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 either (a) the use of summaries that over-approximate possible heaps, or (b) by materializing a concrete heap lazily during generalized symbolic execution. In this work, we present a novel heap initialization and analysis technique which takes inspiration from both approaches and constructs precise heap summaries lazily during symbolic execution. Our approach is 1) *scalable*: it reduces the points of non-determinism compared to generalized symbolic execution and explores each control-flow 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. We demonstrate the precision and scalability of our approach by implementing it as an extension to 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.

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

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 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. Overview

In this section we present an overview of the methodology for computing heap summaries lazily for generalized symbolic execution and illustrate its application to a small example. We begin with a brief explanation of the two supporting technologies.

2.1 Generalized Symbolic Execution

The traditional notion of symbolic execution [3, 11] was developed in the context of sequential programs with a fixed number of program variables having primitive types, e.g., integer, Boolean. *Generalized Symbolic Execution (GSE)* [9] extends traditional symbolic execution to enable analysis of heap manipulating software written in commonly used imperative languages such as Java and C++. GSE enables context and flow sensitive analysis of complex and potentially unbounded data structures by constructing the heap "lazily" during symbolic execution, generating potential heap structures based on program semantics. The original GSE algorithm uses lazy initialization in which symbolic execution explores different heap structures by materializing the heap at the first memory access (read) of an uninitialized symbolic object. At this point, a non-deterministic choice point of heap locations is created that includes three cases: (1) a null object, (2) a new instance of the object, and (3) aliases to other type-compatible symbolic objects that have been materialized along the same execution path [9]. Improvements to the original lazy initializtion algorithm include the *lazier* and *lazier#* algorithms [5, 6] which reduce the amount of nondeterminism in the lazy algorithm by delaying initialization, sometimes indefinitely. Although the lazy initialization algorithms extend symbolic analysis to heap manipulating software through partially initialized structures created on-the-fly, the materialization process exacerbates the state space explosion problem in symbolic execution and may generate heap structures which, because of their size, exceed the capabilities of the materialization technique.

2.2 Heap Summarization Techniques

2.3 Our Approach

Consider the code for a (partial) `LinkedList` implementation shown in Figure 1. Generalized Symbolic Execution will materialize the heap by adding non-deterministic choice points representing the different potential heap configurations at each location in the program where a field is read. The number of choice points introduced by each field access is dependent on how many type compatible heap locations were materialized during previous field reads on the current path and on the lazy initialization algorithm used by GSE. The graph shown in Figure 2 illustrates the growth in the symbolic execution state space in terms of time as the number of calls to the `contains()` method which reads the `head` field. The blue and green lines show growth for the lazy and lazier# initialization algorithms respectively. Notice that the runtime, i.e., non-determinism, increases rapidly with only a small number of invocations of the `contains()` method with the lazy and lazier# approaches.

Our technique, blah, builds on the lazy initialization algorithm for Generalized Symbolic Execution by materializing the heap on-the-fly, i.e., "lazily" during symbolic execution, but and heap summariesOur approach is inspired by two state-of-the-art techniques: Generalized Symbolic Execution (GSE) and heap summarization techniques. Like GSE, our approach constructs the heap lazily, i.e., materializing the heap on-the-fly during symbolic execution. However, unlike GSE, we create a compact representation of the heap, i.e., a heap summary, which eliminates isomorphic heap shapes without losing precision. The heap summaries computed by our approach have two important advantages: 1) they are capable of handling arbitrary data structures, including recursive data structures, and, 2) because of lazy initialization, dynamic bounding can be use to bound input data structures.

```
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

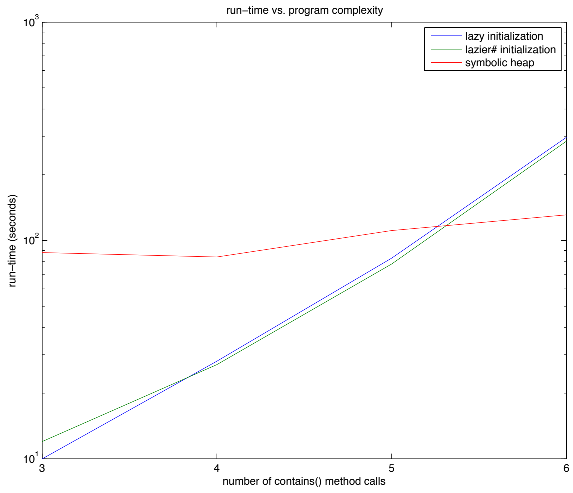


Figure 2. Time versus complexity for the linked list example

3. Preliminaries

Figure 3 defines the surface syntax for the Javalite language [16]. Figure 4 is the machine syntax. Javalite is syntactic machine de-

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

Figure 3. The Javalite surface syntax.

fined 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.

3.1 Environment

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.

3.2 Transition System

In Javalite, states are strings that match a certain pattern.

Definition 1. *The set of* ***states*** $\mathcal{S}$ *is defined as the set of strings matching the pattern s in 4.*

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 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}*.*

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

Definition 3. *A* ***state transition relation*** $\rightarrow_\Phi$ *is a binary relation* $\rightarrow_\Phi \subseteq \mathcal{S} \times \mathcal{S}$*, which relates machine states with successor states. Two important state transition relations are* ***GSE*** $\rightarrow_g$ *and* ***symbolic*** $\rightarrow_\varsigma$*. Each of these use a separate relation for initialization:* $\rightarrow_I$ *for GSE and* $\rightarrow_S$ *for symbolic. All of these transition relations are defined in Figure 10, Figure 6, Figure* **??***, and Figure 7.*

4. Heap: A Bi-partiate Graph

The heap in this work is a labeled bipartite graph consisting of references, r, and locations, l. The machine syntax in Figure 4

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

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

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 $L(r) = \{(\phi\ l)\ ...\}$ 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 set of references.

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

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 $L(r_{null}) = \{(\mathbf{true}\ l_{null})\}$

You can think references as a pointers to sets of locations. More concretely, a reference is an integer that the R function maps to a set of constraint, location pairs that define where it points to. Within a machine state, references are string encodings of integers.

Definition 4. *The set of* ***references*** $\mathcal{R}$ *is defined as the set of natural numbers*

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

In order to make the distinction between different types of references, we partition the set of references using modular arithmetic. Stack references are those references which are created as a result of a field read. The total number of references in a representing state and a represented state are generally not the same. However, the number of references on the stack in either state is always the same.

Definition 5. *The set of* ***stack references*** $\mathcal{R}_t$ *is defined as*

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

.

Input heap references are references that exist prior to program execution in the symbolic input heap. While this set of references may be infinite, they are discovered one at a time via lazy initialization.

Variable lookup

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

New

$$\frac{\begin{array}{l}r = \mathrm{stack}_r() \qquad l = \mathrm{fresh}_l(C) \\ R' = R[\forall f \in \mathit{fields}(\mathrm{C})\ ((l\,f) \mapsto r_{null})] \\ L' = L[r \mapsto \{(\mathbf{true}\ l)\}]\end{array}}{(L\,R\,\phi\,\eta\,(\mathbf{new}\ C)\,k) \rightarrow_J (L'\,R'\,\phi\,\eta\,r\,k)}$$

Field Access(eval)

$$\begin{array}{l}(L\,R\,\phi\,\eta\,(e\,\$f)\,k) \rightarrow_J \\ (L\,R\,\phi\,\eta\,e\,(*\,\$f \rightarrow k))\end{array}$$

Field Write (eval)

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

Equals (l-operand eval)

$$\begin{array}{l}(L\,R\,\phi\,\eta\,(e_0 = e)\,k) \rightarrow_J \\ (L\,R\,\phi\,\eta\,e_0\,(* = e \rightarrow k))\end{array}$$

Equals (r-operand eval)

$$\begin{array}{l}(L\,R\,\phi\,\eta\,v\,(* = e \rightarrow k)) \rightarrow_J \\ (L\,R\,\phi\,\eta\,e\,(v = * \rightarrow k))\end{array}$$

Equals (bool)

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

If-then-else (eval)

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

If-then-else (true)

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

If-then-else (false)

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

Variable Declaration (eval)

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

Variable Declaration

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

Method Invocation (object eval)

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

Method Invocation (arg eval)

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

Method Invocation

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

Variable Assignment (eval)

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

Variable Assignment

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

Begin (no args)

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

Begin (arg0 eval)

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

Begin (argi eval)

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

Begin (argN eval)

$$\begin{array}{l}(L\,R\,\phi\,\eta\,v\,(\mathbf{begin}\ *\ (e_n) \rightarrow k)) \rightarrow_J \\ (L\,R\,\phi\,\eta\,e_n\,(\mathbf{begin}\ *\ () \rightarrow k))\end{array}$$

Begin

$$\begin{array}{l}(L\,R\,\phi\,\eta\,v\,(\mathbf{begin}\ *\ () \rightarrow k)) \rightarrow_J \\ (L\,R\,\phi\,\eta\,v\,k)\end{array}$$

NULL

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

Pop

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

Figure 5. Javalite rewrite rules, indicated by $\rightarrow_J$, that are common to generalized symbolic execution and precise heap summaries.

Definition 6. *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 7. *The set of* ***new heap references*** $\mathcal{R}_f$ *is defined as*

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

.

Definition 8. *For a given function $f : A \mapsto B$, the* ***image*** $f^{\rightarrow}$ *and* ***preimage*** $f^{\leftarrow}$ *are defined as*

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

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

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

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

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

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

A special reference, r_{un}, and location, l_{un}, is introduced to support lazy initialization. The 'un' is to indicate the reference or location is uninitialized at the point of execution.

5. Generating Heap Summaries

5.1 Initialization of Symbolic References

In this section we present the Javalite rewrite rules for the concrete as well as summary initialization of symbolic references. The initialization rules are defined on the bi-partite graph consisting of references and locations. The lazy initialization of symbolic references consists of three key points of non-determinism where each symbolic reference can be initialized non-deterministically to null, a new instance of the symbolic reference, or aliases to symbolic references of the same type previously initialized. The initialization in GSE consists of creating branches in the execution tree for all the non-deterministic choices. On the other hand, the heap summarization approach creates a single branch that contains the summarization for all the initialization in a single bi-partitate graph.

The initialization rules are invoked when an uninitialized field in a symbolic reference is accessed. The function $\mathbb{UN}(L, R, r, f) = \{l\ \ldots\}$ returns constraint-location pairs in which the field f is uninitialized:

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

$$\textsc{Initialize (null)}\quad \frac{\begin{array}{l}\Lambda = \mathbb{UN}(L,R,r,f) \qquad \Lambda \neq \emptyset \qquad (\phi_x\ l_x) = \min_l(\Lambda) \\ r' = \text{fresh}_r() \qquad \theta_{null} = \{(\textbf{true}\ l_{null})\}\end{array}}{(L\ R\ r\ f\ C) \rightarrow_I (L[r' \mapsto \theta_{null}]\ R[(l_x,f) \mapsto r']\ r\ f\ C)}$$

$$\textsc{Initialize (new)}\quad \frac{\begin{array}{l}\Lambda = \mathbb{UN}(L,R,r,f) \qquad \Lambda \neq \emptyset \qquad (\phi_x\ l_x) = \min_l(\Lambda) \\ r' = \text{init}_r() \qquad l' = \text{fresh}_l(C) \\ \rho = \{(r_a\ l_a) \mid \text{isInit}(r_a) \wedge r_a = \min_r(R^{\leftarrow}[l_a]) \wedge \text{type}(l_a) = C\} \\ \theta_{new} = \{(\textbf{true}\ l')\} \\ R' = R[\forall f \in \mathit{fields}(\mathrm{C})\ ((l'\ f) \mapsto r_{un})]\end{array}}{(L\ R\ r\ f\ C) \rightarrow_I (L[r' \mapsto \theta_{new}]\ R'[(l_x,f) \mapsto r']\ r\ f\ C)}$$

$$\textsc{Initialize (alias)}\quad \frac{\begin{array}{l}\Lambda = \mathbb{UN}(L,R,r,f) \qquad \Lambda \neq \emptyset \qquad (\phi_x\ l_x) = \min_l(\Lambda) \\ r' = \text{fresh}_r() \\ \rho = \{(r_a\ l_a) \mid \text{isInit}(r_a) \wedge r_a = \min_r(R^{\leftarrow}[l_a]) \wedge \text{type}(l_a) = C\} \\ (r_a\ l_a) \in \rho \qquad \theta_{alias} = \{(\textbf{true}\ l_a)\}\end{array}}{(L\ R\ r\ f\ C) \rightarrow_I (L[r' \mapsto \theta_{alias}]\ R[(l_x,f) \mapsto r']\ r\ f\ C)}$$

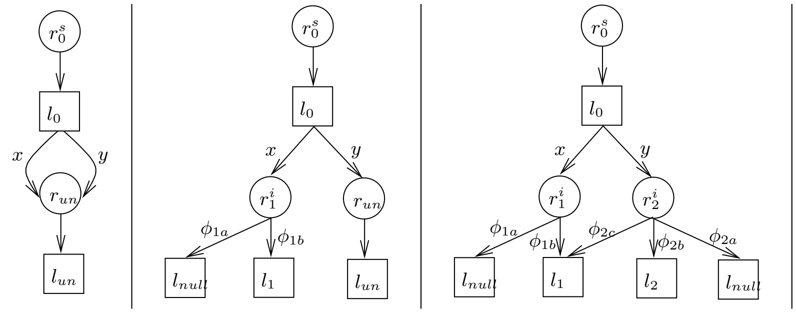
$$\textsc{Initialize (end)}\quad \frac{\Lambda = \mathbb{UN}(L,R,r,f) \qquad \Lambda = \emptyset}{(L\ R\ r\ f\ C) \rightarrow_I (L\ R\ r\ f\ C)}$$

Figure 6. The initialization machine, $s ::= (L\ R\ \phi_g\ r f)$, with $s \rightarrow_I^* s' = s \rightarrow_I \cdots \rightarrow_I s' \rightarrow_I s'$.

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

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

Figure 7. The summary machine, $s ::= (L\ R\ r\ f\ C)$, with $s \rightarrow_S^* s' = s \rightarrow_S \cdots \rightarrow_S s' \rightarrow_S s'$.



$$\begin{array}{l}\rho := \{(r_1^i, r_1^i \neq r_{null}, l_1\} \\ \theta_{null} := \{(r_2^i = r_{null}, l_{null})\} \\ \theta_{new} := \{(r_2^i \neq r_{null} \wedge r_2^i \neq r_1^i, l_2)\} \\ \theta_{alias} := \{(r_1^i \neq r_{null} \wedge r_2^i \neq r_{null} \wedge r_2^i = r_1^i, l_1)\} \\ \theta_{orig} := \{\} \\ \phi_{1a} := r_1^i = r_{null} \\ \phi_{1b} := r_1^i \neq r_{null} \\ \phi_{2a} := r_2^i = r_{null} \\ \phi_{2b} := r_2^i \neq r_{null} \wedge r_2^i \neq r_1^i \\ \phi_{2c} := r_2^i \neq r_{null} \wedge r_2^i = r_1^i\end{array}$$

Figure 8. initialize this.x and this.y

where $\mathbb{S}(\phi)$ returns true if ϕ is satisfiable. Intutively, for the reference, *r*, it constructs the set, θ, that contains all contraint-location pairs that point to the field *f* and *f* points to l_{un}. The cardinality of the set, θ 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.

The rules in Figure 6 present the rewrite rules for the concrete initialization of symbolic heap objects. These rules are invoked until a fix pointed is reached.

The initialize (null) rewrite rule in Figure 6 first checks that the field, *r* is uninitialized. The fresh method returns a new input heap reference from the partition

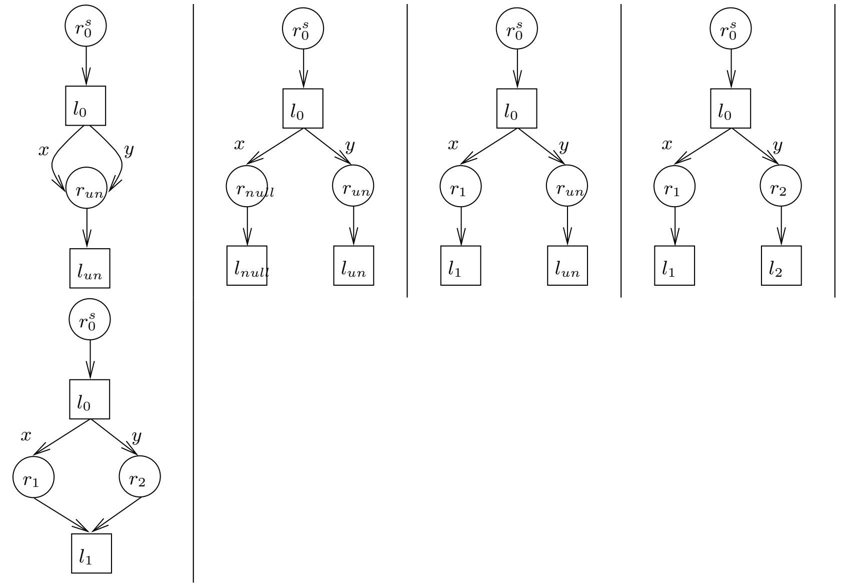


Figure 9. initialize this.x and this.y

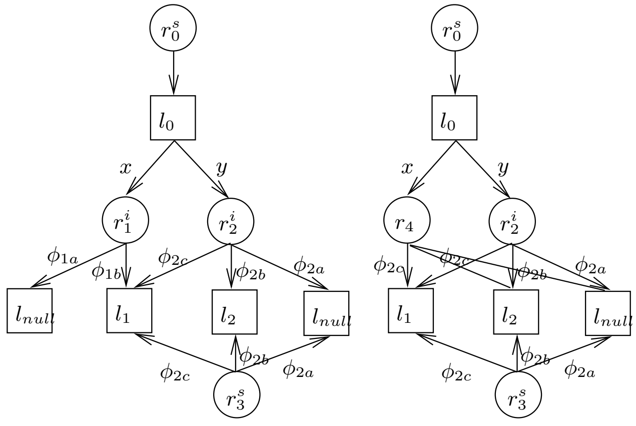


Figure 12. field access for this.y and field write for this.x = this.y

$$\Psi_x = \{(true, l_0, r_1^i)\}$$
$$ST(L, r_3^s, \phi, \phi_g)$$
$$\theta = \{(\phi_{2a}\ l_{null})(\phi_{2b}\ l_2)(\phi_{2c}\ l_1)\}$$
$$ST(L, r_0, \phi, \phi_g)$$
$$\theta = \{\}$$

Figure 13. field write for this.x = this.y

$$L(r_1^i) = \{(\phi_{1a}\ l_{null})\ (\phi_{1b}\ l_1)\}$$
$$L(r_2^i) = \{(\phi_{2a}\ l_{null}),\ (\phi_{2b}\ l_2),\ (\phi_{2c}\ l_1)\}$$
$$\theta_0 = \{\}$$
$$\theta_1 = \{\phi_{2b}\}$$

Equals true

$$\theta_\alpha = \{(\phi_{1a}\ \wedge\ \phi_{2a})(\phi_{1b}\ \wedge\ \phi_{2c})\}$$
$$\phi' = \mathit{true} \wedge [(\phi_{1a}\ \wedge\ \phi_{2a}) \vee (\phi_{1b}\ \wedge\ \phi_{2c})] \wedge \neg\phi_{2b}$$

Equals false

$$\theta_\alpha = \{(\phi_{1a}\ \implies\ \neg\phi_{2a})(\phi_{1b}\ \implies\ \neg\phi_{2c})\}$$
$$\phi' = \mathit{true} \wedge (\phi_{1a}\ \implies\ \neg\phi_{2a}) \wedge (\phi_{1b}\ \implies\ \neg\phi_{2c}) \wedge \phi_{2b}$$

Figure 15. equals true for this.x == this.y

5.2 Accessing and Writing to Field References

Definition 9. *The function* $\mathbb{VS}(L, R, \phi_g, r, f)$ *constructs the value-set given a heap, reference, and desired field:*

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

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

Definition 10. *The strengthen function* $\mathbb{ST}(\mathrm{L}, \mathrm{r}, \phi, \phi_g)$ *strengthens every constraint from the reference* r *with* ϕ *and keeps only location-constraint pairs that are satisfiable after this strengthening with the inclusion of the global heap constraint* ϕ_g*:*

$$\begin{aligned}\mathbb{ST}(\mathrm{L}, \mathrm{r}, \phi, \phi_g) \quad = \quad & \{(\phi \wedge \phi'\ \mathrm{l}') \mid \\ & (\phi'\ \mathrm{l}') \in \mathrm{L}(\mathrm{r}) \wedge \mathbb{S}(\phi \wedge \phi' \wedge \phi_g)\}\end{aligned}$$

5.3 Equality and InEquality of References

6. Proofs

6.1 Heap Properties

Definition 11. *A heap,* (L R)*, is* ***deterministic*** *if and only if*

$$\begin{aligned}& \forall \mathrm{r} \in \mathrm{L}^{\leftarrow}\ (\forall(\phi\ \mathrm{l}), (\phi'\ \mathrm{l}') \in \mathrm{L}(r)\ (\\ & \quad (\mathrm{l} \neq \mathrm{l}' \vee \phi \neq \phi') \Rightarrow (\phi \wedge \phi' = \textbf{\textit{false}}))\end{aligned}$$

At times, it is useful to classify states in terms of patterns that the state strings match. In concrete terms, this is similar to asking what will be the next instruction to execute. For example, we know that left-hand states matching the pattern $(L\ R\ \phi_g\ \eta\ r\ (\texttt{*}\ \$f \to k))$ only appear in the Field Access rule.

Definition 12. *The universe of reachable states is* S*. The universe is further partitioned into states that activate different rules.*

FIELD ACCESS

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

FIELD WRITE

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

Figure 10. GSE with lazy initialization indicated by $\rightarrow_g = \rightarrow_\ell \cup \rightarrow_J$.

FIELD ACCESS

$$\frac{\begin{array}{l}\exists(\phi\ l) \in L(r)\ (l \neq l_{null} \wedge \mathbb{S}(\phi \wedge \phi_g)) \\ \theta = \{\phi \mid (\phi\ l_{null}) \wedge \mathbb{S}(\phi \wedge \phi_g)\} \\ \{C\} = \{C \mid \exists(\phi\ l) \in L(r)\ (C = \text{type}(l,f))\} \\ (L\ R\ r\ f\ C) \rightarrow_S^* (L'\ R'\ r\ f\ C) \qquad r' = \text{stack}_r() \\ \phi'_g = \phi_g \wedge (\wedge_{\phi \in \theta} \neg\phi)\end{array}}{(L\ R\ \phi_g\ \eta\ r\ (\textbf{*}\ \$ f \rightarrow k)) \rightarrow_{FA} (L'[r' \mapsto \mathbb{VS}(L', R', r, f, \phi_g)]\ R'\ \phi_g\ \eta\ r'\ k)}$$

FIELD WRITE

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

Figure 11. Precise symbolic heap summaries from symbolic execution indicated by $\rightarrow_\varsigma = \rightarrow_{FA} \cup \rightarrow_{FW} \cup \rightarrow_{EQ} \cup \rightarrow_J$.

EQUALS (REFERENCE-TRUE)

$$\frac{L(r_0) = L(r_1) \qquad \phi' = (\phi \wedge r_0 = r_1)}{\begin{array}{c}(L\ R\ \phi\ \eta\ r_0\ (r_1 \textbf{ = *} \rightarrow k)) \rightarrow_\ell \\ (L\ R\ \phi'\ \eta\ \textbf{true}\ k)\end{array}}$$

EQUALS (REFERENCE-FALSE)

$$\frac{L(r_0) \neq L(r_1) \qquad \phi' = (\phi \wedge r_0 \neq r_1)}{\begin{array}{c}(L\ R\ \phi\ \eta\ r_0\ (r_1 \textbf{ = *} \rightarrow k)) \rightarrow_\ell \\ (L\ R\ \phi'\ \eta\ \textbf{false}\ k)\end{array}}$$

Figure 14. GSE with lazy initialization indicated by $\rightarrow_g = \rightarrow_\ell \cup \rightarrow_J$.

- *Field Access,* $S_{FA} = \{(\mathrm{L\ R}\ \phi\ \eta\ \mathrm{e\ k}) \in S \mid \exists \mathrm{r}\ (\mathrm{e} = \mathrm{r}) \wedge \exists \mathrm{f}, \mathrm{k}_0\ (\mathrm{k} = (\textbf{*}\ \$\ \mathrm{f} \rightarrow \mathrm{k}_0))\}$
- *Field Write,* $S_{FW} = \{(\mathrm{L\ R}\ \phi\ \eta\ \mathrm{e\ k}) \in S \mid \exists \mathrm{r}\ (\mathrm{e} = \mathrm{r}) \wedge \exists \mathrm{x}, \mathrm{f}, \mathrm{k}_0\ (\mathrm{k} = (\mathrm{x}\ \$\ \mathrm{f} \textbf{ := *} \ \rightarrow\ \mathrm{k}_0))\}$
- *Equals,* $S_{EQ} = \{(\mathrm{L\ R}\ \phi\ \eta\ \mathrm{e\ k}) \in S \mid \exists \mathrm{r}_0(\mathrm{e} = \mathrm{r}_0) \wedge \exists r_1, k_1(\mathrm{k} = (\mathrm{r}_1 \textbf{ = *} \rightarrow \mathrm{k}_1))\}$
- *New,* $S_N = \{(\mathrm{L\ R}\ \phi\ \eta\ \mathrm{e\ k}) \in S \mid \exists \mathrm{C}\ (\mathrm{e} = (\textbf{\textit{new}}\ \mathrm{C}))\}$

Definition 13. *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*

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

We will later be concerned with establishing whether one state represents another state. We want to say that one state represents another state if equivalent paths lead out from each state. This path-centric notion of equivalence is known as functional equivalence. In establishing functional equivalence between states, it is important to determine whether the heaps within the states are themselves functionally equivalent. Two heaps are functionally equivalent if the same sequence of field accesses in each heap produces equivalent results. We define heap functional equivalence using a co-inductive definition of homomorphism over the access paths in the heaps.

Definition 14. *A* ***homomorphism*** $s_p \rightharpoonup_h s_q$*, from state* $s_p = (\mathrm{L}_p\ \mathrm{R}_p\ \phi_p\ \eta_p\ \mathrm{e}_p\ \mathrm{k}_p)$ *to state* $s_q = (\mathrm{L}_q\ \mathrm{R}_q\ \phi_q\ \eta_q\ \mathrm{e}_q\ \mathrm{k}_q)$*, is defined as follows:*

$$\begin{array}{l} s_p \rightharpoonup_h s_q \Leftrightarrow \\ \quad \exists h : \mathcal{L} \mapsto \mathcal{L}\ (\forall \mathrm{l}_\alpha\ (\forall \mathrm{l}_\beta\ (\forall f \in \mathcal{F}(\exists \phi_\alpha\ (\exists \phi_\beta\ (\\ \qquad (\phi_\alpha\ \mathrm{l}_\alpha) \in \mathrm{L}_p(\mathrm{R}_p(\mathrm{l}_\beta, f)) \Rightarrow (\phi_\beta\ h(\mathrm{l}_\alpha)) \in \mathrm{L}_q(\mathrm{R}_q(h(\mathrm{l}_\beta), f)) \\ \qquad\quad)))))) \end{array}$$

Since the access paths in any given heap are bound by certain constraints, to preserve control flow equivalence we must establish whether the collection of any constraints in a given heap are collectively feasible. The homomorphism constraint is the conjunction of all constraints in the image of the represented heap in the representer heap.

Definition 15. *Given the homomorphism* $(\mathrm{L}_p\ \mathrm{R}_p) \rightharpoonup_h (\mathrm{L}_q\ \mathrm{R}_q)$*, the* ***homomorphism constraint*** $\mathbb{HC}((\mathrm{L}_p\ \mathrm{R}_p) \rightharpoonup_h (\mathrm{L}_q\ \mathrm{R}_q))$ *is defined as:*

$$\mathbb{HC}((\mathrm{L}_p\ \mathrm{R}_p) \rightharpoonup_h (\mathrm{L}_q\ \mathrm{R}_q)) =$$

$$\bigwedge\{\phi_b \mid \exists(\phi_a\ l) \in \mathrm{L}_p^{\rightarrow}((\phi_b\ h(l)) \in \mathrm{L}_q^{\rightarrow})\}$$

EQUALS (REFERENCES-TRUE)

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

EQUALS (REFERENCES-FALSE)

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

Figure 16. Precise symbolic heap summaries from symbolic execution indicated by $\rightarrow_\varsigma = \rightarrow_{FA} \cup \rightarrow_{FW} \cup \rightarrow_{EQ} \cup \rightarrow_J$.

The representation relation combines the previously established notions of heap homomorphism and feasibility with the added constraint that the variables, expressions, and continuation strings must match between the pairs of states.

Definition 16. *The **representation relation** $\sqsubset$ is defined as follows: given state $s_p = (\mathrm{L}_p\ \mathrm{R}_p\ \phi_p\ \eta_p\ \mathrm{e}_p\ \mathrm{k}_p)$ and state $s_q = (\mathrm{L}_q\ \mathrm{R}_q\ \phi_q\ \eta_q\ \mathrm{e}_q\ \mathrm{k}_q)$, $s_p \sqsubset s_q$ if and only if $\eta_p = \eta_q$, $\mathrm{e}_p = \mathrm{e}_q$, $\mathrm{k}_p = \mathrm{k}_q$, and there exists a homomorphism $(\mathrm{L}_p\ \mathrm{R}_p) \rightharpoonup_h (\mathrm{L}_q\ \mathrm{R}_q)$ such that*

$$\mathbb{S}(\phi_q \wedge \mathbb{HC}(s_p \rightharpoonup_h s_q)) \qquad (5)$$

The represented relation is extended to sets of states P and Q as

$$P \sqsubset Q \iff \forall q \in Q\ (\forall p \in S\ (p \sqsubset q \Rightarrow p \in P))$$

Definition 17. *A state relation $p \rightarrow_x q$ is extended to sets of states $P \hookrightarrow_x Q$ as*

$$P \hookrightarrow_x Q \iff \forall q \in S\ (p \rightarrow_x q \Rightarrow q \in Q)$$

The $\rightarrow_g$ relations over sets of states is The $\rightarrow_\varsigma$ relation extended to sets of states

Definition 18. *The functional associated to bisimulation, $F_\sim(\sqsubset)$, is the set of all pairs of sets of states $(P\ Q)$ such that*

$$\forall p \in P\ (\forall p'\ (p \rightarrow_g p' \Rightarrow \exists q'\ ((q \rightarrow_\varsigma q') \wedge (p' \sqsubset q')))) \quad (6)$$

$$\forall q \in Q\ (\forall q'\ (q \rightarrow_\varsigma q' \Rightarrow \exists p'\ ((p \rightarrow_g p') \wedge (p' \sqsubset q')))) \quad (7)$$

The bisimularity relation is the greatest fixed point of the functional.

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. A bisimulation is a relation over pairs of states such that whenever two states, s_g and s_ς, are related in the bisimulation, $s_g \sqsubset s_\varsigma$, every successor, from either state, $s_g \rightarrow_g s'_g$ 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: $s'_g \sqsubset s'_\varsigma$.

If $\rightarrow_\varsigma$ is considered to be a model of the $\rightarrow_g$ (i.e., a representation of that machine), then the representation relation, as a bisimulation, ensures that the $\rightarrow_\varsigma$ is complete in that any property that can be shown in states related on $\rightarrow_g$ can also be shown in the $\rightarrow_\varsigma$; and further, the representation relation as a bisimulation ensures that the $\rightarrow_\varsigma$ is also sound in that any property that can be shown to hold in states related by $\rightarrow_\varsigma$ can also be shown to hold in the $\rightarrow_g$.

The proof of the representation relation as a bisimulation reasons over individual rules in $\rightarrow_\varsigma$ to show that for each rule, the representation relation, $\sqsubset$, exists. The heart of the representation relation is the homomorphism that maps locations in one heap to locations in the other heap. The proof reasons over of each rule and as mentioned previously, and derives from the current homomorphism in the representation relation for the current states, a new homomorphism that is sufficient to use in the a new representation relation that includes the successor states. The proof is constructive in that it shows how given a valid homomorphism for the current states, it is possible to derive a new homomorphism that includes successor states. With such a new homomorphism, it is possible to state that for $\rightarrow_\varsigma$, when restricted to a specific rule (i.e., that is the only rule available in the relation), the representation relation is a bisimulation for the restricted $\rightarrow_\varsigma$. If such a bisimulation exists for all the individual rules, then it exists for the rules collectively.

There are two slight complications in the proof: first, the field access rule relies on $\rightarrow^*_S$ that operates on a different state than $\rightarrow_\varsigma$; and second, constructing the new homomorphism in the equals-reference rule relies on the incoming heap being deterministic. The relation $\rightarrow^*_S$ 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 $\rightarrow_\varsigma$ 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 $\rightarrow_\varsigma$ and the state on the right of $\rightarrow_\varsigma$. 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 $\rightarrow_\varsigma$ preserves determinism when the incoming heap is deterministic. Once that is shown, the exactness of the equals reference rule is given.

The final statement that $\sqsubset$ is a bisimulation is in Theorem 11.

Lemma 1 ($\sqsubset$ is a bisimulation for $\rightarrow_I$ and $\rightarrow_S$)**.** *If* $s_\varsigma \cong \mathbb{FS}(\rightarrow_\phi, s_0, \pi_n)$ *for symbolic state* s_ς = (L$_\varsigma$ R$_s$ ϕ_ς η r *(* $ f* $\rightarrow$ k*))*, *initial state* s_0, *and control flow path* π_n, *and if there exists some intermediate state state* s'_ς *such that* (L$_\varsigma$ R$_s$ r f C) $\rightarrow^*_S$ (L$_{s'}$ R$_{s'}$ $\phi_{s'}$ r f C), *then:*

$$s'_\varsigma \cong \{\forall s'_g | \exists s_g (s_g \sqsubset s_\varsigma \land (s_g \rightarrow^*_I s'_g))\}$$

Proof. In order for a state to be equivalent to a set of GSE 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_ς. 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 GSE state s_g such that $s_g \sqsubset s_\varsigma$. By Definition 16, we know s_g = (L_g R_g ϕ_g η r (* $ $f \rightarrow$ k)). Take any state s'_g where $s_g \rightarrow^*_I s'_g$, and state s'_ς where $s_\varsigma \rightarrow^*_S s'_\varsigma$. Note that state s'_g has the form: s'_g = ($L_{g'}$ $R_{g'}$ $\phi_{g'}$ η r (* $ $f \rightarrow k$)).Take any location, field pair (l_g f) such that (l_g f) $\in R^{\leftarrow}_g$, and let $l_\varsigma = h(l_g)$. We may classify l_g into one of three ways, based on the values of the R function in each of the states s_g, s'_g, s_ς, and s'_ς, and we may define a function $h' : \mathcal{L} \mapsto \mathcal{L}$ based on that classification.

Class 1: $R_g(l_g,f) = R_{g'}(l_g,f)$ and $R_s(l_\varsigma,f) = R_{s'}(l_\varsigma,f)$. Let l_α be the location such that $(\phi_a\ l_\alpha) = L_g(R_g(l_g, f))$. In this case, let $h'(l_\alpha) = h(l_\alpha)$. Since $s_g \rightharpoonup_h s_\varsigma$, we may surmise that:

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

Class 2: $R_g(l_g,f) = R_{g'}(l_g,f)$ and $R_s(l_\varsigma,f) \neq R_{s'}(l_\varsigma,f)$. Since $R_s(l_\varsigma,f) \neq R_{s'}(l_\varsigma,f)$, 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 $R_g(l_g,f) = R_{g'}(l_g,f)$, we know that the location we want to map to lies in θ_{orig}. Let l_α be the location such that $(\phi_a\ l_\alpha) = L_g(R_g(l_g, f))$, and let $l_{orig} = h(l_\alpha)$. In this case, we let $h'(l_\alpha) = h(l_\alpha)$. Since $(\phi_a\ l_\alpha) \in L_{g'}(R_{g'}(l_g, f))$. Let $l_{orig} = h(l_\alpha)$. We can see that by the Summarize rule $(\phi_b\ l_{orig}) \in L_{s'}(R_{s'}(l_\varsigma, f))$, so therefore:

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

Class 3: $R_g(l_g,f) \neq R_{g'}(l_g,f)$ and $R_s(l_\varsigma,f) \neq R_{s'}(l_\varsigma,f)$. In this case, there are two possibilities: either the new reference $R_{g'}(l_g,f)$ 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 $(\phi_a\ l_\alpha) \in L_{g'}(R_{g'}(l_g, f))$. If there exists ϕ_α such that $(\phi_\alpha\ l_\alpha) \in L^{\rightarrow}_g$, let $h'(l_\alpha) = h(l_\alpha)$. Otherwise, let l_β be the location such that $(\phi_b\ l_\beta) \in L_{s'}(R_{s'}(l_\varsigma, f))$ and $(\phi_b\ l_\beta) \notin L_s(R_s(l_\varsigma, f))$. Now, let $h'(l_\alpha) = l_\beta$. Observe that either way,

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

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

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

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

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

By Definition 14 we know that $s'_g \rightharpoonup_{h'} s'_\varsigma$.

It remains to show that $\mathbb{S}(\phi'_\varsigma \land \mathbb{HC}(s'_g \rightharpoonup_h s'_\varsigma))$. For locations in Class 1, no new conjuncts are added to $\mathbb{HC}(s'_g \rightharpoonup_h s'_\varsigma)$, 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 $s'_g \sqsubset s'_\varsigma$. We have therefore shown that for some symbolic state s_ς and an arbitrary GSE state s_g such that $s_g \sqsubset s_\varsigma$:

$$(s_g \rightarrow^*_I s'_g \land s_\varsigma \rightarrow^*_S s'_\varsigma) \Rightarrow s'_g \sqsubset s'_\varsigma \qquad (8)$$

We now prove the reverse case, that s^*_ς represents no infeasible states. Suppose that s'_ς represents some infeasible state. This means that we represent some GSE 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}, θ_{orig}, and θ_{alias}. Since the null, and uninitialized locations are already accounted for by the homomorphism $s_g \rightharpoonup_h s_\varsigma$, and since a new location was created symmetrically for both s'_g and s'_ς, the problem must be with some alias location that is part of s_ς but not s_g. This means that there must be a feasible path to a target location that does not exist for any GSE heap. So, pick an arbitrary GSE heap containing the location and field in question. If said target location does not exist, then there is no reference in the GSE heap pointing to that location. In the symbolic 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 GSE 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

$$s^*_g \sqsubset s^*_\varsigma \Rightarrow s^*_g \in \{\forall s'_g | \exists s_g (s_g \sqsubset s_\varsigma \land (s_g \rightarrow^*_I s'_g))\}$$

This fact, combined with our previous result, proves that

$$s^*_\varsigma \cong \{\forall s'_g | \exists s_g (s_g \sqsubset s_\varsigma \land (s_g \rightarrow^*_I s'_g))\}$$

□

Lemma 2 (FIELD ACCESS preserves $\sqsubset\ \subseteq F_\sim(\sqsubset)$)**.** *If two field access states are related in the represented by relation, then they are also related in the functional associated to bisumlation.*

$$\forall p \in S_{FA}\ (\forall q \in S_{FA}\ (p \sqsubset q \Rightarrow (p\ q) \in F_\sim(\sqsubset)))$$

Proof. Proof by contradiction: assume $p \sqsubset q \land (p\ q) \notin F_\sim(\sqsubset)$.

The case where neither p nor q have successors is trivially satisfied by Definition 18 since the conditions for inclusion in $F_{\sim}(\sqsubset)$ are vacuously met. That is a contradiction.

Consider now the case where p and q have successor states. The statement $p \sqsubset q$ (Definition 16) means that p and q are only differentiated by their heaps and global constraints since the environment (η), expression (e), and continuation (k) are the same in both states.

Ignoring the heaps, and given that p and q currently have the same environment, expression, and continuation, all successors of p related by FIELD ACCESS in $\rightarrow_g$ (Figure 10) have the same unchanged environment from p, η, new expression $r' = \mathtt{stack}_r()$, and continuation k from the old continuation in p having completed the field access. Similarly, the one successor of q related by $\rightarrow_\varsigma$ (Figure 11), has that same η, $r' = \mathtt{stack}_r()$, and k. As such, every successor of p/q has a matching successor of q/p that agrees on the environment, expression, and continuation meeting the first condition necessary to relate the successors in $\sqsubset$.

Turning now to the heaps, FIELD ACCESS, in $\rightarrow_g$ and $\rightarrow_\varsigma$, initializes uninitialized locations in the heap on reference r for field f. The heaps in p and q are still homomorphic after initialization, and the heap constraint in the homomorphism is still valid by Lemma 1. Let $(L'_p\ R'_p) \rightharpoonup_h (L'_q\ R'_q)$ be those new heaps and the homomorphism after initialization on the heap in p with $\rightarrow_I^*$ (Figure 6) and the heap in q with $\rightarrow_S^*$ (Figure 7).

After initialization, FIELD ACCESS for $\rightarrow_g$ relates a new heap in any successor state as $(L'_p[r' \mapsto (\phi'\ l')]\ R'_p)$, and for $\rightarrow_\varsigma$ it relates the heap in the only successor state as $(L'_q[r' \mapsto \mathbb{VS}(L'_q, R'_q, r, f, \phi_g)]\ R'_q)$; each version adding in the constraint-location pair or value set respectively on the same reference r'.

The functional in Definition 18 requires and forward (6) and backward (7) relationship between successor heaps.

(6) (7)

We now show that $s'_g \sqsubset s'_\varsigma$. Since η, e, and k are identical between s'_ς and s'_g, the first condition is met by default. Now we construct the function h' such that $h' = h$. Observe that since $s_g \rightharpoonup_h s_\varsigma$, and since R_g and R_ς are unchanged from states s_g to s'_g and s_ς to s'_ς respectively, we are guaranteed that $r = R_g(l, f) \Rightarrow r = R_\varsigma(h'(l), f)$. Let $\{(\phi'_g\ l')\} = L_g(R_g(l, f))$. Since $\mathbb{S}(\phi_g \wedge \mathbb{HC}(s_g \rightharpoonup_h s_\varsigma))$ is valid, we know that:

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

From this, we may deduce that:

$$(\phi_g\ l) \in L'_g(r') \Rightarrow (\phi_\varsigma \wedge \phi'_\varsigma\ h'(l)) \in L'_\varsigma(r')$$

Since r' is the only new addition to L'_g and L'_ς, we now know that the assertion above holds for all $l \in \mathcal{L}$. Thus, we have shown that $s'_g \rightharpoonup_h s'_\varsigma$. Furthermore, since the constraints in $\mathbb{HC}(s'_g \rightharpoonup_{h'} s'_\varsigma)$ are constructed using conjuncts already present in $\mathbb{HC}(s_g \rightharpoonup_h s_\varsigma)$, we are guaranteed that $\mathbb{HC}(s'_g \rightharpoonup_{h'} s'_\varsigma) \Leftrightarrow \mathbb{HC}(s_g \rightharpoonup_h s_\varsigma)$, and therefore $\mathbb{S}(\phi_g \wedge \mathbb{HC}(s'_g \rightharpoonup_{h'} s'_\varsigma))$. This fact, and the fact that $\eta_g = \eta_s$, $e_g = e_s$, $k_g = k_s$, means that by Definition 16 we know $s'_g \sqsubset s'_\varsigma$. We have now shown that:

$$\forall s'_g(s_g \rightarrow_g s'_g \Rightarrow \exists s'_\varsigma((s_\varsigma \rightarrow_\varsigma s'_\varsigma) \wedge (s'_g \sqsubset s'_\varsigma))) \tag{9}$$

Now, suppose that there exists a state s'_i such that $s'_i \sqsubset s'_\varsigma$. Since $s'_i \sqsubset s'_\varsigma$, then by Definition 16, we know there exists a homomorphism $s'_i \rightharpoonup_{h'} s'_\varsigma$, and that $\mathbb{S}(\phi'_i \wedge \mathbb{HC}(s'_i \rightharpoonup_{h'} s'_\varsigma))$. From state s'_i, construct state s_i such that

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

Observe that by virtue of the GSE Field Access rule, $s_i \rightarrow_g s'_i$. Now, construct function h_i so that $h_i = h'$. Observe that by Definition 14 $s_i \rightharpoonup_{h_i} s_\varsigma$, and that $\mathbb{S}(\phi_i \wedge \mathbb{HC}(s_i \rightharpoonup_{h_i} s_\varsigma))$, so $s_i \sqsubset s_\varsigma$. Therefore:

$$\forall s'_\varsigma(s_\varsigma \rightarrow_\varsigma s'_\varsigma \Rightarrow \exists s'_g((s_g \rightarrow_g s'_g) \wedge (s'_g \sqsubset s'_\varsigma))) \tag{10}$$

This concludes the proof. □

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

$$\forall s'_g(s_g \rightarrow_g s'_g \Rightarrow \exists s'_\varsigma((s_\varsigma \rightarrow_\varsigma s'_\varsigma) \wedge (s'_g \sqsubset s'_\varsigma))) \tag{11}$$

and

$$\forall s'_\varsigma(s_\varsigma \rightarrow_\varsigma s'_\varsigma \Rightarrow \exists s'_g((s_g \rightarrow_g s'_g) \wedge (s'_g \sqsubset s'_\varsigma))) \tag{12}$$

Proof. Begin by assuming the conditions from Lemma 3.

The first step is to show that there exists a state s'_ς that is complete with respect to the feasible set. Take state s_ς and compute state s'_ς such that $s_\varsigma \rightarrow_\varsigma s'_\varsigma$. Take any GSE state s_g such that $s_g \sqsubset s_\varsigma$, and find state s'_g such that $s_g \rightarrow_g s'_g$. Let l_g be the location such that $\{(\phi_a\ l_g)\} = L_g(r_x)$ for some ϕ_a. To show that $s'_g \sqsubset s'_\varsigma$, we need to demonstrate that there exists a function h' such that $s'_g \rightharpoonup_{h'} s'_\varsigma$, and that $\mathbb{S}(\phi_{s'} \wedge \mathbb{HC}(s'_g \rightharpoonup_{h'} s'_\varsigma))$. Since $s_h \sqsubset s_\varsigma$, we know that there exists a function h such that $s_g \rightharpoonup_h s_\varsigma$. Let $h' = h$.

First, we consider how $s'_g \rightharpoonup_{h'} s'_\varsigma$. Let l_α and l_β be arbitrary locations in $L_{\overrightarrow{g'}}$ such that $\{(\phi_a\ l_\alpha)\} = L_{g'}(R_{g'}(l_\beta, f))$, let $\theta = L_\varsigma(R_\varsigma(h(l_g), f))$, and let $\theta' = L'_\varsigma(R'_\varsigma(h(l_g), f))$.

Suppose $l_\beta \neq l_g$. 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 symbolic Field Write rule that any feasible location in the set θ must also be in the set θ'. Since $s_g \sqsubset s_\varsigma$, we know that $h(l_\alpha)$ is in θ, and is likewise in θ'. We have now established that in either case where $l_\beta \neq l_g$, $(\phi_b\ h(l_\alpha)) \in L_{s'}(R_{s'}(h(l_\beta), f))$.

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

Thus, for arbitrary l_α and l_β:

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

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

Establishing the fact that $\mathbb{S}(\phi_{s'} \wedge \mathbb{HC}(s'_g \rightharpoonup_{h'} s'_\varsigma))$ 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'_g such that $s_g \rightarrow_g s'_g$, then the state s'_ς such that $s_\varsigma \rightarrow_\varsigma s'_g$ represents s'_g. Therefore, $\forall s'_g(s_g \rightarrow_g s'_g \Rightarrow \exists s'_\varsigma((s_\varsigma \rightarrow_\varsigma s'_\varsigma) \wedge (s'_g \sqsubset s'_\varsigma)))$. This concludes the proof of completeness.

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

□

Lemma 4 ($\rightarrow_S^*$ preserves heap determinism)**.** *Given a deterministic heap,* $(\mathrm{L}_0\ \mathrm{R}_0)$*, from a state with a reference* r *and field* f*, the new heap,* $(\mathrm{L}'\ \mathrm{R}')$*, from the summary machine,* $(\mathrm{L}_0\ \mathrm{R}_0\ \mathrm{r}\ \mathrm{f}) \rightarrow_S^* (\mathrm{L}'\ \mathrm{R}'\ \mathrm{r}\ \mathrm{f})$*, is also deterministic.*

Proof. Induction over the number of steps in $\rightarrow_S^*$ in Figure 7.

Base Case. The relation makes 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 deterministic 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.

- The original heap is deterministic by definition, so any constraint in any member of the set must have some term such that
$$\begin{aligned}&\forall(\phi\ l), (\phi'\ l') \in \theta_{orig}\\ &\quad((l \neq l' \vee \phi \neq \phi') \Rightarrow (\phi \wedge \phi' = \textbf{false}))\end{aligned}$$
Further, any member of θ_{orig} has a constraint of the form $\phi = \neg\phi_x \wedge \ldots$ while any member of θ_{null}, θ_{new}, and θ_{alias} has a constraint of the form $\phi' = \phi_x \wedge \ldots$; thus
$$\begin{aligned}&\forall(\phi\ l) \in \theta_{orig}\ (\forall(\phi'\ l') \in \theta_{null} \cup \theta_{new} \cup \theta_{alias}\ (\\ &\quad\phi \wedge \phi' = \textbf{false}))\end{aligned}$$
- The only member of $\theta_{null} = \{(\phi\ l_{null})\}$ has the form $\phi = \ldots \wedge r_f = r_{null}$ while any member of θ_{new} and θ_{alias} has the form $\phi' = \ldots \wedge r_f \neq r_{null} \wedge \ldots$; thus
$$\forall(\phi'\ l') \in \theta_{new} \cup \theta_{alias}\ (\phi \wedge \phi' = \textbf{false})$$
- The only member of $\theta_{new} = \{(\phi\ l_f)\}$ has a constraint of the form $\phi = \ldots \wedge (\wedge_{(r_a,\ \phi_a,\ l_a) \in \rho} r_f \neq r_a))$ to assert it does not alias anything, while any member of θ_{alias} has the form $\phi' = \ldots \wedge r_f = r_a \wedge \ldots$ to assert it aliases some r_a with both partitions reasoning over the same set of aliases ρ; thus
$$\forall(\phi'\ l') \in \theta_{alias}\ (\phi \wedge \phi' = \textbf{false})$$
- Any member of θ_{alias} has the form
$$\ldots \wedge r_f = r_a \wedge (\wedge_{(r'_a\ \phi'_a\ l'_a) \in \rho\ (r'_a \neq r_a)} r_f \neq r'_a)$$
And thus,
$$\begin{aligned}&\forall(\phi\ l), (\phi'\ l') \in \theta_{alias}\\ &\quad((l \neq l' \vee \phi \neq \phi') \Rightarrow (\phi \wedge \phi' = \textbf{false}))\end{aligned}$$

As θ is mapped to a single reference $r_f = init_r()$ in an already deterministic heap, the resulting heap $(L_1\ R_1)$ is likewise deterministic.

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 deterministic. This matches the base case, in that the heap on the left side of $\rightarrow_S$ is deterministic, and by the same argument as in the base case, $(L_{n+1}\ R_{n+1})$ is thus deterministic. □

Lemma 5 ($\rightarrow_{FA}$ preserves heap determinism)**.** *Given a state, s, with a deterministic heap,* $(\mathrm{L}\ \mathrm{R}) = \mathrm{heap}(s)$*, the new heap,* $(\mathrm{L}'\ \mathrm{R}') = \mathrm{heap}(s')$*, in any state related by the field access rule, $s \rightarrow_{FA} s'$, is also deterministic.*

Proof. Proof by definition of $\rightarrow_{FA}$ in Figure **??**.

Lemma 4 establishes that the heap in the state on the right side of $\rightarrow_S^*$ is deterministic if the heap in the state on the left side is deterministic, so it is only needed to show that determinism is preserved by the call to the value function, $\mathbb{VS}$, in the rule. Let $(L\ R)$ be the new deterministic heap related by $\rightarrow_S^*$.

Recall from Definition 9 that each constraint in each member of the value set has 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 $\phi_\alpha = \phi_\beta$, then by Definition 9
$$\begin{aligned}&\exists(\phi_\alpha\ l) \in L(r)\ (\exists r' \in R(l, f)\ (\\ &\quad(\phi'_\alpha\ l'_\alpha) \in L(r') \wedge (\phi'_\beta\ l'_\beta) \in L(r')))\end{aligned}$$
As $(\phi'_\alpha\ l'_\alpha)$ and $(\phi'_\beta\ l'_\beta)$ are distinct and connected to the same reference r' in a deterministic heap, $\phi'_\alpha \wedge \phi'_\beta = \textbf{false}$ by definition.
- If $\phi_\alpha \neq \phi_\beta$, then by Definition 9
$$\exists l\ ((\phi_\alpha\ l) \in L(r) \wedge \exists l' \neq l\ ((\phi_\beta\ l') \in L(r)))$$
As $(\phi_\alpha\ l)$ and $(\phi_\beta\ l')$ are distinct and connected to the same reference r in a deterministic heap, $\phi_\alpha \wedge \phi_\beta = \textbf{false}$ by definition.

The only change to the heap after the S-relation is the addition of the new reference $r' = \mathrm{stack}_r()$ to point to the value set. As the value set meets the conditions for determinism, the new heap with r' and the value set, $(L'\ R') = \mathrm{heap}(s')$, is also deterministic. □

Lemma 6 ($\rightarrow_{FW}$ preserves heap determinism)**.** *Given a state, s_ς, with a deterministic heap,* $(\mathrm{L}_\varsigma\ \mathrm{R}_\varsigma) = \mathrm{heap}(s_\varsigma)$*, the new heap,* $(\mathrm{L}'_\varsigma\ \mathrm{R}'_\varsigma) = \mathrm{heap}(s'_\varsigma)$*, in any state related by the field write rule, $s_\varsigma \rightarrow_{FW} s'_\varsigma$, is also deterministic.*

Proof. Proof by definition of $\rightarrow_{FW}$ in Figure **??**.

The $\rightarrow_{FW}$ rule relies on the $\mathbb{ST}$ function. Recall from Definition 10 that each constraint in each member of the strengthened set has the form $(\phi \wedge \phi'\ l')$ where every member, $(\phi'\ l')$, comes from $L_\varsigma(r)$ on the same reference in a deterministic heap $(L_\varsigma\ R_\varsigma)$; thus, that set of meets the criteria for determinism by definition. So any application of $\mathbb{ST}$ preserves that criteria for determinism.

The $\rightarrow_{FW}$ makes two uses of the $\mathbb{ST}$ function, $\theta = \mathbb{ST}(L, r, \phi, \phi_g) \cup \mathbb{ST}(L, r_{cur}, \neg\phi, \phi_g)$, to build individual θ sets. Choose any two distinct members of θ, $(\phi_\alpha \wedge \phi'_\alpha\ l'_\alpha)$ and $(\phi_\beta \wedge \phi'_\beta\ l'_\beta)$.

- If $\phi_\alpha = \phi_\beta$, then the constraints came from the same call to $\mathbb{ST}$ so $\phi'_\alpha \wedge \phi'_\beta = \textbf{false}$ by definition.
- If $\phi_\alpha \neq \phi_\beta$, then the constraints came from the different calls and are distinguished by the phase of ϕ in the call so $\phi_\alpha \wedge \phi_\beta = \textbf{false}$.

Each θ is mapped to a new reference from $\mathrm{fresh}_r()$. These are added to an already deterministic heap $(L_\varsigma\ R_\varsigma)$ and meet the criteria so that $(L'_\varsigma\ R'_\varsigma)$ is also deterministic. □

Lemma 7 ($\rightarrow_J$ preserves heap determinism)**.** *Given a state, s_ς, with a deterministic heap,* $(\mathrm{L}_\varsigma\ \mathrm{R}_\varsigma) = \mathrm{heap}(s_\varsigma)$*, the new heap,* $(\mathrm{L}'_\varsigma\ \mathrm{R}'_\varsigma) = \mathrm{heap}(s'_\varsigma)$*, in any state related by the Javalite relation, $s_\varsigma \rightarrow_J s'_\varsigma$, is also deterministic.*

Proof. Proof by definition of $\rightarrow_J$ in Figure 5.

Every rule in $\rightarrow_J$ except `New` leaves the heap unmodified. The rule for `New` adds a single new location ($\mathrm{fresh}_l(C)$) to the heap on a single new reference ($\mathrm{stack}_r()$). The rule also points every field in the new location to r_{null}. As none of these mutations alter the determinism of the heap, the new heap $(L'_\varsigma\ R'_\varsigma)$ is also deterministic. □

Theorem 8 ($\rightarrow_\varsigma$ preserves heap determinism)**.** *Given a state, s_ς, with a deterministic heap,* $(\mathrm{L}_\varsigma\ \mathrm{R}_\varsigma) = \mathrm{heap}(s_\varsigma)$*, the new heap,* $(\mathrm{L}'_\varsigma\ \mathrm{R}'_\varsigma) = \mathrm{heap}(s'_\varsigma)$*, in any state related by the symbolic relation, $s_\varsigma \rightarrow_\varsigma s'_\varsigma$, is also deterministic.*

Proof. Proof by Lemma 5, Lemma 6, and Lemma 7 which represent all the rules that relate states in $\rightarrow_\varsigma$. □

Lemma 9 (Exactness of Reference Compare Rule)**.** *If there exists states s_g and s_ς such that $s_\varsigma \in \mathcal{RC}$ and $s_g \sqsubset s_\varsigma$, then:*

$$\forall s'_g(s_g \rightarrow_g s'_g \Rightarrow \exists s'_\varsigma((s_\varsigma \rightarrow_\varsigma s'_\varsigma) \wedge (s'_g \sqsubset s'_\varsigma))) \qquad (13)$$

and

$$\forall s'_\varsigma(s_\varsigma \rightarrow_\varsigma s'_\varsigma \Rightarrow \exists s'_g((s_g \rightarrow_g s'_g) \wedge (s'_g \sqsubset s'_\varsigma))) \qquad (14)$$

There are two rules that apply to state s_ς, 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_g and s_ς such that $s_\varsigma \in \mathcal{RC}$ and $s_g \sqsubset s_\varsigma$. Let s'_ς be any state such that $s_\varsigma \rightarrow_\varsigma s_\varsigma$ and let $\zeta_T = \forall s'_g(s_g \rightarrow_g s'_g)$. Since $s_g \sqsubset s_\varsigma$, we know that $s_g \in \mathcal{RC}$, and that there exists a homomorphism $s_g \rightharpoonup_h s_\varsigma$ such that $\mathbb{S}(\phi_\varsigma \wedge \mathbb{HC}(s_g \rightharpoonup_h s_\varsigma))$. We partition ζ_T based on the values of $L_g(r_0)$ and $L_g(r_1)$ as follows: Let

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

and let

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

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

Case 1: Assume that $L_g(r_0) = L_g(r_1)$. Compute state s'_g such that $s_g \rightarrow_g s'_g$. In this case, the GSE "equals - references true" rule applied, therefore s'_g is in ζ_t. Observe that by applying Theorem **??**, $\phi'_\varsigma \wedge \phi_0 \wedge \phi_1$ reduces to ϕ_ς. Therefore, $\mathbb{S}(\phi'_\varsigma \wedge \mathbb{HC}(s'_g \rightharpoonup_h s'_\varsigma))$ is true, and by extension, $s'_g \sqsubset s'_\varsigma$. Since this relation holds for arbitrary $s'_g \in \zeta_t$, we now know that

$$((L_g(r_0) = L_g(r_1)) \wedge (s'_g \in \zeta_t)) \Rightarrow s'_g \sqsubset s'_\varsigma \qquad (15)$$

Case 2: Assume that s'_ς 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_ς and R_ς are unchanged from s_ς, and ϕ'_ς is only a strengthened version of ϕ_ς, we know that

$$\{s'_g | s'_g \sqsubset s'_\varsigma\} \subseteq \{s'_g | \exists s_g(s_g \sqsubset s_\varsigma) \wedge s_g \rightarrow_\varsigma s'_g\} \qquad (16)$$

Suppose that there exists state s'_i such that $s'_i \sqsubset s'_\varsigma$ and $s'_i \notin \zeta_t$. Because of Equation 16, we know that

$$s'_i \in \{s'_g | \exists s_g(s_g \sqsubset s_\varsigma) \wedge s_g \rightarrow_\varsigma s'_g\}$$

Combining this with the assumption that $s'_i \notin \zeta_t$, we must conclude that $L_g(r_0) \neq L_g(r_1)$. Because of this, and because of Theorem **??**, 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'_ς does not represent s'_i. We have a contradiction. Therefore:

$$((s'_\varsigma = (L\ R\ \phi'_g\ \eta\ \textbf{true}\ k)) \wedge (s'_g \sqsubset s'_\varsigma)) \Rightarrow s'_g \in \zeta_t \qquad (17)$$

Case 3: Assume that $L_g(r_0) \neq L_g(r_1)$. This means that the GSE "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:

$$((L_g(r_0) = L_g(r_1)) \wedge (s'_g \in \zeta_t)) \Rightarrow s'_g \sqsubset s'_\varsigma \qquad (18)$$

Case 4: Assume that s'_ς 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:

$$s'_\varsigma = (L\ R\ \phi'_g\ \eta\ \textbf{false}\ k) \wedge s'_g \sqsubset s'_\varsigma \Rightarrow s'_g \in \zeta_f \qquad (19)$$

Since $\zeta_T = \zeta_t \cup \zeta_f$, we can combine Equation 15 with 18 to find that

$$\forall s'_g(s_g \rightarrow_g s'_g \Rightarrow \exists s'_\varsigma((s_\varsigma \rightarrow_\varsigma s'_\varsigma) \wedge (s'_g \sqsubset s'_\varsigma))) \qquad (20)$$

Likewise, we can combine Equation 17 with Equation 19 to find that

$$\forall s'_\varsigma(s_\varsigma \rightarrow_\varsigma s'_\varsigma \Rightarrow \exists s'_g((s_g \rightarrow_g s'_g) \wedge (s'_g \sqsubset s'_\varsigma))) \qquad (21)$$

□

Lemma 10 (Exactness of New Rule)**.** *If there exists states s_g and s_ς such that $s_\varsigma \in \mathcal{NW}$ and $s_g \sqsubset s_\varsigma$, then:*

$$\forall s'_g(s_g \rightarrow_g s'_g \Rightarrow \exists s'_\varsigma((s_\varsigma \rightarrow_\varsigma s'_\varsigma) \wedge (s'_g \sqsubset s'_\varsigma))) \qquad (22)$$

and

$$\forall s'_\varsigma(s_\varsigma \rightarrow_\varsigma s'_\varsigma \Rightarrow \exists s'_g((s_g \rightarrow_g s'_g) \wedge (s'_g \sqsubset s'_\varsigma))) \qquad (23)$$

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

Theorem 11. *The representation relation $\sqsubset$ is a bisimulation.*

Proof. Take any two states s_g and s_ς such that $s_g \sqsubset s_\varsigma$. If $s_\varsigma \in \mathcal{FA} \cup \mathcal{FW} \cup \mathcal{RC} \cup \mathcal{NW}$, then by Lemmas 2, 3, 9, and 10 we know Equations 6 and 7 hold. If s_ς has any other form, the heap is not modified for s'_g or s'_ς, so then Equations 6 and 7 hold by default. Thus, Equations 6 and 7 hold for all s_g and s_ς such that $s_g \sqsubset s_\varsigma$. By Definition 18, $\sqsubset$ is a bisimulation.

□

Corollary 12. *For any given initial state, the set of possible control flow sequences under the GSE transition relation is exactly the set of possible control flow sequences under the symbolic transition relation.*

Corollary 13. *For any given initial state, the number of final symbolic states is exactly the number of possible control flow sequences.*

7. Empirical Evaluation

GSESH creates fewer execution paths than GSE, but it does so with increased path constraint complexity. Determining the impact of this tradeoff in real-world applications is an important research question.

Since the number of paths in GSE is exponential in the number of lazy initializations, it is expected that programs for which the number of lazy initializations is proportional to program complexity will evaluate more quickly using GSESH. On the other hand, since GSESH places a greater burden on the constraint solver, it is expected that programs where the lazy initialization count is constant or slowly increasing with respect to program complexity will evaluate more quickly using GSESH.

7.1 JPF Implementation

The Java Pathfinder (JPF) symbolic execution engine contains an implementation of GSE, which we extended with our own implementation of GSESH.

7.2 Bounding

In order to perform a practical evaluation of programs involving recursive data structures, it is generally necessary to impose some sort of bounding to eliminate the number of lazy initializations to a manageable level. While these bounds may be asserted as invariants within the program in-situ, it is more practical to apply bounding via external means. There are several bounding techniques that may be applied to both GSE and GSESH, including execution path length bounding, k-bounding and n-bounding. k-bounding and n-bounding both operate by imposing limits on the size of the symbolic input heap, but differ in how such bounds are defined. n-bounding limits the number of objects created by lazy initialization, while k-bounding limits the length of initialized reference chains. k-bounding has the property of exhaustively exploring the full breadth of a given data structure, while n-bounding tends to explore deeper heap shapes while being somewhat less thorough in breadth. In addition, k-bounding and n-bounding may be combined to adjust the desired level of approximation between depth and breadth.

While k-bounding and n-bounding are applicable to both GSE and GSESH, they are not both equally suited for comparing the two methods. The main reason for this lies in how how each bounding technique impacts the functional equivalence of the respective methods. k-bounding preserves the functional equivalence of GSE and GSESH, a fact which can be established by induction over the length of reference chains from a given initial state. In contrast, GSE and GSESH are not generally functionally equivalent under n-bounding. The reason is that since a GSESH state must represent all possible GSE states on an execution path, it generally includes more locations than any single GSE state it represents. This difference in state counts makes it difficult to externally assert n-bounding in a manner that preserves functional equivalence between GSE and GSESH. Functional equivalence may be preserved in the case where n-bounds are asserted via in-situ invariants, but the significant increase in constraint complexity makes this approach impractical. For this reason, n-bounding is was not selected for comparing GSE to GSESH in the tests performed in this paper.

In contrast to the heap-based bounds provided by k-bounding and n-bounding, execution path-length bounding works by limiting the number along any given execution path. As with other bounding techniques, the usefulness of path-length bounding hinges on whether it preserves functional equivalence. While path-length bounding preserves functional equivalence for GSE and GSESH as formulated in this paper, it does not preserve it for the algorithms as they are implemented in JPF. The reason for the discrepancy lies in how states are defined in JPF. In JPF, the notion of states is closely tied to the concept of nondeterminism. New states are created in JPF only when a nondeterministic choice is made, meaning the number of states on an execution path is tied to the number of nondeterministic choices. A discrepancy in the length of the respective execution paths may be caused by any difference in nondeterminism between the two methods. This is significant, since the nondeterminism in GSESH and GSE differs in both field reads and in reference compares. Field reads in GSESH are deterministic, whereas in GSE they may initiate a nondeterministic lazy initialization. Conversely, in GSE address compares are deterministic, but in GSESH address compares require a nondeterministic assertion over the truth of the branch condition. For this reason, path-length bounding was not used for this experimental evaluation. For future tests, we are considering inserting artificial points of nondeterminism in order to establish functional equivalence of path-length bounds between GSE and GSESH.

7.3 Evaluation

Experimental set-up: TO BE ADDED

Test Cases

In order to evaluate the performance of GSESH, the implementation was tested against three examples: LinkedList, BinarySearchTree, TreeMap. The LinkedList example tests performance on linked list data structures. The test program begins by assuming an object invariant before initiating a sequence of contains() method calls. This is a challenging example for GSE because each call to the contains() method contains a new lazy initialization, which in turn creates an exponential path expansion throughout program execution. While the constraints for each path are easily solved, the sheer number of paths quickly becomes overwhelming.

The BinarySearchTree example test program consists simply of asserting the object invariant for a binary search tree. This example is interesting because it demonstrates nondeterminism from both aliasing constraints created during lazy initialization, and linear constraints from evaluating conditionals over numeric symbols.

Like the BinarySearchTree test, the TreeMap test is simply an assertion of an object invariant, in this case for a red/black tree. This example serves as a stress test for GSESH, since most of the nondeterminism comes from linear constraints over the shape of the tree, rather than aliasing constraints created during lazy initialization. In this case, the path advantage for GSESH is minimized.

For all tests, program complexity was adjusted via k-bounding. For the LinkedList test, the number of contains() method calls was added as an additional parameter. After the conclusion of each test, the test run-time, state count, and path counts were recorded.

The results of the experiments are shown in table 1.

Method	k	Time		Paths	
		GSE	SH	GSE	SH
LinkedList	3	1	1	1656	25
	4	3	2	17485	39
	5	20	3	232743	56
	6	338	8	3731094	76
BinarySearchTree	1	0	1	4	4
	2	1	1	26	17
	3	12	7	305	118
TreeMap	1	0	1	9	9
	2	1	3	100	46
	3	21	206	3026	547

Table 1. test results

7.4 Analysis

LinkedList As expected, the repeated use of lazy initialization throughout the program causes an exponential increase in the number of paths explored by GSE. By combining many of the GSE paths, SH avoids the exponential path explosion, demonstrating dramatically reduced, though still exponential, execution time growth. Remarkably, path growth in the k-bound appears to be sub-linear.

BST The BinarySearchTree example is somewhat of a mixed bag.

TreeMap

8. 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.