Uber-lazy Symbolic Execution

Suzette Person and Neha Rungta
NASA

Eric Mercer and Benjamin Hillery
Brigham Young University

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

Fig. 1. The Javalite surface syntax.

$$
\begin{array}{rcl}
e & ::= & (.... \mid (\textbf{raw}\ v\ @\ m\ (v\ ...)\ )\ ) \\
\phi & ::= & constraint \\
l & ::= & \textbf{number} \\
s & ::= & (\mu\ L\ R\ \eta\ e\ k) \\
k & ::= & \textbf{end} \\
 & \mid & (*\ \$\ f\ \rightarrow\ k)
\end{array}
$$

Fig. 2. The machine syntax for Javalite.

***Abstract*—The abstract goes here.**

I. Pseudo-code

Figure 1 defines the surface syntax for the Javalite language [1]. 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
- $L(r)$ 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

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

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

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

ACKNOWLEDGMENT

REFERENCES

[1] S. O. Wesonga, "Javalite - an operational semantics for modeling Java programs," Master's thesis, Brigham Young University, Provo UT, 2012.

VARIABLE LOOKUP

$$(L\ R\ \eta\ x\ k) \to (L\ R\ \eta\ \eta(x)\ k)$$

FIELD ACCESS(EVAL)

$$(L\ R\ \eta\ (e\ \$\ f)\ k) \to (L\ R\ \eta\ e\ (*\ \$\ f \to k))$$

FIELD ACCESS (NULL)

$$\frac{L(r) = \emptyset}{(L\ R\ \eta\ r\ (*\ \$\ f \to k)) \to (L[r \mapsto \{(\bot, \phi_T)\}]\ R\ \eta\ r\ (*\ \$\ f \to k))}$$

FIELD ACCESS (NON-NULL)

$$\frac{\begin{array}{c}L(r) = \emptyset \qquad \text{type}(r) = \text{C}_r \qquad \text{fresh}_l(\text{C}_r) = l \\ R' = R[\forall \text{f} \in \text{C}_r\ ((l\ f) \mapsto \text{fresh}_r(\text{type}(f)))]\end{array}}{(L\ R\ \eta\ r\ (*\ \$\ f \to k)) \to (L[r \mapsto \{(l, \phi_T)\}]\ R'\ \eta\ r\ (*\ \$\ f \to k))}$$

NEW

$$\frac{\begin{array}{l}\text{fresh}_r(C) = r \qquad \text{fresh}_l(C) = l \\ R' = R[\forall f \in \text{C}\ ((l\ f) \mapsto \text{fresh}_r(\text{type}(f)))] \\ L' = L[r \mapsto \{(l\ \phi_T)\}]\end{array}}{(L\ R\ \eta\ (\textbf{new}\ C)\ k) \to (L'\ R'\ \eta\ r\ k)}$$

FIELD ACCESS

$$\frac{L(r) \neq \emptyset \qquad \text{type}(r) = C_r \qquad \text{fresh}_r(C_r) = r_f}{(L\ R\ \eta\ r\ (*\ \$\ f \to k)) \to (L[r_f \mapsto \mathbb{VS}(L, R, r, f)]\ R\ \eta\ r_f\ k)}$$

EQUALS (L-OPERAND EVAL)

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

EQUALS (R-OPERAND EVAL)

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

EQUALS (BOOL)

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

EQUALS (REFERENCES-TRUE)

$$\frac{\begin{array}{l}v_0 \notin \{\textbf{true}, \textbf{false}\} \qquad v_1 \notin \{\textbf{true}, \textbf{false}\} \\ \theta_\alpha = \{\phi_1 \wedge \phi_2 \mid \exists(l\ \phi_1) \in L(v_1)(\exists(l\ \phi_2) \in L(v_2)(\mathbb{S}(\phi_1 \wedge \phi_2)))\} \\ \theta_1 = \{\phi_1 \mid \exists(l_1\ \phi_1) \in L(v_1) \wedge \forall(l_2\ \phi_2) \in L(v_2)(l_1 \neq l_2)\} \\ \theta_2 = \{\phi_2 \mid \exists(l_2\ \phi_2) \in L(v_2) \wedge \forall(l_1\ \phi_1) \in L(v_1)(l_1 \neq l_2)\} \\ \phi_g = (\vee_{\phi_\alpha \in \theta_\alpha} \phi_\alpha) \wedge (\wedge_{\phi_1 \in \theta_1} \neg\phi_1) \wedge (\wedge_{\phi_2 \in \theta_2} \neg\phi_2) \\ L' = L[\forall r(\exists(l\ \phi) \in L(r)(r \mapsto \mathbb{ST}(L, r, \phi_g)))] \\ X = \mathbb{ER}(L, \phi_g) \qquad \mathbb{C}(L', R, X, \eta, k) = 1\end{array}}{(L\ R\ \eta\ v_0\ (v_1 = * \to k)) \to (L'\ R\ \eta\ \textbf{true}\ k)}$$

EQUALS (REFERENCES-FALSE)

$$\frac{\begin{array}{l}v_0 \notin \{\textbf{true}, \textbf{false}\} \qquad v_1 \notin \{\textbf{true}, \textbf{false}\} \\ \theta_\alpha = \{\phi_1 \to \neg\phi_2 \mid \exists(l\ \phi_1) \in L(v_1)(\exists(l\ \phi_2) \in L(v_2)(\mathbb{S}(\phi_1 \wedge \phi_2)))\} \\ \theta_1 = \{\phi_1 \mid \exists(l_1\ \phi_1) \in L(v_1) \wedge \forall(l_2\ \phi_2) \in L(v_2)(l_1 \neq l_2)\} \\ \theta_2 = \{\phi_2 \mid \exists(l_2\ \phi_2) \in L(v_2) \wedge \forall(l_1\ \phi_1) \in L(v_1)(l_1 \neq l_2)\} \\ \phi_g = (\wedge_{\phi_\alpha \in \theta_\alpha} \phi_\alpha) \wedge ((\vee_{\phi_1 \in \theta_1} \phi_1) \vee (\vee_{\phi_2 \in \theta_2} \phi_2)) \\ L' = L[\forall r(\exists(l\ \phi) \in L(r)(r \mapsto \mathbb{ST}(L, r, \phi_g)))] \\ X = \mathbb{ER}(L, \phi_g) \qquad \mathbb{C}(L', R, X, \eta, k) = 1\end{array}}{(L\ R\ \eta\ v_0\ (v_1 = * \to k)) \to (L'\ R\ \eta\ \textbf{false}\ k)}$$

METHOD INVOCATION (OBJECT EVAL)

$$(L\ R\ \eta\ (e_0\ @\ m\ (e_1\ \ldots)\ )\ k) \to (L\ R\ \eta\ e_0\ (*\ @\ m\ (e_1\ \ldots) \to k))$$

METHOD INVOCATION (ARG0 EVAL)

$$(L\ R\ \eta\ v_0\ (*\ @\ m\,(e_1\ e_2\ \ldots) \to k)) \to (L\ R\ \eta\ e_1\ (v_0\ @\ m\ ()\ *\ (e_2\ \ldots) \to k))$$

METHOD INVOCATION (ARGI EVAL)

$$(L\ R\ \eta\ v_i\ (v_0\ @\ m\ (v_1\ \ldots)\ *\ (e_{i+1}\ e_{i+2}\ \ldots) \to k)) \to (L\ R\ \eta\ e_{i+1}\ (v_0\ @\ m\ (v_1\ \ldots\ v_i)\ *\ (e_{i+2}\ \ldots) \to k))$$

METHOD INVOCATION (ARGS)

$$(L\ R\ \eta\ v_n\ (v_0\ @\ m\ (v_1\ \ldots)\ *\ () \to k)) \to (L\ R\ \eta\ (\textbf{raw}\ v_0\ @\ m\ (v_1\ \ldots\ v_n))\ k)$$

METHOD INVOCATION (NO ARGS)

$$(L\ R\ \eta\ e_0\ (*\ @\ m\ () \to k)) \to (L\ R\ \eta\ (\textbf{raw}\ v_0\ @\ m\ ()\ k)$$

METHOD INVOCATION (RAW)

$$\frac{\text{lookup}(m) = (T\ m\ (\,[T\ x]\ \ldots)\ e_m) \qquad \eta_m = \eta[\textbf{this} \mapsto v_0][x \mapsto v_1]\ \ldots}{(L\ R\ \eta\ (\textbf{raw}\ v_0\ @\ m\ (v_1\ \ldots)\ k) \to (L'\ R'\ \eta_m\ e_m\ (\textbf{pop}\ \eta\ k))}$$

Fig. 3. Uber-lazy state reductions

ITE (EXPR-EVAL)
$(L\ R\ \eta\ (\textbf{if}\ e_0\ e_1\ \textbf{else}\ e_2)\ k) \to (L\ R\ \eta\ e_0\ (\textbf{if}\ *\ e_1\ \textbf{else}\ e_2) \to k)$

ITE (TRUE)
$(L\ R\ \eta\ \textbf{true}\ (\textbf{if}\ *\ e_1\ \textbf{else}\ e_2) \to k) \to (L\ R\ \eta\ e_1\ k)$

ITE (FALSE)
$(L\ R\ \eta\ \textbf{false}\ (\textbf{if}\ *\ e_1\ \textbf{else}\ e_2) \to k) \to (L\ R\ \eta\ e_2\ k)$

FIELD WRITE (EXPR-EVAL)
$(L\ R\ \eta\ (x\ \$\ f\ \textbf{:=}\ e)\ k) \to (L\ R\ \eta\ e\ (x\ \$\ f\ \textbf{:=}\ *)\ \to\ k)$

FIELD WRITE (NULL)

$$\frac{L(r) = \emptyset}{(L\ R\ \eta\ r\ (x\ \$\ f\ \textbf{:=}\ *)\ \to\ k) \to (L[r \mapsto \{(\bot, \phi_T)\}]\ R\ \eta\ r\ (x\ \$\ f\ \textbf{:=}\ *)\ \to\ k)}$$

FIELD WRITE (NON-NULL)

$$\frac{\begin{array}{c}L(r) = \emptyset \qquad \text{type}(r) = \text{C}_r \qquad \text{fresh}_l(\text{C}_r) = l \\ R' = R[\forall \text{f} \in \text{C}_r\ ((l\ f) \mapsto \text{fresh}_r(\text{type}(f)))]\end{array}}{(L\ R\ \eta\ r\ (x\ \$\ f\ \textbf{:=}\ *)\ \to\ k) \to (L[r \mapsto \{(l, \phi_T)\}]\ R'\ \eta\ r\ (x\ \$\ f\ \textbf{:=}\ *)\ \to\ k))}$$

FIELD WRITE

$$\frac{\begin{array}{c}L(r) \neq \emptyset \qquad r_x = \eta(\mathbf{x}) \qquad \Psi_x = \{(r', \phi) \mid \exists (l, \phi) \in L(r_x) \wedge \exists r' \in R(l,f)\} \\ X = \cup_{(r',\phi') \in \Psi_x} \mathbb{ST}(L, r, \phi') \cup \mathbb{ST}(L, r', \neg\phi') \qquad L' = [r' \mapsto X]\end{array}}{(L\ R\ \eta\ r\ (x\ \$\ f\ \textbf{:=}\ *)\ \to\ k) \to (L'R\ \eta\ k))}$$

Fig. 4. Uber-lazy state reductions