Exact 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. 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 enumerates 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

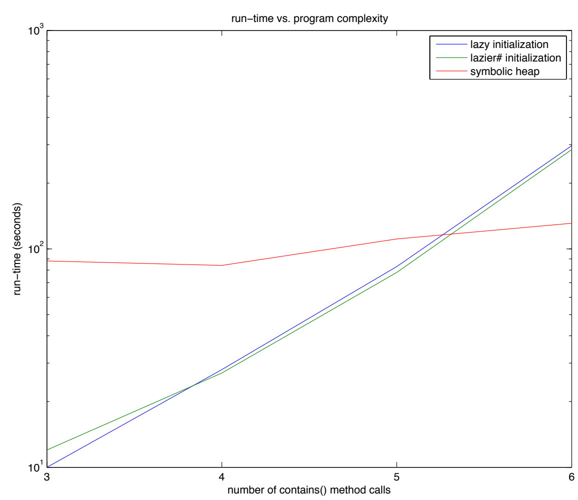

Figure 2. Time versus complexity for the linked list example

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

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

Figure 3. The Javalite surface syntax.

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

Figure 4. The machine syntax for Javalite 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 $\mathbb{VS}(L, R, \phi_g, r, f)$ 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 1. *A* ***state transition function*** $\to_\Phi$ *is a mapping* $\to_\Phi$: $s \mapsto s$*, which takes one machine state and transforms it into another machine state.*

Definition 2. *A* ***state sequence*** *is as a sequence of states denoted as* $\Pi_n = s_0, s_1, ..., s_n$*. A* ***feasible state sequence***, $\Pi_n^\phi = s_0, s_1, ..., s_n$ *is consistent with the transition:* $\forall i\ (0 \leq i < n \Rightarrow s_i \to_\Phi s_{i+1})$.

Definition 3. *Given a sequence of states*

$$\Pi_n = s_0, s_1, ..., s_n$$

where

$$s_i = (\mu_i\ \mathrm{L}_i\ \mathrm{R}_i\ \phi_i\ \eta_i\ \mathrm{e}_i\ \mathrm{k}_i)$$

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

$$\pi_n = \mathbb{CF}(\Pi_n) = (\eta_0\ \mathrm{e}_0\ \mathrm{k}_0), (\eta_1\ \mathrm{e}_1\ \mathrm{k}_1), ..., (\eta_n\ \mathrm{e}_n\ \mathrm{k}_n)$$

Definition 4. *Given a state transition function* $\to_\Phi$*, an initial state* s_0 *and a control flow sequence* π_n*, the* ***feasible state set***, $\mathbb{FS}(\to_\Phi, s_0, \pi_n)$*, is defined as*

$$
\begin{array}{l}
\mathbb{FS}(\to_\Phi, s_0, \pi_n) = \\
\quad \{s \mid \exists \Pi_n^\phi\ (\pi_n = \mathbb{CF}(\Pi_n^\Phi) \wedge s = last(\Pi_n))\}
\end{array}
$$

where $last(\Pi_n)$ *returns the last state on the feasible sequence.*

Definition 5. *A* ***heap homomorphism*** $(g\ h)$ *between two states* s_x *and* s_y *is defined as a pair of functions* $g : \mathrm{r} \mapsto \mathrm{r}$ *and* $h : \mathrm{l} \mapsto \mathrm{l}$ *such that for any reference* $r \in refs(s_x)$*, location* $l \in locs(s_x)$*, and field* f*,*

$$(\phi_x\ l) \in \mathrm{L}_x(r) \Rightarrow (\phi_y\ h(l)) \in \mathrm{L}_y(g(r))$$

and

$$r = \mathrm{R}_x(l, f) \Rightarrow g(r) = \mathrm{R}_y(h(l), f)$$

If a such a pair of functions exists from one state s_x *to another state* s_y *we say that state* s_x *is* ***heap homomorphic*** *to state* s_y*, indicated by the notation* $s_x \rightharpoonup_{(g\ h)} s_y$.

Suppose we take the set of constraints from the image of s_x *in* s_y *under* $(g\ h)$ *:*

$$\chi = \{\phi \mid \exists (r \in \mathrm{R}_x,\ l \in \mathrm{L}_x)((\phi\ h(l)) \in \mathrm{L}_y(g(r))\}$$

and we conjoin those constraints with the global invariant ϕ_y *from* s_y *:*

$$\phi_G = \phi_y \wedge (\bigwedge_{\phi_i \in \chi} \phi_i)$$

If the expression ϕ_G *is satisfiable, we say that the heap homomorphism* $s_x \rightharpoonup_{(g\ h)} s_y$ *is* ***valid***.

$$\textsc{Variable lookup}\quad (L\,R\,\phi_g\,\eta\,x\,k) \rightarrow (L\,R\,\phi_g\,\eta\,\eta(x)\,k)$$

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

$$\textsc{Field Access(eval)}\quad (L\,R\,\phi_g\,\eta\,(e\,\$f)\,k) \rightarrow (L\,R\,\phi_g\,\eta\,e\,(*\,\$f \rightarrow k))$$

$$\textsc{Field Write (eval)}\quad (L\,R\,\phi_g\,\eta\,(x\,\$f := e)\,k) \rightarrow (L\,R\,\phi_g\,\eta\,e\,(x\,\$f := * \rightarrow k))$$

$$\textsc{Equals (l-operand eval)}\quad (L\,R\,\phi_g\,\eta\,(e_0 = e)\,k) \rightarrow (L\,R\,\phi_g\,\eta\,e_0\,(* = e \rightarrow k))$$

$$\textsc{Equals (r-operand eval)}\quad (L\,R\,\phi_g\,\eta\,v\,(* = e \rightarrow k)) \rightarrow (L\,R\,\phi_g\,\eta\,e\,(v = * \rightarrow k))$$

$$\textsc{Equals (bool)}\quad \frac{\begin{array}{l} v_0 \in \{\mathbf{true},\mathbf{false}\} \qquad v_1 \in \{\mathbf{true},\mathbf{false}\} \\ v_r = \mathrm{eq?}(v_0, v_1)\end{array}}{\begin{array}{l}(L\,R\,\phi_g\,\eta\,v_0\,(v_1 = * \rightarrow k)) \rightarrow \\ (L\,R\,\phi_g\,\eta\,v_r\,k)\end{array}}$$

$$\textsc{If-then-else (eval)}\quad (L\,R\,\phi_g\,\eta\,(\mathbf{if}\ e_0\ e_1\ \mathbf{else}\ e_2)\,k) \rightarrow (L\,R\,\phi_g\,\eta\,e_0\,(\mathbf{if}\ *\ e_1\ \mathbf{else}\ e_2) \rightarrow k)$$

$$\textsc{If-then-else (true)}\quad (L\,R\,\phi_g\,\eta\,\mathbf{true}\,(\mathbf{if}\ *\ e_1\ \mathbf{else}\ e_2) \rightarrow k) \rightarrow (L\,R\,\phi_g\,\eta\,e_1\,k)$$

$$\textsc{If-then-else (false)}\quad (L\,R\,\phi_g\,\eta\,\mathbf{false}\,(\mathbf{if}\ *\ e_1\ \mathbf{else}\ e_2) \rightarrow k) \rightarrow (L\,R\,\phi_g\,\eta\,e_2\,k)$$

$$\textsc{Variable Declaration (eval)}\quad (L\,R\,\phi_g\,\eta\,(\mathbf{var}\ T\,x := e_0\ \mathbf{in}\ e_1)\,k) \rightarrow (L\,R\,\phi_g\,\eta\,e_0\,(\mathbf{var}\ T\,x := *\ \mathbf{in}\ e_1 \rightarrow k))$$

$$\textsc{Variable Declaration}\quad (L\,R\,\phi_g\,\eta\,v\,(\mathbf{var}\ T\,x\ * := \mathbf{in}\ e_1 \rightarrow k)) \rightarrow (L\,R\,\phi_g\,\eta[x \mapsto v]\,e_1\,(\mathbf{pop}\ \eta\,k))$$

$$\textsc{Method Invocation (object eval)}\quad (L\,R\,\phi_g\,\eta\,(e_0\ @\ m\ e_1)\,k) \rightarrow (L\,R\,\phi_g\,\eta\,e_0\,(*\ @\ m\ e_1 \rightarrow k))$$

$$\textsc{Method Invocation (arg eval)}\quad (L\,R\,\phi_g\,\eta\,v_0\,(*\ @\ m\ e_1 \rightarrow k)) \rightarrow (L\,R\,\phi_g\,\eta\,e_1\,(v_0\ @\ m\ * \rightarrow k))$$

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

$$\textsc{Variable Assignment (eval)}\quad (L\,R\,\phi_g\,\eta\,(x := e)\,k) \rightarrow (L\,R\,\phi_g\,\eta\,e\,(x := * \rightarrow k))$$

$$\textsc{Variable Assignment}\quad (L\,R\,\phi_g\,\eta\,v\,(x := * \rightarrow k)) \rightarrow (L\,R\,\phi_g\,\eta[x \mapsto v]\,v\,k)$$

$$\textsc{Begin (no args)}\quad (L\,R\,\phi_g\,\eta\,(\mathbf{begin})\,k) \rightarrow (L\,R\,\phi_g\,\eta\,k)$$

$$\textsc{Begin (arg0 eval)}\quad (L\,R\,\phi_g\,\eta\,(\mathbf{begin}\ e_0\ e_1\ \ldots)\,k) \rightarrow (L\,R\,\phi_g\,\eta\,e_0\,(\mathbf{begin}\ *\ (e_1\ \ldots) \rightarrow k))$$

$$\textsc{Begin (argi eval)}\quad (L\,R\,\phi_g\,\eta\,v\,(\mathbf{begin}\ *\ (e_i\ e_{i+1}\ \ldots) \rightarrow k)) \rightarrow (L\,R\,\phi_g\,\eta\,e_i\,(\mathbf{begin}\ *\ (e_{i+1}\ \ldots) \rightarrow k))$$

$$\textsc{Begin (argN eval)}\quad (L\,R\,\phi_g\,\eta\,v\,(\mathbf{begin}\ *\ (e_n) \rightarrow k)) \rightarrow (L\,R\,\phi_g\,\eta\,e_n\,(\mathbf{begin}\ *\ () \rightarrow k))$$

$$\textsc{Begin}\quad (L\,R\,\phi_g\,\eta\,v\,(\mathbf{begin}\ *\ () \rightarrow k)) \rightarrow (L\,R\,\phi_g\,\eta\,v\,k)$$

$$\textsc{NULL}\quad (L\,R\,\phi_g\,\eta\,\mathbf{null}\,k) \rightarrow (L\,R\,\phi_g\,\eta\,r_{null}\,k)$$

$$\textsc{Pop}\quad (L\,R\,\phi_g\,\eta\,v\,(\mathbf{pop}\ \eta_0\,k)) \rightarrow (L\,R\,\phi_g\,\eta_0\,v\,k)$$

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

Definition 6. *The **representation relation** is defined as follows: given two states s_ℓ and s_s, $s_\ell \sqsubset s_s$ if and only if $\eta_\ell = \eta_s$, $\mathrm{e}_\ell = \mathrm{e}_s$, $\mathrm{k}_\ell = \mathrm{k}_s$, and there exists a valid heap homomorphism $s_\ell \rightharpoonup_{(g\ h)} s_s$. The expression $s_\ell \sqsubset s_s$ can be read as "state s_ℓ is represented by state s_s .*

Definition 7. *A state s is **congruent** to a set of states $\mathcal{S}$ if and only if s represents every state in $\mathcal{S}$ and represents no other state:*

$$s \cong \mathcal{S} : s_i \in \mathcal{S} \Leftrightarrow s_i \sqsubset s$$

Need to define a *lazy state* and a *summary state* (or whatever name we are using).

Definition 8. *A state s is **exact** with respect to a transition relation, $\rightarrow_\phi$, an initial state, s_0, and specific control flow path, π_n, if and only if it is congruent to the set of feasible state set:*

$$s \cong \mathbb{FS}(\rightarrow_\phi, s_0, \pi)$$

4.2 Theorems

Theorem 1. *In any symbolic state s_s, the constraints in the value set of any reference are mutually exclusive.*

Proof. □

Lemma 2. *If symbolic state $s_s = (\mathrm{L}_\mathcal{S}\ \mathrm{R}_\mathcal{S}\ \phi_g\ \eta\ \mathrm{r}\ (*\ \$\ \mathrm{f} \rightarrow \mathrm{k}))$ is exact with respect to some initial state s_0 and control flow path π_n, then the intermediate state state $s_f : s_s \rightarrow_S s_f$ is congruent to $\{\forall s'_\ell | \exists s_\ell \sqsubset s_s(s_\ell \rightarrow^*_I s'_\ell)\}$.*

Proof. Given a lazy state $s_\ell \sqsubset s_s$, there are four distinct possibilities for s'_ℓ: either the field is already initialized, or the field is initialized to a new reference, the null reference, or an alias.

Case 1: Suppose the field is initialized. In this case, the rule does nothing and the theorem is trivially true.

Case 2: The "new" rule gets executed.

Case 3: The "null" rule gets executed.

Case 4: The "alias" rule gets executed. □

Lemma 3. *If symbolic state $s_s = (\mathrm{L}_\mathcal{S}\ \mathrm{R}_\mathcal{S}\ \phi_g\ \eta\ \mathrm{r}\ (*\ \$\ \mathrm{f} \rightarrow \mathrm{k}))$ is exact with respect to some initial state s_0 and control flow path π_n, then the state $s'_s : s_s \rightarrow_s s_f$ is exact with respect to s_0 and π_{n+1}.*

Initialize (null)

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

Initialize (new)

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

Initialize (alias)

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

Initialize (end)

$$\frac{\Lambda = \{l \mid \exists\phi\ ((\phi\ l) \in L(r) \wedge R(l,f) = \bot\ )\} \qquad \Lambda = \emptyset}{(L\, R\, \phi_g\ r f) \rightarrow_I (L\, R\, \phi_g\ r f)}$$

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

Field Access

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

Field Write

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

Equals (reference-true)

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

Equals (reference-false)

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

Figure 7. Generalized symbolic execution with lazy initialization.

Summarize

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

Summarize (end)

$$\frac{\Lambda = \{l \mid \exists\phi\ ((\phi\ l) \in L(r) \wedge R(l,f) = \bot\ )\} \qquad \Lambda = \emptyset}{(L\, R\, r f) \rightarrow_I (L\, R\, r f)}$$

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

Proof. We will consider two cases for this proof. In the first case, we assume that all of the fields involved in the read are initialized. In the second case we consider the case of uninitialized fields.

Case 1: suppose all of the pertinent fields in s_s are initialized. Take an arbitrary lazy state $s_\ell \sqsubset s_s$. Since s_s is exact, $s_\ell = (L_\ell\, R_\ell\, \phi_L\ \eta\ r\ (\texttt{*}\ \texttt{\$}f \rightarrow k))$, and that $s_\ell \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_n)$. If we apply the state transition functions to achieve states $s'_\ell : s_\ell \rightarrow_\ell s'_\ell$ and $s'_s : s_s \rightarrow_s s'_s$, we find that:

$$s'_\ell = (L_\ell[r' \mapsto (\phi'\ l')]\ R_\ell\ \phi_L\ \eta\ r'\ k)$$

and

$$s'_s = (L_s[r' \mapsto \mathbb{VS}(L_{\mathcal{S}}, R_s, r, f, \phi_g)]\ R_{\mathcal{S}}\ \phi_g\ \eta\ r'\ k)$$

We now show that $s'_\ell \sqsubset s'_s$. Since η, e, and k are identical between s'_s and s'_ℓ, the first condition is met by default. Now we

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

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

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

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

Figure 9. Precise symbolic heap summaries from symbolic execution.

construct functions $g' : g' = g[r' \mapsto r']$ and $h' : h' = h$. Observe that since $s_\ell \rightarrow s_s$, and since R_ℓ and R_s are unchanged from states s_ℓ to s'_ℓ and s_s to s'_s respectively, we are guaranteed that $r = R_\ell(l, f) \Rightarrow g'(r) = R_s(h'(l), f)$. Since $(\phi'_\ell\ l') = L_\ell(R_\ell(l, f))$, and since $(g\ h)$ is valid, we know that:

$$(\phi_s \wedge \phi'_s\ l') \in \mathbb{VS}(L_{\mathcal{S}}, R_s, r, f, \phi_g)$$

From this, we may deduce that:

$$(\phi_\ell\ l) \in L'_\ell(r') \Rightarrow (\phi_s \wedge \phi'_s\ h'(l)) \in L'_s(g'(r'))$$

Since r' the only new addition to L'_ℓ and L'_s, we now know that the assertion above holds for all $l \in s'_\ell$. Thus, we have shown that $(g'\ h')$ is a heap homomorphism from s'_ℓ to s'_s. Furthermore, since $\mathbb{S}(\phi_s \wedge \phi'_s \wedge \phi_g)$ holds true, we know that $(g'\ h') : s'_\ell \rightarrow s'_s$ is valid. Since there is a valid heap homomorphism, and since $\eta_\ell = \eta_s,\ e_\ell = e_s,\ k_\ell = k_s$, we by definition 6 know $s'_\ell \sqsubset s'_s$. We have now shown that for any lazy state s_ℓ:

$$s_\ell \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_n) \Rightarrow s'_\ell \sqsubset s'_s$$

Now, suppose that there exists a state s'_i such that $s'_i \sqsubset s'_s$, but $s'_i \notin \mathbb{FS}(\rightarrow_\ell, s_0, \pi_{n+1})$. Since $s'_i \sqsubset s'_s$, there must exist a valid homomorphism $(g'_i h'_i) : s'_i \rightarrow s'_s$. If $(g'_i\ h'_i)$ is valid, then there must also exist a valid homomorphism $(g_i\ h_i) : s_i \rightarrow s_s$, and by extension, state $s_i \sqsubset s_s$. However, since s_s is exact, $s_i \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_n)$, and by our previous result, $s'_i \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_{n+1})$. We have a contradiction. Therefore,

$$s'_i \sqsubset s'_s \Rightarrow s'_i \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_{n+1})$$

Combining the two previous equations, we find that

$$s'_\ell \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_n) \Leftrightarrow s'_\ell \sqsubset s'_s)$$

By definition 7, $s'_s \equiv \mathbb{FS}(\rightarrow_\ell, s_0, \pi_n)$, and so by definition 8, s'_s is exact.

Case 2: If there are uninitialized fields, then the lazy initialization machine will make an intermediate state s_t. By lemma 2, the symbolic state is congruent to the set of lazy initialized states. From the intermediate state, the proof is the same as for case 1. ☐

Lemma 4. *The field write rule is correct.*

Proof. ☐

Lemma 5. *If symbolic state $s_s = (L_s\ R_s\ \phi_s\ \eta\ r_0\ (r_1 \texttt{ = *} \rightarrow k))$ is exact with respect to some initial state s_0 and control flow path π_n, then the state $s'_s : s_s \rightarrow_s s'_s$ is exact with respect to s_0 and π_{n+1}.*

There are two rules that apply to state s_s, one for the **true** branch and one for the **false** branch. Since the proofs for both rules are nearly identical, for clarity we will only prove the case for the **true** branch here.

Proof. Choose any $s_\ell \sqsubset s_s$, and let $\zeta_T = \mathbb{FS}(\rightarrow_\ell, s_0, (\pi_n, (\eta\ \textbf{true}\ k)))$. Since s_s is exact, we know that $s_\ell \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_n)$, $s_l = (L_\ell\ R_\ell\ \phi_\ell\ \eta\ r_0\ (r_1 \texttt{ = *} \rightarrow k))$, and that there exists $(g\ h) : s_\ell \rightarrow s_s$. Depending on the values of $L_\ell(r_0)$ and $L_\ell(r_1)$, there are two different rules that might apply to s_ℓ.

Case 1: Assume $s_\ell : L_\ell(r_0) = L_\ell(r_1)$, and let

$$\zeta_t = \zeta_T \setminus \{s_f | L_f(r_0) \neq L_f(r_1)\}$$

In this case, the lazy "equals - references true" rule applies, and we know state $s'_\ell : s_\ell \rightarrow_\ell s'_\ell$ is in ζ_t. Since $(g\ h)$ is valid, we know that $\exists(\phi_0\ l) \in L_\ell(g(r_0))$, $\exists(\phi_1\ l) \in L_\ell(g(r_1))$, and that $\mathbb{S}(\phi_s \wedge \phi_0 \wedge \phi_1)$ holds true. Observe that by applying theorem 1, $\phi_s' \wedge \phi_0 \wedge \phi_1$ reduces to ϕ_s. Therefore, $(g\ h) : s_\ell' \to s_s'$ is a valid heap homomorphism, and by extension, $s_\ell' \sqsubset s_s'$. Since this relation holds for arbitrary $s_\ell' \in \zeta_t$, we now know that

$$s_\ell' \in \zeta_t \Rightarrow s_\ell' \sqsubset s_s'$$

Now we prove the case for the other direction. Consider the state $s_s' : s_s \to_s s_s'$. Define θ_α, θ_0 and θ_1 as in the "equals (references-true) rule". Since L_s and R_s are unchanged from s_s, and ϕ_g is only strengthened, we know that:

$$\{s_\ell' | s_\ell' \sqsubset s_s'\} \subseteq \{s_\ell' | \exists s_\ell(s_\ell \sqsubset s_s) \wedge s_\ell \to_s s_\ell'\}$$

Suppose that there exists state s_i' such that $s_i' \sqsubset s_s'$ and $s_i' \notin \zeta_t$. Because of the above conclusion, we know that

$$s_i' \in \{s_\ell' | \exists s_\ell(s_\ell \sqsubset s_s) \wedge s_\ell \to_s s_\ell'\}$$

Combining this with the assumption that $s_i' \notin \zeta_t$, we must conclude that $L_\ell(r_0) \neq L_\ell(r_1)$. Because of this, and because of theorem 1, we know that either all constraints

$$\phi_i : \exists\phi_\alpha(\phi_\alpha \in \theta_\alpha) \wedge \phi_i = (\phi_\alpha \wedge \phi_0 \wedge \phi_1)$$

are unsatisfiable, or that at least one constraint

$$\phi_i : \exists\phi_\alpha(\phi_\alpha \in (\theta_0 \cup \theta_1)) \wedge (\phi_i = \phi_\alpha \wedge \phi_0 \wedge \phi_1)$$

is valid. Either way, $\mathbb{S}(\phi_i' \wedge \phi_0 \wedge \phi_1)$ is false, $(g\ h)$ is not a valid homomorphism, and s_s' does not represent s_i'. We have a contradiction. Therefore:

$$s_\ell' \sqsubset s_s' \Rightarrow s_\ell' \in \zeta_t$$

Combining this with our previous result, we conclude that

$$s_\ell' \in \zeta_t \Leftrightarrow s_\ell' \sqsubset s_s'$$

Case 2: Assume $s_\ell : L_\ell(r_0) \neq L_\ell(r_1)$, and let

$$\zeta_f = \zeta_T \setminus \{s_t | L_t(r_0) = L_t(r_1)\}$$

This means that the lazy "equals - references false" rule applies. The proof for the "equals - references false" rule is highly similar to the proof for "equals - references true", so we omit it for the sake of brevity. The result for this case is:

$$s_\ell' \in \zeta_f \Leftrightarrow s_\ell' \sqsubset s_s'$$

Since $\zeta_T = \zeta_t \cup \zeta_f$, we can combine the results of the two cases to find that

$$s_\ell' \in \zeta_T \Leftrightarrow s_\ell' \sqsubset s_s'$$

. By definition 7, $s_s' \equiv \zeta_T$, and by definition 8, s_s' is exact. □

Theorem 6. *Every symbolic state on every execution path is exact.*

Proof. Combine all of the production-rule lemmas to inductively prove the theorem. □

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.

[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.