# Uber-lazy Symbolic Execution

Suzette Person and Neha Rungta
NASA

Eric Mercer and Benjamin Hillery
Brigham Young University

*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.

## I. 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], [2], [3], [4], [5], [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], [16], [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.

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...

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

Fig. 1.   The Javalite surface syntax.

- 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.

## II.   BACKGROUND AND MOTIVATION

Example

## III.   PSEUDO-CODE

Figure 1 defines the surface syntax for the Javalite language [18]. The Figure 2 is the machine syntax. The semantics of Javalite is syntax based and defined as rewrites on a string. The semantics use a CEKS machine 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. This paper only defines salient features of the language and machine relevant to understanding the new algorithm.

This paper uses a standard definition of constraints $\phi \in \Phi$ assuming all the usual relational operators and connectives. The heap is a labeled bipartite graph consisting of references $R$ and locations $L$ in the store. The functions $R$ and $L$ are defined for convenience in manipulating the labeled bipartite graph.

- $R(l, f)$ maps location-field pairs from the store to a reference in $R$; and

$$
\begin{aligned}
e &::= (\,...\ \mid\ (v\ @\ m\ v\,)\,) \\
\phi &::= constraint \\
l &::= \textbf{number} \\
\eta &::= (mt\ (\eta\ [x \to v])) \\
s &::= (\mu\ L\ R\ \phi_g\ \eta\ e\ k) \\
k &::= \textbf{end} \\
&\quad\mid\ (*\ \$\ f \to k\,) \\
&\quad\mid\ (*\ @\ m\ e \to k\,) \\
&\quad\mid\ (v\ @\ m\ * \to k\,) \\
&\quad\mid\ (*\ =\ e \to k\,) \\
&\quad\mid\ (v\ =\ * \to k\,) \\
&\quad\mid\ (x := * \to k\,) \\
&\quad\mid\ (x\ \$\ f := * \to k\,) \\
&\quad\mid\ (\textbf{if}\ *\ e\ \textbf{else}\ e \to k\,) \\
&\quad\mid\ (\textbf{var}\ T\ x := *\ \textbf{in}\ e \to k\,) \\
&\quad\mid\ (\textbf{begin}\ *\ (e\ ...) \to k\,) \\
&\quad\mid\ (\textbf{pop}\ \eta\ k) \\
&\quad\mid\ (\textbf{init}\ f\ k)
\end{aligned}
$$

Fig. 2.   The machine syntax for Javalite.

- $L(r)$ maps references to a set of location-constraint pairs in the store.

A reference is a node that gathers the possible store locations for an object during symbolic execution. Each store location is guarded by a constraint that determines the aliasing in the heap. Intuitively, the reference is a level of indirection between a variable and the store, and the reference is used to group a set of possible store locations each predicated on the possible aliasing in the associated constraint. For a variable (or field) to access any particular store location associated with its reference, the corresponding constraint must be satisfied.

Locations are boxes in the graphical representation and indicated with the letter $l$ in the math. References are circles in the graphical representation and indicated with the letter $r$ in the math. Edges from locations are labeled with field names $f \in F$. Edges from the references are labeled with constraints $\phi \in \Phi$ (we assume $\Phi$ is a power set over individual constraints and $\phi$ is a set of constraints for the edge).

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

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

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

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

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

## IV.   RELATED WORK

The related work goes here.

ACKNOWLEDGMENT

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.

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

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

NEW
$\text{fresh}_r\,()\ =\ r\qquad \text{fresh}_l\,(C)\ =\ l$
$R'\ =\ R[\forall f \in \textit{fields}\,(\text{C})\ \ (\,(l\ f)\ \mapsto\ \text{fresh}_r\,()\,)\,]$
$\rho\ =\ \{r'\ |\ \exists f \in \textit{fields}\,(\text{C})\ \ (r'\ =\ R'\,(l,f)\,)\}$
$L'\ =\ L[r\ \mapsto\ \{\,(\phi_T\ l)\,\}][\forall r' \in \rho\ (r'\ \mapsto\ (\phi_T\ l_{null})\,)\,]$
_____
$(L\ R\ \phi_g\ \eta\ (\textbf{new}\ C)\ k)\ \rightarrow$
$(L'\ R'\ \phi_g\ \eta\ r\ k)$

FIELD ACCESS(EVAL)
$(L\ R\ \phi_g\ \eta\ (e\ \$\ f)\ k)\ \rightarrow$
$(L\ R\ \phi_g\ \eta\ e\ (\textbf{init}\ f\ (*\ \$\ f \rightarrow k)\,))$

EQUALS (L-OPERAND EVAL)
$(L\ R\ \phi_g\ \eta\ (e_0\ \textbf{=}\ e)\ k)\ \rightarrow$
$(L\ R\ \phi_g\ \eta\ e_0\ (*\ \textbf{=}\ e \rightarrow k)\,)$

EQUALS (R-OPERAND EVAL)
$(L\ R\ \phi_g\ \eta\ v\ (*\ \textbf{=}\ e \rightarrow k)\,)\ \rightarrow$
$(L\ R\ \phi_g\ \eta\ e\ (v\ \textbf{=}\ * \rightarrow k)\,)$

EQUALS (BOOL)
$v_0 \in \{\textbf{true},\textbf{false}\}\qquad v_1 \in \{\textbf{true},\textbf{false}\}$
$\text{eq?}\,(v_0, v_1)\ =\ v_r$
_____
$(L\ R\ \phi_g\ \eta\ v_0\ (v_1\ \textbf{=}\ * \rightarrow k)\,)\ \rightarrow$
$(L\ R\ \phi_g\ \eta\ v_r\ k)$

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
$\text{lookup}\,(m)\ =\ (T\ m\ [T\ x]\ \ e_m)$
$\eta[\textbf{this} \mapsto v_0][x \mapsto v_1]\ =\ \eta_m$
_____
$(L\ R\ \phi_g\ \eta\ v_1\ (v_0\ @\ m\ *\ \rightarrow k)\,)\ \rightarrow$
$(L\ R\ \phi_g\ \eta_m\ e_m\ (\textbf{pop}\ \eta\ k)\,)$

IF-THEN-ELSE (EVAL)
$(L\ R\ \phi_g\ \eta\ (\textbf{if}\ e_0\ e_1\ \textbf{else}\ e_2)\ k)\ \rightarrow$
$(L\ R\ \phi_g\ \eta\ e_0\ (\textbf{if}\ *\ e_1\ \textbf{else}\ e_2\ \rightarrow k)$

IF-THEN-ELSE (TRUE)
$(L\ R\ \phi_g\ \eta\ \textbf{true}\ (\textbf{if}\ *\ e_1\ \textbf{else}\ e_2\ \rightarrow k)\ \rightarrow$
$(L\ R\ \phi_g\ \eta\ e_1\ k)$

IF-THEN-ELSE (FALSE)
$(L\ R\ \phi_g\ \eta\ \textbf{false}\ (\textbf{if}\ *\ e_1\ \textbf{else}\ e_2\ \rightarrow k)\ \rightarrow$
$(L\ R\ \phi_g\ \eta\ e_2\ k)$

FIELD WRITE (EVAL)
$(L\ R\ \phi_g\ \eta\ (x\ \$\ f\ \textbf{:=}\ e)\ k)\ \rightarrow$
$(L\ R\ \phi_g\ \eta\ e\ (\textbf{init}\ f\ (x\ \$\ f\ \textbf{:=}\ *\ \rightarrow\ k)\,))$

BEGIN (NO ARGS)
$(L\ R\ \phi_g\ \eta\ (\textbf{begin})\ k)\ \rightarrow$
$(L\ R\ \phi_g\ \eta\ k)$

BEGIN (ARG0 EVAL)
$(L\ R\ \phi_g\ \eta\ (\textbf{begin}\ e_0\ e_1\ ...)\ k)\ \rightarrow$
$(L\ R\ \phi_g\ \eta\ e_0\ (\textbf{begin}\ *\ (e_1\ ...)\ \rightarrow k)\,)$

BEGIN (ARGi EVAL)
$(L\ R\ \phi_g\ \eta\ v\ (\textbf{begin}\ *\ (e_i\ e_{i+1}\ ...)\ \rightarrow k)\,)\ \rightarrow$
$(L\ R\ \phi_g\ \eta\ e_i\ (\textbf{begin}\ *\ (e_{i+1}\ ...)\ \rightarrow k)\,)$

BEGIN (ARGN EVAL)
$(L\ R\ \phi_g\ \eta\ v\ (\textbf{begin}\ *\ (e_n)\ \rightarrow k)\,)\ \rightarrow$
$(L\ R\ \phi_g\ \eta\ e_n\ (\textbf{begin}\ *\ ()\ \rightarrow k)\,)$

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

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

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

VARIABLE ASSIGNMENT (EVAL)
$(L\ R\ \phi_g\ \eta\ (x\ \textbf{:=}\ e)\ k)\ \rightarrow$
$(L\ R\ \phi_g\ \eta\ e\ (x\ \textbf{:=}\ *\ \rightarrow\ k)\,)$

VARIABLE ASSIGNMENT
$(L\ R\ \phi_g\ \eta\ v\ (x\ \textbf{:=}\ *\ \rightarrow\ k)\,)\ \rightarrow$
$(L\ R\ \phi_g\ \eta[x \mapsto v]\ v\ k)$

POP
$(L\ R\ \phi_g\ \eta\ (\textbf{pop}\ \eta_0\ k)\,)\ \rightarrow$
$(L\ R\ \phi_g\ \eta_0\ k)$

LAZY INITIALIZATION (END)
$\forall\,(\phi\ l) \in L\,(r)\ \ (R\,(l,f)\ \neq\ \emptyset)$
_____
$(L\ R\ \phi_g\ \eta\ r\ (\textbf{init}\ f\ k)\,)\ \rightarrow\ (L\ R\ \phi_g\ \eta\ r\ k)$

Fig. 3.  Rules common for classic symbolic execution with lazy initialization and symbolic execution with the fully symbolic heap.

LAZY INITIALIZATION (NULL)
$$\Lambda = \{l \mid \exists \phi \ ((\phi \ l) \in L(r) \wedge R(l,f) = \emptyset \ )\} \qquad \Lambda \neq \emptyset \qquad l_x = \min_l(\Lambda)$$
$$\mathrm{init}_r() = r_f \qquad \{(\phi \ l_{null}) \mid \phi = (r_f = r_{null})\} = \theta_{null}$$
$$\overline{(L \ R \ \phi_g \ \eta \ r \ (\mathbf{init} \ f \ k)) \ \rightarrow \ (L[r_f \mapsto \theta] \ R[(l_x,f) \mapsto r_f] \ \phi_g \ \eta \ r \ (\mathbf{init} \ f \ k))}$$

LAZY INITIALIZATION
$$\Lambda = \{l \mid \exists \phi \ ((\phi \ l) \in L(r) \wedge R(l,f) = \emptyset \ )\} \qquad \Lambda \neq \emptyset \qquad l_x = \min_l(\Lambda)$$
$$\mathrm{type}(f) = \mathrm{C}_r \qquad \mathrm{init}_r() = r_f \qquad \mathrm{init}_l(\mathrm{C}_r) = l_f$$
$$\rho = \{(r' \ \phi' \ l') \mid \mathrm{isInit}(r') \wedge (\phi' \ l') \in L(r') \wedge \mathrm{type}(l') = \mathrm{C}_r\}$$
$$\theta_{alias} = \{(\phi \ l) \mid (r' \ \phi' \ l) \in \rho \wedge \phi = (\phi' \wedge r_f \neq r_{null} \wedge r_f = r' \wedge (\wedge_{(r'' \ \phi'' \ l'') \in \rho} \ (r'' \neq r') \ r_f \neq r''))\}$$
$$\theta_{new} = \{(\phi \ l_f) \mid \phi = (r_f \neq r_{null} \wedge (\wedge_{(r', \ \phi', \ l') \in \rho} \ r_f \neq r'))\}$$
$$\theta_{null} = \{(\phi \ l_{null}) \mid \phi = (r_f = r_{null})\}$$
$$\theta = \theta_{alias} \cup \theta_{new} \cup \theta_{null}$$
$$\overline{(L \ R \ \phi_g \ \eta \ r \ (\mathbf{init} \ f \ k)) \ \rightarrow \ (L[r_f \mapsto \theta] \ R[(l_x,f) \mapsto r_f] \ \phi_g \ \eta \ r \ (\mathbf{init} \ f \ k))}$$

LAZY INITIALIZATION
$$\Lambda = \{l \mid \exists \phi \ ((\phi \ l) \in L(r) \wedge R(l,f) = \emptyset \ )\} \qquad \Lambda \neq \emptyset \qquad l_x = \min_l(\Lambda)$$
$$\mathrm{type}(f) = \mathrm{C}_r \qquad \mathrm{init}_r() = r_f \qquad \mathrm{init}_l(\mathrm{C}_r) = l_f$$
$$\rho = \{(r' \ \phi' \ l') \mid \mathrm{isInit}(r') \wedge (\phi' \ l') \in L(r') \wedge \mathrm{type}(l') = \mathrm{C}_r\}$$
$$\theta_{alias} = \{(\phi \ l) \mid (r' \ \phi' \ l) \in \rho \wedge \phi = (\phi' \wedge r_f \neq r_{null} \wedge r_f = r' \wedge (\wedge_{(r'' \ \phi'' \ l'') \in \rho} \ (r'' \neq r') \ r_f \neq r''))\}$$
$$\theta_{new} = \{(\phi \ l_f) \mid \phi = (r_f \neq r_{null} \wedge (\wedge_{(r', \ \phi', \ l') \in \rho} \ r_f \neq r'))\}$$
$$\theta_{null} = \{(\phi \ l_{null}) \mid \phi = (r_f = r_{null})\}$$
$$\theta = \theta_{alias} \cup \theta_{new} \cup \theta_{null}$$
$$\overline{(L \ R \ \phi_g \ \eta \ r \ (\mathbf{init} \ f \ k)) \ \rightarrow \ (L[r_f \mapsto \theta] \ R[(l_x,f) \mapsto r_f] \ \phi_g \ \eta \ r \ (\mathbf{init} \ f \ k))}$$

FIELD ACCESS
$$\frac{r \neq r_{null} \qquad L(r) = \{(\phi \ l)\} \qquad \mathrm{fresh}_r() = r'}{\begin{array}{c}(L \ R \ \phi_g \ \eta \ r \ (* \ \$ \ f \rightarrow k)) \ \rightarrow \\ (L[r' \mapsto (\phi \ l)] \ R \ \phi_g \ \eta \ r' \ k)\end{array}}$$

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

FIELD WRITE
$$\eta(x) = r_x \qquad r_x \neq r_{null} \qquad L(r_x) = \{(\phi \ l)\}$$
$$\mathrm{fresh}_r() = r'$$
$$\frac{R[(l \ f) \mapsto r'] = R' \qquad L[r' \mapsto L(r)] = L'}{\begin{array}{c}(L \ R \ \phi_g \ \eta \ r \ (x \ \$ \ f := * \ \rightarrow \ k)) \ \rightarrow \\ (L'R' \ \phi_g \ \eta \ k)\end{array}}$$

Fig. 4. Rules specific to classic symbolic execution with lazy initialization.

LAZY INITIALIZATION

$\Lambda = \{l \mid \exists \phi \; ((\phi \; l) \in L(r) \wedge R(l,f) = \emptyset )\}$     $\Lambda \neq \emptyset$     $l_x = \min_l(\Lambda)$

$\text{type}(f) = C_r$     $\text{init}_r() = r_f$     $\text{init}_l(C_r) = l_f$

$\rho = \{(r' \; \phi' \; l') \mid \text{isInit}(r') \wedge (\phi' \; l') \in L(r') \wedge \text{type}(l') = C_r\}$

$\theta_{alias} = \{(\phi \; l) \mid (r' \; \phi' \; l) \in \rho \wedge \phi = (\phi' \wedge r_f \neq r_{null} \wedge r_f = r' \wedge (\wedge_{(r'' \; \phi'' \; l'') \in \rho \; (r'' \neq r')} \; r_f \neq r''))\}$

$\theta_{new} = \{(\phi \; l_f) \mid \phi = (r_f \neq r_{null} \wedge (\wedge_{(r', \; \phi', \; l') \in \rho} \; r_f \neq r'))\}$

$\theta_{null} = \{(\phi \; l_{null}) \mid \phi = (r_f = r_{null})\}$

$\theta = \theta_{alias} \cup \theta_{new} \cup \theta_{null}$

$$(L \; R \; \phi_g \; \eta \; r \; (\textbf{init} \; f \; k)) \rightarrow (L[r_f \mapsto \theta] \; R[(l_x,f) \mapsto r_f] \; \phi_g \; \eta \; r \; (\textbf{init} \; f \; k))$$

FIELD ACCESS

$$\frac{\forall (\phi \; l) \in L(r) \;\; (l = l_{null} \rightarrow \neg \mathbb{S}(\phi \wedge \phi_g)) \qquad \text{fresh}_r() = r'}{(L \; R \; \phi_g \; \eta \; r \; (* \; \$ f \rightarrow k)) \rightarrow (L[r' \mapsto \mathbb{VS}(L,R,r,f,\phi_g)] \; R \; \phi_g \; \eta \; r' \; k)}$$

EQUALS (REFERENCES-TRUE)

$v_0 \notin \{\textbf{true}, \textbf{false}\}$     $v_1 \notin \{\textbf{true}, \textbf{false}\}$

$\theta_\alpha = \{\phi_0 \wedge \phi_1 \mid \exists l \; ((\phi_0 \; l) \in L(v_0) \wedge (\phi_1 \; l) \in L(v_1))\}$

$\theta_0 = \{\phi_0 \mid \exists l_0 \; ((\phi_0 \; l_0) \in L(v_0) \wedge \forall (\phi_1 \; l_1) \in L(v_1) \;\; (l_0 \neq l_1))\}$

$\theta_1 = \{\phi_1 \mid \exists l_1 \; ((\phi_1 \; l_1) \in L(v_1) \wedge \forall (\phi_0 \; l_0) \in L(v_0) \;\; (l_0 \neq l_1))\}$

$\phi'_g = \phi_g \wedge (\vee_{\phi_\alpha \in \theta_\alpha} \phi_\alpha) \wedge (\wedge_{\phi_0 \in \theta_0} \neg \phi_0) \wedge (\wedge_{\phi_1 \in \theta_1} \neg \phi_1)$

$\mathbb{S}(\phi'_g)$

$$\frac{}{(L \; R \; \phi_g \; \eta \; v_0 \; (v_1 = * \rightarrow k)) \rightarrow (L \; R \; \phi'_g \; \eta \; \textbf{true} \; k)}$$

EQUALS (REFERENCES-FALSE)

$v_0 \notin \{\textbf{true}, \textbf{false}\}$     $v_1 \notin \{\textbf{true}, \textbf{false}\}$

$\theta_\alpha = \{\phi_0 \rightarrow \neg \phi_1 \mid \exists l \; ((\phi_0 \; l) \in L(v_0) \wedge (\phi_1 \; l) \in L(v_1))\}$

$\theta_0 = \{\phi_0 \mid \exists l_0 \; ((\phi_0 \; l_0) \in L(v_0) \wedge \forall (\phi_1 \; l_1) \in L(v_1) \;\; (l_0 \neq l_1))\}$

$\theta_1 = \{\phi_1 \mid \exists l_1 \; ((\phi_1 \; l_1) \in L(v_1) \wedge \forall (\phi_0 \; l_0) \in L(v_0) \;\; (l_0 \neq l_1))\}$

$\phi'_g = \phi_g \wedge (\wedge_{\phi_\alpha \in \theta_\alpha} \phi_\alpha) \vee ((\vee_{\phi_0 \in \theta_0} \phi_0) \vee (\vee_{\phi_1 \in \theta_1} \phi_1))$

$\mathbb{S}(\phi'_g)$

$$\frac{}{(L \; R \; \phi_g \; \eta \; v_0 \; (v_1 = * \rightarrow k)) \rightarrow (L \; R \; \phi'_g \; \eta \; \textbf{false} \; k)}$$

FIELD WRITE

$r_x = \eta(x)$     $\forall (\phi \; l) \in L(r_x) \;\; (l = l_{null} \rightarrow \neg \mathbb{S}(\phi \wedge \phi_g))$

$\Psi_x = \{(r_{old} \; \phi \; l) \mid (\phi \; l) \in L(r_x) \wedge r_{old} \in R(l,f)\}$

$X = \{(r_{old} \; \theta \; l) \mid \exists \phi \; ((r_{old} \; \phi \; l) \in \Psi_x \wedge \theta = \mathbb{ST}(L,r,\phi) \cup \mathbb{ST}(L,r_{old},\neg\phi))\}$

$R' = R[\forall (r_{old} \; \theta \; l) \in X \;\; ((l \; f) \mapsto \text{fresh}_r())]$

$L' = L[\forall (r_{old} \; \theta \; l) \in X \;\; (\exists r' = R'(l,f) \;\; (r' \mapsto \theta))]$

$$(L \; R \; \phi_g \; \eta \; r \; (x \; \$ f := * \rightarrow k)) \rightarrow (L'R' \; \phi_g \; \eta \; k)$$

Fig. 5.   Rules specific to the fully symbolic heap.