \in L(r) \wedge R(l,f) = \bot\ )\} \qquad \Lambda \neq \emptyset \\ \text{C} = \text{type}(f) \qquad r' = \text{fresh}_r() \\ \rho = \{(r_a\ l_a) \mid \text{isInit}(r_a) \wedge \text{type}(l_a) = \text{C} \wedge \exists\phi_a\ ((\phi_a\ l_a) \in L(r_a))\} \\ (r_a\ l_a) \in \rho \qquad \theta_{alias} = \{(\phi_T\ l_a)\} \qquad l_x = \text{min}_l(\Lambda) \\ \phi'_g = (\phi_g \wedge r' \neq r_{null} \wedge r' = r_a \wedge (\ \wedge_{(r'_a\ l_a)\in\rho\ (r'_a \neq r_a)}\, r' \neq r'_a))\end{array}}{(L\,R\,\phi_g\,r\,f) \to_I (L[r' \mapsto \theta_{alias}]\ R[(l_x,f) \mapsto r']\ \phi'_g\ r\,f)}$$

Figure 6. The initialization machine, $s ::= (L\,R\,\phi_g\,r\,f)$, for generalized symbolic execution with lazy initialization denoted by $\to_I$.

Field Access

$$\frac{\begin{array}{l}\{(\phi\ l)\} = L(r) \qquad l \neq l_{null} \\ (L\,R\,\phi_g\,r\,f) \to_I^* (L'\,R'\,\phi'_g\,r\,f) \\ \{(\phi'\ l')\} = L'(R'(l,f)) \qquad \text{fresh}_r() = r'\end{array}}{\begin{array}{l}(L\,R\,\phi_g\,\eta\,r\,(\textbf{*}\ \textbf{\$}f \to k)) \to \\ (L'[r' \mapsto (\phi'\ l')]\ R'\ \phi'_g\ \eta\ r'\ k)\end{array}}$$

Field Write

$$\frac{\begin{array}{l}r_x = \eta(x) \qquad \{(\phi\ l)\} = L(r_x) \qquad l \neq l_{null} \\ (L\,R\,\phi_g\,r_x\,f) \to_I^* (L'\,R'\,\phi'_g\,r_x\,f) \\ \text{fresh}_r() = r' \qquad \theta = L'(r)\end{array}}{\begin{array}{l}(L\,R\,\phi_g\,\eta\,r\,(x\ \textbf{\$}f \textbf{:=} \textbf{*}\ \to\ k)) \to \\ (L'[r' \mapsto \theta]\ R'[(l\,f) \mapsto r']\ \phi_g\ \eta\ r'\ k)\end{array}}$$

Equals (reference-true)

$$\frac{L(r_0) = L(r_1) \qquad \phi'_g = (\phi_g \wedge r_0 = r_1)}{\begin{array}{l}(L\,R\,\phi_g\,\eta\,r_0\ (r_1 \textbf{=} \textbf{*} \to k)) \to \\ (L\,R\,\phi'_g\ \eta\ \textbf{true}\ k)\end{array}}$$

Equals (reference-false)

$$\frac{L(r_0) \neq L(r_1) \qquad \phi'_g = (\phi_g \wedge r_0 \neq r_1)}{\begin{array}{l}(L\,R\,\phi_g\,\eta\,r_0\ (r_1 \textbf{=} \textbf{*} \to k)) \to \\ (L\,R\,\phi'_g\ \eta\ \textbf{false}\ k)\end{array}}$$

Figure 7. Generalized symbolic execution with lazy initialization.

Summarize

$$\frac{\begin{array}{l}\Lambda = \{l \mid \exists\phi\ ((\phi\ l) \in L(r) \wedge l \neq l_{null} \wedge R(l,f) = \bot\ )\} \qquad \Lambda \neq \emptyset \\ \text{C} = \text{type}(f) \qquad r_f = \text{init}_r() \qquad l_f = \text{init}_l(\text{C}) \\ \rho = \{(r_a\ \phi_a\ l_a) \mid \text{isInit}(r_a) \wedge (\phi_a\ l_a) \in L(r_a) \wedge \text{type}(l_a) = \text{C}\} \\ \theta_{alias} = \{(\phi\ l_a) \mid \exists r_a\ (\exists\phi_a\ ((r_a\ \phi_a\ l_a) \in \rho \wedge \phi = (\phi_a \wedge r_f \neq r_{null} \wedge r_f = r_a \wedge (\ \wedge_{(r'_a\ \phi'_a\ l'_a)\in\rho\ (r'_a \neq r_a)}\, r_f \neq r'_a))))\} \\ \theta_{new} = \{(\phi\ l_f) \mid \phi = (r_f \neq r_{null} \wedge (\ \wedge_{(r'_a,\ \phi'_a,\ l'_a)\in\rho}\, r_f \neq r'_a))\} \\ \theta_{null} = \{(\phi\ l_{null}) \mid \phi = (r_f = r_{null})\} \\ \theta = \theta_{alias} \cup \theta_{new} \cup \theta_{null} \qquad l_x = \text{min}_l(\Lambda)\end{array}}{(L\,R\,r\,f) \to_S (L[r_f \mapsto \theta]\ R[(l_x,f) \mapsto r_f]\ r\,f)}$$

Figure 8. The summary machine, $s ::= (L\,R\,r\,f)$, for precise heap summaries from symbolic execution denoted $\to_S$.

Field Access

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

Field Write

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

Equals (references-true)

$$\frac{\begin{array}{l}\theta_\alpha = \{\phi_0 \wedge \phi_1 \mid \exists l\ ((\phi_0\ l) \in L(r_0) \wedge (\phi_1\ l) \in L(r_1))\}\\ \theta_0 = \{\phi_0 \mid \exists l_0\ ((\phi_0\ l_0) \in L(r_0) \wedge \forall(\phi_1\ l_1) \in L(r_1)\ (l_0 \neq l_1))\}\\ \theta_1 = \{\phi_1 \mid \exists l_1\ ((\phi_1\ l_1) \in L(r_1) \wedge \forall(\phi_0\ l_0) \in L(r_0)\ (l_0 \neq l_1))\}\\ \phi'_g = \phi_g \wedge (\vee_{\phi_\alpha \in \theta_\alpha}\ \phi_\alpha) \wedge (\wedge_{\phi_0 \in \theta_0}\ \neg\phi_0) \wedge (\wedge_{\phi_1 \in \theta_1}\ \neg\phi_1)\\ \mathbb{S}(\phi'_g)\end{array}}{(L\ R\ \phi_g\ \eta\ r_0\ (r_1 \mathbf{=} \mathbf{*} \rightarrow k)) \rightarrow (L\ R\ \phi'_g\ \eta\ \mathbf{true}\ k)}$$

Equals (references-false)

$$\frac{\begin{array}{l}\theta_\alpha = \{\phi_0 \rightarrow \neg\phi_1 \mid \exists l\ ((\phi_0\ l) \in L(r_0) \wedge (\phi_1\ l) \in L(r_1))\}\\ \theta_0 = \{\phi_0 \mid \exists l_0\ ((\phi_0\ l_0) \in L(r_0) \wedge \forall(\phi_1\ l_1) \in L(r_1)\ (l_0 \neq l_1))\}\\ \theta_1 = \{\phi_1 \mid \exists l_1\ ((\phi_1\ l_1) \in L(r_1) \wedge \forall(\phi_0\ l_0) \in L(r_0)\ (l_0 \neq l_1))\}\\ \phi'_g = \phi_g \wedge (\wedge_{\phi_\alpha \in \theta_\alpha}\ \phi_\alpha) \vee ((\vee_{\phi_0 \in \theta_0}\ \phi_0) \vee (\vee_{\phi_1 \in \theta_1}\ \phi_1))\\ \mathbb{S}(\phi'_g)\end{array}}{(L\ R\ \phi_g\ \eta\ r_0\ (r_1 \mathbf{=} \mathbf{*} \rightarrow k)) \rightarrow (L\ R\ \phi'_g\ \eta\ \mathbf{false}\ k)}$$

Figure 9. Precise symbolic heap summaries from symbolic execution.