Path-Optimal Symbolic Execution of Heap-Manipulating Programs

Anonymous Authors

APPENDIX

In this Appendix we introduce an example imperative, object-oriented programming language and give operational semantics for path-optimal symbolic execution. Figure 1 reports the syntax of the language. A program is a sequence of class declarations followed by an expression to be evaluated. Classes have exactly one superclass (there is no multiple inheritance nor interfaces) and multiple fields and methods. The body of a method is a single expression.

There are many types of expressions, but here we want focus on expressions that read the content of object fields (getfield expressions $e.f$), modify the content of object fields (putfield expressions $e.f := e$), conditional expressions $\texttt{if } e \ e \ e$, and method invocation expressions $e.m(\overline{e})$. These are the expressions whose semantics is more affected by the fact that their subexpressions may be symbolic.

Figure 2 reports the syntax of the configurations and the evaluation contexts used in the operational semantics definition. The small-step operational semantics transition relation $\Rightarrow$ is defined in Figure 3 as a suitable composition of two distinct transition relations, the *computation* transition relation $\rightarrow$, and the (reflexive-transitive closure of the) *refinement* transition relation $\rightsquigarrow$. These relations are formally defined in Figure 4, Figure 5 and Figure 6. The computation transition accounts for the steps that implement a computational step performed to progress the evaluation of the current expression in the program. The refinement transition adds information to the current configuration that is already implicit in it, without progressing the evaluation of the program. The added information may come in the shape of a heap object, or of a value in a field of an object.

In POSE each symbolic reference points to a *symbolic object* in the initial heap. In the beginning of symbolic execution these objects do not actually exist in the initial heap, and are materialized on demand (through a suitable refinement step) when the corresponding symbolic reference is used to access the object. This is the only similarity of POSE with generalized symbolic execution. Differently from the latter, where an input reference is initialized to point to the object it may alias to, in POSE every symbolic reference points to a distinct symbolic object, and the alternative effects of the alias relationship are encoded *inside each object*, by means of if-then-else-values (ite-values, with shape $\texttt{ite}(\ldots)$).

Let us consider the computation semantics of getfield expressions $e.f$, that read the content of field f from a heap object denoted by expression e. Rule Getfield (c5) considers the case where e evaluates to the location l of a *concrete object*, i.e., of an object created after the start of symbolic execution by means of a `new` expression. In this case, the value is simply read from the object field. Rule Getfield (c1) considers the case where e evaluates to a symbol Y with reference type. The rule may fire only if a symbolic object for Y exists in the heap ($Y \in \mathrm{dom}(H)$), the field f is present in the symbolic object ($f \in \mathrm{dom}(H(Y))$), and the slot corresponding to the field f has been initialized with some value ($H(Y)(f) \neq \bot$). We will discuss all these conditions when explaining the refinement transitions. When all the conditions are met, the value is read from the object field, exactly as for concrete objects. Finally, rules Getfield (c2)-(c4) cope with $\texttt{ite}(\sigma, \sigma_1, \sigma_2)$ symbolic references. These symbolic references may emerge as expressions of alias conditions and are handled by reading the field f from the objects denoted by the expressions σ_1 and σ_2, and returning the ite-expression combining the results (rule Getfield (c2)). The resulting path condition is obtained by suitably merging the path conditions obtained by evaluating $\sigma_1.f$ and $\sigma_2.f$, by means of the function $\Sigma' = mergePC(\sigma, \Sigma'_1, \Sigma'_2)$ defined as follows: If $\Sigma'_1 = \sigma_{11}, \ldots, \sigma_{1m}$ and $\Sigma'_2 = \sigma_{21}, \ldots, \sigma_{2w}$, then $\Sigma' = (\sigma \implies \sigma_{11}), \ldots, (\sigma \implies \sigma_{1m}), (\neg\sigma \implies \sigma_{21}), \ldots, (\neg\sigma \implies \sigma_{2w})$. Similarly, the resulting heap is obtained by merging the updates that the evaluations of $\sigma_1.f$ and $\sigma_2.f$ apply to the heap, by means of the function $H' = mergeHGf(H'_1, H'_2, f, \sigma)$ defined as follows: $\mathrm{dom}(H') = \mathrm{dom}(H'_1) \cup \mathrm{dom}(H'_2)$, and for all $u \in \mathrm{dom}(H')$ it is:

1) if $u \in \mathrm{dom}(H'_1)$ and $u \notin \mathrm{dom}(H'_2)$, then $H'(u) = H'_1(u)$;
2) if $u \in \mathrm{dom}(H'_2)$ and $u \notin \mathrm{dom}(H'_1)$, then $H'(u) = H'_2(u)$;
3) if $u \in \mathrm{dom}(H'_1)$, and $u \in \mathrm{dom}(H'_2)$, and $H'_1(u) = H'_2(u)$, then $H'(u) = H'_1(u) = H'_2(u)$;
4) if $u \in \mathrm{dom}(H'_1)$, and $u \in \mathrm{dom}(H'_2)$, and $H'_1(u) \neq H'_2(u)$, then $\mathrm{dom}(H'(u)) = \mathrm{dom}(H'_1(u)) \cup \mathrm{dom}(H'_2(u))$, and for all $f \in \mathrm{dom}(H'(u))$:
   a) if $f \in \mathrm{dom}(H'_1(u))$ and $f \notin \mathrm{dom}(H'_2(u))$, then $H'(u)(f) = H'_1(u)(f)$;
   b) if $f \in \mathrm{dom}(H'_2(u))$ and $f \notin \mathrm{dom}(H'_1(u))$, then $H'(u)(f) = H'_2(u)(f)$;
   c) if $f \in \mathrm{dom}(H'_1(u))$, and $f \in \mathrm{dom}(H'_2(u))$, and $H'_1(u)(f) = H'_2(u)(f)$ then $H'(u)(f) = H'_1(u)(f) = H'_2(u)(f)$;
   d) if $f \in \mathrm{dom}(H'_1(u))$, and $f \in \mathrm{dom}(H'_2(u))$, and

Programs	P	::=	$\overline{C}\ e$
Classes	C	::=	$\texttt{class}\ c : c\ \{\overline{t\ f}; \overline{D}\}$
Methods	D	::=	$t\ m(\overline{t\ x}) := e$
Program expressions	e	::=	$x \mid v \mid \texttt{new}\ c \mid e.f \mid e.f := e \mid \texttt{let}\ x := e\ \texttt{in}\ e$
			$\mid e\ op\ e \mid op\ e \mid \texttt{if}\ e\ e\ e \mid e.m(\overline{e})$
Types	t	::=	$c \mid \texttt{bool} \mid \texttt{int}$
Values	v	::=	$p \mid u$
Primitive values	p	::=	$r \mid X$
Reference values	u	::=	$\texttt{null} \mid l \mid Y$
Literals	r	::=	$b \mid n$
Booleans	b	::=	$\texttt{true} \mid \texttt{false}$
Integers	n	::=	$\ldots -1 \mid 0 \mid 1 \ldots$
Operators	op	::=	$+ \mid = \ldots$
Locations	l		
Identifiers	c, f, m, x, X, Y		

Fig. 1. Language syntax

Configurations	J	::=	$H\ \Sigma\ \eta$
Configuration expressions	η	::=	$x \mid \sigma \mid \texttt{new}\ c \mid \eta.f \mid \eta.f := \eta \mid \texttt{let}\ x := \eta\ \texttt{in}\ \eta$
			$\mid \eta\ op\ \eta \mid op\ \eta \mid \texttt{if}\ \eta\ \eta\ \eta \mid \eta.m(\overline{\eta})$
Heaps	H	::=	$\overline{u \mapsto o},\ u \neq \texttt{null}$
Objects	o	::=	$(\overline{f \mapsto \sigma})^c \mid (\overline{f \mapsto \sigma})$
Path conditions	Σ	::=	$\emptyset \mid \Sigma, \sigma$
Symbolic values	σ	::=	$\bot \mid v \mid \sigma\ op\ \sigma \mid op\ \sigma \mid Y \leq: c \mid Y.f = Z \mid \texttt{ite}\,(Y = Y, \sigma, \sigma)$
Evaluation contexts	E	::=	$- \mid E.f \mid E.f := \eta \mid \sigma.f := E \mid \texttt{let}\ x := E\ \texttt{in}\ \eta$
			$\mid E\ op\ \eta \mid \sigma\ op\ E \mid op\ E \mid \texttt{if}\ E\ \eta\ \eta \mid E.m(\overline{\eta})$
			$\mid \sigma.m(\overline{\sigma}, E, \overline{\eta})$

Fig. 2. Syntax of configurations and evaluation contexts

$$\text{Step:}\ \frac{H\ \Sigma\ \eta \rightsquigarrow^* H'\ \Sigma'\ \eta \quad H'\ \Sigma'\ \eta \to H''\ \Sigma''\ \eta''}{H\ \Sigma\ \eta \Rightarrow H''\ \Sigma''\ \eta''}$$

Fig. 3. Small-step operational semantics

$H'_2(u)(f) = \bot$ then $H'(u)(f) = H'_1(u)(f)$;

e) if $f \in \mathrm{dom}(H'_1(u))$, and $f \in \mathrm{dom}(H'_2(u))$, and $H'_1(u)(f) = \bot$ then $H'(u)(f) = H'_2(u)(f)$;

f) if $f \in \mathrm{dom}(H'_1(u))$, and $f \in \mathrm{dom}(H'_2(u))$, and $H'_1(u)(f) \neq H'_2(u)(f)$, and $H'_1(u)(f) \neq \bot$, and $H'_2(u)(f) \neq \bot$ then $H'(u)(f) = \texttt{ite}\,(\sigma, H'_1(u)(f), H'_2(u)(f))$.

In the case one of the field access expression $\sigma_1.f$, $\sigma_2.f$ cannot evaluate, the ite-based getfield reduces to the other field access expression (rules Getfield (c3) and (c4)).

Now we analyze the refinement semantics of getfield expressions. Rule Getfield (r1) handles the situation where a symbolic reference has no corresponding symbolic object in the heap yet—i.e., is *unbound*. For the sake of simplicity we suppose that each field f is contained in exactly one class. If $c = class(f)$ is the class that contains the field f, then the refinement step adds a symbolic object of class c to the heap, whose fields are all uninitialized (the function $fields(c)$ yields the set of all the fields declared in c and in its superclasses), binds Y to the new object, and updates the path condition to reflect the fact that field access has success ($\neg Y = \texttt{null}$) and that the symbolic object's type is a subclass of c ($Y \leq: c$). Rule

$$\text{New:}\ \frac{fields(c) = \overline{f} \quad l \notin \mathrm{dom}(H)}{H\ \Sigma\ \texttt{new}\ c \to H, l \mapsto (\overline{f \mapsto default})^c\ \Sigma\ l}$$

$$\text{Getfield (c1):}\ \frac{Y \in \mathrm{dom}(H) \quad f \in \mathrm{dom}(H(Y)) \quad H(Y)(f) \neq \bot}{H\ \Sigma\ Y.f \to H\ \Sigma\ H(Y)(f)}$$

$$\text{Getfield (c2):}\ \frac{\begin{array}{c} H\ \Sigma\ \sigma_1.f \Rightarrow H_1'\ \Sigma_1'\ \sigma_1' \quad H\ \Sigma\ \sigma_2.f \Rightarrow H_2'\ \Sigma_2'\ \sigma_2' \\ H' = mergeHGf(H_1', H_2', f, \sigma) \quad \Sigma' = mergePC(\sigma, \Sigma_1', \Sigma_2') \end{array}}{H\ \Sigma\ \texttt{ite}\,(\sigma, \sigma_1, \sigma_2)\,.f \to H'\ \Sigma'\ \texttt{ite}\,(\sigma, \sigma_1', \sigma_2')}$$

$$\text{Getfield (c3):}\ \frac{H\ \Sigma\ \sigma_1.f \Rightarrow H'\ \Sigma'\ \sigma_1' \quad H\ \Sigma\ \sigma_2.f \not\Rightarrow}{H\ \Sigma\ \texttt{ite}\,(\sigma, \sigma_1, \sigma_2)\,.f \to H'\ \Sigma', \sigma\ \sigma_1'}$$

$$\text{Getfield (c4):}\ \frac{H\ \Sigma\ \sigma_1.f \not\Rightarrow \quad H\ \Sigma\ \sigma_2.f \Rightarrow H'\ \Sigma'\ \sigma_2'}{H\ \Sigma\ \texttt{ite}\,(\sigma, \sigma_1, \sigma_2)\,.f \to H'\ \Sigma', \neg\sigma\ \sigma_2'}$$

$$\text{Getfield (c5):}\ \frac{}{H\ \Sigma\ l.f \to H\ \Sigma\ H(l)(f)}$$

$$\text{Putfield (c1):}\ \frac{Y \in \mathrm{dom}(H) \quad f \in \mathrm{dom}(H(Y)) \quad H_r = update(H, Y, f, \sigma')}{H\ \Sigma\ Y.f := \sigma' \to H_r[Y \mapsto H(Y)[f \mapsto \sigma']]\ \Sigma\ \sigma'}$$

$$\text{Putfield (c2):}\ \frac{\begin{array}{c} H\ \Sigma\ \sigma_1.f := \sigma' \Rightarrow H_1'\ \Sigma_1'\ \sigma' \quad H\ \Sigma\ \sigma_2.f := \sigma' \Rightarrow H_2'\ \Sigma_2'\ \sigma' \\ H' = mergeHPf(H_1', H_2', f, \sigma) \\ \Sigma' = mergePC(\sigma, \Sigma_1', \Sigma_2'), mergeClauses(H_1', H_2', f, \sigma) \end{array}}{H\ \Sigma\ \texttt{ite}\,(\sigma, \sigma_1, \sigma_2)\,.f := \sigma' \to H'\ \Sigma'\ \sigma'}$$

$$\text{Putfield (c3):}\ \frac{H\ \Sigma\ \sigma_1.f := \sigma' \Rightarrow H'\ \Sigma'\ \sigma' \quad H\ \Sigma\ \sigma_2.f := \sigma' \not\Rightarrow}{H\ \Sigma\ \texttt{ite}\,(\sigma, \sigma_1, \sigma_2)\,.f := \sigma' \to H'\ \Sigma', \sigma\ \sigma'}$$

$$\text{Putfield (c4):}\ \frac{H\ \Sigma\ \sigma_1.f := \sigma' \not\Rightarrow \quad H\ \Sigma\ \sigma_2.f := \sigma' \Rightarrow H'\ \Sigma'\ \sigma'}{H\ \Sigma\ \texttt{ite}\,(\sigma, \sigma_1, \sigma_2)\,.f := \sigma' \to H'\ \Sigma', \neg\sigma\ \sigma'}$$

$$\text{Putfield (c5):}\ \frac{}{H\ \Sigma\ l.f := \sigma' \to H[l \mapsto H(l)[f \mapsto \sigma']]\ \Sigma\ \sigma'}$$

$$\text{Let:}\ \frac{}{H\ \Sigma\ \texttt{let}\ x := \sigma\ \texttt{in}\ e \to H\ \Sigma\ e[\sigma/x]} \qquad \text{Op (c1):}\ \frac{r_1\ op\ r_2 = r}{H\ \Sigma\ r_1\ op\ r_2 \to H\ \Sigma\ r}$$

$$\text{Op (c2):}\ \frac{op\ r_1 = r}{H\ \Sigma\ op\ r_1 \to H\ \Sigma\ r} \qquad \text{Eq (c1):}\ \frac{}{H\ \Sigma\ v = v \to H\ \Sigma\ \texttt{true}}$$

$$\text{Eq (c2):}\ \frac{r_1 \neq r_2}{H\ \Sigma\ r_1 = r_2 \to H\ \Sigma\ \texttt{false}} \qquad \text{Eq (c3):}\ \frac{l_1 \neq l_2}{H\ \Sigma\ l_1 = l_2 \to H\ \Sigma\ \texttt{false}}$$

$$\text{Eq (c4):}\ \frac{}{H\ \Sigma\ l = \texttt{null} \to H\ \Sigma\ \texttt{false}} \qquad \text{Eq (c5):}\ \frac{}{H\ \Sigma\ \texttt{null} = l \to H\ \Sigma\ \texttt{false}}$$

$$\text{Eq (c6):}\ \frac{}{H\ \Sigma\ Y = l \to H\ \Sigma\ \texttt{false}} \qquad \text{Eq (c7):}\ \frac{}{H\ \Sigma\ l = Y \to H\ \Sigma\ \texttt{false}}$$

Fig. 4. Small-step operational semantics, computation transition (pt. 1)

$$\text{If (c1):}\ \frac{}{H\ \Sigma\ \texttt{if true}\ e_1\ e_2 \to H\ \Sigma\ e_1}$$

$$\text{If (c2):}\ \frac{}{H\ \Sigma\ \texttt{if false}\ e_1\ e_2 \to H\ \Sigma\ e_2}$$

$$\text{If (c3):}\ \frac{}{H\ \Sigma\ \texttt{if}\ \sigma\ e_1\ e_2 \to H\ \Sigma, \sigma\ e_1} \qquad \text{If (c4):}\ \frac{}{H\ \Sigma\ \texttt{if}\ \sigma\ e_1\ e_2 \to H\ \Sigma, \neg\sigma\ e_2}$$

$$\text{Invoke (c1):}\ \frac{H(l) = o^c \quad recv(c, m) = c' \quad method(c', m) = t\ m\ (\overline{t\ x})\ \{e_m\}}{H\ \Sigma\ l.m(\overline{\sigma}) \to H\ \Sigma\ e_m[l/\texttt{this}][\overline{\sigma/x}]}$$

$$\text{Invoke (c2):}\ \frac{method(c', m) = t\ m\ (\overline{t\ x})\ \{e_m\} \quad O = \{c \mid overrides(c, m, c')\}}{H\ \Sigma\ Y.m(\overline{\sigma}) \to H\ \Sigma, \neg Y = null, Y \leq: c', \overline{\neg Y \leq: c}_{c \in O}\ e_m[l/\texttt{this}][\overline{\sigma/x}]}$$

$$\text{Invoke (c3):}\ \frac{H\ \Sigma\ \sigma_1.m(\overline{\sigma}) \to H\ \Sigma_1'\ \eta_1'}{H\ \Sigma\ \texttt{ite}\,(\sigma, \sigma_1, \sigma_2)\,.m(\overline{\sigma}) \to H\ \Sigma_1', \sigma\ \eta_1'}$$

$$\text{Invoke (c4):}\ \frac{H\ \Sigma\ \sigma_2.m(\overline{\sigma}) \to H\ \Sigma_2'\ \eta_2'}{H\ \Sigma\ \texttt{ite}\,(\sigma, \sigma_1, \sigma_2)\,.m(\overline{\sigma}) \to H\ \Sigma_2', \neg\sigma\ \eta_2'}$$

$$\text{Ctx (c):}\ \frac{H\ \Sigma\ \eta \to H'\ \Sigma'\ \eta'}{H\ \Sigma\ E[\eta/-] \to H'\ \Sigma'\ E[\eta'/-]}$$

Fig. 5. Small-step operational semantics, computation transition (pt. 2)

$$\text{Getfield (r1):}\ \frac{Y \notin \mathrm{dom}(H) \quad fields(class(f)) = \overline{f'}}{H\ \Sigma\ Y.f \rightsquigarrow H, Y \mapsto (\overline{f' \mapsto \bot})\ \Sigma, \neg Y = \texttt{null}, Y \leq: c\ Y.f}$$

$$\text{Getfield (r1.b):}\ \frac{Y \in \mathrm{dom}(H) \quad f \notin \mathrm{dom}(H(Y)) \quad fields(class(f)) \setminus fields(H(Y)) = \overline{f'}}{H\ \Sigma\ Y.f \rightsquigarrow H[Y \mapsto (H(Y), \overline{f' \mapsto \bot})]\ \Sigma, Y \leq: c\ Y.f}$$

$$\text{Getfield (r2):}\ \frac{Y \in \mathrm{dom}(H) \quad f \in \mathrm{dom}(H(Y)) \quad H(Y)(f) = \bot \quad Z \text{ fresh}}{H\ \Sigma\ Y.f \rightsquigarrow H[Y \mapsto H(Y)[f \mapsto assume(H, Y, f, Z)]]\ \Sigma, Y.f = Z\ Y.f}$$

$$\text{Putfield (r1):}\ \frac{Y \notin \mathrm{dom}(H) \quad fields(class(f)) = \overline{f'}}{H\ \Sigma\ Y.f := \sigma' \rightsquigarrow H, Y \mapsto (\overline{f' \mapsto \bot})\ \Sigma, \neg Y = \texttt{null}, Y \leq: c\ Y.f := \sigma'}$$

$$\text{Putfield (r1.b):}\ \frac{Y \in \mathrm{dom}(H) \quad f \notin \mathrm{dom}(H(Y)) \quad fields(class(f)) \setminus fields(H(Y)) = \overline{f'}}{H\ \Sigma\ Y.f := \sigma' \rightsquigarrow H[Y \mapsto (H(Y), \overline{f' \mapsto \bot})]\ \Sigma, Y \leq: c\ Y.f := \sigma'}$$

$$\text{Ctx (r):}\ \frac{H\ \Sigma\ \eta \rightsquigarrow H'\ \Sigma'\ \eta'}{H\ \Sigma\ E[\eta/-] \rightsquigarrow H'\ \Sigma'\ E[\eta'/-]}$$

Fig. 6. Small-step operational semantics, refinement transition

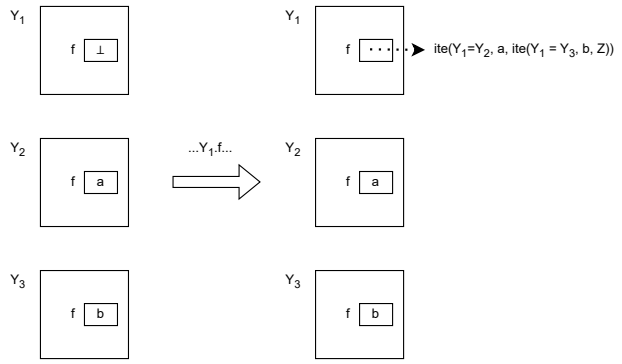

Fig. 7. Example semantics of field reading expressions

Getfield (r1.b) handles the situation where Y has an associated symbolic object in the heap, but this does not contain the field f. If we assume the programs to be type-correct, this situation may only arise if the symbolic object was previously created with the help of a superclass of c. Note that, upon creation, a symbolic object is assumed to possibly be of *any subclass* of its creation class. If later during symbolic execution a finer type assumption must be performed, as in this case, the missing fields are added to the object, with all the new slots left uninitialized. Also, the path condition is updated to reflect the new type assumption on the symbolic object bound to Y ($Y \leq: c$). Finally, the rule Getfield (r2) handles the situation where the field f in the symbolic object pointed by Y is uninitialized ($H(Y)(f) = \bot$). The refinement rule therefore initializes the field with a special value $assume(H, Y, f, Z)$ that accounts for all the possible alias relations between the symbolic reference Y and all the other symbolic references that are bound. Formally the function $assume(H, Y, f, Z)$ is defined as follows:

$$assume(H, Y, f, Z) = \texttt{ite}\,(Y = Y_1, H(Y_1)(f), \ldots, \\ \texttt{ite}\,(Y = Y_m, H(Y_m)(f), Z)\ \ldots\,)$$

where $\{Y_1 \ldots Y_m\} = \{Y' \in \mathrm{dom}(H) \mid Y' \neq Y \wedge f \in \mathrm{dom}(H(Y')) \wedge H(Y')(f) \neq \bot\}$. The set $\{Y_1 \ldots Y_m\}$ is the *alias set* of Y, and is composed by the set of all the bound symbolic references that point to a type-compatible object with c, and whose f field is initialized. The symbolic value Z stands for the initial value of $Y.f$, the value that would have the field $Y.f$ if symbolic execution up to the current instant of time did not modify the field. The $assume$ value that is assigned to the $Y.f$ field reflects the possible current value of the field: For all Y_i, If Y aliases Y_i, then $Y.f$ has the value stored in $Y_i.f$. If Y does not alias any object in its alias set, then it still contains the unmodified initial value Z.

Figure 7 reports an example illustrating the effect of the evaluation of a field reading expression. Let us suppose that the field $Y_1.f$ is uninitialized, that the expression $Y_1.f$ must be evaluated, and that two symbolic references Y_2 and Y_3 are bound in the heap with a value for field f. The effect of refinement is to initialize field $Y_1.f$ to the ite value shown in figure, which takes into account the possible alias relations of Y_1 with Y_2 and Y_3. This value is returned as a result of the evaluation of the expression.

The computation semantics of field assignment expressions, $e.f := e$, can be described as follows. Rule Putfield (c5) is for concrete objects: If l is the location of a concrete object, the field f of the object is directly updated. More complex is the effect of updating a field of a symbolic object, as described by rule Putfield (c1). The update can be performed only if Y is bound and the symbolic object has field f: Otherwise, refinement is necessary: The corresponding refinement transitions, Putfield (r1) and Putfield (r1.b), are almost identical to Getfield (r1) and Getfield (r1.b), and will not be discussed. The effect of a field update does not only affect the symbolic object bound to Y: It affects all the symbolic objects in the alias set of Y. Formally the impact of the update on the alias set is described by the definition of $H_r = update(H, Y, f, \sigma')$:

$$\mathrm{dom}(H_r) = \mathrm{dom}(H) \wedge \forall\, u' \in \mathrm{dom}(H),\ f' \in \mathrm{dom}(H)(Y') \mid \\ u' = Y' \wedge Y' \neq Y \wedge f' = f \wedge H(Y')(f) \neq \bot \\ \Longrightarrow H_r(u')(f') = \texttt{ite}\,(Y = Y', \sigma', H(u')(f')) \wedge \\ \neg(u' = Y' \wedge Y' \neq Y \wedge f' = f \wedge H(Y')(f) \neq \bot) \\ \Longrightarrow H_r(u')(f') = H(u')(f').$$

In plain terms, if Y' is a bound symbolic reference to a type-compatible object, whose f field is initialized, the field $Y'.f$ must also be updated to an ite reflecting the potential alias relation: If Y aliases Y', then $Y'.f$ also assumes the value assigned to $Y.f$, otherwise it retains its current value. Finally, rules Putfield (c2)-(c4) handle the case where an $\texttt{ite}\,(\sigma, \sigma_1, \sigma_2)$ symbolic reference is used to refer the object to be updated. Rules Putfield (c3) and (c4) are analogous to Getfield (c3) and (c4), and will not be discussed. Rule Putfield (c2) defines the effects of a field update on the heap and the path condition in terms of a suitable merging between the effects of updating the objects referred by σ_1 and σ_2 respectively. The formal definitions of the $mergeHPf$ and $mergeClauses$ functions are as follows. It is $H' = mergeHPf(H'_1, H'_2, f, \sigma)$ iff $\mathrm{dom}(H') = \mathrm{dom}(H'_1) \cup \mathrm{dom}(H'_2)$, and for all $u \in \mathrm{dom}(H')$ it is:

1) if $u \in \mathrm{dom}(H'_1)$, and $u \notin \mathrm{dom}(H'_2)$, then $\mathrm{dom}(H'(u)) = \mathrm{dom}(H'_1(u))$, and for all $f \in \mathrm{dom}(H'(u))$:
   a) if $H'_1(u)(f) = \bot$ then it is $H'(u)(f) = \bot$;
   b) otherwise it is $H'(u)(f) = \texttt{ite}\,(\sigma, \sigma_1, Z)$, where $\sigma_1 = H'_1(u)(f)$ and Z is a fresh symbol;
2) if $u \notin \mathrm{dom}(H'_1)$, and $u \in \mathrm{dom}(H'_2)$, then $\mathrm{dom}(H'(u)) = \mathrm{dom}(H'_2(u))$, and for all $f \in \mathrm{dom}(H'(u))$:
   a) if $H'_2(u)(f) = \bot$ then it is $H'(u)(f) = \bot$;
   b) otherwise it is $H'(u)(f) = \texttt{ite}\,(\sigma, Z, \sigma_2)$, where $\sigma_2 = H'_2(u)(f)$ and Z is a fresh symbol;
3) if $u \in \mathrm{dom}(H'_1)$, and $u \in \mathrm{dom}(H'_2)$, and $H'_1(u) = H'_2(u)$, then $H'(u) = H'_1(u) = H'_2(u)$;
4) if $u \in \mathrm{dom}(H'_1)$, and $u \in \mathrm{dom}(H'_2)$, and $H'_1(u) \neq H'_2(u)$, then $\mathrm{dom}(H'(u)) = \mathrm{dom}(H'_1(u)) \cup \mathrm{dom}(H'_2(u))$, and for all $f \in \mathrm{dom}(H'(u))$:

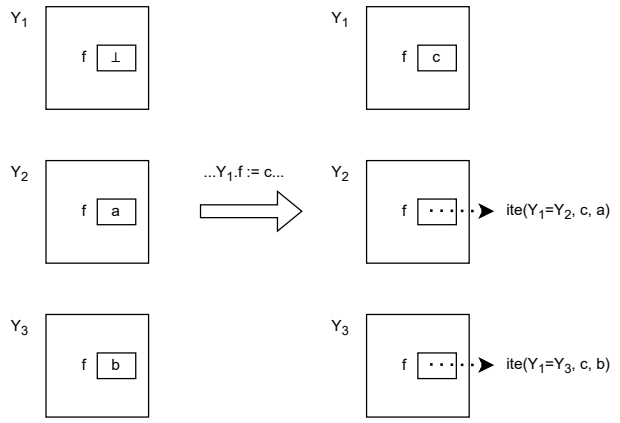

Fig. 8. Example semantics of field assignment expressions

a) if $f \in \mathrm{dom}(H'_1(u))$ and $f \notin \mathrm{dom}(H'_2(u))$, then:
   i) if $H'_1(u)(f) = \bot$ then $H'(u)(f) = \bot$;
   ii) otherwise, $H'(u)(f) = \texttt{ite}(\sigma, \sigma_1, Z)$, where $\sigma_1 = H'_1(u)(f)$ and Z is a fresh symbol;
b) if $f \in \mathrm{dom}(H'_2(u))$ and $f \notin \mathrm{dom}(H'_1(u))$, then:
   i) if $H'_2(u)(f) = \bot$ then $H'(u)(f) = \bot$;
   ii) otherwise, $H'(u)(f) = \texttt{ite}(\sigma, Z, \sigma_2)$, where $\sigma_2 = H'_2(u)(f)$ and Z is a fresh symbol;
c) if $f \in \mathrm{dom}(H'_1(u))$, and $f \in \mathrm{dom}(H'_2(u))$, and $H'_1(u)(f) = H'_2(u)(f)$ then $H'(u)(f) = H'_1(u)(f) = H'_2(u)(f)$;
d) if $f \in \mathrm{dom}(H'_1(u))$, and $f \in \mathrm{dom}(H'_2(u))$, and $H'_1(u)(f) \neq \bot$, and $H'_2(u)(f) = \bot$, then $H'(u)(f) = \texttt{ite}(\sigma, \sigma_1, Z)$, where $\sigma_1 = H'_1(u)(f)$ and Z is a fresh symbol;
e) if $f \in \mathrm{dom}(H'_1(u))$, and $f \in \mathrm{dom}(H'_2(u))$, and $H'_1(u)(f) = \bot$, and $H'_2(u)(f) \neq \bot$, then $H'(u)(f) = \texttt{ite}(\sigma, Z, \sigma_2)$, where $\sigma_2 = H'_2(u)(f)$ and Z is a fresh symbol;
f) if $f \in \mathrm{dom}(H'_1(u))$, and $f \in \mathrm{dom}(H'_2(u))$, and $H'_1(u)(f) \neq H'_2(u)(f)$, and $H'_1(u)(f) \neq \bot$, and $H'_2(u)(f) \neq \bot$ then $H'(u)(f) = \texttt{ite}(\sigma, \sigma_1, \sigma_2)$, where $\sigma_1 = H'_1(u)(f)$ and $\sigma_2 = H'_2(u)(f)$.

The function $mergeClauses(H'_1, H'_2, f, \sigma)$ yields the set of all the clauses $Y.f = Z$, where Z are all (and only) the fresh symbols introduced in the definition of $mergeHPf$, and Y is the symbolic input reference that points to the symbolic object where Z has been injected.

Figure 8 reports an example of the effect of the evaluation of a field assignment expression. We again suppose that three symbolic references, Y_1, Y_2, Y_3, are bound to symbolic objects, that all the symbolic objects define the field f, that the object Y_2 and Y_3 have field f initialized, and that the expression $Y_1.f := c$ must be evaluated. In this situation no refinement transition fires: The computation transition updates the field f of Y_1 to the c value, and as a further effect updates the fields f of all the other symbolic objects with `ite` terms that reflect the possible effect of the update on these objects in case an alias relation subsists.

Conditional expressions `if` e e e have associated only computation transitions. Rules If (c1) and If (c2) manage the cases `if true...` and `if false...`, that reduce to the evaluation of the corresponding branch. Rules If (c3) and If (c4) manage the case where the condition of the `if` expression is symbolic. In this case both transitions may fire, and the path condition is updated to reflect the assumption associated to the branch taken.

Finally, method invocation expressions $e.m(\overline{e})$ are handled only by computation transitions. We introduce the following definitions:

- The function $recv(c, m)$ yields the class that provides the implementation of method m to class c (it is c itself iff c implements m, or an ancestor of c iff c inherits the implementation of m).
- The predicate $impl(c, m)$ is true iff c implements m.
- The predicate $sees(c, m, c')$ is true iff c is a subclass of c', c' implements the method m, and no superclass of c and subclass of c' implements m (note that c may implement m).
- The predicate $overrides(c, m, c')$ is defined as $c \neq c' \wedge sees(c, m, c') \wedge impl(c, m)$.

We can deduce that $recv(c, m) = c'$, for $c \neq c'$, iff $sees(c, m, c') \wedge \neg impl(c, m)$, and that $recv(c, m) = c$ iff $impl(c, m)$. Given the above definitions it is easy to see that rule Invoke (c1) (method invocation on concrete objects) finds the implementation of the method of the object pointed by location l, and invokes it. Different is the case (rule Invoke (c2)) where the receiver of the method invocation is a symbolic reference. In this case, since the symbolic reference might refer to an object of any class that declares the method m, symbolic execution branches to all possible implementations of m. The path condition at each branch is assumed to reflect the assumption on the class of the symbolic receiver object: When the method m is executed, if m is declared by class c', the object may have any subclass of c' that does not override m. The final rules Invoke (c3) and Invoke (c4) handle the case where the reference to the receiver object is an $\texttt{ite}(\sigma, \sigma_1, \sigma_2)$ symbolic reference. In synthesis, when such a reference is encountered, symbolic execution branches.

We conclude this Appendix by briefly discussing the remaining rules. The Let computation rule has the effect of performing a term substitution in its main argument. The Op (c1) and Op (c2) computation rules yield the result of the application of a binary or unary operator to concrete operands. The Eq (c1)-(c7) computation rules account for the semantics of equality, where syntactically identical values are always considered equal, distinct concrete locations are always considered different each other and from `null`, and symbolic references are always considered different from concrete locations. Finally, the context rules Ctx (c) and Ctx (r) extend the reduction semantics of expressions under the admissible evaluation contexts.