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.

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.

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 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. Preliminaries

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.

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 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 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.

```
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

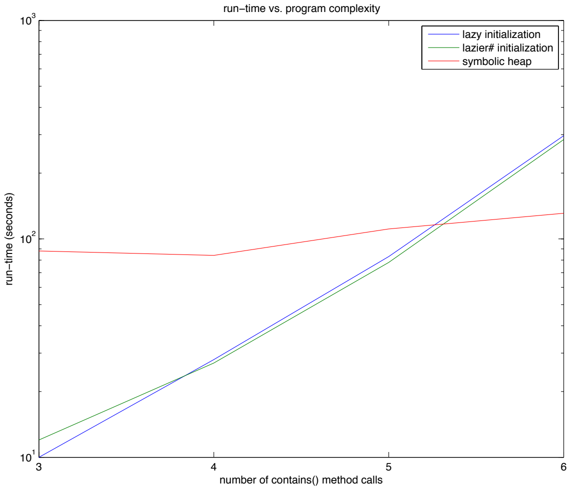

Figure 2. Time versus complexity for the linked list example

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.

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

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

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

Figure 3. The Javalite surface syntax.

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

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

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 1. *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 2. *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.

Definition 3. *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 4. *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 5. *For a given function* $f : A \mapsto B$*, the* ***image*** $f^{\rightarrow}$ *and* ***preimage*** $f^{\leftarrow}$ *are defined as*

$$f^{\rightarrow} = \{f(a) \mid a \in A\} \tag{1}$$
$$f^{\leftarrow} = \{a \mid f(a) \in B\} \tag{2}$$

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

$$f^{\rightarrow}[C] = \{f(a) \mid a \in C\} \tag{3}$$
$$f^{\leftarrow}[D] = \{a \mid f(a) \in D\} \tag{4}$$

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.

3.3 Common Execution Semantics

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

Definition 6. *The set of* ***states*** $\mathcal{S}$ *is defined as the set of strings matching the pattern s in 4.*

Definition 7. $\mathcal{S}_0$ *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.

4. 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 $\mathbb{UN}(L, R, r, f) = \{l\ ...\}$ returns constraint-location pairs in which the field f is unini-

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)

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

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)

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

Variable Declaration

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

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)

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

Variable Assignment

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

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.

tialized:

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

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

5. Accessing and Writing to Field References

6. Equality and InEquality of References

Definition 8. *The function* $\mathbb{VS}(L, R, \phi_g, r, f)$ *constructs the value-set given a heap, reference, and desired field:*

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

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

Definition 9. *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*:*

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

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

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

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

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

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

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

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

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

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

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

$$\textsc{Equals (reference-true)}\quad \frac{L(r_0) = L(r_1) \qquad \phi' = (\phi \wedge r_0 = r_1)}{\begin{array}{l}(L\,R\,\phi\,\eta\ r_0\ (r_1\ \textbf{=}\ \text{*} \rightarrow k)) \rightarrow_\ell\\ (L\,R\,\phi'\ \eta\ \textbf{true}\ k)\end{array}}$$

$$\textsc{Equals (reference-false)}\quad \frac{L(r_0) \neq L(r_1) \qquad \phi' = (\phi \wedge r_0 \neq r_1)}{\begin{array}{l}(L\,R\,\phi\,\eta\ r_0\ (r_1\ \textbf{=}\ \text{*} \rightarrow k)) \rightarrow_\ell\\ (L\,R\,\phi'\ \eta\ \textbf{false}\ k)\end{array}}$$

Figure 8. GSE with lazy initialization indicated by $\rightarrow_g = \rightarrow_\ell \cup \rightarrow_J$.

7. Proofs

7.1 Heap Properties

Definition 10. *A heap,* (L R)*, is* ***deterministic*** *if and only if*

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

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 $(L\,R\,\phi_g\ \eta\ r\ (\text{*}\ \$f \rightarrow k))$ only appear in the Field Access rule.

Definition 11. *The sets $\mathcal{FA}$, $\mathcal{FW}$, $\mathcal{RC}$, and $\mathcal{NW}$ are defined as the sets of states having the forms* $(\mathrm{L\ R}\ \phi_g\ \eta\ \mathrm{r}\ (\text{*}\ \$\ \mathrm{f} \rightarrow \mathrm{k}))$*,* $(\mathrm{L\ R}\ \phi_g\ \eta\ \mathrm{r}\ (\mathrm{x}\ \$\ \mathrm{f}\ \textbf{:=}\ \text{*} \rightarrow \mathrm{k}))$*,* $(\mathrm{L\ R}\ \phi_g\ \eta\ \mathrm{r}_0\ (\mathrm{r}_1\ \textbf{=}\ \text{*} \rightarrow \mathrm{k}))$*, and* $(\mathrm{L\ R}\ \phi_g\ \eta\ (\textbf{new}\ \mathrm{C})\ \mathrm{k})$ *respectively.*

Field Access

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

Field Write

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

Equals (references-true)

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

Equals (references-false)

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

Figure 9. Precise symbolic heap summaries from symbolic execution indicated by $\to_\varsigma = \to_{FA} \cup \to_{FW} \cup \to_{EQ} \cup \to_J$.

Definition 12. *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 13. *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:*

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

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

$$\mathbb{HC}(s_x \rightharpoonup_h s_y) =$$

$$\bigwedge\{\phi_b\ |\exists(\phi_a\ l) \in \mathrm{L}_x^{\rightarrow}((\phi_b\ h(l)) \in \mathrm{L}_y^{\rightarrow})\}$$

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 15. *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*

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

Definition 16. *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:*

$$\forall s'_x(s_x \to_x s'_x \Rightarrow \exists s'_y((s_y \to_y s'_y) \wedge (s'_x\ \mathcal{R}\ s'_y))) \tag{6}$$

$$\forall s'_y(s_y \to_y s'_y \Rightarrow \exists s'_x((s_x \to_x s'_x) \wedge (s'_x\ \mathcal{R}\ s'_y))) \tag{7}$$

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 [**?**].

7.2 Theorems

The goal of this section is to prove that the representation relation, $\sqsubset$, is a bisimulation as defined in (6) and (7). 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* $s_\varsigma = (\mathrm{L}_\varsigma\ \mathrm{R}_s\ \phi_\varsigma\ \eta\ \mathrm{r}\ (*\ \$\ \mathrm{f} \rightarrow \mathrm{k}))$*, initial state* s_0*, and control flow path* π_n*, and if there exists some intermediate state state* s'_ς *such that* $(\mathrm{L}_\varsigma\ \mathrm{R}_s\ \mathrm{r}\ \mathrm{f}\ \mathrm{C}) \rightarrow^*_S (\mathrm{L}_{s'}\ \mathrm{R}_{s'}\ \phi_{s'}\ \mathrm{r}\ \mathrm{f}\ \mathrm{C})$*, then:*

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

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 15, we know $s_g = (L_g\ R_g\ \phi_g\ \eta\ r\ (*\ \$ f \rightarrow k))$. 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: $s'_g = (L_{g'}\ R_{g'}\ \phi_{g'}\ \eta\ r\ (*\ \$ f \rightarrow k))$.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 13 we know that $s'_g \rightharpoonup_{h'} s'_\varsigma$.

It remains to show that $\mathbb{S}(\phi'_\varsigma \wedge \mathbb{HC}(s'_g \rightharpoonup_h s'_\varsigma))$. 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 $\phi_x \wedge \phi_{orig}$, or

$\phi_x \wedge (r_f \; op \; r_a) \wedge (r_f \; op \; r_b) \wedge \ldots$. Constraints of the form $\phi_x \wedge \phi_{orig}$ 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 $\phi_x \wedge (r_f \; op \; r_a) \wedge (r_f \; op \; r_b) \wedge \ldots$, 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 $s_g \sqsubset s_\varsigma$:

$$(s_g \to_I^* s'_g \wedge s_\varsigma \to_S^* s'_\varsigma) \Rightarrow s'_g \sqsubset s'_\varsigma \tag{8}$$

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

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

This fact, combined with our previous result, proves that

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

□

Lemma 2 (Bisumlation for $\to_{\varsigma|FA}$)**.** *If there exists states s_g and s_ς such that $s_\varsigma \in \mathcal{FA}$ and $s_g \sqsubset s_\varsigma$, then:*

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

and

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

Proof. Begin by assuming the conditions stated in Lemma 2. By Lemma 1, the symbolic intermediate state is equivalent to the set of GSE intermediate states, so we may assume without loss of generality that all of the pertinent fields in s_ς are initialized. Take an arbitrary GSE state s_g such that $s_g \sqsubset s_\varsigma$. Since s_ς is exact, $s_g = (L_g \; R_g \; \phi_g \; \eta \; r \; (\mathbf{*}\,\mathbf{\$}f \to k))$, and $s_g \in \mathbb{FS}(\to_g, s_0, \pi_n)$. If we apply the state transition functions to achieve states s'_g and s'_ς such that $s_g \to_g s'_g$ and $s_\varsigma \to_\varsigma s'_\varsigma$, we find that according to the Field Access rule:

$$s'_g = (L_g[r' \mapsto (\phi' \; l')] \; R_g \; \phi_L \; \eta \; r' \; k)$$

and

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

We now show that $s'_g \sqsubset s'_\varsigma$. 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 15 we know $s'_g \sqsubset s'_\varsigma$. We have now shown that:

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

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 15, 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

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

Observe that by virtue of the GSE Field Access rule, $s_i \to_g s'_i$. Now, construct function h_i so that $h_i = h'$. Observe that by Definition 13 $s_i \rightharpoonup_{h_i} s_\varsigma$, and that $\mathbb{S}(\phi_i \wedge \mathbb{HC}(s_i \rightharpoonup_{h_i} s_\varsigma))$, so $s_i \sqsubset s_\varsigma$. Therefore:

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

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:*

$$\forall s'_g (s_g \to_g s'_g \Rightarrow \exists s'_\varsigma ((s_\varsigma \to_\varsigma s'_\varsigma) \wedge (s'_g \sqsubset s'_\varsigma))) \tag{13}$$

and

$$\forall s'_\varsigma (s_\varsigma \to_\varsigma s'_\varsigma \Rightarrow \exists s'_g ((s_g \to_g s'_g) \wedge (s'_g \sqsubset s'_\varsigma))) \tag{14}$$

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 $s_\varsigma \to_\varsigma s'_\varsigma$. Take any GSE state s_g such that $s_g \sqsubset s_\varsigma$, and find state s'_g such that $s_g \to_g s'_g$. 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 $L_{g'}^{\rightarrow}$ 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: $(L_0\ R_0\ rf) \rightarrow_S (L_1\ R_1\ rf)$. 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

  $$\begin{array}{l}\forall(\phi\ l), (\phi'\ l') \in \theta_{orig} \\ \quad ((l \neq l' \vee \phi \neq \phi') \Rightarrow (\phi \wedge \phi' = \textbf{false}))\end{array}$$

  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

  $$\begin{array}{l}\forall(\phi\ l) \in \theta_{orig}\ (\forall(\phi'\ l') \in \theta_{null} \cup \theta_{new} \cup \theta_{alias}\ (\\ \quad \phi \wedge \phi' = \textbf{false}))\end{array}$$

- 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,

  $$\begin{array}{l}\forall(\phi\ l), (\phi'\ l') \in \theta_{alias} \\ \quad ((l \neq l' \vee \phi \neq \phi') \Rightarrow (\phi \wedge \phi' = \textbf{false}))\end{array}$$

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:

$$(L_0\ R_0\ rf) \rightarrow_S (L_1\ R_1\ rf) \rightarrow_S \ldots \rightarrow_S (L_n\ R_n\ rf)$$

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 9.

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 8 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 8

  $$\begin{array}{l}\exists(\phi_\alpha\ l) \in L(r)\ (\exists r' \in R(l,f)\ (\\ \quad (\phi'_\alpha\ l'_\alpha) \in L(r') \wedge (\phi'_\beta\ l'_\beta) \in L(r')))\end{array}$$

  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 8

  $$\exists l\ ((\phi_\alpha\ l) \in L(r) \wedge \exists l' \neq l\ ((\phi_\beta\ l') \in L(r)))$$

  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 9.

The $\rightarrow_{FW}$ rule relies on the $\mathbb{ST}$ function. Recall from Definition 9 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 $\phi'_\alpha \wedge \phi'_\beta =$ **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 $\phi_\alpha \wedge \phi_\beta =$ **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:*

$$\forall s'_g (s_g \rightarrow_g s'_g \Rightarrow \exists s'_\varsigma ((s_\varsigma \rightarrow_\varsigma s'_\varsigma) \wedge (s'_g \sqsubset s'_\varsigma))) \tag{15}$$

and

$$\forall s'_\varsigma (s_\varsigma \rightarrow_\varsigma s'_\varsigma \Rightarrow \exists s'_g ((s_g \rightarrow_g s'_g) \wedge (s'_g \sqsubset s'_\varsigma))) \tag{16}$$

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: $(L \; R \; \phi'_g \; \eta$ **true** $k)$ and $(L \; R \; \phi'_g \; \eta$ **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

$$((L_g(r_0) = L_g(r_1)) \wedge (s'_g \in \zeta_t)) \Rightarrow s'_g \sqsubset s'_\varsigma \tag{17}$$

Case 2: Assume that s'_ς has the form $(L \; R \; \phi'_g \; \eta$ **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

$$\{s'_g | s'_g \sqsubset s'_\varsigma\} \subseteq \{s'_g | \exists s_g (s_g \sqsubset s_\varsigma) \wedge s_g \rightarrow_\varsigma s'_g\} \tag{18}$$

Suppose that there exists state s'_i such that $s'_i \sqsubset s'_\varsigma$ and $s'_i \notin \zeta_t$. Because of Equation 18, 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:

$$((s'_\varsigma = (L \, R \, \phi'_g \; \eta \; \textbf{true} \; k)) \wedge (s'_g \sqsubset s'_\varsigma)) \Rightarrow s'_g \in \zeta_t \tag{19}$$

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:

$$((L_g(r_0) = L_g(r_1)) \wedge (s'_g \in \zeta_t)) \Rightarrow s'_g \sqsubset s'_\varsigma \tag{20}$$

Case 4: Assume that s'_ς has the form $(L \; R \; \phi'_g \; \eta$ **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:

$$s'_\varsigma = (L \, R \, \phi'_g \; \eta \; \textbf{false} \; k) \wedge s'_g \sqsubset s'_\varsigma \Rightarrow s'_g \in \zeta_f \tag{21}$$

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

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

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

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

□

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:*

$$\forall s'_g (s_g \rightarrow_g s'_g \Rightarrow \exists s'_\varsigma ((s_\varsigma \rightarrow_\varsigma s'_\varsigma) \wedge (s'_g \sqsubset s'_\varsigma))) \tag{24}$$

and

$$\forall s'_\varsigma (s_\varsigma \rightarrow_\varsigma s'_\varsigma \Rightarrow \exists s'_g ((s_g \rightarrow_g s'_g) \wedge (s'_g \sqsubset s'_\varsigma))) \tag{25}$$

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 16, $\sqsubset$ 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.*

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.