Exact Heap Summaries from Symbolic Execution

Anonymous

Abstract

One of the fundamental challenges of using symbolic execution to analyze software has been the treatment of dynamically allocated data. State-of-the-art symbolic execution techniques have addressed this challenge by constructing the heap *lazily*, materializing objects on the concrete heap "as needed" and using non-deterministic choice points to explore each feasible concrete heap configuration. Because analysis of the materialized heap locations relies on concrete program semantics, the lazy initialization approach exacerbates the state space explosion problem that limits the scalability of symbolic execution. In this work we present a novel approach for lazy symbolic execution of heap manipulating software which utilizes a fully symbolic heap constructed on-the-fly during symbolic execution. Our approach is 1) *scalable* – it does not create the additional points of non-determinism introduced by existing lazy initialization techniques and it explores each execution path only once for any given set of isomorphic heaps, 2) *precise* – at any given point during symbolic execution, the symbolic heap represents the exact set of feasible concrete heap structures for the program under analysis, and 3) *expressive* – the symbolic heap can represent recursive data structures and heaps resulting from loops and recursive control structures in the code. We report on a case-study of an implementation of our technique in the Symbolic PathFinder tool to illustrate its scalability, precision and expressiveness. We also discuss how test case generation – a common use for symbolic execution results – can benefit from symbolic execution which uses a fully symbolic heap.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

In recent years symbolic execution – a program analysis technique for systematic exploration of program execution paths using symbolic input values – has provided the basis for various software testing and analysis techniques. For each execution path explored during symbolic execution, constraints on the symbolic inputs are collected to create a *path condition*. The set of path conditions computed by symbolic execution characterize the observed program execution behaviours and can be used as an enabling technology for various applications, e.g., regression analysis [1–6], data structure repair [7], dynamic discovery of invariants [8, 9], and debugging[10].

Initial work on symbolic execution largely focused on checking properties of programs with primitive types, such as integers and booleans [11, 12]. Despite recent advances in constraint solving technologies, improvements in raw computing power, and advances in reduction and abstraction techniques [13, 14] 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 [15–17]. 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.

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

This paper makes the following contributions:

- We present a novel lazy symbolic execution technique for analyzing heap manipulating programs that constructs a fully symbolic representatino of the heap on-the-fly durig symbolic execution.
- We prove the soundness and completeness of our algorithm...
- We implement our approach in the Symbolic PathFinder tool
- We demonstrate experimentally that our technique improves the scalability of symbolic execution of heap maninpulating software over state-of-the-art techniques, while maintaining efficiency and precision.
- We discuss the benefits of using a symbolic heap that can be realized by the client analysis that uses the results of symbolic execution.

2. Background and Motivation

In this section we present the background on state of the art techniques that have been developed to handle data non-determinism arising from complex data structures. We present an overview of lazy initialization and lazier# initialization. We also present a brief description of the two bounding strategies used in symbolic execution in heap manipulating programs. Next we present a motivating examples where current concrete initialization of the heap structures struggle to scale to medium sized program due to non-determinism introduced in the symbolic execution tree. We use this example to motivate the need for a more truly symbolic and compact representation of the heap in a manner similar to that of primitive types.

Generalized symbolic execution technique generates a concrete representation of connected memory structures using only the implicit information from the program itself. In the original lazy initialization algorithm, symbolic execution explores different heap shapes by concretizing the heap at the first memory access (read) to an un-initialized symbolic object. At this point, a non-deterministic choice point of concreate heap locations is created that includes: (a) null, (b) an access to a new instance of the object, and (c) aliases to other type-compatible symbolic objects that have been concretized along the same execution path [**?**]. The number of choices explored in lazy initialization greatly increases the non-determinism and often makes the exploration of the program state space intractable.

The Lazier# algorithm is an improvement of the lazy initialization and it pushes the non-deterministic choices further into the execution tree. In the case of a memory access to an uninitialized reference location, by default, no choice point is created. Instead, the read returns a unique symbolic reference representing the contents of the location. The reference may assume any one of three states: uninitialized, non-null, or initialized. The reference is returned in an uninitialized state, and only in a subsequent memory access is the reference concretely initialized.

3. Javalite

Figure 3 defines the surface syntax for the Javalite language [18]. Figure 4 is the machine syntax. Javalite is syntactic machine defined as rewrites on a string. The semantics use a CEKS model with a (C)ontrol string representing the expression being evaluated, an (E)nvironment for local variables, a (K)ontinuation for what is to be executed next, and a (S)tore for the heap.

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

```
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

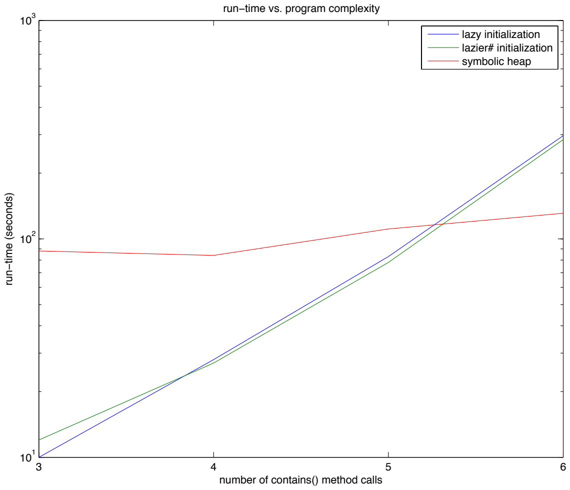

Figure 2. Time versus complexity for the linked list example

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

```
 P ::= (μ (C m))
 μ ::= (CL ...)
 T ::= bool | C
CL ::= (class C ([T f] ...) (M ...)
 M ::= (T m [T x] e)
 e ::= x
     | (new C)
     | (e $f)
     | (x $f := e)
     | (e = e)
     | (if e e else e)
     | (var T x := e in e)
     | (e @ m e )
     | (x := e)
     | (begin e ...)
     | v
 x ::= this | id
 f ::= id
 m ::= id
 C ::= id
 v ::= r | null | true | false
 r ::= number
id ::= variable-not-otherwise-mentioned
```

Figure 3. The Javalite surface syntax.

```
φ ::= (φ) | φ ⋈ φ | ¬φ | true | false | r = r | r ≠ r
l ::= number
L ::= (mt | (L [r → {(φ l) ...}]))
R ::= (mt | (R [(l f) → r]))
η ::= (mt | (η [x → v]))
s ::= (μ L R φ_g η e k)
k ::= end
    | (* $f → k)
    | (x $f := * → k)
    | (* = e → k)
    | (v = * → k)
    | (if * e else e → k )
    | (var T x := * in e → k )
    | (* @ m e → k )
    | (v @ m * → k )
    | (x := * → k)
    | (begin * (e ...) → k )
    | (pop η k)
```

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

set of references.

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

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

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

The initial state of the machine needs to be defined.

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

4. GSE with Lazy Initialization

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

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

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

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

5. GSE with Heap Summaries

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

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

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

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

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

6. Proofs

6.1 Definitions

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

Definition 2. *The set of **initial states** $\mathcal{S}_0$ is defined as*

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

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

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

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

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

.

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

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

.

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

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

.

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

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

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

VARIABLE LOOKUP

$$(L\,R\,\phi_g\,\eta\,x\,k) \rightarrow \\ (L\,R\,\phi_g\,\eta\,\eta(x)\,k)$$

NEW

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

FIELD ACCESS(EVAL)

$$(L\,R\,\phi_g\,\eta\,(e\,\$f)\,k) \rightarrow \\ (L\,R\,\phi_g\,\eta\,e\,(*\,\$f \rightarrow k))$$

FIELD WRITE (EVAL)

$$(L\,R\,\phi_g\,\eta\,(x\,\$f := e)\,k) \rightarrow \\ (L\,R\,\phi_g\,\eta\,e\,(x\,\$f := * \rightarrow k))$$

EQUALS (L-OPERAND EVAL)

$$(L\,R\,\phi_g\,\eta\,(e_0 = e)\,k) \rightarrow \\ (L\,R\,\phi_g\,\eta\,e_0\,(* = e \rightarrow k))$$

EQUALS (R-OPERAND EVAL)

$$(L\,R\,\phi_g\,\eta\,v\,(* = e \rightarrow k)) \rightarrow \\ (L\,R\,\phi_g\,\eta\,e\,(v = * \rightarrow k))$$

EQUALS (BOOL)

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

IF-THEN-ELSE (EVAL)

$$(L\,R\,\phi_g\,\eta\,(\mathbf{if}\ e_0\ e_1\ \mathbf{else}\ e_2)\,k) \rightarrow \\ (L\,R\,\phi_g\,\eta\,e_0\,(\mathbf{if}\ *\ e_1\ \mathbf{else}\ e_2) \rightarrow k)$$

IF-THEN-ELSE (TRUE)

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

IF-THEN-ELSE (FALSE)

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

VARIABLE DECLARATION (EVAL)

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

VARIABLE DECLARATION

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

METHOD INVOCATION (OBJECT EVAL)

$$(L\,R\,\phi_g\,\eta\,(e_0\ @\ m\,e_1)\,k) \rightarrow \\ (L\,R\,\phi_g\,\eta\,e_0\,(*\ @\ m\,e_1 \rightarrow k))$$

METHOD INVOCATION (ARG EVAL)

$$(L\,R\,\phi_g\,\eta\,v_0\,(*\ @\ m\,e_1 \rightarrow k)) \rightarrow \\ (L\,R\,\phi_g\,\eta\,e_1\,(v_0\ @\ m\,* \rightarrow k))$$

METHOD INVOCATION

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

VARIABLE ASSIGNMENT (EVAL)

$$(L\,R\,\phi_g\,\eta\,(x := e)\,k) \rightarrow \\ (L\,R\,\phi_g\,\eta\,e\,(x := * \rightarrow k))$$

VARIABLE ASSIGNMENT

$$(L\,R\,\phi_g\,\eta\,v\,(x := * \rightarrow k)) \rightarrow \\ (L\,R\,\phi_g\,\eta[x \mapsto v]\,v\,k)$$

BEGIN (NO ARGS)

$$(L\,R\,\phi_g\,\eta\,(\mathbf{begin})\,k) \rightarrow \\ (L\,R\,\phi_g\,\eta\,k)$$

BEGIN (ARG0 EVAL)

$$(L\,R\,\phi_g\,\eta\,(\mathbf{begin}\ e_0\ e_1\ \ldots)\,k) \rightarrow \\ (L\,R\,\phi_g\,\eta\,e_0\,(\mathbf{begin}\ *\ (e_1\ \ldots) \rightarrow k))$$

BEGIN (ARGI EVAL)

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

BEGIN (ARGN EVAL)

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

BEGIN

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

NULL

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

POP

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

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

Definition 8. *A **state transition function** $\rightarrow_\Phi$ is a mapping $\rightarrow_\Phi$: $s \mapsto s$, which takes one machine state and transforms it into another machine state. Two important state transition functions are the **lazy state transition function** $\rightarrow_\ell$ and the **summary state transition function** $\rightarrow_s$.*

Definition 9. *A **state sequence** is a sequence of states denoted as $\Pi_n = s_0, s_1, ..., s_n$. A **feasible state sequence**, $\Pi^\phi_n = s_0, s_1, ..., s_n$ is consistent with the transition: $\forall i\ (0 \leq i < n \Rightarrow s_i \rightarrow_\Phi s_{i+1})$, where $s_0 \in \mathcal{S}_0$.*

Definition 10. *The set of **lazy states** $\mathcal{S}_\ell$ is defined as*

$$\mathcal{S}_\ell = \{s_\ell \mid \exists\Pi^\ell_n\ (\Pi^\ell_n = s_0, \ldots, s_\ell)\} \tag{3}$$

Definition 11. *The set of **summary states** $\mathcal{S}_s$ is defined as*

$$\mathcal{S}_s = \{s_s \mid \exists\Pi^s_n\ (\Pi^s_n = s_0, \ldots, s_s)\} \tag{4}$$

Definition 12. *Given a sequence of states*

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

where

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

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

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

Definition 13. *Given a state transition function $\rightarrow_\Phi$, an initial state s_0 and a control flow sequence π_n, the **feasible state set**, $\mathbb{FS}(\rightarrow_\Phi, s_0, \pi_n)$, is defined as*

$$\mathbb{FS}(\rightarrow_\Phi, s_0, \pi_n) = \\ \{s \mid \exists\Pi^\phi_n\ (\pi_n = \mathbb{CF}(\Pi^\Phi_n) \wedge s = \mathit{last}(\Pi_n))\}$$

where $\mathit{last}(\Pi_n)$ returns the last state on the feasible sequence.

Definition 14. *A **field access descriptor** γ_i is a tuple*

$$\gamma_i = (\mathrm{r}_i\ \phi_i\ \mathrm{l}_i\ \mathrm{f}_i)$$

Definition 15. *An **access path** Γ_n is a sequence of field access descriptors $\Gamma_n = \gamma_0, \gamma_1, ..., \gamma_n$.*

Definition 16. *For a given access path Γ_n the **access path constraint** $\mathbb{PC}(\Gamma_n)$ is defined as*

$$\mathbb{PC}(\Gamma_n) = \bigwedge\{\phi \mid \exists\gamma \in \Gamma_n\ (\gamma = (\mathrm{r}\ \phi\ \mathrm{e}\ \mathrm{f}))\}$$

Initialize (null)

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

Initialize (new)

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

Initialize (alias)

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

Initialize (end)

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

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

Field Access

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

Field Write

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

Equals (reference-true)

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

Equals (reference-false)

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

Figure 7. GSE with lazy initialization.

Summarize

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

Summarize (end)

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

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

Definition 17. *For a given state $s = (\mathrm{L}_s\ \mathrm{R}_s\ \phi_s\ \eta_s\ \mathrm{e}_s\ \mathrm{k}_s)$, a **valid access path** $\Gamma_n^s = \gamma_0, \gamma_1, ..., \gamma_n$ satisfies the properties*

$$\begin{array}{l} \mathrm{r}_0 \in \mathit{refs}(\eta_s) \\ \mathbb{S}(\phi_s \wedge \mathbb{PC}(\Gamma_n^s)) \\ \forall i \in \mathbb{N}\ (0 \leq i < n \Leftrightarrow \gamma_i \in \Gamma_n^s \wedge \\ \quad \gamma_{i+1} \in \Gamma_n^s \wedge \\ \quad (\phi_i\ \mathrm{l}_i) \in \mathrm{L}_s(\mathrm{r}_i) \wedge \\ \quad \mathrm{r}_{i+1} = \mathrm{R}_s(\mathrm{l}_i, \mathrm{f}_i) \wedge \\ \quad (\phi_{i+1}\ \mathrm{l}_{i+1}) = \mathrm{L}_s(\mathrm{r}_{i+1})) \end{array}$$

where $\gamma_i = (\mathrm{r}_i\ \phi_i\ \mathrm{l}_i\ \mathrm{f}_i)$

Definition 18. *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*

Field Access

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

Field Write

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

Equals (references-true)

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

Equals (references-false)

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

Figure 9. Precise symbolic heap summaries from symbolic execution.

as follows:

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

Definition 19. *Given homomorphism $s_x \rightharpoonup_h s_y$, the **homomorphism constraint** $\mathbb{HC}(s_x \rightharpoonup_h s_y)$ is defined as:*

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

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

Definition 20. *The **representation relation** is defined as follows: given lazy state $s_\ell = (\mathrm{L}_\ell\ \mathrm{R}_\ell\ \phi_\ell\ \eta_\ell\ \mathrm{e}_\ell\ \mathrm{k}_\ell)$ and summary state $s_s = (\mathrm{L}_s\ \mathrm{R}_s\ \phi_s\ \eta_s\ \mathrm{e}_s\ \mathrm{k}_s)$, $s_\ell \sqsubset s_s$ if and only if $\eta_\ell = \eta_s$, $\mathrm{e}_\ell = \mathrm{e}_s$, $\mathrm{k}_\ell = \mathrm{k}_s$, and there exists a homomorphism $s_\ell \rightharpoonup_h s_s$ such that*

$$\mathbb{S}(\phi_s \wedge \mathbb{HC}(s_\ell \rightharpoonup_h s_s)) \qquad (5)$$

Definition 21. *A summary state s_s is **equivalent** to a set of lazy states P if and only if s_s represents every state in P and represents no other state:*

$$s_s \cong P \Leftrightarrow \forall s_i \in \mathcal{S}\ (s_i \in P \Leftrightarrow s_i \sqsubset s_s)$$

Definition 22. *A state s is **sound** with respect to a transition relation, $\to_\phi$, initial state, s_0, and control flow path, π_n, if and only if*

$$\forall s_\ell \in \mathcal{S}_\ell\ (s_\ell \sqsubset s_s \Rightarrow s_\ell \in \mathbb{FS}(\to_\phi, s_0, \pi_n))$$

Definition 23. *A state s is **complete** with respect to a transition relation, $\to_\phi$, initial state, s_0, and control flow path, π_n, if and only if*

$$\forall s_\ell \in \mathcal{S}_\ell\ (s_\ell \in \mathbb{FS}(\to_\phi, s_0, \pi_n) \Rightarrow s_\ell \sqsubset s_s)$$

Definition 24. *A state s is **exact** with respect to a transition relation, $\to_\phi$, initial state, s_0, and control flow path, π_n, if and only if it is both sound and complete:*

$$s \cong \mathbb{FS}(\to_\phi, s_0, \pi_n)$$

6.2 Theorems

Theorem 1. *If we have a summary state $s_s = (\mathrm{L}_s\ \mathrm{R}_s\ \phi_s\ \eta_s\ \mathrm{e}_s\ \mathrm{k}_s)$, then for any reference $r \in \mathrm{L}_s^{\leftarrow}$, and any two pairs $(\phi_\alpha\ l_\alpha) \in \mathrm{L}_s(r)$ and $(\phi_\beta\ l_\beta) \in \mathrm{L}_s(r)$ such that $l_\alpha \neq l_\beta$, then*

$$(\phi_\alpha \wedge \phi_\beta) = \textbf{\textit{false}}$$

Proof. The proof will proceed inductively. For the base case, consider any initial state s_0. The property holds by construction. (Hands waving around in the air for emphasis)

Now we will show that if the exclusivity property holds for some state s_s, then it holds for any state s_s' where $s_s \to_s s_s'$.

First, suppose we have state s_s and intermediate state s_s' where $s_s \to_S s_s'$. If all fields are initialized, then s_s and s_s' are identical, so the property holds for s_s'. If any field is uninitialized, then the fields are initialized to new references, for which the exclusivity property holds by construction. (More hands waving)

Since the exclusivity property holds for every intermediate state, for any instruction that relies on the summary machine we can assume without loss of generality that all pertinent fields are initialized.

Now, suppose we have a field read instruction. When we read the field, a new value set is created based on the $\mathbb{VS}$ function. The members of the value set have the form $(\phi \wedge \phi' \;\, l)$. Choose any two distinct members of the value set, $(\phi_\alpha \wedge \phi'_\alpha \;\, l_\alpha)$ and $(\phi_\beta \wedge \phi'_\beta \;\, l_\beta)$. If $\phi_\alpha \neq \phi_\beta$, we know that exclusivity holds because ϕ_α and ϕ_β came from the same value set in s_s, and are therefore exclusive. If $\phi_\alpha = \phi_\beta$, we know that exclusivity holds because ϕ'_α and ϕ'_β came from the same value set in s_s and are therefore exclusive. Thus, the exclusivity property holds for any pair of constraints in the value set. Since the only new value set in $L^{\rightarrow}_{s'}$ is generated by the $\mathbb{VS}$ function, we are guaranteed that if exclusivity holds for s_s, then exclusivity holds for s'_s.

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

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

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

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

Theorem 2. *If we have a summary state $s_s = (\mathrm{L}_s\ \mathrm{R}_s\ \phi_s\ \eta_s\ \mathrm{e}_s\ \mathrm{k}_s)$, then for any reference $r \in \mathrm{L}^{\leftarrow}_s$,*

$$\bigvee\{\phi \mid (\phi\ \mathrm{l}) \in \mathrm{L}_s(\mathrm{r})\} = \textbf{\textit{true}}$$

Proof. The proof will proceed inductively.

Base Case: In the initial state s_0 there is only one location, and all of it's fields are initialized to contain the uninitiated reference r_{un}. Since the only reference is the uninitialized reference, for which the desired property holds by construction, the property is true for all references on the heap.

Inductive Step: Suppose we have a summary state s_s for which the property holds, and some state s'_s such that $s_s \rightarrow_S s'_s$. If all relevant fields are initialized, then $s_s = s'_s$, and the property holds trivially. If there is some uninitialized fields, then on of those fields will have a new reference r_{new} added to it. The disjunction

$$\bigvee\{\phi \mid (\phi\ l) \in L_s(r_{new})\}$$

Takes the form

$$(\phi_0) \vee (\neg\phi_0 \wedge \phi_1) \vee (\neg\phi_0 \wedge \neg\phi_1 \wedge \phi_2) \vee ...$$
$$... \vee (\neg\phi_0 \wedge \neg\phi_1 \wedge ... \wedge \phi_n) \vee (\neg\phi_0 \wedge \neg\phi_1 \wedge ... \wedge \neg\phi_n)$$

Which reduces to **true**. Thus, the collective exhaustion property holds for intermediate states. Because the property holds for intermediate states, for summary machine rules we will assume without loss of generality that all relevant fields are initialized. Suppose that the rule executed in the transition from s_s to s'_s is the field read rule. Suppose we have a state s_s for which the property holds, and some state s'_s such that $s_s \rightarrow_s s'_s$.

□

Lemma 3. *If symbolic state $s_s =$ ($\mathrm{L}_s\ \mathrm{R}_s\ \phi_s\ \eta\ \mathrm{r}$ (* \$ $\mathrm{f} \rightarrow \mathrm{k}$)) is exact with respect to some initial state s_0 and control flow path π_n, then the intermediate state state s'_s such that $s_s \rightarrow^*_S s'_s$ is equivalent to $\{\forall s'_\ell | \exists s_\ell \sqsubset s_s(s_\ell \rightarrow^*_I s'_\ell)\}$.*

Proof. Take any lazy state s_ℓ such that $s_\ell \sqsubset s_s$. By Definition 20, we know $s_\ell =$ ($L_\ell\ R_\ell\ \phi_\ell\ \eta\ r$ (* \$ $f \rightarrow k$)). Take any state s'_ℓ where $s_\ell \rightarrow^*_I s'_\ell$, and state s'_s where $s_s \rightarrow^*_S s'_s$. Note that state s'_ℓ has the form: $s'_\ell =$ ($L_{\ell'}\ R_{\ell'}\ \phi_\ell\ \eta\ r$ (* \$ $f \rightarrow k$)).Take any location, field pair ($l_\ell\ f$) such that ($l_\ell\ f$) $\in R^{\leftarrow}_\ell$, and let $l_s = h(l_\ell)$. We may classify l_ℓ into one of three categories, based on the values of the R function in each of the states s_ℓ, s'_ℓ, s_s, and s'_s, and we may define a function $h' : \mathcal{L} \mapsto \mathcal{L}$ based on that classification.

Class 1: $R_\ell(l_\ell,f) = R_{\ell'}(l_\ell,f)$ and $R_s(l_s,f) = R_{s'}(l_s,f)$. In this case, let $h'(l_\ell) = h(l_\ell)$. Since $s_\ell \rightharpoonup_h s_s$, obviously:

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

Class 2: $R_\ell(l_\ell,f) = R_{\ell'}(l_\ell,f)$ and $R_s(l_s,f) \neq R_{s'}(l_s,f)$. Notice that since $R_s(l_s,f) \neq R_{s'}(l_s,f)$, we may deduce that $(\phi_b\ \bot) \in L_s(R_s(l_s, f))$, and by extension, $(\phi_a\ \bot) = L_\ell(R_\ell(l_\ell, f))$. In this case, we let $h'(l_\ell) = h(l_\ell)$. Since $(\phi_a\ \bot) \in L_{\ell'}(R_{\ell'}(l_\ell, f))$, and since by rule $(\phi_b\ \bot) \in L_{s'}(R_{s'}(l_s, f))$, we can see that in this case:

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

Class 3: $R_\ell(l_\ell,f) \neq R_{\ell'}(l_\ell,f)$ and $R_s(l_s,f) \neq R_{s'}(l_s,f)$. In this case, let l_α be any location such that $(\phi_a\ l_\alpha) \in L_{\ell'}(R_{\ell'}(l_\ell, f))$. If $(*\ l_\alpha) \in L^{\rightarrow}_\ell$, let $h'(l_\alpha) = h(l_\alpha)$. Otherwise, let l_β be the location such that $(\phi_b\ l_\beta) \in L_{s'}(R_{s'}(l_s, f))$ and $(\phi_b\ l_\beta) \notin L_s(R_s(l_s, f))$. Now, let $h'(l_\alpha) = l_\beta$. Observe that either way,

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

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

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

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

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

. By Definition 18 we know that $s'_\ell \rightharpoonup_{h'} s'_s$. Furthermore, since the heap constraint is satisfiable (hands waving here), we know that $s'_\ell \sqsubset s'_s$. We have therefore shown that for some summary state s_s and an arbitrary lazy state s_ℓ such that $s_\ell \sqsubset s_s$:

$$(s_\ell \rightarrow^*_I s'_\ell \wedge s_s \rightarrow^*_S s'_s) \Rightarrow s'_\ell \sqsubset s'_s \qquad (6)$$

We now prove the reverse case, that s^*_s represents no infeasible states. Suppose that s'_s represents some infeasible state. This means that we represent some lazy state that has some reference r which points somewhere that no place in the feasible set points to. Since we don't change the path condition, all the old references still point exactly to the same places they used to. So, the problem must be with one of the new references. All of the new references point to either a new location, the null location, the uninitialized location, or some alias. The new, null, and uninitialized locations are pretty straightforward and easy to show that they are all pointing to the correct places at the correct times. This means that there must be a feasible path to a target location that does not exist for any lazy heap. So, pick an arbitrary lazy heap containing the location and field in question. If said target location does not exist, then there is no reference in the lazy heap pointing to that location. In the summary heap, the path constraint on the path leading to the undesired target contains an aliasing condition that states that the source reference only points to this target location on condition that the parent reference points there. However, since we already know that no other reference in the lazy heap points there, this condition must be infeasible. Therefore, it is not part of the represented state. We have a contradiction. Therefore, there is no alias that points somewhere it's not supposed to.

We have now proven that

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

This fact, combined with our previous result, proves that

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

□

Lemma 4. *If there are symbolic states s_s and s_s', control sequences π_n and π_{n+1}, initial state s_0, and reference* r′ *such that the following conditions hold:*

$$s_s = (\mathrm{L}_{\mathcal{S}}\ \mathrm{R}_{\mathcal{S}}\ \phi_g\ \eta\ \mathrm{r}\ (*\ \$\ \mathrm{f} \rightarrow \mathrm{k})) \tag{7}$$

$$s_s \cong \mathbb{FS}(\rightarrow_\phi, s_0, \pi_n) \tag{8}$$

$$\mathrm{r}' = \mathrm{fresh}_r() \tag{9}$$

$$\pi_{n+1} = \pi_n\ (\eta\ \mathrm{r}'\ \mathrm{k}) \tag{10}$$

$$s_s \rightarrow_s s_s' \tag{11}$$

then

$$s_s' \cong \mathbb{FS}(\rightarrow_\phi, s_0, \pi_{n+1})$$

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

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

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

and

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

We now show that $s_\ell' \sqsubset s_s'$. Since η, e, and k are identical between s_s' and s_ℓ', the first condition is met by default. Now we construct the function h' such that $h' = h$. Observe that since $s_\ell \rightharpoonup_h s_s$, and since R_ℓ and R_s are unchanged from states s_ℓ to s_ℓ' and s_s to s_s' respectively, we are guaranteed that $r = R_\ell(l, f) \Rightarrow r = R_s(h'(l), f)$. Let $\{(\phi_\ell'\ l')\} = L_\ell(R_\ell(l, f))$. Since $\mathbb{S}(\phi_g \wedge \mathbb{HC}(s_\ell \rightharpoonup_h s_s))$ is valid, we know that:

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

From this, we may deduce that:

$$(\phi_\ell\ l) \in L_\ell'(r') \Rightarrow (\phi_s \wedge \phi_s'\ h'(l)) \in L_s'(r')$$

Since r' is the only new addition to L_ℓ' and L_s', we now know that the assertion above holds for all $l \in \mathcal{L}$. Thus, we have shown that $s_\ell' \rightharpoonup_{(g\ h)} s_s'$. Furthermore, since $\mathbb{S}(\phi_g \wedge \mathbb{HC}(s_\ell' \rightharpoonup_{h'} s_s'))$, and since $\eta_\ell = \eta_s$, $e_\ell = e_s$, $k_\ell = k_s$, by Definition 20 we know $s_\ell' \sqsubset s_s'$. We have now shown that for any lazy state s_ℓ:

$$s_\ell \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_n) \Rightarrow s_\ell' \sqsubset s_s' \tag{12}$$

Since there is only one possible control flow sequence π_{n+1}, this means that if $s_\ell \rightarrow_\ell s_\ell'$, then

$$s_\ell \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_n) \Leftrightarrow s_\ell' \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_{n+1}) \tag{13}$$

Combining Equations 12 and 13, we may finally conclude that s_s' is complete with respect to $\mathbb{FS}(\rightarrow_\ell, s_0, \pi_{n+1})$

$$s_\ell' \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_{n+1}) \Rightarrow s_\ell' \sqsubset s_s' \tag{14}$$

Now, suppose that there exists a state s_i' such that $s_i' \sqsubset s_s'$, but $s_i' \notin \mathbb{FS}(\rightarrow_\ell, s_0, \pi_{n+1})$. Since $s_i' \sqsubset s_s'$, then by Definition 20, we know there exists a homomorphism $s_i' \rightharpoonup_{h'} s_s'$, and that $\mathbb{S}(\phi_i' \wedge \mathbb{HC}(s_i' \rightharpoonup_{h'} s_s'))$. From state s_i', construct state s_i such that

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

Observe that $s_i \rightarrow_\ell s_i'$. Now, construct function h_i so that $h_i = h'$. Observe that $s_i \rightharpoonup_{h_i} s_s$, and that $\mathbb{S}(\phi_i \wedge \mathbb{HC}(s_i \rightharpoonup_{h_i} s_s))$, so $s_i \sqsubset s_s$. By the hypothesis that s_s is exact, $s_i \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_n)$. Combining this with the fact that $s_i \rightarrow_\ell s_i'$, we conclude that $s_i' \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_{n+1})$. We have a contradiction.

Therefore, s_s' is sound with respect to $\mathbb{FS}(\rightarrow_\ell, s_0, \pi_{n+1})$

$$s_i' \sqsubset s_s' \Rightarrow s_i' \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_{n+1}) \tag{15}$$

Since s_s is both sound and complete, we may combine Equations 14 and 15 to find that

$$s_\ell' \in \mathbb{FS}(\rightarrow_\ell, s_0, \pi_{n+1}) \Leftrightarrow s_\ell' \sqsubset s_s'$$

By Definition 21, $s_s' \cong \mathbb{FS}(\rightarrow_\ell, s_0, \pi_n)$, and so by Definition 24, s_s' is exact.

Case 2: If there are uninitialized fields, then the lazy initialization machine will make an intermediate state s_t. By lemma 3, the symbolic state is equivalent to the set of lazy initialized states. From the intermediate state, the proof is the same as for case 1. □

Lemma 5. *If symbolic state $s_s = (\mathrm{L}_s\ \mathrm{R}_s\ \phi_s\ \eta\ \mathrm{r}\ (\mathrm{x}\ \$\ \mathrm{f} := * \rightarrow \mathrm{k}))$ is exact with respect to some initial state s_0 and control flow path π_n, and if $s_s \rightarrow_s s_s'$ for some state s_s', then s_s' is equal to $(\mathrm{L}_{s'}\ \mathrm{R}_{s'}\ \phi_{s'}\ \eta\ \mathrm{k})$ and is exact with respect to s_0 and $\pi_n\ (\eta\ \mathrm{k}\ \emptyset)$.*

Proof. By Lemma 3 we know that if s_s is exact, then the intermediate state s_x such that $s_s \rightarrow_S^* s_x$ is also exact. Thus, for s_s and any state s_i such that $s_i \sqsubset s_s$ we will assume, without loss of generality, that all relevant fields have been initialized.

First, we show that every state in the feasible set is represented by the s_s'. Take some lazy state s_i such that $s_i \sqsubset s_s$. It is easy to see that s_i' is represented by s_s'.

Next, we show that every state represented by s_s' is in the feasible set. We use the same argument as in the field read proof, that because s_s' represents every state in the feasible set, and since the cardinality of the set of states represented by s_s' is less than or equal to the cardinality of the feasible set, that s_s' can only represent feasible states.

□

Lemma 6. *If symbolic state $s_s = (\mathrm{L}_s\ \mathrm{R}_s\ \phi_s\ \eta\ \mathrm{r}_0\ (\mathrm{r}_1 = * \rightarrow \mathrm{k}))$ is exact with respect to some initial state s_0 and control flow path π_n, and if $s_s \rightarrow_s s_s'$ for some state s_s', then s_s' is equal to $(\mathrm{L}_s\ \mathrm{R}_s\ \phi_{s'}\ \eta\ \mathrm{v}_{s'}\ \mathrm{k})$ and is exact with respect to s_0 and $\pi_n\ (\eta\ \mathrm{v}_{s'}\ \mathrm{k})$.*

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

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

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

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

In this case, the lazy “equals - references true" rule applies, and we know state $s'_\ell : s_\ell \rightarrow_\ell s'_\ell$ is in ζ_t. Observe that by applying theorem 1, $\phi'_s \wedge \phi_0 \wedge \phi_1$ reduces to ϕ_s. Therefore, $\mathbb{S}(\phi'_s \wedge \mathbb{HC}(s'_\ell \rightharpoonup_{(g\ h)} s'_s))$ is true, and by extension, $s'_\ell \sqsubset s'_s$. Since this relation holds for arbitrary $s'_\ell \in \zeta_t$, we now know that

$$s'_\ell \in \zeta_t \Rightarrow s'_\ell \sqsubset s'_s$$

Now we prove the case for the other direction. Consider a state s'_s where $s_s \rightarrow_s s'_s$. Define θ_α, θ_0 and θ_1 as in the “equals (references-true) rule”. Since L_s and R_s are unchanged from s_s, and ϕ'_s is only a strengthened version of ϕ_s, we know that

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

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

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

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

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

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

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

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

$$s'_\ell \sqsubset s'_s \Rightarrow s'_\ell \in \zeta_t$$

Combining this with our previous result, we conclude that

$$s'_\ell \in \zeta_t \Leftrightarrow s'_\ell \sqsubset s'_s$$

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

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

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

$$s'_\ell \in \zeta_f \Leftrightarrow s'_\ell \sqsubset s'_s$$

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

$$s'_\ell \in \zeta_T \Leftrightarrow s'_\ell \sqsubset s'_s$$

By Definition 21, $s'_s \equiv \zeta_T$, and by definition 24, s'_s is exact. □

Theorem 7. *If we have some initial state s_0, and we apply the summary state transition function n times in order to get a sequence of summary states Π^s_n, then the following statement is true:*

$$last(\Pi^s_n) \cong \mathbb{FS}(\rightarrow_\ell, s_0, \mathbb{CF}(\Pi^s_n)) \quad (16)$$

Proof. The proof will proceed inductively.

Base case: The initial state is exact. (hands waving)

Inductive step: If summary state s_s is exact, then any state s'_s such that $s_s \rightarrow_s s'_s$ is also exact. Suppose s_s has the form for a field read, field write, or reference compare rule. By Lemmas 4, 5, and 6, s'_s will also be exact.

□

7. Related Work

The related work goes here.

Acknowledgments

Acknowledgments, if needed.

References

[1] J. Backes, S. Person, N. Rungta, and O. Tkachuk, “Regression verification using impact summaries,” in *Model Checking Software*. Springer, 2013, pp. 99–116.

[2] P. Godefroid, S. K. Lahiri, and C. Rubio-González, “Statically validating must summaries for incremental compositional dynamic test generation,” in *SAS*, 2011, pp. 112–128.

[3] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, “Differential symbolic execution,” in *FSE*, 2008, pp. 226–237.

[4] S. Person, G. Yang, N. Rungta, and S. Khurshid, “Directed incremental symbolic execution,” in *PLDI*, 2011, pp. 504–515.

[5] D. A. Ramos and D. R. Engler, “Practical, low-effort equivalence verification of real code,” in *CAV*, 2011, pp. 669–685.

[6] G. Yang, C. S. Păsăreanu, and S. Khurshid, “Memoized symbolic execution,” in *ISSTA*, 2012, pp. 144–154.

[7] S. Khurshid, I. García, and Y. L. Suen, “Repairing structurally complex data,” in *SPIN*, 2005, pp. 123–138.

[8] C. Csallner, N. Tillmann, and Y. Smaragdakis, “Dysy: Dynamic symbolic execution for invariant inference,” in *ICSE*, 2008, pp. 281–290.

[9] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid, “Feedback-driven dynamic invariant discovery,” in *ISSTA*, 2014, pp. 362–372.

[10] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, “Directed symbolic execution,” in *SAS*, 2011, pp. 95–111.

[11] L. A. Clarke, “A system to generate test data and symbolically execute programs,” *IEEE Transactions on Software Engineering*, vol. SE–2, no. 3, pp. 215–222, 1976.

[12] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[13] S. Anand, C. S. Pasareanu, and W. Visser, “Symbolic execution with abstraction,” *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 11, pp. 53–67, January 2009.

[14] P. Godefroid, “Compositional dynamic test generation,” in *POPL*, 2007, pp. 47–54.

[15] 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*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 157–166.

[16] 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*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 273–282.

[17] S. Khurshid, C. S. Păsăreanu, and W. Visser, “Generalized symbolic execution for model checking and testing,” in *TACAS*, 2003, pp. 553–568.

[18] S. O. Wesonga, “Javalite - an operational semantics for modeling Java programs,” Master’s thesis, Brigham Young University, Provo UT, 2012.

[19] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba, “A syntactic theory of sequential control,” *Theoretical Computer Science*, vol. 52, no. 3, pp. 205 – 237, 1987.