Exact Heap Summaries from Symbolic Execution

Anonymous

Abstract

A fundamental challenge of using symbolic execution for software analysis has been the treatment of dynamically allocated data. State-of-the-art techniques have addressed this challenge through the use of heap summaries and by constructing the concrete heap "lazily." In this work, we present a novel heap analysis technique which takes inspiration from both approaches and builds exact heap summaries lazily during symbolic execution. Our technique enables exact analysis of program properties and analysis of arbitrary recursive data structures. It also supports dynamic bounds for linked data structure inputs, and efficient sub-division of the underlying constraint problem. We demonstrate the precision and scalability of our approach in the Symbolic PathFinder framework for analyzing Java programs.

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.

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

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:

- 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. Javalite

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 environment, η, associates a variable x with a value v. The value can be a reference, r or one of the special values **null**, **true**, or **false**. Although the Javalite machine is purely syntactic, for clarity and brevity in the presentation, the more complex structures such as the environment are treated as partial functions. As such, $\eta(x) = r$ is the reference mapped to the variable in the environment. The notation $\eta' = \eta[x \mapsto v]$ defines a new partial function η' that is just like η only the variable x now maps to v.

The heap is a labeled bipartite graph consisting of references, r, and locations, l. The machine syntax in Figure 4 defines that graph in L, the location map, and R, the reference map. As done with the environment, L and R are treated as partial functions where $L(r) = \{(\phi\ l)\ \ldots\}$ is the set of location-constraint pairs in the heap associated with the given reference, and $R(l,f) = r$ is the reference associated with the given location-field pair in the heap.

As the updates to L and R are complex in the machine semantics, predicate calculus is used to describe updates to the functions. Consider the following example where l is some location andρ is a

```
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

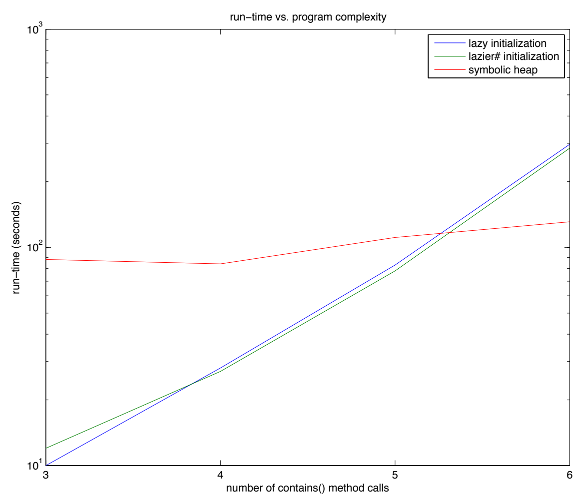

Figure 2. Time versus complexity for the linked list example

set of references.

$$L' = L[r \mapsto \{(\textbf{true}\ l)\}][\forall r' \in \rho\ (r' \mapsto (\textbf{true}\ l_{null}))]$$

The new partial function L' is just like L only it remaps r, and it remaps all the references in ρ.

The location l_{null} is a special location in the heap to represent null. It has a companion reference r_{null}. The initial heap for the machine is defined such that $L(r_{null}) = \{(\textbf{true}\ l_{null})\}$

The initial state of the machine needs to be defined.

The rewrite rules that define the Javalite semantics are in Figure 5.

Variable lookup
$$(L\,R\,\phi_g\,\eta\,x\,k) \to_J (L\,R\,\phi_g\,\eta\,\eta(x)\,k)$$

New
$$\frac{r = \text{stack}_r() \quad l = \text{fresh}_l(C) \quad R' = R[\forall f \in \mathit{fields}(\mathrm{C})\ ((l\,f) \mapsto r_{null})] \quad L' = L[r \mapsto \{(\textbf{true}\ l)\}]}{(L\,R\,\phi_g\,\eta\,(\textbf{new}\ C)\,k) \to_J (L'\,R'\,\phi_g\,\eta\,r\,k)}$$

Field Access(eval)
$$(L\,R\,\phi_g\,\eta\,(e\,\$f)\,k) \to_J (L\,R\,\phi_g\,\eta\,e\,(*\,\$f \to k))$$

Field Write (eval)
$$(L\,R\,\phi_g\,\eta\,(x\,\$f := e)\,k) \to_J (L\,R\,\phi_g\,\eta\,e\,(x\,\$f := * \to k))$$

Equals (l-operand eval)
$$(L\,R\,\phi_g\,\eta\,(e_0 = e)\,k) \to_J (L\,R\,\phi_g\,\eta\,e_0\,(* = e \to k))$$

Equals (r-operand eval)
$$(L\,R\,\phi_g\,\eta\,v\,(* = e \to k)) \to_J (L\,R\,\phi_g\,\eta\,e\,(v = * \to k))$$

Equals (bool)
$$\frac{v_0 \in \{\textbf{true}, \textbf{false}\} \quad v_1 \in \{\textbf{true}, \textbf{false}\} \quad v_r = \text{eq?}(v_0, v_1)}{(L\,R\,\phi_g\,\eta\,v_0\,(v_1 = * \to k)) \to_J (L\,R\,\phi_g\,\eta\,v_r\,k)}$$

If-then-else (eval)
$$(L\,R\,\phi_g\,\eta\,(\textbf{if}\ e_0\ e_1\ \textbf{else}\ e_2)\,k) \to_J (L\,R\,\phi_g\,\eta\,e_0\,(\textbf{if}\ *\ e_1\ \textbf{else}\ e_2) \to k)$$

If-then-else (true)
$$(L\,R\,\phi_g\,\eta\,\textbf{true}\,(\textbf{if}\ *\ e_1\ \textbf{else}\ e_2) \to_J k) \to (L\,R\,\phi_g\,\eta\,e_1\,k)$$

If-then-else (false)
$$(L\,R\,\phi_g\,\eta\,\textbf{false}\,(\textbf{if}\ *\ e_1\ \textbf{else}\ e_2) \to_J k) \to (L\,R\,\phi_g\,\eta\,e_2\,k)$$

Variable Declaration (eval)
$$(L\,R\,\phi_g\,\eta\,(\textbf{var}\ T\,x := e_0\ \textbf{in}\ e_1)\,k) \to_J (L\,R\,\phi_g\,\eta\,e_0\,(\textbf{var}\ T\,x := *\ \textbf{in}\ e_1 \to k))$$

Variable Declaration
$$(L\,R\,\phi_g\,\eta\,v\,(\textbf{var}\ T\,x\,* := \textbf{in}\ e_1 \to k)) \to_J (L\,R\,\phi_g\,\eta[x \mapsto v]\,e_1\,(\textbf{pop}\ \eta\,k))$$

Method Invocation (object eval)
$$(L\,R\,\phi_g\,\eta\,(e_0\ @\ m\ e_1)\,k) \to_J (L\,R\,\phi_g\,\eta\,e_0\,(*\ @\ m\ e_1 \to k))$$

Method Invocation (arg eval)
$$(L\,R\,\phi_g\,\eta\,v_0\,(*\ @\ m\ e_1 \to k)) \to_J (L\,R\,\phi_g\,\eta\,e_1\,(v_0\ @\ m\ * \to k))$$

Method Invocation
$$\frac{(T\,m\ [T\,x]\ \ e_m) = \text{lookup}(m) \quad \eta_m = \eta[\textbf{this} \mapsto v_0][x \mapsto v_1]}{(L\,R\,\phi_g\,\eta\,v_1\,(v_0\ @\ m\ * \to k)) \to_J (L\,R\,\phi_g\,\eta_m\,e_m\,(\textbf{pop}\ \eta\,k))}$$

Variable Assignment (eval)
$$(L\,R\,\phi_g\,\eta\,(x := e)\,k) \to_J (L\,R\,\phi_g\,\eta\,e\,(x := * \to k))$$

Variable Assignment
$$(L\,R\,\phi_g\,\eta\,v\,(x := * \to k)) \to_J (L\,R\,\phi_g\,\eta[x \mapsto v]\,v\,k)$$

Begin (no args)
$$(L\,R\,\phi_g\,\eta\,(\textbf{begin})\,k) \to (L\,R\,\phi_g\,\eta\,k)$$

Begin (arg0 eval)
$$(L\,R\,\phi_g\,\eta\,(\textbf{begin}\ e_0\ e_1\ \ldots)\,k) \to_J (L\,R\,\phi_g\,\eta\,e_0\,(\textbf{begin}\ *\ (e_1\ \ldots) \to k))$$

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

Begin (argN eval)
$$(L\,R\,\phi_g\,\eta\,v\,(\textbf{begin}\ *\ (e_n) \to k)) \to (L\,R\,\phi_g\,\eta\,e_n\,(\textbf{begin}\ *\ () \to k))$$

Begin
$$(L\,R\,\phi_g\,\eta\,v\,(\textbf{begin}\ *\ () \to k)) \to (L\,R\,\phi_g\,\eta\,v\,k)$$

NULL
$$(L\,R\,\phi_g\,\eta\,\textbf{null}\,k) \to (L\,R\,\phi_g\,\eta\,r_{null}\,k)$$

Pop
$$(L\,R\,\phi_g\,\eta\,v\,(\textbf{pop}\ \eta_0\,k)) \to (L\,R\,\phi_g\,\eta_0\,v\,k)$$

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

4. GSE with Lazy Initialization

A special reference, r_{un}, and location, l_{un}, is introduced to support lazy initialization in GSE. The 'un' is to indicate the reference or location is uninitialized at the point of execution. The initial state of the machine maps r_{null} as before and adds $L(r_{un}) = \{(\textbf{true}\ l_{un})\}$

A field in an object is symbolic, meaning it is uninitialized, if the location for the field is l_{un} on some constraint. The function $\mathbb{UN}(L, R, r, f) = \{l\ \ldots\}$ returns constraint-location pairs in which the field f is uninitialized:

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

where $\mathbb{S}(\phi)$ returns true if ϕ is satisfiable. The cardinality of the set is never greater than one in GSE and the constraint is always satisfiable because all constraints are constant. This property is relaxed in GSE with heap summaries.

5. GSE with Heap Summaries

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

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

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

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

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

6. Proofs

6.1 Definitions

Definition 1. *The set of **states** $\mathcal{S}$ is defined as*

Definition 2. *$\mathcal{S}_0$ is defined as the set of **initial states**. An initial state is a state meeting the following conditions: The range of L has*

INITIALIZE (NULL)

$$\frac{\Lambda = \mathbb{UN}(L,R,r,f) \qquad \Lambda \neq \emptyset \qquad (\phi_x\ l_x) = \min_l(\Lambda) \\ r' = \mathrm{fresh}_r() \qquad \theta_{null} = \{(\mathbf{true}\ l_{null})\} \\ \phi'_g = (\phi_g \wedge r' = r_{null})}{(L\ R\ \phi_g\ r\ f\ C) \rightarrow_I (L[r' \mapsto \theta_{null}]\ R[(l_x,f) \mapsto r']\ \phi'_g\ r\ f\ C)}$$

INITIALIZE (NEW)

$$\frac{\Lambda = \mathbb{UN}(L,R,r,f) \qquad \Lambda \neq \emptyset \qquad (\phi_x\ l_x) = \min_l(\Lambda) \\ r_f = \mathrm{init}_r() \qquad l_f = \mathrm{fresh}_l(C) \\ \rho = \{(r_a\ l_a) \mid \mathrm{isInit}(r_a) \wedge r_a = \min_r(R^{\leftarrow}[l_a]) \wedge \mathrm{type}(l_a) = C\} \\ \theta_{new} = \{(\mathbf{true}\ l_f)\} \\ R' = R[\forall f \in \mathit{fields}(\mathrm{C})\ ((l_f\ f) \mapsto r_{un})] \\ \phi'_g = (\phi_g \wedge r_f \neq r_{null} \wedge (\wedge_{(r_a\ l_a) \in \rho}\ r_f \neq r_a))}{(L\ R\ \phi_g\ r\ f\ C) \rightarrow_I (L[r_f \mapsto \theta_{new}]\ R'[(l_x,f) \mapsto r_f]\ \phi'_g\ r\ f\ C)}$$

INITIALIZE (ALIAS)

$$\frac{\Lambda = \mathbb{UN}(L,R,r,f) \qquad \Lambda \neq \emptyset \qquad (\phi_x\ l_x) = \min_l(\Lambda) \\ r' = \mathrm{fresh}_r() \\ \rho = \{(r_a\ l_a) \mid \mathrm{isInit}(r_a) \wedge r_a = \min_r(R^{\leftarrow}[l_a]) \wedge \mathrm{type}(l_a) = C\} \\ (r_a\ l_a) \in \rho \qquad \theta_{alias} = \{(\mathbf{true}\ l_a)\} \\ \phi'_g = (\phi_g \wedge r' \neq r_{null} \wedge r' = r_a \wedge (\wedge_{(r'_a\ l_a) \in \rho\ (r'_a \neq r_a)}\ r' \neq r'_a))}{(L\ R\ \phi_g\ r\ f\ C) \rightarrow_I (L[r' \mapsto \theta_{alias}]\ R[(l_x,f) \mapsto r']\ \phi'_g\ r\ f\ C)}$$

INITIALIZE (END)

$$\frac{\Lambda = \mathbb{UN}(L,R,r,f) \qquad \Lambda = \emptyset}{(L\ R\ \phi_g\ r\ f\ C) \rightarrow_I (L\ R\ \phi_g\ r\ f\ C)}$$

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

$$\frac{\{(\phi\ l)\} = L(r) \qquad l \neq l_{null} \qquad C = \mathrm{type}(l,f) \\ (L\ R\ \phi_g\ r\ f\ C) \rightarrow_I^* (L'\ R'\ \phi'_g\ r\ f\ C) \\ \{(\phi'\ l')\} = L'(R'(l,f)) \qquad r' = \mathrm{stack}_r()}{(L\ R\ \phi_g\ \eta\ r\ (*\ \$f \rightarrow k)) \rightarrow_\ell \\ (L'[r' \mapsto (\phi'\ l')]\ R'\ \phi'_g\ \eta\ r'\ k)}$$

FIELD WRITE

$$\frac{r_x = \eta(x) \qquad \theta = \{(\phi\ l)\} = L(r_x) \\ l \neq l_{null} \qquad r' = \mathrm{fresh}_r()}{(L\ R\ \phi_g\ \eta\ r\ (x\ \$f := *\ \rightarrow\ k)) \rightarrow_\ell \\ (L[r' \mapsto \theta]\ R[(l\ f) \mapsto r']\ \phi_g\ \eta\ r\ k)}$$

EQUALS (REFERENCE-TRUE)

$$\frac{L(r_0) = L(r_1) \qquad \phi'_g = (\phi_g \wedge r_0 = r_1)}{(L\ R\ \phi_g\ \eta\ r_0\ (r_1 = * \rightarrow k)) \rightarrow_\ell \\ (L\ R\ \phi'_g\ \eta\ \mathbf{true}\ k)}$$

EQUALS (REFERENCE-FALSE)

$$\frac{L(r_0) \neq L(r_1) \qquad \phi'_g = (\phi_g \wedge r_0 \neq r_1)}{(L\ R\ \phi_g\ \eta\ r_0\ (r_1 = * \rightarrow k)) \rightarrow_\ell \\ (L\ R\ \phi'_g\ \eta\ \mathbf{false}\ k)}$$

Figure 7. GSE with lazy initialization indicated by $\rightarrow_\ell$.

SUMMARIZE

$$\frac{\Lambda = \mathbb{UN}(L,R,r,f) \qquad \Lambda \neq \emptyset \qquad (\phi_x\ l_x) = \min_l(\Lambda) \qquad r_f = \mathrm{init}_r() \qquad l_f = \mathrm{fresh}_l(\mathrm{C}) \\ \rho = \{(r_a\ \phi_a\ l_a) \mid \mathrm{isInit}(r_a) \wedge r_a = \min_r(R^{\leftarrow}[l_a]) \wedge (\phi_a\ l_a) \in L(r_a) \wedge \mathrm{type}(l_a) = \mathrm{C}\} \\ \theta_{null} = \{(\phi\ l_{null}) \mid \phi = (\phi_x \wedge r_f = r_{null})\} \\ \theta_{new} = \{(\phi\ l_f) \mid \phi = (\phi_x \wedge r_f \neq r_{null} \wedge (\wedge_{(r'_a,\ \phi'_a,\ l'_a) \in \rho}\ r_f \neq r'_a))\} \\ \theta_{alias} = \{(\phi\ l_a) \mid \exists r_a\ (\exists \phi_a\ ((r_a\ \phi_a\ l_a) \in \rho \wedge \phi = (\phi_x \wedge \phi_a \wedge r_f \neq r_{null} \wedge r_f = r_a \wedge (\wedge_{(r'_a\ \phi'_a\ l'_a) \in \rho\ (r'_a \neq r_a)}\ r_f \neq r'_a))))\} \\ \theta_{orig} = \{(\phi\ l_{orig}) \mid \exists \phi_{orig}((\phi_{orig}\ l_{orig}) \in L(R(l_x,f)) \wedge \phi = (\neg\phi_x \wedge \phi_{orig})\} \\ \theta = \theta_{null} \cup \theta_{new} \cup \theta_{alias} \cup \theta_{orig} \qquad R' = R[\forall f \in \mathit{fields}(\mathrm{C})\ ((l_f\ f) \mapsto r_{un})]}{(L\ R\ r\ f\ C) \rightarrow_S (L[r_f \mapsto \theta]\ R'[(l_x,f) \mapsto r_f]\ r\ f\ C)}$$

SUMMARIZE (END)

$$\frac{\Lambda = \mathbb{UN}(L,R,r,f) \qquad \Lambda = \emptyset}{(L\ R\ r\ f\ C) \rightarrow_S (L\ R\ r\ f\ C)}$$

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

exactly three locations: l_{null}, l_{un}, and l_0, the function R is defined only for location l_0, and for any field f, $R(l_0, f)$ returns r_{un}.

Definition 3. *The set of **references** $\mathcal{R}$ is defined as the set of natural numbers*

$$\mathcal{R} = \mathbb{N}$$

The total number of references in a summary state and a lazy state that it represents are generally not the same. However, the number of references on the stack in either state is always the same. In order to make the distinction between different types of references, we partition the set of natural numbers using modular arithmetic.

Definition 4. *The set of **stack references** $\mathcal{R}_t$ is defined as*

$$\mathcal{R}_t = \{i \in \mathbb{N} \mid (i \mod 3) = 0\}$$

.

FIELD ACCESS

$$\frac{\begin{array}{l}\forall(\phi\ l) \in L(r)\ \ (l = l_{null} \to \neg\mathbb{S}(\phi \wedge \phi_g)) \\ \{C\} = \{C \mid \exists(\phi\ l) \in L(r)\ \ (C = \text{type}(l,f))\} \\ (L\ R\ r\ f\ C) \to^*_S (L'\ R'\ r\ f\ C) \qquad r' = \text{stack}_r()\end{array}}{(L\ R\ \phi_g\ \eta\ r\ (\textbf{*}\ \$ f \to k)) \to_\varsigma (L'[r' \mapsto \mathbb{VS}(L', R', r, f, \phi_g)]\ R'\ \phi_g\ \eta\ r'\ k)}$$

FIELD WRITE

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

EQUALS (REFERENCES-TRUE)

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

EQUALS (REFERENCES-FALSE)

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

Figure 9. Precise symbolic heap summaries from symbolic execution indicated by $\to_\varsigma$.

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

Figure 3. The Javalite surface syntax.

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

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

Definition 5. *The set of* ***input heap references*** $\mathcal{R}_h$ *is defined as*

$$\mathcal{R}_h = \{i \in \mathbb{N} \mid (i \mod 3) = 1\}$$

.

Definition 6. *The set of* ***new heap references*** $\mathcal{R}_f$ *is defined as*

$$\mathcal{R}_n = \{i \in \mathbb{N} \mid (i \mod 3) = 2\}$$

.

Definition 7. *For a given function $f : A \mapsto B$, the **image** $f^{\rightarrow}$ and **preimage** $f^{\leftarrow}$ are defined as*

$$f^{\rightarrow} = \{f(a) \mid a \in A\} \quad (1)$$

$$f^{\leftarrow} = \{a \mid f(a) \in B\} \quad (2)$$

The bracket notation $f^{\rightarrow}[C]$ is used to denote that the image is drawn from a specific subset:

$$f^{\rightarrow}[C] = \{f(a) \mid a \in C\} \quad (3)$$

$$f^{\leftarrow}[D] = \{a \mid f(a) \in D\} \quad (4)$$

Where $C \subset A$ and $D \subset B$

Definition 8. *A **state transition relation** $\rightarrow_{\Phi}$ is a binary relation $\rightarrow_{\Phi} \subseteq \mathcal{S} \times \mathcal{S}$, which relat es machine states with successor states. Two important state transition relations are the **lazy state transition relation** $\rightarrow_{\ell}$ and the **summary state transition relation** $\rightarrow_{\varsigma}$. Each of these use a separate relation for initialization: $\rightarrow_{I}$ for initialization the lazy transition relation and $\rightarrow_{S}$ for initialization the summary transition relation. All of these transition relations are defined in Figure 7, Figure 6, Figure 9, and Figure 8.*

Definition 9. *A state $s = (\mathrm{L}\ \mathrm{R}\ \phi\ \eta\ \mathrm{e}\ \mathrm{k})$ is a **lazy state** if and only if $\forall \mathrm{r} \in \mathrm{L}^{\leftarrow}\ (|\mathrm{L}(\mathrm{r})| = 1)$]*

Definition 10. *A state $s = (\mathrm{L}\ \mathrm{R}\ \phi\ \eta\ \mathrm{e}\ \mathrm{k})$ is a **summary state** if and only if $\forall \mathrm{r} \in \mathrm{L}^{\leftarrow}\ (|\mathrm{L}(\mathrm{r})| \geq 1)$. A lazy state is also a summary state by definition.*

Definition 11. *A heap, $(\mathrm{L}\ \mathrm{R})$, is **location mutual exclusive** if and only if*

$$\forall \mathrm{r} \in \mathrm{L}^{\leftarrow}\ (\forall (\phi_\alpha\ \mathrm{l}_\alpha), (\phi_\beta\ \mathrm{l}_\beta) \in \mathrm{L}(r)\ (\\ \mathrm{l}_\alpha \neq \mathrm{l}_\beta \Rightarrow \phi_\alpha \wedge \phi_\beta = \textit{\textbf{false}}))$$

Definition 12. *The sets $\mathcal{FA}$, $\mathcal{FW}$, $\mathcal{RC}$, and $\mathcal{NW}$ are defined as the sets of states having the forms (L R ϕ_g η r (*** $** f $\rightarrow$ k)), (L R ϕ_g η r (x **$** f **:= *** $\rightarrow$ k)), (L R ϕ_g η r_0 (r_1 **= *** $\rightarrow$ k)), and (L R ϕ_g η (**new** C) k) respectively.*

Definition 13. ***Intermediate states** are imaginary placeholder states used when reasoning about complex transition rules in terms of simpler sub-rules. For example, the transition $s_x \rightarrow_{\phi} s_y$ may be equivalent to a sequence of simpler transitions $s_x \rightarrow_{\alpha} s_a \rightarrow_{\beta} s_b \rightarrow_{\gamma} s_y$. When reasoning about this equivalent transition sequence, it can be useful to discuss the notional intermediate states s_a and s_b. However, it is important to remember that s_a and s_b are not technically involved in the transition $s_x \rightarrow_{\phi} s_y$, and indeed may not be part of any feasible state sequence under transition relation $\rightarrow_{\phi}$.*

Definition 14. *Given a sequence of states*

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

where

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

*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 15. *A **homomorphism** $s_x \rightharpoonup_h s_y$, from state $s_x = (\mathrm{L}_x\ \mathrm{R}_x\ \phi_x\ \eta_x\ \mathrm{e}_x\ \mathrm{k}_x)$ to state $s_y = (\mathrm{L}_y\ \mathrm{R}_y\ \phi_y\ \eta_y\ \mathrm{e}_y\ \mathrm{k}_y)$, is defined as follows:*

$$\begin{aligned} & s_x \rightharpoonup_h s_y \Leftrightarrow \\ & \quad \exists h : \mathcal{L} \mapsto \mathcal{L}\ (\forall \mathrm{l}_\alpha\ (\forall \mathrm{l}_\beta\ (\forall f \in \mathcal{F}(\exists \phi_\alpha\ (\exists \phi_\beta\ (\\ & \qquad (\phi_\alpha\ \mathrm{l}_\alpha) \in \mathrm{L}_x(\mathrm{R}_x(\mathrm{l}_\beta, f)) \Rightarrow (\phi_\beta\ h(\mathrm{l}_\alpha)) \in \mathrm{L}_y(\mathrm{R}_y(h(\mathrm{l}_\beta), f)) \\ & \qquad)))))) \end{aligned}$$

Definition 16. *Given homomorphism $s_x \rightharpoonup_h s_y$, the **homomorphism constraint** $\mathbb{HC}(s_x \rightharpoonup_h s_y)$ is defined as:*

$$\mathbb{HC}(s_x \rightharpoonup_h s_y) = \\ \bigwedge \{\phi_b \mid \exists (\phi_a\ l) \in \mathrm{L}_x^{\rightarrow}((\phi_b\ h(l)) \in \mathrm{L}_y^{\rightarrow})\}$$

Definition 17. *The **representation relation** $\sqsubset$ is defined as follows: given state $s_y = (\mathrm{L}_y\ \mathrm{R}_y\ \phi_y\ \eta_y\ \mathrm{e}_y\ \mathrm{k}_y)$ and state $s_x = (\mathrm{L}_x\ \mathrm{R}_x\ \phi_x\ \eta_x\ \mathrm{e}_x\ \mathrm{k}_x)$, $s_y \sqsubset s_x$ if and only if $\eta_y = \eta_x$, $\mathrm{e}_y = \mathrm{e}_x$, $\mathrm{k}_y = \mathrm{k}_x$, and there exists a homomorphism $s_y \rightharpoonup_h s_x$ such that*

$$\mathbb{S}(\phi_x \wedge \mathbb{HC}(s_y \rightharpoonup_h s_x)) \quad (5)$$

Definition 18. *A state relation $\mathcal{R}$ is a **bisimulation** if and only if for every state s_x and s_y such that $s_x\ \mathcal{R}\ s_y$, the following two properties hold:*

$$\forall s'_x (s_x \rightarrow_x s'_x \Rightarrow \exists s'_y ((s_y \rightarrow_y s'_y) \wedge (s'_x\ \mathcal{R}\ s'_y))) \quad (6)$$

$$\forall s'_y (s_y \rightarrow_y s'_y \Rightarrow \exists s'_x ((s_x \rightarrow_x s'_x) \wedge (s'_x\ \mathcal{R}\ s'_y))) \quad (7)$$

Note that in the literature it is customary to define bisimulation in terms of a single labeled transition system, whereas for the purposes of this paper the definition of bisimulation refers to a pair of transition relations $\rightarrow_x$ and $\rightarrow_y$ defined by reduction rules. Since it is possible to create a union of the two rule systems $\rightarrow_x \cup \rightarrow_y$, and since none of the transitions in the reduction rules in this paper are labeled, this definition is sufficient for all of the customary properties of bisimulation to apply. For a more detailed treatment on the application of bisimulation to reduction rule systems see [**?**].

6.2 Theorems

The goal of this section is to prove that the representation relation, $\sqsubset$, is a bisimulation as defined in (6) and (7). This proof is accomplished by showing that the requisite homomorphism, $\rightharpoonup_h$, used in the representation relation exists. A bisimulation is a relation over pairs of states such that whenever two states, s_ℓ and s_ς, are related in the bisimulation, $s_\ell \sqsubset s_\varsigma$, every successor, from either state, $s_\ell \rightarrow_\ell s'_\ell$ or $s_\varsigma \rightarrow_\varsigma s'_\varsigma$, has a corresponding mutual successor in the other state such that both of those successors are also related in the bisimulation: $s'_\ell \sqsubset s'_\varsigma$.

If the ς-relation is considered to be a model of the $\ell - relation$ (i.e., a representation of that machine), then the representation relation, as a bisimulation, ensures that the ς-relation is complete in that it any property that can be shown in the ℓ-relation can also be shown in the ς-relation; and further, the representation relation as a bisimulation ensures that the ς-relation is also sound in that any property that can be shown to hold in the ς-relation can also be shown to hold in the ℓ-relation.

The proof of the representation relation as a bisimulation reasons over individual rules in the ς-relation to show that each rule preserves exactness. That is to say that whenever the state on the left of the ς-relation has a corresponding state that it represents in the ℓ-relation, its successor set is both sound and complete relative the ℓ-relation and subject to the representation relation. Intuitively, the ς-relation, at a given state, has a successor to account for every successor from the ℓ-relation at the corresponding state, and the ell-machine has a successor to account for every successor in the ς-relation.

The proof of exactness of each rule relies on the ability to derive from the homomorphism in the representation relation in the current state, a new homomorphism that is sufficient to use in the representation relation that reasons over the successor states. The proof is constructive in that it shows how given a valid homomorphism for the current state, it is possible to derive a new homomorphism to reason about successor states.

There are two slight complications in the proof: the field access rule relies on the S-relation that operates on a different state that then ς-relation and constructing the new homomorphism in the equals-reference rule relies on the incoming heap being location mutual exclusive. The S-relation effectively produces an intermediary state that is the state where uninitialized references are initialized before the field access actually takes place. In essence, a state on the left of the ς-relation that is a field access, undergoes a transition in which its heap is changed to initialize fields. Once the fields are initialized, then the actual field access takes place. The state with the heap that has the initialized fields is the intermediary state between the state on the left of the ς-relation and the state on the right of the ς-relation. The proof reasons separately about this intermediary state to prove that it too is exact.

The equals-reference proof must show that any given reference in the heap is not able to point to two distinct locations at the same time. If the incoming heap is able to point to two valid locations at a given reference at the same time, then the help is non-deterministic, and it is not possible to construct a valid homomorphism from the existing homomorphism: which location should be used in the map? As such, the proof first establishes that ς-relation preserves location mutual exclusive in the incoming heap. Once that is shown, the exactness of the equals reference rule is given.

The final statement that $\sqsubset$ is a bisimulation is in Theorem 9.

Lemma 1 (Exactness of Summarize Rule)**.** *If* $s_s \cong \mathbb{FS}(\rightarrow_\phi, s_0, \pi_n)$ *for symbolic state* $s_s =$ *(*$\mathrm{L}_s\ \mathrm{R}_s\ \phi_s\ \eta\ \mathrm{r}$ *(* \$* $\mathrm{f} \rightarrow \mathrm{k}$*)), initial state* s_0*, and control flow path* π_n*, and if there exists some intermediate state state* s_s' *such that* $(\mathrm{L}_s\ \mathrm{R}_s\ \mathrm{r}\ \mathrm{f}\ \mathrm{C}) \rightarrow_S^* (\mathrm{L}_{s'}\ \mathrm{R}_{s'}\ \phi_{s'}\ \mathrm{r}\ \mathrm{f}\ \mathrm{C})$*, then:*

$$s_s' \cong \{\forall s_\ell' | \exists s_\ell (s_\ell \sqsubset s_s \land (s_\ell \rightarrow_I^* s_\ell'))\}$$

Proof. In order for a state to be equivalent to a set of lazy states, it must be both sound and complete with respect to the set. We will begin the proof by showing completeness, and then finish by demonstrating soundness.

To show completeness, we must show that any state in the set is represented by s_s. The definition of representation requires the both the existence of a homomorphism, and proof that the homomorphism constraint is satisfiable. To show that a homomorphism exits, take any lazy state s_ℓ such that $s_\ell \sqsubset s_s$. By Definition 17, we know $s_\ell =$ ($L_\ell\ R_\ell\ \phi_\ell\ \eta\ r$ (* \$$f \rightarrow k$)). Take any state s_ℓ' where $s_\ell \rightarrow_I^* s_\ell'$, and state s_s' where $s_s \rightarrow_S^* s_s'$. Note that state s_ℓ' has the form: $s_\ell' =$ ($L_{\ell'}\ R_{\ell'}\ \phi_{\ell'}\ \eta\ r$ (* \$$f \rightarrow k$)).Take any location, field pair (l_ℓ f) such that (l_ℓ f) $\in R_\ell^{\leftarrow}$, and let $l_s = h(l_\ell)$. We may classify l_ℓ into one of three ways, based on the values of the R function in each of the states s_ℓ, s_ℓ', s_s, and s_s', and we may define a function $h' : \mathcal{L} \mapsto \mathcal{L}$ based on that classification.

Class 1: $R_\ell(l_\ell,f) = R_{\ell'}(l_\ell,f)$ and $R_s(l_s,f) = R_{s'}(l_s,f)$. Let l_α be the location such that $(\phi_a\ l_\alpha) = L_\ell(R_\ell(l_\ell, f))$. In this case, let $h'(l_\alpha) = h(l_\alpha)$. Since $s_\ell \rightharpoonup_h s_s$, we may surmise that:

$$(\phi_a\ l_\alpha) \in L_{\ell'}(R_{\ell'}(l_\ell, f)) \Rightarrow (\phi_b\ h'(l_\alpha)) \in L_{s'}(R_{s'}(l_s, f))$$

Class 2: $R_\ell(l_\ell,f) = R_{\ell'}(l_\ell,f)$ and $R_s(l_s,f) \neq R_{s'}(l_s,f)$. Since $R_s(l_s,f) \neq R_{s'}(l_s,f)$, the Summarize rule must have altered this reference. A reference created by the Summarize rule has a value set θ_{all} with four subsets: θ_{null}, θ_{new}, θ_{alias}, and θ_{orig}. Because $R_\ell(l_\ell,f) = R_{\ell'}(l_\ell,f)$, we know that the location we want to map to lies in θ_{orig}. Let l_α be the location such that $(\phi_a\ l_\alpha) = L_\ell(R_\ell(l_\ell, f))$, and let $l_{orig} = h(l_\alpha)$. In this case, we let $h'(l_\alpha) = h(l_\alpha)$. Since $(\phi_a\ l_\alpha) \in L_{\ell'}(R_{\ell'}(l_\ell, f))$. Let $l_{orig} = h(l_\alpha)$. We can see that by the Summarize rule $(\phi_b\ l_{orig}) \in L_{s'}(R_{s'}(l_s, f))$, so therefore:

$$(\phi_a\ l_\alpha) \in L_{\ell'}(R_{\ell'}(l_\ell, f)) \Rightarrow (\phi_b\ h'(l_\alpha)) \in L_{s'}(R_{s'}(l_s, f))$$

Class 3: $R_\ell(l_\ell,f) \neq R_{\ell'}(l_\ell,f)$ and $R_s(l_s,f) \neq R_{s'}(l_s,f)$. In this case, there are two possibilities: either the new reference $R_{\ell'}(l_\ell,f)$ points to some location we've seen before l_α, or it points to a previously unobserved location l_β. By establishing which of these possibilities has happened, we can build h'. To construct h', let l_α be any location such that $(\phi_a\ l_\alpha) \in L_{\ell'}(R_{\ell'}(l_\ell, f))$. If there exists ϕ_α such that $(\phi_\alpha\ l_\alpha) \in L_\ell^{\rightarrow}$, let $h'(l_\alpha) = h(l_\alpha)$. Otherwise, let l_β be the location such that $(\phi_b\ l_\beta) \in L_{s'}(R_{s'}(l_s, f))$ and $(\phi_b\ l_\beta) \notin L_s(R_s(l_s, f))$. Now, let $h'(l_\alpha) = l_\beta$. Observe that either way,

$$(\phi_a\ l_\alpha) \in L_{\ell'}(R_{\ell'}(l_\ell, f)) \Rightarrow (\phi_b\ h'(l_\alpha)) \in L_{s'}(R_{s'}(l_s, f))$$

Furthemore, since l_α and l_β are new locations with uninitialized fields, we know that for any field f', $\{(\phi_p\ \bot)\} = L_{\ell'}(R_{\ell'}(l_\alpha, f'))$ and $\{(\phi_p\ \bot)\} = L_{s'}(R_{s'}(l_\beta, f'))$ therefore, we know that:

$$(\phi_p\ l_x) \in L_{\ell'}(R_{\ell'}(l_\alpha, f')) \Rightarrow (\phi_q\ h'(l_x)) \in L_{s'}(R_{s'}(h'(l_\alpha), f))$$

We have now shown that there exists a mapping $h' : \mathcal{L} \mapsto \mathcal{L}$ for all $l_{\ell'} \in L_{\ell'}^{\rightarrow}$ such that:

$$(\phi_a\ l_\alpha) \in L_{\ell'}(R_{\ell'}(l_{\ell'}, f)) \Rightarrow (\phi_b\ h'(l_\alpha)) \in L_{s'}(R_{s'}(l_{\ell'}, f))$$

By Definition 15 we know that $s_\ell' \rightharpoonup_{h'} s_s'$.

It remains to show that $\mathbb{S}(\phi_s' \land \mathbb{HC}(s_\ell' \rightharpoonup_h s_s'))$. For locations in Class 1, no new conjuncts are added to $\mathbb{HC}(s_\ell' \rightharpoonup_h s_s')$, and therefore the satisfiability cannot be changed. For locations in Class 2 or Class 3, the new constraints take either the form $\phi_x \land \phi_{orig}$, or $\phi_x \land (r_f\ op\ r_a) \land (r_f\ op\ r_b) \land$.... Constraints of the form $\phi_x \land \phi_{orig}$ contain terms ϕ_x and ϕ_{orig} which were already conjoined to prior heap constraint, so satisfiability is not affected. In constraints of the form $\phi_x \land (r_f\ op\ r_a) \land (r_f\ op\ r_b) \land$ ..., the term ϕ_x is conjoined to the prior heap constraint, and all the other terms involve the new variable r_f, so satisfiability is not affected. Since the previous heap constraint is satisfiable, and none of the new terms can impact the satisfiability, we know that the new heap constraint must also be satisfiable.

Since the heap constraint is satisfiable, we know that $s_\ell' \sqsubset s_s'$. We have therefore shown that for some summary state s_s and an arbitrary lazy state s_ℓ such that $s_\ell \sqsubset s_s$:

$$(s_\ell \rightarrow_I^* s_\ell' \land s_s \rightarrow_S^* s_s') \Rightarrow s_\ell' \sqsubset s_s' \tag{8}$$

We now prove the reverse case, that s_s^* represents no infeasible states. Suppose that s_s' represents some infeasible state. This means that we represent some lazy state that has some reference r which points somewhere that no place in the feasible set points to. Since we don't change the path condition, all the old references still point exactly to the same places they used to.

So, the problem must be with one of the new references. All of the new references point to either a new location, the null location, the uninitialized location, or some alias. In the Summarize rule, the values and constraints for the new, null, uninitialized, and alias locations are contained in the sets θ_{new}, θ_{null}, $\theta orig$, and θ_{alias}. Since the null, and uninitialized locations are already accounted for by the homomorphism $s_\ell \rightharpoonup_h s_s$, and since a new location was created symmetrically for both s_ℓ' and s_s', the problem must be with some alias location that is part of s_s but not s_ℓ. This means that there must be a feasible path to a target location that does not exist for any lazy heap. So, pick an arbitrary lazy heap containing the location and field in question. If said target location does not exist, then there is no reference in the lazy heap pointing to that location. In the summary heap, the path constraint on the path leading to the undesired target contains an aliasing condition that states that the source reference only points to this target location on condition that the parent reference points there. However, since we already know that no other reference in the lazy heap points there, this condition must be infeasible. Therefore, it is not part of the represented state.

We have a contradiction. Therefore, there is no alias that points somewhere it's not supposed to.

We have now proven that

$$s_\ell^* \sqsubset s_s^* \Rightarrow s_\ell^* \in \{\forall s'_\ell | \exists s_\ell (s_\ell \sqsubset s_s \wedge (s_\ell \rightarrow_I^* s'_\ell))\}$$

This fact, combined with our previous result, proves that

$$s_s^* \cong \{\forall s'_\ell | \exists s_\ell (s_\ell \sqsubset s_s \wedge (s_\ell \rightarrow_I^* s'_\ell))\}$$

□

Lemma 2 (Exactness of Field Access Rule)**.** *If there exists states s_ℓ and s_s such that $s_s \in \mathcal{FA}$ and $s_\ell \sqsubset s_s$, then:*

$$\forall s'_\ell (s_\ell \rightarrow_\ell s'_\ell \Rightarrow \exists s'_s ((s_s \rightarrow_s s'_s) \wedge (s'_\ell \sqsubset s'_s))) \quad (9)$$

and

$$\forall s'_s (s_s \rightarrow_s s'_s \Rightarrow \exists s'_\ell ((s_\ell \rightarrow_\ell s'_\ell) \wedge (s'_\ell \sqsubset s'_s))) \quad (10)$$

Proof. Begin by assuming the conditions stated in Lemma 2. By Lemma 1, the summary intermediate state is equivalent to the set of lazy intermediate states, so we may assume without loss of generality that all of the pertinent fields in s_s are initialized. Take an arbitrary lazy state s_ℓ such that $s_\ell \sqsubset s_s$. Since s_s is exact, $s_\ell = (L_\ell\ R_\ell\ \phi_\ell\ \eta\ r\ (\texttt{*}\ \texttt{\$}f \rightarrow k))$, and $s_\ell \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_n)$. If we apply the state transition functions to achieve states s'_ℓ and s'_s such that $s_\ell \rightarrow_\ell s'_\ell$ and $s_s \rightarrow_s s'_s$, we find that according to the Field Access rule:

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

and

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

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 construct the function h' such that $h' = h$. Observe that since $s_\ell \rightharpoonup_h s_s$, 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 $r = R_\ell(l, f) \Rightarrow r = R_s(h'(l), f)$. Let $\{(\phi'_\ell\ l')\} = L_\ell(R_\ell(l, f))$. Since $\mathbb{S}(\phi_g \wedge \mathbb{HC}(s_\ell \rightharpoonup_h s_s))$ is valid, we know that:

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

From this, we may deduce that:

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

Since r' is the only new addition to L'_ℓ and L'_s, we now know that the assertion above holds for all $l \in \mathcal{L}$. Thus, we have shown that $s'_\ell \rightharpoonup_h s'_s$. Furthermore, since the constraints in $\mathbb{HC}(s'_\ell \rightharpoonup_{h'} s'_s)$ are constructed using conjuncts already present in $\mathbb{HC}(s_\ell \rightharpoonup_h s_s)$, we are guaranteed that $\mathbb{HC}(s'_\ell \rightharpoonup_{h'} s'_s) \Leftrightarrow \mathbb{HC}(s_\ell \rightharpoonup_h s_s)$, and therefore $\mathbb{S}(\phi_g \wedge \mathbb{HC}(s'_\ell \rightharpoonup_{h'} s'_s))$. This fact, and the fact that $\eta_\ell = \eta_s$, $e_\ell = e_s$, $k_\ell = k_s$, means that by Definition 17 we know $s'_\ell \sqsubset s'_s$. We have now shown that:

$$\forall s'_\ell (s_\ell \rightarrow_\ell s'_\ell \Rightarrow \exists s'_s ((s_s \rightarrow_s s'_s) \wedge (s'_\ell \sqsubset s'_s))) \quad (11)$$

Now, suppose that there exists a state s'_i such that $s'_i \sqsubset s'_s$. Since $s'_i \sqsubset s'_s$, then by Definition 17, we know there exists a homomorphism $s'_i \rightharpoonup_{h'} s'_s$, and that $\mathbb{S}(\phi'_i \wedge \mathbb{HC}(s'_i \rightharpoonup_{h'} s'_s))$. From state s'_i, construct state s_i such that

$$\begin{aligned}
s_i &= (L_i\ R_i\ \phi_i\ \eta\ r\ (\texttt{*}\ \texttt{\$}f \rightarrow k)) \\
L_i &= L_{i'} \setminus \{r'\} \\
R_i &= R_{i'} \\
\phi_i &= \phi'_i
\end{aligned}$$

Observe that by virtue of the lazy Field Access rule, $s_i \rightarrow_\ell s'_i$. Now, construct function h_i so that $h_i = h'$. Observe that by Definition 15 $s_i \rightharpoonup_{h_i} s_s$, and that $\mathbb{S}(\phi_i \wedge \mathbb{HC}(s_i \rightharpoonup_{h_i} s_s))$, so $s_i \sqsubset s_s$. Therefore:

$$\forall s'_s (s_s \rightarrow_s s'_s \Rightarrow \exists s'_\ell ((s_\ell \rightarrow_\ell s'_\ell) \wedge (s'_\ell \sqsubset s'_s))) \quad (12)$$

This concludes the proof.

□

Lemma 3 (Exactness of Field Write Rule)**.** *If there exists states s_ℓ and s_s such that $s_s \in \mathcal{FW}$ and $s_\ell \sqsubset s_s$, then:*

$$\forall s'_\ell (s_\ell \rightarrow_\ell s'_\ell \Rightarrow \exists s'_s ((s_s \rightarrow_s s'_s) \wedge (s'_\ell \sqsubset s'_s))) \quad (13)$$

and

$$\forall s'_s (s_s \rightarrow_s s'_s \Rightarrow \exists s'_\ell ((s_\ell \rightarrow_\ell s'_\ell) \wedge (s'_\ell \sqsubset s'_s))) \quad (14)$$

Proof. Begin by assuming the conditions from Lemma 3.

The first step is to show that there exists a state s'_s that is complete with respect to the feasible set. Take state s_s and compute state s'_s such that $s_s \rightarrow_s s'_s$. Take any lazy state s_ℓ such that $s_\ell \sqsubset s_s$, and find state s'_ℓ such that $s_\ell \rightarrow_\ell s'_\ell$. Let l_ℓ be the location such that $\{(\phi_a\ l_\ell)\} = L_\ell(r_x)$ for some ϕ_a. To show that $s'_\ell \sqsubset s'_s$, we need to demonstrate that there exists a function h' such that $s'_\ell \rightharpoonup_{h'} s'_s$, and that $\mathbb{S}(\phi_{s'} \wedge \mathbb{HC}(s'_\ell \rightharpoonup_{h'} s'_s))$. Since $s_h \sqsubset s_s$, we know that there exists a function h such that $s_\ell \rightharpoonup_h s_s$. Let $h' = h$.

First, we consider how $s'_\ell \rightharpoonup_{h'} s'_s$. Let l_α and l_β be arbitrary locations in $L_{\ell'}^{\rightarrow}$ such that $\{(\phi_a\ l_\alpha)\} = L_{\ell'}(R_{\ell'}(l_\beta, f))$, let $\theta = L_s(R_s(h(l_\ell), f))$, and let $\theta' = L'_s(R'_s(h(l_\ell), f))$.

Suppose $l_\beta \neq l_\ell$. In this case either $\theta = \theta'$ or $\theta \neq \theta'$. In the first case, we are guaranteed that the homomorphism works by default. Otherwise, if $\theta \neq \theta'$. We can see from the construction of the set X in the summary Field Write rule that any feasible location in the set θ must also be in the set θ'. Since $s_\ell \sqsubset s_s$, we know that $h(l_\alpha)$ is in θ, and is likewise in θ'. We have now established that in either case where $l_\beta \neq l_\ell$, $(\phi_b\ h(l_\alpha)) \in L_{s'}(R_{s'}(h(l_\beta), f))$.

On the other hand, suppose $l_\beta = l_\ell$. In this case we know that $\{(\phi_a\ l_\alpha)\} = L_{\ell'}(R_{\ell'}(l_\ell, f))$. From the lazy field rule, we can surmise that $(\phi_a\ l_\alpha) \in L_\ell(r)$, and since $s_\ell \sqsubset s_s$, we know that $(\phi_b\ h(l_\alpha)) \in L_s(r)$ for some constraint ϕ_b. Using this fact, we can apply the summary Field Write rule to infer that l_α must be one of the locations in θ', and therefore $(\phi_c\ h(l_\alpha)) \in L_{s'}(R'(l_\ell, f))$

Thus, for arbitrary l_α and l_β:

$$(\phi_a\ l_\alpha) \in L_{\ell'}(R_{\ell'}(l_\beta, f)) \Rightarrow (\phi_b\ h(l_\alpha)) \in L_{s'}(R_{s'}(h(l_\beta), f))$$

Therefore, we have shown that $s'_\ell \rightharpoonup_{h'} s'_s$.

Establishing the fact that $\mathbb{S}(\phi_{s'} \wedge \mathbb{HC}(s'_\ell \rightharpoonup_{h'} s'_s))$ is left as an exercise to the reader (waving hands in the air).

By proving the existence of a valid homomorphism, we have shown that for any state s'_ℓ such that $s_\ell \rightarrow_\ell s'_\ell$, then the state s'_s such that $s_s \rightarrow_s s'_\ell$ represents s'_ℓ. Therefore, $\forall s'_\ell (s_\ell \rightarrow_\ell s'_\ell \Rightarrow \exists s'_s ((s_s \rightarrow_s s'_s) \wedge (s'_\ell \sqsubset s'_s)))$. This concludes the proof of completeness.

To show that s'_s is sound with respect to the feasible set, we use the same argument as in the field read proof: that any state s'_ℓ represented by s'_s must have a counterpart state s_ℓ such that $s_\ell \rightarrow_\ell s'_\ell$ and $s_\ell \sqsubset s_s$. Because s_s is exact, s'_ℓ must be a part of the feasible set. Therefore, $\forall s'_s (s_s \rightarrow_s s'_s \Rightarrow \exists s'_\ell ((s_\ell \rightarrow_\ell s'_\ell) \wedge (s'_\ell \sqsubset s'_s)))$.

□

Lemma 4 ($\rightarrow_S^*$ preserves location mutual exclusion)**.** *Given a location mutual exclusive heap,* $(\mathrm{L}_0\ \mathrm{R}_0)$*, from a summary state with a reference* r *and field* f*, the new heap,* $(\mathrm{L}'\ \mathrm{R}')$*, from the summary initialization machine,* $(\mathrm{L}_0\ \mathrm{R}_0\ \mathrm{r}\ \mathrm{f}) \rightarrow_S^* (\mathrm{L}'\ \mathrm{R}'\ \mathrm{r}\ \mathrm{f})$*, after any number of machine steps, is also location mutual exclusive.*

Proof. Induction over the number of machine steps on the summary initialization machine in Figure 8.

Base Case. The machine takes one step: $(L_0\, R_0\, r f) \rightarrow_S (L_1\, R_1\, r f)$. Let $\Lambda = \mathbb{UN}(L_0, R_0, r, f)$ be the set of uninitialized locations. If $\Lambda = \emptyset$, then the `Summarize (end)` rule is active and $(L_1\, R_1) = (L_0\, R_0)$, which is location mutual exclusive by the initial conditions in the lemma.

If $\Lambda \neq \emptyset$, then the `Summarize` rule is active, and each new constraint location pair must be considered individually. These pairs are partitioned into the sets θ_{null}, θ_{new}, θ_{alias}, and θ_{orig} by the rule.

- Any member of θ_{null}, θ_{new}, and θ_{alias} has a constraint of the form $\phi_x \wedge \ldots$ while any member of θ_{orig} has a constraint of the form $\neg\phi_x \wedge \ldots$. As every $(\phi_{orig}\ l_{orig}) \in L_n(R_n(l_x, f))$ is location mutual exclusive by the initial conditions on $(L_0\ R_0)$ in the lemma statement, members of θ_{orig} are location mutual exclusive with each other and with members in θ_{null}, θ_{new}, and θ_{alias}.
- Any member of θ_{null} has the form $\ldots \wedge r_f = r_{null}$ while any member of θ_{new} and θ_{alias} has the form $\ldots \wedge r_f \neq r_{null} \wedge \ldots$. As θ_{null} is a singleton set, its location is location mutual exclusive from those in θ_{new} and θ_{alias}.
- Any member of θ_{new} has a constraint of the form $\ldots \wedge (\wedge_{(r_a,\, \phi_a,\, l_a) \in \rho} r_f \neq r_a))$ to assert it does not alias anything while any member of θ_{alias} has the form $\ldots \wedge r_f = r_a \wedge \ldots$ for some alias r_a with both partitions reasoning over the same set of aliases ρ. As θ_{new} is a singleton set, its location is location mutual exclusive from any member of θ_{alias}.
- Any member of θ_{alias} has the form

  $$\ldots \wedge r_f = r_a \wedge (\wedge_{(r'_a\ \phi'_a\ l'_a) \in \rho\ (r'_a \neq r_a)} r_f \neq r'_a)$$

  Each is mutually exclusive over the particular alias it asserts.

Every member in $\theta = \theta_{null} \cup \theta_{new} \cup \theta_{alias} \cup \theta_{orig}$ is thus location mutual exclusive making $(L_1\ R_1\ r f)$ location mutual exclusive as well.

Inductive Step. The machine takes n-steps:

$$(L_0\, R_0\, r f) \rightarrow_S (L_1\, R_1\, r f) \rightarrow_S \ldots \rightarrow_S (L_n\, R_n\, r f)$$

By the induction hypothesis, $(L_n\ R_n)$ is location mutual exclusive. This matches the base case, in that the heap on the left side of $\rightarrow_S$ is location mutual exclusive, and by the same argument as in the base case, $(L_{n+1}\ R_{n+1})$ is thus location mutual exclusive. $\square$

Theorem 5 ($\rightarrow_\varsigma$ preserves location mutual exclusion)**.** *Given a summary state, $s_s = (\mathrm{L}_s\ \mathrm{R}_s\ \phi_s\ \eta_s\ \mathrm{e}_s\ \mathrm{k}_s)$, such that $s_s \rightarrow_\varsigma s'_s$, where $s'_s = (\mathrm{L}'_s\ \mathrm{R}'_s\ \phi'_s\ \eta'_s\ \mathrm{e}'_s\ \mathrm{k}'_s)$, if the heap $(\mathrm{L}_s\ \mathrm{R}_s)$ is location mutual exclusive then the heap after the transition, $(\mathrm{L}'_s\ \mathrm{R}'_s)$ is also location mutual exclusive.*

Proof. The proof will proceed inductively.

Base Case: Let s_s be an initial state. By Definition 2, for every reference $r_i \in L_s^{\leftarrow}$, the set $L_s(r_i)$ contains at most one element. Thus, the requirement that $l_\alpha \neq l_\beta$ is never met, and the Theorem is vacuously true.

Inductive Step: Now we will show that if the exclusivity property holds for some state s_s, then it holds for any state s'_s where $s_s \rightarrow_s s'_s$. In order to evaluate whether Theorem 5 holds for state any $s_s\prime$, we must consider the rule that applied during the transition from s_s to $s_s\prime$. There are two broad classes of rules: rules where $L_s \neq L_{s'}$, and rules where $L_s = L_{s'}$. Rules in the $L_s \neq L_{s'}$ class modify the structure of the heap, and must be considered carefully to to consider the impact of those modifications. Only three rules belong to the class $L_s \neq L_{s'}$: Field Access, Field Read, and New.

We begin by considering the Field Access rule. Suppose s_s has the form $(L_s\ R_s\ \phi_s\ \eta\ r$ (* \$ $f \rightarrow k$)). In this case, the relationship between s_s and s'_s is described by the Field Access rule. intermediate state.

We now use this result to show that the mutual exclusion property holds for state s'_s. The relation between the L-function from the final intermediate state L' and the L-function in s'_s is $L_{s'} = L'[r' \mapsto \mathbb{VS}(L', R', r, f, \phi_g)]$, where a new value set is created based on the $\mathbb{VS}$ function. The members of the value set have the form $(\phi \wedge \phi'\ \ l)$. Choose any two distinct members of the value set, $(\phi_\alpha \wedge \phi'_\alpha\ \ l_\alpha)$ and $(\phi_\beta \wedge \phi'_\beta\ \ l_\beta)$. If $\phi_\alpha \neq \phi_\beta$, we know that exclusivity holds because ϕ_α and ϕ_β came from the same value set in s_s, and are therefore exclusive. If $\phi_\alpha = \phi_\beta$, we know that exclusivity holds because ϕ'_α and ϕ'_β came from the same value set in s_s and are therefore exclusive. Thus, the exclusivity property holds for any pair of constraints in the value set. Since the only new value set in $L_{s'}^{\rightarrow}$ is generated by the $\mathbb{VS}$ function, we are guaranteed that if exclusivity holds for s_s, then exclusivity holds for s'_s.

Suppose we have a field write instruction. This case is nearly identical as the field read. In this instruction, a new value set is created. The members of the value set have the form $(\phi \wedge \phi'\ \ l)$. Choose any two distinct members of the value set, $(\phi_\alpha \wedge \phi'_\alpha\ \ l_\alpha)$ and $(\phi_\beta \wedge \phi'_\beta\ \ l_\beta)$. If $\phi_\alpha \neq \phi_\beta$, we know that exclusivity holds because $\phi_\alpha = \neg\phi_\beta$, so ϕ_α and ϕ_β are therefore exclusive. If $\phi_\alpha = \phi_\beta$, we know that exclusivity holds because ϕ'_α and ϕ'_β came from the same value set in s_s and are therefore exclusive. Thus, the exclusivity property holds for any pair of constraints in the value set. Since exclusivity holds for the only new value set in $L_{s'}^{\rightarrow}$, we are guaranteed that if exclusivity holds for s_s, then exclusivity holds for s'_s.

Suppose we have a "new" instruction. In this case, only one value set is added to $L_{s'}^{\rightarrow}$, and that value set contains only one member, so exclusivity holds by default.

Suppose we have any instruction other that a read, write, or new. No machine rule other than those three listed instructions modifies the L function. Therefore, the exclusivity property must hold for s'_s in these cases.

Since the exclusivity property holds for any initial state, and since it holds for any "next" state if the property holds for the previous state, we have proven the property for every symbolic state. $\square$

Corollary 6. *Any state represented by any reachable summary state is a lazy state.*

Proof. By Theorem 5 since the constraints on every reference are mutually exclusive, the heap constraint for any heap with a reference pointing to more than one location will be unsatisfiable, and by Definition 17 will not be represented by the summary state. $\square$

Lemma 7 (Exactness of Reference Compare Rule)**.** *If there exists states s_ℓ and s_s such that $s_s \in \mathcal{RC}$ and $s_\ell \sqsubset s_s$, then:*

$$\forall s'_\ell(s_\ell \rightarrow_\ell s'_\ell \Rightarrow \exists s'_s((s_s \rightarrow_s s'_s) \wedge (s'_\ell\ \sqsubset\ s'_s))) \tag{15}$$

and

$$\forall s'_s(s_s \rightarrow_s s'_s \Rightarrow \exists s'_\ell((s_\ell \rightarrow_\ell s'_\ell) \wedge (s'_\ell\ \sqsubset\ s'_s))) \tag{16}$$

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 brevity we will only show the proofs for the case for the **true** branch.

Proof. Assume there exists states s_ℓ and s_s such that $s_s \in \mathcal{RC}$ and $s_\ell \sqsubset s_s$. Let s'_s be any state such that $s_s \rightarrow_s s_s$ and let $\zeta_T = \forall s'_\ell(s_\ell \rightarrow_\ell s'_\ell)$. Since $s_\ell \sqsubset s_s$, we know that $s_\ell \in \mathcal{RC}$, and that there exists a homomorphism $s_\ell \rightharpoonup_h s_s$ such that

$\mathbb{S}(\phi_s \wedge \mathbb{HC}(s_\ell \rightharpoonup_h s_s))$. We partition ζ_T based on the values of $L_\ell(r_0)$ and $L_\ell(r_1)$ as follows: Let

$$\zeta_t = \zeta_T \setminus \{s_f | (s_f = (L_f\ R_f\ \phi_\ell\ \eta\ e\ k)) \wedge (L_f(r_0) \neq L_f(r_1))\}$$

and let

$$\zeta_f = \zeta_T \setminus \zeta_t$$

Furthermore, there are two possible configurations for s'_s: $(L\ R\ \phi'_g\ \eta\ \textbf{true}\ k)$ and $(L\ R\ \phi'_g\ \eta\ \textbf{false}\ k)$. We now consider the partitions of ζ_T and configurations of s'_s in separate cases.

Case 1: Assume that $L_\ell(r_0) = L_\ell(r_1)$. Compute state s'_ℓ such that $s_\ell \rightarrow_\ell s'_\ell$. In this case, the lazy "equals - references true" rule applied, therefore s'_ℓ is in ζ_t. Observe that by applying Theorem 5, $\phi'_s \wedge \phi_0 \wedge \phi_1$ reduces to ϕ_s. Therefore, $\mathbb{S}(\phi'_s \wedge \mathbb{HC}(s'_\ell \rightharpoonup_h s'_s))$ is true, and by extension, $s'_\ell \sqsubset s'_s$. Since this relation holds for arbitrary $s'_\ell \in \zeta_t$, we now know that

$$((L_\ell(r_0) = L_\ell(r_1)) \wedge (s'_\ell \in \zeta_t)) \Rightarrow s'_\ell \sqsubset s'_s \qquad (17)$$

Case 2: Assume that s'_s has the form $(L\ R\ \phi'_g\ \eta\ \textbf{true}\ k)$, and 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'_s is only a strengthened version of ϕ_s, we know that

$$\{s'_\ell | s'_\ell \sqsubset s'_s\} \subseteq \{s'_\ell | \exists s_\ell (s_\ell \sqsubset s_s) \wedge s_\ell \rightarrow_s s'_\ell\} \qquad (18)$$

Suppose that there exists state s'_i such that $s'_i \sqsubset s'_s$ and $s'_i \notin \zeta_t$. Because of Equation 18, we know that

$$s'_i \in \{s'_\ell | \exists s_\ell (s_\ell \sqsubset s_s) \wedge s_\ell \rightarrow_s s'_\ell\}$$

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 5, we know that either all constraints in the set

$$\{\phi_i \mid \exists \phi_\alpha (\phi_\alpha \in \theta_\alpha) \wedge \phi_i = (\phi_\alpha \wedge \phi_0 \wedge \phi_1)\}$$

are unsatisfiable, or that at least one constraint in the set

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

is valid. Either way, $\mathbb{S}(\phi'_i \wedge \phi_0 \wedge \phi_1)$ is false and s'_s does not represent s'_i. We have a contradiction. Therefore:

$$((s'_s = (L\ R\ \phi'_g\ \eta\ \textbf{true}\ k)) \wedge (s'_\ell \sqsubset s'_s)) \Rightarrow s'_\ell \in \zeta_t \qquad (19)$$

Case 3: Assume that $L_\ell(r_0) \neq L_\ell(r_1)$. 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 Case 1, so we omit it for the sake of brevity. The result for this case is:

$$((L_\ell(r_0) = L_\ell(r_1)) \wedge (s'_\ell \in \zeta_t)) \Rightarrow s'_\ell \sqsubset s'_s \qquad (20)$$

Case 4: Assume that s'_s has the form $(L\ R\ \phi'_g\ \eta\ \textbf{false}\ k)$. The proof for this case is highly similar to the proof for Case 2, so we omit it for the sake of brevity. The result for this case is:

$$s'_s = (L\ R\ \phi'_g\ \eta\ \textbf{false}\ k) \wedge s'_\ell \sqsubset s'_s \Rightarrow s'_\ell \in \zeta_f \qquad (21)$$

Since $\zeta_T = \zeta_t \cup \zeta_f$, we can combine Equation 17 with 20 to find that

$$\forall s'_\ell (s_\ell \rightarrow_\ell s'_\ell \Rightarrow \exists s'_s ((s_s \rightarrow_s s'_s) \wedge (s'_\ell \sqsubset s'_s))) \qquad (22)$$

Likewise, we can combine Equation 19 with Equation 21 to find that

$$\forall s'_s (s_s \rightarrow_s s'_s \Rightarrow \exists s'_\ell ((s_\ell \rightarrow_\ell s'_\ell) \wedge (s'_\ell \sqsubset s'_s))) \qquad (23)$$

□

Lemma 8 (Exactness of New Rule)**.** *If there exists states s_ℓ and s_s such that $s_s \in \mathcal{NW}$ and $s_\ell \sqsubset s_s$, then:*

$$\forall s'_\ell (s_\ell \rightarrow_\ell s'_\ell \Rightarrow \exists s'_s ((s_s \rightarrow_s s'_s) \wedge (s'_\ell \sqsubset s'_s))) \qquad (24)$$

and

$$\forall s'_s (s_s \rightarrow_s s'_s \Rightarrow \exists s'_\ell ((s_\ell \rightarrow_\ell s'_\ell) \wedge (s'_\ell \sqsubset s'_s))) \qquad (25)$$

Proof. The proof is left as an exercise to the reader. □

Theorem 9. *The representation relation $\sqsubset$ is a bisimulation.*

Proof. Take any two states s_ℓ and s_s such that $s_\ell \sqsubset s_s$. If $s_s \in \mathcal{FA} \cup \mathcal{FW} \cup \mathcal{RC} \cup \mathcal{NW}$, then by Lemmas 2, 3, 7, and 8 we know Equations 6 and 7 hold. If s_s has any other form, the heap is not modified for s'_ℓ or s'_s, so then Equations 6 and 7 hold by default. Thus, Equations 6 and 7 hold for all s_ℓ and s_s such that $s_\ell \sqsubset s_s$. By Definition 18, $\sqsubset$ is a bisimulation.

□

Corollary 10. *For any given initial state, the set of possible control flow sequences under the lazy transition relation is exactly the set of possible control flow sequences under the summary transition relation.*

Corollary 11. *For any given initial state, the number of final summary states is exactly the number of possible control flow sequences.*

7. 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.