Uber-lazy Symbolic Execution

Neha Rungta
NASA Ames

Eric Mercer and Benjamin Hillery
Brigham Young University

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

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.

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

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

$$(L\ R\ \eta\ \mathrm{x}\ k) \to (L\ R\ \eta\ \eta(\mathrm{x})\ k)$$

FIELD ACCESS(EVAL)

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

FIELD ACCESS (NULL)

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

FIELD ACCESS (NON-NULL)

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

FIELD ACCESS

$$\frac{\begin{array}{l} L(r) \neq \emptyset \qquad \mathrm{type}(r) = C_r \qquad \mathrm{fresh}_r(C_r) = r_f \\ Q = \{(r'\ \phi) \mid (l\ \phi) \in L(r) \wedge r' \in R(l\ \mathrm{f})\} \\ L_f = L[r_f \mapsto \cup_{(r'\ \phi) \in Q}\{(l\ \phi'\phi) \mid (l\ \phi') \in L(r')\}] \end{array}}{(L\ R\ \eta\ r\ (*\ \$\ \mathrm{f} \to k)) \to (L_f\ R\ \eta\ r_f\ k)}$$

Fig. 3. Uber-lazy state reductions