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...

- 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

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.

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.

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

Fig. 1. The Javalite surface syntax.

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

Fig. 2. The machine syntax for Javalite.

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
- $L(r)$ 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 Φ is a power set over individual constraints and ϕ is a set of constraints for the edge).

The function $\mathbb{VS}(L, R, r, f)$ constructs the value-set given a heap, reference, and desired field such that $(l' \ \phi' \wedge \phi) \in \mathbb{VS}(L, R, r, f)$ if and only if

$$\begin{array}{l}\exists(l \ \phi) \in L(r)(\\ \quad \exists r' \in R(l, f)(\\ \quad\quad \exists(l' \ \phi') \in L(r')(\mathbb{S}(\phi \wedge \phi'))))\end{array}$$

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

The strengthen function $\mathbb{ST}(L, r, \phi')$ strengthens every constraint from the reference r and keeps only location-constraint pairs that are satisfiable after strengthening. Formally, $(l \ \phi \wedge \phi') \in \mathbb{ST}(L, r, \phi')$ if and only if $\exists(l \ \phi) \in L(r) \wedge \mathbb{S}(\phi \wedge \phi')$

The empty-reference function $\mathbb{ER}(L, \phi') = \{r \mid L(r) \neq \emptyset \wedge \forall(l, \phi) \in L(r)(\neg\mathbb{S}(\phi \wedge \phi'))\}$ searches the heap for references that become disconnected from all their locations after strengthening. These references, if reachable, imply the heap is no longer valid on the current search path. As such, the symbolic execution algorithm should backtrack. This check is similar to a feasibility check in classic symbolic execution with only primitive data types.

The consistency function is critical to the soundness of the algorithm as it detects when a symbolic heap becomes invalid along a path, similar to a feasibility check when doing classical symbolic execution with just primitives. As the constraints in the heap are strengthened with different aliasing requirements, it is possible to reach a point where the heap is no longer connected. Meaning, a valid reference is live, either in the local environment or the continuation, the reaches another reference that is no longer connected to any locations due to strengthening. The function relies on the empty-reference function to identify disconnected references. The function in essence checks every reference in the local environment and every reference found in the continuation, as these are all considered live. This operation similar to garbage collection where the local environment and stack are inspected to find the roots of the heap for the scan.

The consistency function relies on two auxiliary functions which are informally defined. The function $\mathrm{ref}(\eta, k)$ inspects the local environment and continuation for all live references, and it returns those references in a set. The function $\mathrm{reach}(L, R, r, r')$ returns true if r' is reachable from r in the heap and false otherwise. The consistency function $\mathbb{C}(L, R, X, \eta, k)$ is now defined as

$$\begin{cases} 0 & \exists r \in \mathrm{ref}(\eta, k) \ (\exists r' \in X \ (\mathrm{reach}(L, R, r, r'))) \\ 1 & \text{otherwise} \end{cases}$$

IV. Uber-lazy Operational Semantics

The operational semantics for Uber-lazy symbolic execution are specified using the Javalite language [18]. Javalite is an imperative model of Java that includes many features of the Java language. In this work, we present only the salient features of the Javalite language relevant to understanding the Uber-lazy symbolic execution algorithm. A detailed explanation of Javalite is available in [18].

Javalite is an imperative model of the Java language developed to facilitate rapid prototyping of model checking algorithms and proofs about the algorithms. It is specified with a Java-like syntax and a set of reduction rules for syntacticly executing Javalite programs. Javalite is based on a variant of the CEKS syntactic machine [19]. The operational semantics are defined using the structure (syntax) of the language and a set of reduction rules for syntactically executing Javalite programs. In a CEKS machine, the (C)ontrol string represents a program, command, or instruction to be evaluated; it is initialized to a string representing the entire progam. An (E)nvironment maps (local) variables to their values. The (K)ontinuation specifies what is to be executed next, and the (S)tore is used to store dynamically allocated data, i.e., the heap.

A. Symbolic Store

At the core of the Uber-lazy symbolic execution algorithm is a fully symbolic heap, i.e., a heap in which objects are never materialized as concrete objects during symbolic execution, but instead are represented using constraints charactering feasible heap shapes. To represent the heap store (S), our algorithm defines a labeled bi-partite graph where $S = (R, L, E)$. R contains the set of nodes representing program *references*. L contains the set of nodes representing *locations* in the store. The store is initialized with two special locations: $null$ and $\perp$ representing a null object and an uninitialized location respectively. Each edge in the set of labeled edges, E, is unidirectional. An edge from a reference $r \in R$ to a location $l \in L$ is labeled with a constraint ϕ indicating the conditions under which r references i.e., points to, that location in the store. This paper uses a standard definition of constraints $\phi \in \Phi$ assuming all of the usual relation operators and connectives. Reference nodes collect the the feasible points-to relations for a given program execution path during symbolic execution. Each edge from a location $l \in L$ to a reference $r \in R$ is labeled with the name of a field $f \in F$.

More details here including we assume the input program is typesafe or if there is a type and a mismatch occurs, then the machine halts.

The machine also halts if an exception is thrown.

B. Syntax

Javalite programs and expressions are written in a syntax specified by the grammar in Figure 2. The production rules correspond to the various features supported by Javalite such as classes, fields, methods, and expressions.

Figure 2 specifies the machine syntax for Javalite. The expression e is equivalent to a CEKS machine's control string. The

C. Reduction Rules

We first present the reduction rules

1) Basic Reduction Rules: Most rules presented here - short discussion

2) Store Update Rules: Interesting rules presented here; more elaborate discussion

Start off with helper functions for rules that manipulate the store...

The function $\mathbb{VS}(L, R, r, f)$ constructs the value-set given a heap, reference, and desired field such that $(l'\ \phi' \wedge \phi) \in \mathbb{VS}(L, R, r, f)$ if and only if

$$\begin{array}{l}\exists(l\ \phi) \in L(r)(\\ \quad \exists r' \in R(l, f)(\\ \quad\quad \exists(l'\ \phi') \in L(r')(\mathbb{S}(\phi \wedge \phi'))))\end{array}$$

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 and keeps only location-constraint pairs that are satisfiable after strengthening. Formally, $(l\ \phi \wedge \phi') \in \mathbb{ST}(L, r, \phi')$ if and only if $\exists(l\ \phi) \in L(r) \wedge \mathbb{S}(\phi \wedge \phi')$

The empty-reference function $\mathbb{ER}(L, \phi') = \{r \mid L(r) \neq \emptyset \wedge \forall(l, \phi) \in L(r)(\neg\mathbb{S}(\phi \wedge \phi'))\}$ searches the heap for references that become disconnected from all their locations after strengthening. These references, if reachable, imply the heap is no longer valid on the current search path. As such, the symbolic execution algorithm should backtrack. This check is similar to a feasibility check in classic symbolic execution with only primitive data types.

The consistency function is critical to the soundness of the algorithm as it detects when a symbolic heap becomes invalid along a path, similar to a feasibility check when doing classical symbolic execution with just primitives. As the constraints in the heap are strengthened with different aliasing requirements, it is possible to reach a point where the heap is no longer connected. Meaning, a valid reference is live, either in the local environment or the continuation, the reaches another reference that is no longer connected to any locations due to strengthening. The function relies on the empty-reference function to identify disconnected references. The function in essence checks every reference in the local environment and every reference found in the continuation, as these are all considered live. This operation similar to garbage collection where the local environment and stack are inspected to find the roots of the heap for the scan.

The consistency function relies on two auxiliary functions which are informally defined. The function $\mathrm{ref}(\eta, k)$ inspects the local environment and continuation for all live references, and it returns those references in a set. The function $\mathrm{reach}(L, R, r, r')$ returns true if r' is reachable from r in the heap and false otherwise. The consistency function $\mathbb{C}(L, R, X, \eta, k)$ is now defined as

$$\begin{cases} 0 & \exists r \in \mathrm{ref}(\eta, k)\ (\exists r' \in X\ (\mathrm{reach}(L, R, r, r'))) \\ 1 & \text{otherwise} \end{cases}$$

V. 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.

[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.

VARIABLE LOOKUP

$$(L\ R\ g\ \eta\ x\ k) \rightarrow (L\ R\ g\ \eta\ \eta(x)\ k)$$

FIELD ACCESS(EVAL)

$$(L\ R\ g\ \eta\ (e\ \$\ f)\ k) \rightarrow (L\ R\ g\ \eta\ e\ (\textbf{init}\ f\ (*\ \$\ f \rightarrow k)))$$

LAZY INITIALIZATION

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

NEW

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

LAZY INITIALIZATION (END)

$$\frac{\forall(\phi\ l) \in L(r)\ (R(l,f) \neq \emptyset)}{(L\ R\ g\ \eta\ r\ (\textbf{init}\ f\ k)) \rightarrow (L\ R\ g\ \eta\ r\ k)}$$

FIELD ACCESS

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

EQUALS (L-OPERAND EVAL)

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

EQUALS (R-OPERAND EVAL)

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

EQUALS (BOOL)

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

EQUALS (REFERENCES-TRUE)

$$\frac{\begin{array}{l}v_0 \notin \{\textbf{true}, \textbf{false}\} \qquad v_1 \notin \{\textbf{true}, \textbf{false}\} \\ \theta_\alpha = \{\phi_0 \wedge \phi_1 \mid \exists l\ ((\phi_0\ l) \in L(v_0) \wedge (\phi_1\ l) \in L(v_1))\} \\ \theta_0 = \{\phi_0 \mid \exists l_0\ ((\phi_0\ l_0) \in L(v_0) \wedge \forall(\phi_1\ l_1) \in L(v_1)\ (l_0 \neq l_1))\} \\ \theta_1 = \{\phi_1 \mid \exists l_1\ ((\phi_1\ l_1) \in L(v_1) \wedge \forall(\phi_0\ l_0) \in L(v_0)\ (l_0 \neq l_1))\} \\ \phi_g = (\vee_{\phi_\alpha \in \theta_\alpha}\ \phi_\alpha) \wedge (\wedge_{\phi_0 \in \theta_0}\ \neg\phi_0) \wedge (\wedge_{\phi_1 \in \theta_1}\ \neg\phi_1) \qquad g' = g \wedge \phi_g\end{array}}{(L\ R\ g\ \eta\ v_0\ (v_1 = * \rightarrow k)) \rightarrow (L\ R\ g'\ \eta\ \textbf{true}\ k)}$$

EQUALS (REFERENCES-FALSE)

$$\frac{\begin{array}{l}v_0 \notin \{\textbf{true}, \textbf{false}\} \qquad v_1 \notin \{\textbf{true}, \textbf{false}\} \\ \theta_\alpha = \{\phi_0 \rightarrow \neg\phi_1 \mid \exists l\ ((\phi_0\ l) \in L(v_0) \wedge (\phi_1\ l) \in L(v_1))\} \\ \theta_0 = \{\phi_0 \mid \exists l_0\ ((\phi_0\ l_0) \in L(v_0) \wedge \forall(\phi_1\ l_1) \in L(v_1)\ (l_0 \neq l_1))\} \\ \theta_1 = \{\phi_1 \mid \exists l_1\ ((\phi_1\ l_1) \in L(v_1) \wedge \forall(\phi_0\ l_0) \in L(v_0)\ (l_0 \neq l_1))\} \\ \phi_g = (\wedge_{\phi_\alpha \in \theta_\alpha}\ \phi_\alpha) \vee ((\vee_{\phi_0 \in \theta_0}\ \phi_0) \vee (\vee_{\phi_1 \in \theta_1}\ \phi_1)) \qquad g' = g \wedge \phi_g\end{array}}{(L\ R\ g\ \eta\ v_0\ (v_1 = * \rightarrow k)) \rightarrow (L\ R\ g'\ \eta\ \textbf{true}\ k)}$$

METHOD INVOCATION (OBJECT EVAL)

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

METHOD INVOCATION (ARG EVAL)

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

METHOD INVOCATION (RAW)

$$\frac{\text{lookup}(m) = (T\ m\ [T\ x]\ \ e_m) \qquad \eta_m = \eta[\textbf{this} \mapsto v_0][x \mapsto v_1]}{(L\ R\ g\ \eta\ v_1\ (v_0\ @\ m\ *\ \rightarrow k)) \rightarrow (L\ R\ g\ \eta\ (\textbf{raw}\ v_0\ @\ m\ v_1\ k)}$$

METHOD INVOCATION

$$\frac{\text{lookup}(m) = (T\ m\ [T\ x]\ \ e_m) \qquad \eta_m = \eta[\textbf{this} \mapsto v_0][x \mapsto v_1]}{(L\ R\ g\ \eta\ (\textbf{raw}\ v_0\ @\ m\ v_1\ k) \rightarrow (L\ R\ g\ \eta_m\ e_m\ (\textbf{pop}\ \eta\ k))}$$

Fig. 3. Uber-lazy state reductions

ITE (EXPR-EVAL)

$$(L\ R\ g\ \eta\ \ (\textbf{if}\ e_0\ e_1\ \textbf{else}\ e_2)\ \ k)\ \rightarrow\ (L\ R\ g\ \eta\ e_0\ \ (\textbf{if}\ *\ e_1\ \textbf{else}\ e_2)\ \rightarrow k)$$

ITE (TRUE)

$$(L\ R\ g\ \eta\ \textbf{true}\ \ (\textbf{if}\ *\ e_1\ \textbf{else}\ e_2)\ \rightarrow k)\ \rightarrow\ (L\ R\ g\ \eta\ e_1\ k)$$

ITE (FALSE)

$$(L\ R\ g\ \eta\ \textbf{false}\ \ (\textbf{if}\ *\ e_1\ \textbf{else}\ e_2)\ \rightarrow k)\ \rightarrow\ (L\ R\ g\ \eta\ e_2\ k)$$

FIELD WRITE (EVAL)

$$(L\ R\ g\ \eta\ \ (x\ \$\ f\ \textbf{:=}\ e)\ \ k)\ \rightarrow\ (L\ R\ g\ \eta\ e\ \ (\textbf{init}\ f\ \ (x\ \$\ f\ \textbf{:=}\ *\ \rightarrow\ \ k)\ )\ )$$

FIELD WRITE

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

BEGIN (NO ARGS)

$$(L\ R\ g\ \eta\ \ (\textbf{begin})\ \ k)\ \rightarrow\ (L\ R\ g\ \eta\ k)$$

BEGIN (ARG0 EVAL)

$$(L\ R\ g\ \eta\ \ (\textbf{begin}\ e_0\ e_1\ ...)\ \ k)\ \rightarrow\ (L\ R\ g\ \eta\ e_0\ \ (\textbf{begin}\ *\ \ (e_1\ ...)\ \rightarrow k)\ )$$

BEGIN (ARGI EVAL)

$$(L\ R\ g\ \eta\ v\ \ (\textbf{begin}\ *\ \ (e_i\ e_{i+1}\ ...)\ \rightarrow k)\ )\ \rightarrow\ (L\ R\ g\ \eta\ e_i\ \ (\textbf{begin}\ *\ \ (e_{i+1}\ ...)\ \rightarrow k)\ )$$

BEGIN (ARGN EVAL)

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

BEGIN

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

VARIABLE DECLARATION (EVAL)

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

VARIABLE DECLARATION

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

VARIABLE ASSIGNMENT (EVAL)

$$(L\ R\ g\ \eta\ \ (x\ \textbf{:=}\ e)\ \ k)\ \rightarrow\ (L\ R\ g\ \eta\ e\ \ (x\ \textbf{:=}\ *\ \rightarrow\ \ k)\ )$$

VARIABLE ASSIGNMENT

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

POP

$$(L\ R\ g\ \eta\ \ (\textbf{pop}\ \eta_0\ k)\ )\ \rightarrow\ (L\ R\ g\ \eta_0\ k)$$

Fig. 4. Uber-lazy state reductions