

Symbolic Execution for Implicit Dynamic Frames Soundness Proof

Jan Smans

Bart Jacobs

Frank Piessens

April 28, 2009

In this note we prove the soundness of our symbolic execution approach for implicit dynamic frames. Sections 1 and 2 define the syntax and run-time semantics, respectively, of programs. The run-time semantics is standard and ignores annotations. Before we define our symbolic execution-based verification approach in Section 4, we define a version of it where concrete values are used instead of symbols in Section 3, and we prove its soundness. We call the latter *abstracted execution*, since it uses abstract heaps instead of concrete heaps. In Section 4, we prove that symbolic execution simulates abstracted execution.

1 Language

We define the following sets. \mathcal{X} is the set of variable names with typical element x . \mathcal{C} is the set of class names with typical element C . \mathcal{F} is the set of field names with typical element f . \mathcal{M} is the set of mutator names with typical element m . \mathcal{P} is the set of pure method names with typical element p . \mathcal{Q} is the set of predicate method names with typical element q .

The syntax of programs is shown in Figure 1. Overlining indicates repetition. A program consists of a number of classes and a main routine \bar{s} . Each class declares a number of fields and methods. We distinguish three kinds of methods: mutators, pure methods and predicates. The body of a mutator consists of a list of statements, while the body of a pure method and a predicate consist of a single return statement, returning an expression, respectively an assertion. A statement s is either a local variable declaration, a variable update, a field update, a mutator invocation, an object creation, an if-then-else statement, an assert statement, an open statement, a close statement or a use statement. An expression e is either *null*, a variable, a field read, a pure method invocation, an old expression, an opening expression, a using expression or a conditional expression. An assertion ϕ is **true**, **false**, an access assertion, an equality, a separating conjunction, a predicate invocation, a conditional assertion or an untouched assertion.

2 Concrete execution

A continuation is either **done**, a statement continuation $s; \kappa$, or a return continuation **ret**(Γ, κ).

A configuration consists of a heap, a store, and a continuation.

$prog ::= \overline{class \bar{s}}$
 $class ::= \mathbf{class} C \{ \overline{field} \overline{method} \}$
 $field ::= C f;$
 $method ::= mutator \mid pure \mid pred$
 $mutator ::= \mathbf{void} m(\overline{C} x) \mathbf{requires} \phi; \mathbf{ensures} \phi; \{ \bar{s} \}$
 $pure ::= \mathbf{pure} C p(\overline{C} x) \mathbf{requires} \phi; \{ \mathbf{return} e; \}$
 $pred ::= \mathbf{predicate} q(\overline{t} x) \{ \mathbf{return} \phi; \}$
 $s ::= C x; \mid x := e; \mid e.f := e; \mid e.m(\bar{e}) \mid$
 $x := \mathbf{new} C; \mid \mathbf{if}(e = e) \{ \bar{s} \} \mathbf{else} \{ \bar{s} \} \mid$
 $\mathbf{assert} e_1 = e_2; \mid \mathbf{open} e.q(\bar{e}); \mid \mathbf{close} e.q(\bar{e}); \mid$
 $\mathbf{use} e.p(\bar{e})$
 $e ::= \mathbf{null} \mid x \mid e.f \mid e.p(\bar{e}) \mid \mathbf{old}(e) \mid$
 $\mathbf{opening} e.q(\bar{e}) \mathbf{in} e \mid \mathbf{using} e.p(\bar{e}) \mathbf{in} e \mid$
 $\mathbf{dropping} e.f \mathbf{in} e \mid e = e ? e : e$
 $\phi ::= \mathbf{true} \mid \mathbf{false} \mid \mathbf{acc}(e.f) \mid e = e \mid \phi * \phi \mid e.q(\bar{e}) \mid$
 $e = e ? \phi : \phi \mid \mathbf{untouched}(\phi)$

Figure 1: Syntax of a small Java-like language.

$$\begin{array}{c}
\frac{}{H, \Gamma \vdash \mathbf{null} \Downarrow \mathbf{null}} \quad \frac{}{H, \Gamma \vdash x \Downarrow \Gamma(x)} \quad \frac{H, \Gamma \vdash e \Downarrow v \quad (v, f) \in \text{dom}(H)}{H, \Gamma \vdash e.f \Downarrow H(v, f)} \\
\\
\frac{\frac{H, \Gamma \vdash e_0, \dots, e_n \Downarrow v_0, \dots, v_n \quad v_0 \neq \mathbf{null}}{\mathbf{pure} C p(C_1 x_1, \dots, C_n x_n) \mathbf{requires} \phi_{\text{pre}}; \{ \mathbf{return} e; \}} \quad H, [\mathbf{this} \mapsto v_0, \dots, x_n \mapsto v_n] \vdash e \Downarrow v}{H, \Gamma \vdash e_0.p(e_1, \dots, e_n) \Downarrow v}}{} \\
\\
\frac{}{H, \Gamma \vdash \mathbf{opening} e.q(\bar{e}) \mathbf{in} e \Downarrow v} \quad \frac{}{H, \Gamma \vdash \mathbf{using} e.p(\bar{e}) \mathbf{in} e \Downarrow v} \quad \frac{}{H, \Gamma \vdash \mathbf{dropping} e.f \mathbf{in} e \Downarrow v} \\
\\
\frac{H, \Gamma \vdash e_1, e_2, e_3 \Downarrow v_1, v_1, v_3}{H, \Gamma \vdash e_1 = e_2 ? e_3 : e_4 \Downarrow v_3} \quad \frac{H, \Gamma \vdash e_1, e_2, e_4 \Downarrow v_1, v_2, v_4 \quad v_1 \neq v_2}{H, \Gamma \vdash e_1 = e_2 ? e_3 : e_4 \Downarrow v_4}
\end{array}$$

$$\begin{array}{c}
\text{STEP-DECL} \\
\langle H, \Gamma, Cx; \kappa \rangle \rightsquigarrow \langle H, \Gamma[x \mapsto \text{null}], \kappa \rangle
\end{array}
\qquad
\begin{array}{c}
\text{STEP-ASSIGN} \\
\frac{H, \Gamma \vdash e \Downarrow v}{\langle H, \Gamma, x := e; \kappa \rangle \rightsquigarrow \langle H, \Gamma[x \mapsto v], \kappa \rangle}
\end{array}$$

$$\begin{array}{c}
\text{STEP-WRITE} \\
\frac{H, \Gamma \vdash e_1 \Downarrow v_1 \quad (v_1, f) \in H \quad H, \Gamma \vdash e_2 \Downarrow v_2}{\langle H, \Gamma, e.f := e; \kappa \rangle \rightsquigarrow \langle H[(v_1, f) \mapsto v_2], \Gamma, \kappa \rangle}
\end{array}$$

STEP-CALL

$$\frac{H, \Gamma \vdash e_0, \dots, e_n \Downarrow v_0, \dots, v_n \quad \text{void } m(C_1 x_1, \dots, C_n x_n) \text{ requires } \phi_{\text{pre}}; \text{ ensures } \phi_{\text{post}}; \{ \bar{s} \} \quad \Gamma' = [\text{this} \mapsto v_0, \dots, x_n \mapsto v_n]}{\langle H, \Gamma, e_0.m(e_1, \dots, e_n); \kappa \rangle \rightsquigarrow \langle H, \Gamma', \bar{s}; \text{ret}(\Gamma, \kappa) \rangle}$$

STEP-RETURN

$$\frac{}{\langle H, \Gamma, \text{ret}(\Gamma', \kappa) \rangle \rightsquigarrow \langle H, \Gamma', \kappa \rangle}$$

STEP-NEW

$$\frac{o \notin \text{dom}(H) \quad \text{class } C \{ C_1 f_1; \dots C_n f_n; \dots \} \quad o \neq \text{null} \quad H' = H[(o, f_1) \mapsto \text{null}, \dots, (o, f_n) \mapsto \text{null}]}{\langle H, \Gamma, x := \text{new } C; \kappa \rangle \rightsquigarrow \langle H', \Gamma[x \mapsto o], \kappa \rangle}$$

STEP-IF-TRUE

$$\frac{H, \Gamma \vdash e_1 \Downarrow v \quad H, \Gamma \vdash e_2 \Downarrow v}{\langle H, \Gamma, \text{if } (e_1 = e_2) \{ \bar{s} \} \text{ else } \{ \bar{s}' \} \kappa \rangle \rightsquigarrow \langle H, \Gamma, \bar{s}; \kappa \rangle}$$

STEP-IF-FALSE

$$\frac{H, \Gamma \vdash e_1 \Downarrow v_1 \quad H, \Gamma \vdash e_2 \Downarrow v_2 \quad v_1 \neq v_2}{\langle H, \Gamma, \text{if } (e_1 = e_2) \{ \bar{s} \} \text{ else } \{ \bar{s}' \} \kappa \rangle \rightsquigarrow \langle H, \Gamma, \bar{s}'; \kappa \rangle}$$

STEP-ASSERT

$$\frac{H, \Gamma \vdash e_1 \Downarrow v \quad H, \Gamma \vdash e_2 \Downarrow v}{\langle H, \Gamma, \text{assert } e_1 = e_2; \kappa \rangle \rightsquigarrow \langle H, \Gamma, \kappa \rangle}$$

STEP-OPEN

$$\langle H, \Gamma, \text{open } e.q(\bar{e}); \kappa \rangle \rightsquigarrow \langle H, \Gamma, \kappa \rangle$$

STEP-CLOSE

$$\langle H, \Gamma, \text{close } e.q(\bar{e}); \kappa \rangle \rightsquigarrow \langle H, \Gamma, \kappa \rangle$$

STEP-USE

$$\langle H, \Gamma, \text{use } e.p(\bar{e}); \kappa \rangle \rightsquigarrow \langle H, \Gamma, \kappa \rangle$$

Lemma 1. *Concrete evaluation is unique.*

$$(H, \Gamma \vdash e \Downarrow v_1) \Rightarrow (H, \Gamma \vdash e \Downarrow v_2) \Rightarrow v_1 = v_2$$

Proof. By induction on the derivation of the first premise. □

3 Abstracted execution

Abstracted execution differs from concrete execution in that it abstracts over the heap, over pure method calls, and over mutator calls.

- A concrete heap contains only field chunks; an abstract heap contains *predicate chunks* as well. A predicate chunk abstracts over a *nested heap*. The nested heap is fully determined by its *snapshot*.
- In abstracted execution, pure method calls are evaluated using an interpretation function which takes as input the argument values and the precondition's *snapshot*, rather than by recursively evaluating the body of the pure method. This enables modular reasoning about pure method calls.
- In abstracted execution, mutator calls are verified using their contract instead of by recursively verifying their body.

3.1 Concrete snapshots

Definition 1. The set *Snapshots* of concrete snapshots is defined inductively as follows:

- *unit* is a concrete snapshot
- if $o \in \mathcal{O}$, then $\text{fromRef}(o)$ is a concrete snapshot
- if θ_1 and θ_2 are concrete snapshots, then $\text{combine}(\theta_1, \theta_2)$ is a concrete snapshot

Definition 2. We define the functions toRef , first , and second as follows:

$$\begin{array}{lll}
 \text{toRef}(\text{unit}) = \text{null} & \text{first}(\text{unit}) = \text{unit} & \text{second}(\text{unit}) = \text{unit} \\
 \text{toRef}(\text{fromRef}(o)) = o & \text{first}(\text{fromRef}(o)) = \text{unit} & \text{second}(\text{fromRef}(o)) = \text{unit} \\
 \text{toRef}(\text{combine}(\theta_1, \theta_2)) = \text{null} & \text{first}(\text{combine}(\theta_1, \theta_2)) = \theta_1 & \text{second}(\text{combine}(\theta_1, \theta_2)) = \theta_2
 \end{array}$$

3.2 Abstract heaps

Definition 3. The sets *AbstractHeaps* and *AbstractHeapChunks* are defined mutually inductively as follows:

- A field chunk $o.f \mapsto v$, where $o, v \in \mathcal{O}$, is an abstract heap chunk
- A finite multiset of abstract heap chunks is an abstract heap
- A predicate chunk $o.q[\theta, H](v_1, \dots, v_n)$, where $o, v_1, \dots, v_n \in \mathcal{O}$, $\theta \in \text{Snapshots}$, $H \in \text{AbstractHeaps}$, is an abstract heap chunk

Notice that any concrete heap can be seen as an abstract heap that contains no predicate chunks and that contains no two field chunks for the same location, and where the target of each field chunk is non-null. Conversely, any such abstract heap can be seen as a concrete heap.

Definition 4. The size of an abstract heap is the total number of nodes: the number of elements plus the sum of the sizes of the nested heaps. This means that dropping top-level nodes reduces the size, but replacing a predicate chunk by its nested heap reduces the size as well.

Definition 5. The flattening of an abstract heap is the abstract heap obtained by retaining only the leaf nodes (i.e., the field chunks, including those in transitively nested heaps).

3.3 Abstract expression evaluation

$$\begin{array}{c}
P, H, G, \Gamma \vdash \mathbf{null} \Downarrow \mathbf{null} \quad P, H, G, \Gamma \vdash x \Downarrow \Gamma(x) \quad \frac{P, H, G, \Gamma \vdash e \Downarrow o \quad o.f \mapsto v \in H}{P, H, G, \Gamma \vdash e.f \Downarrow v} \\
\\
\frac{P, H, G, \Gamma \vdash e_0, \dots, e_n \Downarrow v_0, \dots, v_n \quad \mathbf{pure} \ C \ p(C_1 \ x_1, \dots, C_n \ x_n) \ \mathbf{requires} \ \phi; \ \{ \mathbf{return} \ e; \} \quad p \in P \quad \{p' \bullet p' < p\}, H, \emptyset, \Gamma \vdash e \Downarrow v}{P, H, G, \Gamma \vdash e_0.p(e_1, \dots, e_n) \Downarrow v} \\
\\
\frac{P, G, \emptyset, \Gamma \vdash e \Downarrow v}{P, H, G, \Gamma \vdash \mathbf{old}(e) \Downarrow v} \\
\\
\frac{P, H, G, \Gamma \vdash e_0, \dots, e_n \Downarrow v_0, \dots, v_n \quad v_0.q[\theta, H_0](v_1, \dots, v_n) \in H \quad \mathbf{predicate} \ q(C_1 \ x_1, \dots, C_n \ x_n) \ \{ \mathbf{return} \ \phi; \} \quad \mathcal{P}, H - \{v_0.q[\theta, H_0](v_1, \dots, v_n)\} + H_0, G, \Gamma \vdash e \Downarrow v}{P, H, G, \Gamma \vdash \mathbf{opening} \ e_0.q(e_1, \dots, e_n) \ \mathbf{in} \ e \Downarrow v} \\
\\
\frac{P, H, G, \Gamma \vdash e \Downarrow v}{P, H, G, \Gamma \vdash \mathbf{using} \ e_0.p(e_1, \dots, e_n) \ \mathbf{in} \ e \Downarrow v} \\
\\
\frac{P, H, G, \Gamma \vdash e \Downarrow o \quad (o.f \mapsto v_0) \in H \quad \mathcal{P}, H - \{o.f \mapsto v_0\}, G, \Gamma \vdash e \Downarrow v}{P, H, G, \Gamma \vdash \mathbf{dropping} \ e.f \ \mathbf{in} \ e \Downarrow v} \\
\\
\frac{P, H, G, \Gamma \vdash e_1, e_2, e_3 \Downarrow v_1, v_1, v}{P, H, G, \Gamma \vdash \mathbf{if} \ (e_1 = e_2) \ e_3 \ \mathbf{else} \ e_4 \Downarrow v} \quad \frac{P, H, G, \Gamma \vdash e_1, e_2, e_4 \Downarrow v_1, v_2, v \quad v_1 \neq v_2}{P, H, G, \Gamma \vdash \mathbf{if} \ (e_1 = e_2) \ e_3 \ \mathbf{else} \ e_4 \Downarrow v}
\end{array}$$

Lemma 2 (Soundness of abstract expression evaluation). *If $P, H, \emptyset, \Gamma \vdash e \Downarrow v$ and e does not contain **old** expressions, then $\mathbf{flat}(H), \Gamma \vdash e \Downarrow v$.*

Proof. By induction on the derivation of the premise. The only non-trivial case is the case of **opening** expressions. Here the goal follows immediately from the fact that opening an arbitrary predicate chunk does not change the flattening of the heap. \square

Lemma 3 (Expression evaluation monotonicity).

$$P, H, \emptyset, \Gamma \vdash e \Downarrow v \Rightarrow P \subseteq P' \Rightarrow H \subseteq H' \Rightarrow P', H', \emptyset, \Gamma \vdash e \Downarrow v$$

Proof. By induction on the derivation of the premise. \square

3.4 Expression evaluation order

Definition 6. *We define a partial order on triples (H, P, e) where H is an abstract heap, P is a set of pure method names, and e is an expression. We say that $(H_1, P_1, e_1) \prec (H_2, P_2, e_2)$ if either*

- H_1 is smaller than H_2 , or
- H_1 is equal to H_2 but P_1 is a strict prefix of P_2 , i.e. there is a $p \in P_2$ such that $\forall p' \in P_1 \bullet p' < p$, or
- $H_1 = H_2$ and $P_1 = P_2$ but e_1 is syntactically smaller than e_2

Lemma 4. *The \prec order is a well-founded order.*

Lemma 5. *If the flattening of an abstract heap H is a concrete heap, then the value of an expression in H is unique:*

$$(P, H, \emptyset, \Gamma \vdash e \Downarrow v_1) \wedge (P, H, \emptyset, \Gamma \vdash e \Downarrow v_2) \Rightarrow v_1 = v_2$$

Proof. By induction according to the \prec order on (H, P, e) and case analysis on e . \square

3.5 Abstract assertion evaluation

$$\begin{array}{c}
\frac{\emptyset, G, \Gamma \vdash \phi \Downarrow \theta, H}{H, G, \Gamma \vDash \phi \Downarrow \theta} \quad \frac{H, \emptyset, \Gamma \vDash \phi \Downarrow \theta}{H, \Gamma \vDash \phi \Downarrow \theta} \quad H, G, \Gamma \vdash \mathbf{true} \Downarrow \mathbf{unit}, H \\
\\
\frac{H, G, \Gamma \vdash e \Downarrow o}{H, G, \Gamma \vdash \mathbf{acc}(e.f) \Downarrow v, H + \{o.f \mapsto v\}} \quad \frac{H, G, \Gamma \vdash e_1 \Downarrow v \quad H, G, \Gamma \vdash e_2 \Downarrow v}{H, G, \Gamma \vdash e_1 = e_2 \Downarrow \mathbf{unit}, H} \\
\\
\frac{H, G, \Gamma \vdash \phi_1 \Downarrow \theta_1, H_1 \quad H_1, G, \Gamma \vdash \phi_2 \Downarrow \theta_2, H_2}{H, G, \Gamma \vdash \phi_1 * \phi_2 \Downarrow \mathbf{combine}(\theta_1, \theta_2), H_2} \\
\\
\frac{\mathbf{predicate } q(C_1 x_1, \dots, C_n x_n) \{ \mathbf{return } \phi; \} \quad H, G, \Gamma \vdash e_0, \dots, e_n \Downarrow v_0, \dots, v_n \quad H_0, [\mathbf{this} \mapsto v_0, \dots, x_n \mapsto v_n] \vDash \phi \Downarrow \theta}{H, G, \Gamma \vdash e_0.q(e_1, \dots, e_n) \Downarrow \theta, H + \{v_0.q[\theta, H_0](v_1, \dots, v_n)\}} \\
\\
\frac{H, G, \Gamma \vdash e_1 \Downarrow v_1 \quad H, G, \Gamma \vdash e_2 \Downarrow v_2 \quad v_1 = v_2 \quad H, G, \Gamma \vdash \phi_1 \Downarrow \theta, H_1}{H, G, \Gamma \vdash e_1 = e_2 ? \phi_1 : \phi_2 \Downarrow \theta, H_1} \\
\\
\frac{H, G, \Gamma \vdash e_1 \Downarrow v_1 \quad H, G, \Gamma \vdash e_2 \Downarrow v_2 \quad v_1 \neq v_2 \quad H, G, \Gamma \vdash \phi_2 \Downarrow \theta, H_1}{H, G, \Gamma \vdash e_1 = e_2 ? \phi_1 : \phi_2 \Downarrow \theta, H_1} \\
\\
\frac{H, G, \Gamma \vdash e_0, \dots, e_n \Downarrow v_0, \dots, v_n \quad v_0.q[\theta, G_0](v_1, \dots, v_n) \in G \quad v_0.q[\theta, H_0](v_1, \dots, v_n) \in H}{H, G, \Gamma \vdash \mathbf{untouched}(e_0.q(e_1, \dots, e_n)) \Downarrow \mathbf{unit}, H}
\end{array}$$

Lemma 6.

$$(H, G, \Gamma \vdash \phi \Downarrow \theta, H') \Rightarrow H \subseteq H'$$

Proof. By induction on the derivation of $H, G, \Gamma \vdash \phi \Downarrow \theta, H'$. □

Lemma 7. *Suppose H , H_1 , and H_2 are abstract heaps such that $\mathbf{flat}(H + H_1) \in \mathbf{ConcreteHeaps}$ and $\mathbf{flat}(H + H_2) \in \mathbf{ConcreteHeaps}$. Then*

$$(H, \emptyset, \Gamma \vdash \phi \Downarrow \theta, H_1) \Rightarrow (H, \emptyset, \Gamma \vdash \phi \Downarrow \theta, H_2) \Rightarrow H_1 = H_2$$

Proof. By induction on the derivation of $H, \emptyset, \Gamma \vdash \phi \Downarrow \theta, H_1$, using the uniqueness of expression evaluation (Lemma 5). □

Lemma 8. *Suppose H_1 and H_2 are two abstract heaps whose flattenings are concrete heaps. Then if an assertion ϕ holds over both of them, with the same snapshot θ , then $H_1 = H_2$:*

$$(H_1, \Gamma \vDash \phi \Downarrow \theta) \Rightarrow (H_2, \Gamma \vDash \phi \Downarrow \theta) \Rightarrow H_1 = H_2$$

Proof. Follows immediately from Lemma 7 □

Definition 7. *We say an abstract heap is consistent if its flattening is a concrete heap and each for each predicate chunk $o.q[\theta, H_0](v_1, \dots, v_n)$, we have $H_0, \emptyset, [\mathbf{this} \mapsto o, x_1 \mapsto v_1, \dots, x_n \mapsto v_n] \vDash \phi \Downarrow \theta$ where $\mathbf{predicate } q(x_1, \dots, x_n) \{ \mathbf{return } \phi; \}$.*

Notice that if an abstract heap is consistent, then so are its nested heaps.

3.6 Interpretation of pure method calls

Definition 8. We define the value $\mathcal{I}(C.p)(\theta, v_0, \dots, v_n)$ of a pure method call with arguments v_0, \dots, v_n and snapshot θ as follows:

$$\frac{\text{pure } C \ p(C_1 \ x_1, \dots, C_n \ x_n) \ \text{requires } \phi; \ \{ \text{return } e; \} \quad \text{flat}(H) \in \text{ConcreteHeaps}}{H, [\text{this} \mapsto v_0, \dots, x_n \mapsto v_n] \models \phi \Downarrow \theta \quad \{p' \bullet p' < p\}, H, \emptyset, [\text{this} \mapsto v_0, \dots, x_n \mapsto v_n] \vdash e \Downarrow v} \mathcal{I}(C.p)(\theta, v_0, \dots, v_n) = v$$

This equation has a solution since there is at most one such heap H , and expression evaluation is unique. However, we do not claim here that the equation has only one solution. We pick any solution that satisfies the equation.

3.7 Abstract production, consumption, evaluation

The equations below define the mutually recursive functions `aproduce'`, `aproduce`, `aconsume'`, `aconsume`, and `aeval`. If evaluation of a function “call” does not terminate, we define its value to be **false**. That is: let T be the transformation defined by the equations below. It maps an interpretation for the abovementioned functions to a new interpretation. Since the functions return booleans, we can represent an interpretation as the set of the calls that return **true**. We then define the interpretation I of the functions as

$$I = \bigcup_n T^n(\emptyset)$$

Note: it is easy to see that T is monotonic: $I \subseteq T(I)$. Indeed, all calls appear in positive positions, including calls of the postcondition. Therefore, we have that I is the least fixpoint and it satisfies the equations:

$$I = T(I)$$

$$\begin{aligned} \text{aproduce}' &: \wp(\mathcal{P}) \times \text{AbstractHeaps} \times \text{Stores} \times \text{AbstractHeaps} \times \\ &\text{Snapshots} \times \text{Assertions} \times (\text{AbstractHeaps} \rightarrow \mathbb{B}) \rightarrow \mathbb{B} \end{aligned}$$

$$\begin{aligned}
&\text{aproduce}'(P, G, \Gamma, H, \text{snapshot}, \text{true}, Q) \equiv \\
&\quad Q(H) \\
&\text{aproduce}'(P, G, \Gamma, H, \text{snapshot}, \text{false}, Q) \equiv \\
&\quad \text{true} \\
&\text{aproduce}'(P, G, \Gamma, H, \text{snapshot}, \text{acc}(e.f), Q) \equiv \\
&\quad \text{aeval}(P, (H, G, \Gamma), e, (\lambda o \bullet \\
&\quad \quad \text{let } \{o_1.f_1 \mapsto v_1, \dots, o_n.f_n \mapsto v_n\} = \{o'.g \mapsto v \in H \mid f = g\} \text{ in} \\
&\quad \quad o \neq \text{null} \wedge \\
&\quad \quad (o_1 \neq o \wedge \dots \wedge o_n \neq o) \Rightarrow Q(H + \{o.f \mapsto \text{toRef}(\text{snapshot})\}))) \\
&\text{aproduce}'(P, G, \Gamma, H, \text{snapshot}, e_1 = e_2, Q) \equiv \\
&\quad \text{aeval}(P, (H, G, \Gamma), e_1, (\lambda v_1 \bullet \text{eval}(P, (H, G, \Gamma), e_2, (\lambda v_2 \bullet \\
&\quad \quad v_1 = v_2 \Rightarrow Q(H)))))) \\
&\text{aproduce}'(P, G, \Gamma, H, \text{snapshot}, \phi_1 * \phi_2, Q) \equiv \\
&\quad \text{aproduce}'(P, (H, G, \Gamma), \text{first}(\text{snapshot}), \phi_1, (\lambda H' \bullet \text{aproduce}'(P, G, \Gamma, H', \text{second}(\text{snapshot}), \phi_2, Q))) \\
&\text{aproduce}'(P, G, \Gamma, H, \text{snapshot}, e_0.q(e_1, \dots, e_n), Q) \equiv \\
&\quad \text{aeval}(P, (H, G, \Gamma), e_0, (\lambda v_0 \bullet \dots \text{aeval}(P, (H, G, \Gamma), e_n, (\lambda v_n \bullet \\
&\quad \quad v_0 \neq \text{null} \wedge \forall H_0 \bullet Q(H + \{v_0.q[\text{snapshot}, H_0](v_1, \dots, v_n)\})))))) \\
&\text{aproduce}'(P, G, \Gamma, H, \text{snapshot}, e_1 = e_2 ? \phi_1 : \phi_2, Q) \equiv \\
&\quad \text{aeval}(P, (H, G, \Gamma), e_1, (\lambda v_1 \bullet \text{aeval}(P, (H, G, \Gamma), e_2, (\lambda v_2 \bullet \\
&\quad \quad v_1 = v_2 \Rightarrow \text{aproduce}'(P, G, \Gamma, H, \text{snapshot}, \phi_1, Q) \wedge \\
&\quad \quad v_1 \neq v_2 \Rightarrow \text{aproduce}'(P, G, \Gamma, H, \text{snapshot}, \phi_2, Q)))))) \\
&\text{aproduce}'(P, (H, G, \Gamma), \text{snapshot}, \text{untouched}(\phi), Q) \equiv \\
&\quad \text{aconsume}(P, (H, G, \Gamma), \phi, (\lambda(-, -, -), \text{snapshot}' \bullet \\
&\quad \quad \text{aconsume}(P, (G, G, \Gamma), \phi, (\lambda(-, -, -), \text{snapshot}'' \bullet \\
&\quad \quad \text{snapshot}' = \text{snapshot}'' \wedge Q((H, G, \Gamma))))))
\end{aligned}$$

$$\text{aproduce} : \wp(\mathcal{P}) \times \text{AbstractStates} \times \text{Snapshots} \times \text{Assertions} \times (\text{AbstractStates} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$$

$$\begin{aligned}
&\text{aproduce}(P, (H, G, \Gamma), \text{snapshot}, \phi, Q) \equiv \\
&\quad \text{aproduce}'(P, G, \Gamma, \emptyset, \text{snapshot}, \phi, (\lambda H' \bullet Q((H + H', G, \Gamma))))
\end{aligned}$$

$$\begin{aligned}
&\text{aconsume}' : \wp(\mathcal{P}) \times \text{AbstractStates} \times \text{AbstractHeaps} \times \text{Assertions} \times \\
&\quad (\text{AbstractHeaps} \times \text{Snapshots} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}
\end{aligned}$$

$$\begin{aligned}
& \text{aconsume}'(P, (H, G, \Gamma), H_2, \mathbf{true}, Q) \equiv \\
& \quad Q(H_2, \mathbf{unit}) \\
& \text{aconsume}'(P, (H, G, \Gamma), H_2, \mathbf{false}, Q) \equiv \\
& \quad \mathbf{false} \\
& \text{aconsume}'(P, (H, G, \Gamma), H_2, \mathbf{acc}(e.f), Q) \equiv \\
& \quad \text{aeval}(P, (H, G, \Gamma), e, (\lambda o \bullet \\
& \quad \quad \mathbf{let} \text{ matches} = \{o'.g \mapsto v \in H_2 \mid f = g \wedge o = o'\} \mathbf{in} \\
& \quad \quad \exists o'.g \mapsto v \in \text{matches} \bullet Q(H_2 - \{o'.g \mapsto v\}, \mathbf{fromRef}(v)))) \\
& \text{aconsume}'(P, (H, G, \Gamma), H_2, e_1 = e_2, Q) \equiv \\
& \quad \text{aeval}(P, (H, G, \Gamma), e_1, (\lambda v_1 \bullet \text{aeval}((H, G, \Gamma), e_2, (\lambda v_2 \bullet \\
& \quad \quad v_1 = v_2 \wedge Q(H_2, \mathbf{unit})))))) \\
& \text{aconsume}'(P, (H, G, \Gamma), H_2, \phi_1 * \phi_2, Q) \equiv \\
& \quad \text{aconsume}'(P, (H, G, \Gamma), H_2, \phi_1, (\lambda H'_2, \text{lhs_snapshot} \bullet \\
& \quad \quad \text{aconsume}'(P, (H, G, \Gamma), H'_2, \phi_2, (\lambda H''_2, \text{rhs_snapshot} \bullet \\
& \quad \quad \quad Q(H''_2, \text{combine}(\text{lhs_snapshot}, \text{rhs_snapshot})))))) \\
& \text{aconsume}'(P, (H, G, \Gamma), H_2, e_0.q(e_1, \dots, e_n), Q) \equiv \\
& \quad \text{aeval}(P, (H, G, \Gamma), e_0, (\lambda v_0 \bullet \dots \text{aeval}(P, (H, G, \Gamma), e_n, (\lambda v_n \bullet \\
& \quad \quad \mathbf{let} \text{ matches} = \{v'_0.q[\text{snapshot}, H_0](v'_1, \dots, v'_n) \in H_2 \mid v_0 = v'_0 \wedge \dots \wedge v_n = v'_n\} \mathbf{in} \\
& \quad \quad \exists v'_0.q[\text{snapshot}, H_0](v'_1, \dots, v'_n) \in \text{matches} \bullet Q(H_2 - \{v'_0.q'[\text{snapshot}, H_0](v'_1, \dots, v'_n)\}, \text{snapshot})))))) \\
& \text{aconsume}'(P, (H, G, \Gamma), H_2, e_1 = e_2 ? \phi_1 : \phi_2, Q) \equiv \\
& \quad \text{aeval}(P, (H, G, \Gamma), e_1, (\lambda v_1 \bullet \text{aeval}(P, (H, G, \Gamma), e_2, (\lambda v_2 \bullet \\
& \quad \quad (v_1 = v_2 \Rightarrow \text{aconsume}'(P, (H, G, \Gamma), H_2, \phi_1, Q)) \wedge \\
& \quad \quad (v_1 \neq v_2 \Rightarrow \text{aconsume}'(P, (H, G, \Gamma), H_2, \phi_2, Q)))))) \\
& \text{aconsume}'(P, (H, G, \Gamma), H_2, \mathbf{untouched}(\phi), Q) \equiv \\
& \quad \text{aconsume}'(P, (H, G, \Gamma), H, \phi, (\lambda _, \text{snapshot}' \bullet \\
& \quad \quad \text{aconsume}'(P, (G, G, \Gamma), G, \phi, (\lambda _, \text{snapshot}'' \bullet \text{snapshot}' = \text{snapshot}'')))) \wedge \\
& \quad Q(H_2, \mathbf{unit})
\end{aligned}$$

$$\text{aconsume} : \wp(\mathcal{P}) \times \text{AbstractStates} \times \text{Assertions} \times (\text{AbstractStates} \times \text{Snapshots} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$$

$$\begin{aligned}
& \text{aconsume}(P, (H, G, \Gamma), \phi, Q) \equiv \\
& \quad \text{aconsume}'(P, (H, G, \Gamma), H, \phi, (\lambda H', \text{snapshot} \bullet Q((H', G, \Gamma), \text{snapshot})))
\end{aligned}$$

$$\text{aeval} : \wp(\mathcal{P}) \times \text{AbstractStates} \times \text{Expressions} \times (\mathcal{O} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$$

$$\begin{aligned}
& \text{aexec}((H, G, \Gamma), C \ x; , Q) \equiv \\
& \quad Q((H, G, \Gamma[x \mapsto \mathbf{null}])) \\
& \text{aexec}((H, G, \Gamma), x := e; , Q) \equiv \\
& \quad \text{aeval}(\mathcal{P}, (H, G, \Gamma), e, (\lambda v \bullet Q((H, G, \Gamma[x \mapsto v])))) \\
& \text{aexec}((H, G, \Gamma), e_1.f := e_2; , Q) \equiv \\
& \quad \text{aeval}(\mathcal{P}, (H, G, \Gamma), e_1, (\lambda v_1 \bullet \text{aeval}(\mathcal{P}, (H, G, \Gamma), e_2, (\lambda v_2 \bullet \\
& \quad \quad \mathbf{let} \text{ matches} = \{o.g \mapsto v \mid f = g \wedge o = v_1\} \mathbf{in} \\
& \quad \quad \exists o.g \mapsto v \in \text{ matches} \bullet Q((H - \{o.g \mapsto v\} + \{o.g \mapsto v_2\}, G, \Gamma)))))) \\
& \text{aexec}((H, G, \Gamma), e_0.m(e_1, \dots, e_n); , Q) \equiv \\
& \quad \text{aeval}(\mathcal{P}, (H, G, \Gamma), e_0, (\lambda v_1 \bullet \dots \bullet \text{aeval}(\mathcal{P}, (H, G, \Gamma), e_n, (\lambda v_n \bullet \\
& \quad \quad v_0 \neq \mathbf{null} \wedge \text{aconsume}(\mathcal{P}, (H, G, \Gamma), \text{pre}, (\lambda(H', -, -), - \bullet \\
& \quad \quad \text{aproduce}(\mathcal{P}, (H', H, [\mathbf{this} \mapsto v_0, \dots, x_n \mapsto v_n], \Sigma'), \text{mpost}(C.m), (\lambda(H'', -, -) \bullet \\
& \quad \quad \quad Q((H'', G, \Gamma)))))))))) \\
& \quad \text{where } \text{pre} \text{ is } \text{mpre}(C.m)[e_0/\mathbf{this}, e_1/x_1, \dots, e_n/x_n] \\
& \text{aexec}((H, G, \Gamma), x := \mathbf{new} \ C; , Q) \equiv \\
& \quad \forall o \in \mathcal{O} \setminus \{\mathbf{null}\} \bullet \\
& \quad \quad Q((H + \{o.f_1 \mapsto \mathbf{null}, \dots, o.f_n \mapsto \mathbf{null}\}, G, \Gamma[x \mapsto o])) \\
& \quad \text{where } \text{fields}(C) = C_1 \ f_1 \dots C_n \ f_n \\
& \text{aexec}((H, G, \Gamma), \mathbf{if}(e_1 = e_2) \{ \bar{s}_1 \} \mathbf{else} \ \bar{s}_2; , Q) \equiv \\
& \quad \text{aeval}(\mathcal{P}, (H, G, \Gamma), e_1, (\lambda v_1 \bullet \text{aeval}(\mathcal{P}, (H, G, \Gamma), e_2, (\lambda v_2 \bullet \\
& \quad \quad v_1 = v_2 \Rightarrow \text{aexec}((H, G, \Gamma), \bar{s}_1, Q) \wedge \\
& \quad \quad v_1 \neq v_2 \Rightarrow \text{aexec}((H, G, \Gamma), \bar{s}_2, Q)))))) \\
& \text{aexec}((H, G, \Gamma), \mathbf{assert} \ e_1 = e_2; , Q) \equiv \\
& \quad \text{aeval}(\mathcal{P}, (H, G, \Gamma), e_1, (\lambda v_1 \bullet \text{aeval}(\mathcal{P}, (H, G, \Gamma), e_2, (\lambda v_2 \bullet \\
& \quad \quad v_1 = v_2 \wedge Q((H, G, \Gamma)))))) \\
& \text{aexec}((H, G, \Gamma), \mathbf{open} \ e_0.q(e_1, \dots, e_n); , Q) \equiv \\
& \quad \text{aeval}(\mathcal{P}, (H, G, \Gamma), e_0, (\lambda v_0 \bullet \dots \bullet \text{aeval}(\mathcal{P}, (H, G, \Gamma), e_n, (\lambda v_n \bullet \\
& \quad \quad \text{aconsume}(\mathcal{P}, (H, G, \Gamma), e_0.q(e_1, \dots, e_n), (\lambda(H', -, -), \text{snapshot} \bullet \\
& \quad \quad \text{aproduce}(\mathcal{P}, (H, G, [\mathbf{this} \mapsto v_0, \dots, x_n \mapsto v_n]), \text{snapshot}, \text{mbody}(C.q), (\lambda(H'', -, -) \bullet \\
& \quad \quad \quad Q((H'', G, \Gamma)))))))))) \\
& \text{aexec}((H, G, \Gamma), \mathbf{close} \ e_0.q(e_1, \dots, e_n); , Q) \equiv \\
& \quad \text{aeval}(\mathcal{P}, (H, G, \Gamma), e_0, (\lambda v_0 \bullet \dots \bullet \text{aeval}(\mathcal{P}, (H, G, \Gamma), e_n, (\lambda v_n \bullet \\
& \quad \quad \text{aconsume}(\mathcal{P}, (H, G, [\mathbf{this} \mapsto v_0, \dots, x_n \mapsto v_n]), \text{mbody}(C.m), (\lambda(H', -, -), \text{snapshot} \bullet \\
& \quad \quad \quad \forall H_0 \bullet Q((H' + \{v_0.q[\text{snapshot}, H_0](v_1, \dots, v_n)\}, G, \Gamma)))))))))) \\
& \text{aexec}((H, G, \Gamma), \mathbf{use} \ e_0.p(e_1, \dots, e_n); , Q) \equiv \\
& \quad \text{aeval}(\mathcal{P}, (H, G, \Gamma), e_0.p(e_1, \dots, e_n), (\lambda v_{\text{call}} \bullet \text{aeval}(\mathcal{P}, (H, G, \Gamma), \text{body}, (\lambda v_{\text{body}} \bullet \\
& \quad \quad v_{\text{call}} = v_{\text{body}} \Rightarrow Q((H, G, \Gamma)))))) \\
& \quad \text{where } \text{body} \text{ is } \text{mbody}(C.p)[e_0/\mathbf{this}, e_1/x_1, \dots, e_n/x_n]
\end{aligned}$$

3.9 Abstract validity

Definition 9. A mutator

$$\mathbf{void} \ m(C_1 \ t_1, \dots, C_n \ t_n) \ \mathbf{requires} \ \phi_{\text{pre}}; \ \mathbf{ensures} \ \phi_{\text{post}}; \ \{ \bar{s} \}$$

is abstractly valid if all of the following hold:

- The precondition is well-defined and the postcondition is well-defined, provided the precondition held in the method pre-state.

$$\begin{aligned}
& \forall v_0, \dots, v_n \in \mathcal{O} \bullet v_0 \neq \mathbf{null} \Rightarrow \forall \text{snapshot}, \text{snapshot}' \in \text{Snapshots} \bullet \\
& \quad \text{aproduce}(\mathcal{P}, (\emptyset, \emptyset, [\mathbf{this} \mapsto v_0, x_1 \mapsto v_1, \dots, x_n \mapsto v_n]), \text{snapshot}, \phi_{\text{pre}}, (\lambda(H, -, \Gamma) \bullet \\
& \quad \quad \text{aproduce}(\mathcal{P}, (\emptyset, H, \Gamma), \text{snapshot}', \phi_{\text{post}}, (\lambda(-, -) \bullet \mathbf{true}))))
\end{aligned}$$

- The method body satisfies the method contract.

$$\begin{aligned} & \forall v_0, \dots, v_n \in \mathcal{O} \bullet v_0 \neq \mathbf{null} \Rightarrow \forall \text{snapshot}, \text{snapshot}' \in \text{Snapshots} \bullet \\ & \text{aproduce}(\mathcal{P}, (\emptyset, \emptyset, [\mathbf{this} \mapsto v_0, x_1 \mapsto v_1, \dots, x_n \mapsto v_n]), \text{snapshot}, \phi_{\text{pre}}, (\lambda(H, -, \Gamma) \bullet \\ & \quad \text{aexec}((H, H, \Gamma), \bar{s}, (\lambda(H', G', \Gamma') \bullet \\ & \quad \quad \text{aconsume}(\mathcal{P}, (H', G', \Gamma'), \phi_{\text{post}}, (\lambda_-, - \bullet \mathbf{true})))))) \end{aligned}$$

Definition 10. A predicate

$$\text{predicate } q(C_1 t_1, \dots, C_n t_n) \{ \text{return } \phi_{\text{body}}; \}$$

is abstractly valid if the body is a well-defined assertion:

$$\begin{aligned} & \forall v_0, \dots, v_n \in \mathcal{O} \bullet v_0 \neq \mathbf{null} \Rightarrow \forall \text{snapshot} \in \text{Snapshots} \bullet \\ & \text{aproduce}(\mathcal{P}, (\emptyset, \emptyset, [\mathbf{this} \mapsto v_0, x_1 \mapsto v_1, \dots, x_n \mapsto v_n]), \text{snapshot}, \phi_{\text{body}}, (\lambda_- \bullet \mathbf{true})) \end{aligned}$$

Definition 11. A pure method

$$\text{pure } C p(C_1 t_1, \dots, C_n t_n) \text{ requires } \phi_{\text{pre}}; \{ \text{return } e_{\text{body}}; \}$$

is abstractly valid if the precondition is a well-defined assertion and the body is well-defined, provided the precondition holds:

$$\begin{aligned} & \forall v_0, \dots, v_n \in \mathcal{O} \bullet v_0 \neq \mathbf{null} \Rightarrow \forall \text{snapshot} \in \text{Snapshots} \bullet \\ & \text{let } P = \{p' \in \mathcal{P} \bullet p' < p\} \text{ in} \\ & \text{aproduce}(P, (\emptyset, \emptyset, [\mathbf{this} \mapsto v_0, x_1 \mapsto v_1, \dots, x_n \mapsto v_n]), \text{snapshot}, \phi_{\text{pre}}, (\lambda(H, -, \Gamma) \bullet \\ & \quad \text{aeval}(P, (H, \emptyset, \Gamma), e_{\text{body}}, (\lambda_- \bullet \mathbf{true})))) \end{aligned}$$

Notice that the precondition or the body of a pure method may call only pure methods defined earlier in the program text, unless the call is in the body of an **opening** expression or after consuming the call's precondition, at least one field chunk is left in the symbolic heap.

Definition 12. The main routine \bar{s} is valid if the following holds

$$\text{aexec}((\emptyset, \emptyset, \emptyset), \bar{s}, \lambda_- \bullet \mathbf{true})$$

Definition 13. A program is valid if all methods and the main routine are valid.

3.10 Soundness

Definition 14. An assertion ϕ is well-defined with respect to an old state G , a store Γ , and a set of pure method names P , if for all snapshots snapshot , we have $\text{aproduce}(P, G, \Gamma, \emptyset, \text{snapshot}, \phi, (\lambda_- \bullet \mathbf{true}))$.

Lemma 9 (Soundness of aproduce , aconsume , aeval). Assume the program is valid.

Consider a set of pure methods P , and two consistent abstract heaps H and G .

- Consider a store Γ , an expression e , and an abstract expression evaluation postcondition Q . If $\text{aeval}(P, (H, G, \Gamma), e, Q)$, then there exists a value v such that $P, H, G, \Gamma \vdash e \Downarrow v$ and $Q(v)$.
- Consider a store Γ , a snapshot snapshot , an assertion ϕ , and an abstract production postcondition Q , and a subset H_0 of H . Assume that $H_0, G, \Gamma \vdash \phi \Downarrow \text{snapshot}, H$. If $\text{aproduce}'(P, G, \Gamma, H_0, \text{snapshot}, \phi, Q)$, then $Q(H)$.
- Consider a store Γ , a snapshot snapshot , an assertion ϕ , and an abstract production postcondition Q , and a subset H_0 of H . Assume that $H - H_0, G, \Gamma \vDash \phi \Downarrow \text{snapshot}$. If $\text{aproduce}(P, (H_0, G, \Gamma), \text{snapshot}, \phi, Q)$, then $Q(H)$.

- Consider a store Γ , a subset H_2 of H , an assertion ϕ , and an abstract consumption postcondition Q . If for all snapshots $snapshot_0$ we have $\mathbf{aproduce}'(P, G, \Gamma, H - H_2, snapshot_0, \phi, (\lambda_ \bullet \mathbf{true}))$, and we have $\mathbf{aconsume}'(P, (H, G, \Gamma), H_2, \phi, Q)$, then there exists a snapshot $snapshot$ and a heap $H_3 \subseteq H_2$ such that $H - H_2, G, \Gamma \vdash \phi \Downarrow snapshot, H - H_2 + H_3$ and $Q(snapshot, H_2 - H_3)$.
- Consider a store Γ , an assertion ϕ , and an abstract consumption postcondition Q . If for all snapshots $snapshot_0$ we have $\mathbf{aproduce}(P, G, \Gamma, \emptyset, snapshot_0, \phi, (\lambda_ \bullet \mathbf{true}))$, and we have $\mathbf{aconsume}(P, (H, G, \Gamma), \phi, Q)$, then there exists a snapshot $snapshot$ and a heap $H_0 \subseteq H$ such that $H - H_0, G, \Gamma \models \phi \Downarrow snapshot$ and $Q((H_0, G, \Gamma), snapshot)$.

Proof. By induction on the lexicographical order of the size of $H + G$, $\max(P)$, and the number of recursive calls of \mathbf{aeval} , $\mathbf{aproduce}'$, $\mathbf{aproduce}$, $\mathbf{aconsume}'$, and $\mathbf{aconsume}$. We highlight the most interesting cases.

- **Case \mathbf{aeval} .**

- **Case $e = e_0.p(e_1, \dots, e_n)$.** By the induction hypothesis for the recursive call $\mathbf{aconsume}$, and since the precondition of p is well-defined, there exists a snapshot $snapshot$ and a heap H_0 such that $H - H_0, G, \Gamma' \models \mathbf{mpre}(C.p) \Downarrow snapshot$. By the validity of p , and by the induction hypothesis for a call of $\mathbf{aproduce}$ and \mathbf{aeval} with a P' that does not include p , we have that the body of p evaluates to a value v . By the definition of the interpretation I of p , we prove our goal.
- **Case $\mathbf{using} e_0.p(e_1, \dots, e_n)$ in e .** By the induction hypothesis for the recursive calls of \mathbf{aeval} , we have that the call evaluates to a value v_{call} that is the abstract evaluation of the call, and the body evaluates to a value v_{body} that is the abstract evaluation of the body. It is easy to see from the definition of abstract evaluation that these values are equal.
- **Case $\mathbf{opening} e_0.q(e_1, \dots, e_n)$ in e .** By consistency of H , the nested heap of the matched predicate chunk satisfies the predicate body. Therefore, we have that the \mathbf{aeval} call succeeds, which gives us the goal.

- **Case $\mathbf{aconsume}'$.**

- **Case $e_0.q(e_1, \dots, e_n)$.** By consistency of H , we have that the chunk that is removed satisfies the assertion (which implies that the nested heap satisfies the body of the predicate).

□

Definition 15 (Valid Continuation). *Validity of a continuation κ with respect to an abstract heap H and a store Γ is defined as follows:*

$$\begin{aligned} \mathbf{valid_cont}(H, \Gamma, \mathbf{done}) &= \mathbf{true} \\ \mathbf{valid_cont}(H, \Gamma, s; \kappa) &= \mathbf{aexec}(H, \Gamma, s, (\lambda H, \Gamma \bullet \mathbf{valid_cont}(H, \Gamma, \kappa))) \\ \mathbf{valid_cont}(H, \Gamma, \mathbf{ret}(\Gamma', \kappa)) &= \mathbf{valid_cont}(H, \Gamma', \kappa) \end{aligned}$$

Definition 16 (Valid Abstract Configuration). *An abstract configuration (H, Γ, κ) is valid if H is consistent and κ is valid with respect to H and Γ .*

Definition 17. *An abstract configuration (H, Γ, κ) abstracts a concrete configuration (H', Γ', κ') iff $H' = \mathbf{flat}(H)$ and $\Gamma' = \Gamma$ and $\kappa' = \kappa$.*

Definition 18 (Valid Concrete Configuration). *A concrete configuration is valid if there is a valid abstract configuration that abstracts it.*

Definition 19. *The initial concrete configuration consists of the empty heap, the empty store, and the main routine.*

$$\mathbf{program} = \overline{\mathbf{class} \bar{s}} \Rightarrow \gamma_0 = \langle \emptyset, \emptyset, \bar{s} \rangle$$

Lemma 10 (Frame Rule). *Suppose $H + H_F$ is consistent. Then, if $\text{aexec}(H, \Gamma, \bar{s}, Q)$, then $\text{aexec}(H + H_F, \Gamma, \bar{s}, (\lambda H', \Gamma' \bullet H_F \subseteq H' \wedge Q(H' - H_F, \Gamma')))$.*

Proof. By induction on \bar{s} . □

Lemma 11 (Preservation). *Execution steps preserve configuration validity.*

$$\text{valid_conf}(\gamma) \Rightarrow \gamma \rightsquigarrow \gamma' \Rightarrow \text{valid_conf}(\gamma')$$

Proof. By case analysis on the step rule. Relies heavily on the soundness of aeval , aconsume , and aproduce (Lemma 9). We highlight the most interesting cases.

- **Case STEP-CALL.** Uses the Frame Rule, as well as the soundness of production and consumption.
- **Case STEP-RETURN.** Follows immediately from validity of the continuation.
- **Case STEP-OPEN.** By the consistency of the heap, the nested heap of the chunk that is matched satisfies the body of the predicate.
- **Case STEP-CLOSE.** Since consumption succeeds, there is a subset H_ϕ of the heap that satisfies the body of the predicate. The heap obtained by replacing H_ϕ with a predicate chunk whose nested heap is H_ϕ is consistent, satisfies the postcondition, and has the same flattening as the original one.
- **Case STEP-USE.** Similar to the case for **using** expressions in the proof of Lemma 9.

□

Lemma 12. *The initial configuration is valid.*

$$\text{valid_conf}(\gamma_0)$$

Proof. This follows directly from validity of the main routine. □

Lemma 13. *A valid configuration is not stuck.*

$$\text{valid_conf}((H, \Gamma, \kappa)) \Rightarrow (\kappa = \mathbf{done} \vee (\exists \gamma' \bullet \gamma \rightsquigarrow \gamma'))$$

Proof. By case analysis on the continuation.

- **Case κ is a statement continuation.** By case analysis on the statement.
- **Case κ is a return continuation.**

□

Theorem 1 (Soundness). *If the program is valid, then concrete executions do not get stuck.*

$$\text{avalid_program} \Rightarrow (\forall H, \Gamma, \kappa \bullet \gamma_0 \rightsquigarrow^* (H, \Gamma, \kappa) \Rightarrow \kappa = \mathbf{done} \vee (\exists \gamma' \bullet (H, \Gamma, \kappa) \rightsquigarrow \gamma'))$$

Proof. Follows directly from Lemmas 12, 11, and 13. □

4 Symbolic execution

In this section, we define validity of programs and we prove that a valid program is abstractly valid. Since abstract validity is sound, valid programs do not get stuck.

4.1 Logic

We target a multi-sorted, first-order logic with equality. A term t is either a variable or a function application. A formula ψ is either *true*, *false*, an equality between terms, a conjunction, a disjunction, a negation or a quantification.

$$\begin{aligned} t & ::= x \mid f(\bar{t}) \\ \psi & ::= \text{true} \mid \text{false} \mid t = t \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg\psi \mid \forall \bar{x} \bullet \psi \end{aligned}$$

Functions with arity zero are called constants.

4.2 Signature

Each term in the logic has a corresponding sort. The sorts are the following: *ref*, the sort of references and *snapshot*, the sort of heap snapshots. In the remainder of this paper, we omit sorts whenever they are clear from the context.

The signature of the logic contains a number of built-in functions and a number of program-specific functions. The *built-in functions* are the following:

$$\begin{aligned} \text{unit} & : \text{snapshot} \\ \text{combine} & : \text{snapshot} \times \text{snapshot} \rightarrow \text{snapshot} \\ \text{first} & : \text{snapshot} \rightarrow \text{snapshot} \\ \text{second} & : \text{snapshot} \rightarrow \text{snapshot} \\ \text{null} & : \text{ref} \\ \text{fromRef} & : \text{ref} \rightarrow \text{snapshot} \\ \text{toRef} & : \text{snapshot} \rightarrow \text{ref} \end{aligned}$$

In addition to the built-in functions, the signature contains a number of program-specific functions. More specifically, for each pure method p defined in a class C with parameters $C_1 x_1, \dots, C_1 x_n$, the logic includes a function symbol $C.p$ with sort $\text{snapshot} \times \text{ref} \times \text{ref}_1 \times \dots \times \text{ref}_n \rightarrow \text{ref}$. We encode invocations of pure methods as applications of the latter functions. The first argument of the function encodes the state of the heap covered by p 's precondition and is used for framing.

4.3 Interpretation

The built-in function symbols are interpreted as follows: *unit* is interpreted as **unit**, *combine* is interpreted as **combine**, etc.

Pure method symbols are interpreted as in Section 3.6.

4.4 Theory

We define the theory Σ_{prelude} as follows:

$$\begin{aligned} & (\forall l, r \bullet \text{first}(\text{combine}(l, r)) = l) \wedge \\ & (\forall l, r \bullet \text{second}(\text{combine}(l, r)) = r) \wedge \\ & (\forall o \bullet \text{toRef}(\text{fromRef}(o)) = o) \end{aligned}$$

This theory encodes the property that *combine* and *fromRef* are injective. In the remainder of this paper, we omit Σ_{prelude} for brevity. For example, we write $\Sigma \vdash \psi$ to denote ψ is provable from the theory Σ instead of $\Sigma_{\text{prelude}} \cup \Sigma \vdash \psi$.

4.5 Symbolic state

Definition 20. A symbolic heap H is a list of heap chunks. A heap chunk is one of the following:

- a field chunk, $t_1.f \mapsto t_2$.

- a predicate chunk $t_0.q[t_s](t_1, \dots, t_n)$.

Definition 21. A symbolic store Γ is a partial function from variable names to terms.

Definition 22. A path condition Σ is a set of formulas.

Definition 23. A symbolic state σ is a quadruple consisting of:

- a symbolic heap, H .
- a symbolic heap, G . G represents the old value of the heap.
- a symbolic store, Γ .
- a path condition, Σ .

4.6 Symbolic production, consumption, evaluation

The equations below define the mutually recursive functions `produce'`, `produce`, `consume'`, `consume`, and `eval`. If evaluation of a function “call” does not terminate, we define its value to be **false**. That is: let T be the transformation defined by the equations below. It maps an interpretation for the abovementioned functions to a new interpretation. Since the functions return booleans, we can represent an interpretation as the set of the calls that return **true**. We then define the interpretation I of the functions as

$$I = \bigcup_n T^n(\emptyset)$$

Note: it is easy to see that T is monotonic: $I \subseteq T(I)$. Indeed, all calls appear in positive positions, including calls of the postcondition. Therefore, we have that I is the least fixpoint and it satisfies the equations:

$$I = T(I)$$

$$\begin{aligned} \text{produce}' : \wp(\mathcal{P}) \times \text{SymbolicHeaps} \times \text{SymbolicStores} \times \text{PathConditions} \times \text{SymbolicHeaps} \times \\ \text{Terms} \times \text{Assertions} \times (\text{SymbolicHeaps} \times \text{PathConditions} \rightarrow \mathbb{B}) \rightarrow \mathbb{B} \end{aligned}$$

$$\begin{aligned}
&\text{produce}'(P, G, \Gamma, \Sigma, H, \text{snapshot}, \text{true}, Q) \equiv \\
&\quad Q(H, \Sigma) \\
&\text{produce}'(P, G, \Gamma, \Sigma, H, \text{snapshot}, \text{false}, Q) \equiv \\
&\quad \text{true} \\
&\text{produce}'(P, G, \Gamma, \Sigma, H, \text{snapshot}, \text{acc}(e.f), Q) \equiv \\
&\quad \text{eval}(P, (H, G, \Gamma, \Sigma), e, (\lambda t \bullet \\
&\quad \quad \text{let } \{o_1.f_1 \mapsto v_1, \dots, o_n.f_n \mapsto v_n\} = \{o.g \mapsto v \in H \mid f = g\} \text{ in} \\
&\quad \quad (\Sigma \vdash_{Z3} t \neq \text{null}) \wedge \\
&\quad \quad ((\Sigma \not\vdash_{Z3} o_1 = t \vee \dots \vee o_n = t) \Rightarrow Q(H + \{t.f \mapsto \text{toRef}(\text{snapshot})\}, \Sigma \cup \{o_1 \neq t \wedge \dots \wedge o_n \neq t\}))) \\
&\text{produce}'(P, G, \Gamma, \Sigma, H, \text{snapshot}, e_1 = e_2, Q) \equiv \\
&\quad \text{eval}(P, (H, G, \Gamma, \Sigma), e_1, (\lambda t_1 \bullet \text{eval}(P, (H, G, \Gamma, \Sigma), e_2, (\lambda t_2 \bullet \\
&\quad \quad \Sigma \not\vdash_{Z3} t_1 \neq t_2 \Rightarrow Q(H, \Sigma \cup \{t_1 = t_2\})))))) \\
&\text{produce}'(P, G, \Gamma, \Sigma, H, \text{snapshot}, \phi_1 * \phi_2, Q) \equiv \\
&\quad \text{produce}'(P, (H, G, \Gamma, \Sigma), \text{first}(\text{snapshot}), \phi_1, (\lambda H', \Sigma' \bullet \text{produce}'(P, G, \Gamma, \Sigma', H', \text{second}(\text{snapshot}), \phi_2, Q))) \\
&\text{produce}'(P, G, \Gamma, \Sigma, H, \text{snapshot}, e_0.q(e_1, \dots, e_n), Q) \equiv \\
&\quad \text{eval}(P, (H, G, \Gamma, \Sigma), e_0, (\lambda t_0 \bullet \dots \text{eval}(P, (H, G, \Gamma, \Sigma), e_n, (\lambda t_n \bullet \\
&\quad \quad (\Sigma \vdash_{Z3} t_0 \neq \text{null}) \wedge Q(H + \{t_0.q[\text{snapshot}](t_1, \dots, t_n)\}, \Sigma)))))) \\
&\text{produce}'(P, G, \Gamma, \Sigma, H, \text{snapshot}, e_1 = e_2 ? \phi_1 : \phi_2, Q) \equiv \\
&\quad \text{eval}(P, (H, G, \Gamma, \Sigma), e_1, (\lambda t_1 \bullet \text{eval}(P, (H, G, \Gamma, \Sigma), e_2, (\lambda t_2 \bullet \\
&\quad \quad \Sigma \not\vdash_{Z3} t_1 \neq t_2 \Rightarrow \text{produce}'(P, G, \Gamma, \Sigma \cup \{t_1 = t_2\}, H, \text{snapshot}, \phi_1, Q) \wedge \\
&\quad \quad \Sigma \vdash_{Z3} t_1 = t_2 \Rightarrow \text{produce}'(P, G, \Gamma, \Sigma \cup \{t_1 \neq t_2\}, H, \text{snapshot}, \phi_2, Q)))))) \\
&\text{produce}'(P, (H, G, \Gamma, \Sigma), \text{snapshot}, \text{untouched}(\phi), Q) \equiv \\
&\quad \text{consume}(P, (H, G, \Gamma, \Sigma), \phi, (\lambda(-, -, \Sigma') \bullet \text{snapshot}' \bullet \\
&\quad \quad \text{consume}(P, (G, G, \Gamma, \Sigma'), \phi, (\lambda(-, -, \Sigma'') \bullet \text{snapshot}'' \bullet \\
&\quad \quad \quad \Sigma'' \vdash_{Z3} \text{snapshot}' = \text{snapshot}'' \wedge Q((H, G, \Gamma, \Sigma''))))))))
\end{aligned}$$

$$\text{produce} : \wp(\mathcal{P}) \times \text{SymbolicStates} \times \text{Terms} \times \text{Assertions} \times (\text{SymbolicStates} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$$

$$\begin{aligned}
&\text{produce}(P, (H, G, \Gamma, \Sigma), \text{snapshot}, \phi, Q) \equiv \\
&\quad \text{produce}'(P, G, \Gamma, \Sigma, \emptyset, \text{snapshot}, \phi, (\lambda H', \Sigma' \bullet Q((H + H', G, \Gamma, \Sigma'))))
\end{aligned}$$

$$\begin{aligned}
&\text{consume}' : \wp(\mathcal{P}) \times \text{SymbolicStates} \times \text{SymbolicHeaps} \times \text{Assertions} \times \\
&\quad (\text{SymbolicHeaps} \times \text{PathConditions} \times \text{Terms} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}
\end{aligned}$$

$$\begin{aligned}
& \text{consume}'(P, (H, G, \Gamma, \Sigma), H_2, \mathbf{true}, Q) \equiv \\
& \quad Q(H_2, \Sigma, \mathit{unit}) \\
& \text{consume}'(P, (H, G, \Gamma, \Sigma), H_2, \mathbf{false}, Q) \equiv \\
& \quad \mathit{false} \\
& \text{consume}'(P, (H, G, \Gamma, \Sigma), H_2, \mathbf{acc}(e.f), Q) \equiv \\
& \quad \text{eval}(P, (H, G, \Gamma, \Sigma), e, (\lambda t \bullet \\
& \quad \quad \mathbf{let} \text{ matches} = \{o.g \mapsto v \in H_2 \mid f = g \wedge \Sigma \vdash t = o\} \mathbf{in} \\
& \quad \quad \exists o.g \mapsto v \in \text{matches} \bullet Q(H_2 - \{o.g \mapsto v\}, \Sigma, \text{fromRef}(v)))))) \\
& \text{consume}'(P, (H, G, \Gamma, \Sigma), H_2, e_1 = e_2, Q) \equiv \\
& \quad \text{eval}(P, (H, G, \Gamma, \Sigma), e_1, (\lambda t_1 \bullet \text{eval}((H, G, \Gamma, \Sigma), e_2, (\lambda t_2 \bullet \\
& \quad \quad \Sigma \vdash_{Z3} t_1 = t_2 \wedge Q(H_2, \Sigma, \mathit{unit})))))) \\
& \text{consume}'(P, (H, G, \Gamma, \Sigma), H_2, \phi_1 * \phi_2, Q) \equiv \\
& \quad \text{consume}'(P, (H, G, \Gamma, \Sigma), H_2, \phi_1, (\lambda H'_2, \Sigma', \text{lhs_snapshot} \bullet \\
& \quad \quad \text{consume}'(P, (H, G, \Gamma, \Sigma'), H'_2, \phi_2, (\lambda H''_2, \Sigma'', \text{rhs_snapshot} \bullet \\
& \quad \quad \quad Q(H''_2, \Sigma'', \text{combine}(\text{lhs_snapshot}, \text{rhs_snapshot})))))) \\
& \text{consume}'(P, (H, G, \Gamma, \Sigma), H_2, e_0.q(e_1, \dots, e_n), Q) \equiv \\
& \quad \text{eval}(P, (H, G, \Gamma, \Sigma), e_0, (\lambda t_0 \bullet \dots \text{eval}(P, (H, G, \Gamma, \Sigma), e_n, (\lambda t_n \bullet \\
& \quad \quad \mathbf{let} \text{ matches} = \{t'_0.q[t'_s](t'_1, \dots, t'_n) \in H_2 \mid \Sigma \vdash_{Z3} t_0 = t'_0 \wedge \dots \wedge t_n = t'_n\} \mathbf{in} \\
& \quad \quad \exists t'_0.q[t'_s](t'_1, \dots, t'_n) \in \text{matches} \bullet Q(H_2 - \{t'_0.q[t'_s](t'_1, \dots, t'_n)\}, \Sigma, t'_s)))))) \\
& \text{consume}'(P, (H, G, \Gamma, \Sigma), H_2, e_1 = e_2 ? \phi_1 : \phi_2, Q) \equiv \\
& \quad \text{eval}(P, (H, G, \Gamma, \Sigma), e_1, (\lambda t_1 \bullet \text{eval}(P, (H, G, \Gamma, \Sigma), e_2, (\lambda t_2 \bullet \\
& \quad \quad (\Sigma \not\vdash_{Z3} t_1 \neq t_2 \Rightarrow \text{consume}'(P, (H, G, \Gamma, \Sigma \cup \{t_1 = t_2\}), H_2, \phi_1, Q)) \wedge \\
& \quad \quad (\Sigma \not\vdash_{Z3} t_1 = t_2 \Rightarrow \text{consume}'(P, (H, G, \Gamma, \Sigma \cup \{t_1 \neq t_2\}), H_2, \phi_2, Q)))))) \\
& \text{consume}'(P, (H, G, \Gamma, \Sigma), H_2, \mathbf{untouched}(\phi), Q) \equiv \\
& \quad \text{consume}'(P, (H, G, \Gamma, \Sigma), H, \phi, (\lambda _ , _ , \text{snapshot}' \bullet \\
& \quad \quad \text{consume}'(P, (G, G, \Gamma, \Sigma), G, \phi, (\lambda _ , _ , \text{snapshot}'' \bullet \Sigma \vdash_{Z3} \text{snapshot}' = \text{snapshot}'')))) \wedge \\
& \quad Q(H_2, \Sigma, \mathit{unit})
\end{aligned}$$

$$\text{consume} : \wp(\mathcal{P}) \times \text{SymbolicStates} \times \text{Assertions} \times (\text{SymbolicStates} \times \text{Terms} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$$

$$\begin{aligned}
& \text{consume}(P, (H, G, \Gamma, \Sigma), \phi, Q) \equiv \\
& \quad \text{consume}'(P, (H, G, \Gamma, \Sigma), H, \phi, (\lambda H', \Sigma', \text{snapshot} \bullet Q((H', G, \Gamma, \Sigma'), \text{snapshot})))
\end{aligned}$$

$$\text{eval} : \wp(\mathcal{P}) \times \text{SymbolicStates} \times \text{Expressions} \times (\text{Terms} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$$

$$\begin{aligned}
\text{eval}(P, \sigma, \text{null}, Q) &\equiv Q(\text{null}) \\
\text{eval}(P, \sigma, x, Q) &\equiv Q(s_\sigma(x)) \\
\text{eval}(P, \sigma, e.f, Q) &\equiv \text{eval}(P, \sigma, e, (\lambda t \bullet \\
&\quad \text{let matches} = \{o.g \mapsto v \in H_\sigma \mid f = g \wedge \Sigma_\sigma \vdash_{Z3} t = o\} \text{ in} \\
&\quad \exists o.g \mapsto v \in \text{matches} \bullet Q(v))) \\
\text{eval}(P, \sigma, e_0.p(e_1, \dots, e_n), Q) &\equiv \\
&\quad \text{eval}(P, \sigma, e_0, (\lambda t_0 \bullet \dots \text{eval}(P, \sigma, e_n, (\lambda t_n \bullet \\
&\quad (\Sigma_\sigma \vdash_{Z3} t_0 \neq \text{null}) \wedge p \in P \wedge \text{consume}(P, (H_\sigma, G_\sigma, [\text{this} \mapsto t_0, \dots, x_n \mapsto t_n], \Sigma_\sigma), \text{mpre}(C.p), (\lambda \sigma', \text{snapshot} \bullet \\
&\quad Q(C.p(\text{snapshot}, t_0, \dots, t_n)))))))) \\
\text{eval}(P, \sigma, \text{old}(e), Q) &\equiv \\
&\quad \text{eval}(P, \text{old}(\sigma), e, Q) \\
\text{eval}(P, \sigma, \text{opening } e_0.q(e_1, \dots, e_n) \text{ in } e, Q) &\equiv \\
&\quad \text{eval}(P, \sigma, e_0, (\lambda t_0 \bullet \dots \text{eval}(P, \sigma, e_n, (\lambda t_n \bullet \\
&\quad \text{consume}(P, \sigma, e_0.q(e_1, \dots, e_n), (\lambda(H', G, \Gamma, \Sigma'), \text{snapshot} \bullet \\
&\quad \text{produce}(\wp(P), (H', G, [\text{this} \mapsto t_0, \dots, x_n \mapsto t_n], \Sigma'), \text{snapshot}, \text{mbody}(C.q), (\lambda \sigma'' \bullet \\
&\quad \text{eval}(\wp(P), (H_{\sigma''}, G, \Gamma, \Sigma_{\sigma''}), e, Q)))))))) \\
\text{eval}(P, \sigma, \text{using } e_0.p(e_1, \dots, e_n) \text{ in } e, Q) &\equiv \\
&\quad \text{eval}(P, \sigma, e_0.p(e_1, \dots, e_n), (\lambda t_{\text{call}} \bullet \text{eval}(P, \sigma, \text{body}, (\lambda t_{\text{body}} \bullet \\
&\quad \Sigma \not\vdash_{Z3} t_{\text{call}} \neq t_{\text{body}} \Rightarrow \text{eval}(P, \sigma \cup \{t_{\text{call}} = t_{\text{body}}\}, e, Q)))))) \\
&\quad \text{where } \text{body} \text{ is } \text{mbody}(C.p)[e_0/\text{this}, e_1/x_1, \dots, e_n/x_n] \\
\text{eval}(P, \sigma, e_1 = e_2 ? e_3 : e_4, Q) &\equiv \\
&\quad \text{eval}(P, \sigma, e_1, (\lambda t_1 \bullet \text{eval}(P, \sigma, e_2, (\lambda t_2 \bullet \\
&\quad (\Sigma_\sigma \not\vdash_{Z3} t_1 \neq t_2 \wedge \Sigma_\sigma \not\vdash_{Z3} t_1 = t_2 \Rightarrow \\
&\quad \text{eval}(\sigma \cup \{t_1 = t_2\}, e_3, (\lambda t_3 \bullet \text{eval}(P, \sigma \cup \{t_1 \neq t_2\}, e_4, (\lambda t_4 \bullet Q(\text{ite}(t_1 = t_2, t_3, t_4))))))))) \wedge \\
&\quad (\Sigma_\sigma \vdash_{Z3} t_1 = t_2 \Rightarrow \\
&\quad \text{eval}(P, \sigma, e_3, (\lambda t_3 \bullet Q(t_3)))) \wedge \\
&\quad (\Sigma_\sigma \vdash_{Z3} t_1 \neq t_2 \Rightarrow \\
&\quad \text{eval}(P, \sigma, e_4, (\lambda t_4 \bullet Q(t_4)))) \\
&\quad))))
\end{aligned}$$

4.7 Symbolic statement execution

$$\text{exec} : \text{SymbolicStates} \times \text{Statements} \times (\text{SymbolicStates} \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$$

$\text{exec}((H, G, \Gamma, \Sigma), C \ x; , Q) \equiv$
 $Q((H, G, \Gamma[x \mapsto \text{null}], \Sigma))$
 $\text{exec}((H, G, \Gamma, \Sigma), x := e; , Q) \equiv$
 $\text{eval}(\mathcal{P}, (H, G, \Gamma, \Sigma), e, (\lambda t \bullet Q((H, G, \Gamma[x \mapsto t], \Sigma))))$
 $\text{exec}((H, G, \Gamma, \Sigma), e_1.f := e_2; , Q) \equiv$
 $\text{eval}(\mathcal{P}, (H, G, \Gamma, \Sigma), e_1, (\lambda t_1 \bullet \text{eval}(\mathcal{P}, (H, G, \Gamma, \Sigma), e_2, (\lambda t_2 \bullet$
 $\text{let } \text{matches} = \{o.g \mapsto v \mid f = g \wedge \Sigma \vdash_{Z3} o = t_1\} \text{ in}$
 $\exists o.g \mapsto v \in \text{matches} \bullet Q((H - \{o.g \mapsto v\} + \{o.g \mapsto t_2\}, G, \Gamma, \Sigma))))))$
 $\text{exec}((H, G, \Gamma, \Sigma), e_0.m(e_1, \dots, e_n); , Q) \equiv$
 $\text{eval}(\mathcal{P}, (H, G, \Gamma, \Sigma), e_0, (\lambda t_1 \bullet \dots \text{eval}(\mathcal{P}, (H, G, \Gamma, \Sigma), e_n, (\lambda t_n \bullet$
 $\Sigma \vdash_{Z3} t_0 \neq \text{null} \wedge \text{consume}(\mathcal{P}, (H, G, \Gamma, \Sigma), \text{pre}, (\lambda(H', -, -, \Sigma'), - \bullet$
 $\text{produce}(\mathcal{P}, (H', H, [\text{this} \mapsto t_0, \dots, x_n \mapsto t_n], \Sigma'), \text{mpost}(C.m), (\lambda(H'', -, -, \Sigma'') \bullet$
 $Q((H'', G, \Gamma, \Sigma''))))))))$
 where pre is $\text{mpre}(C.m)[e_0/\text{this}, e_1/x_1, \dots, e_n/x_n]$
 $\text{exec}((H, G, \Gamma, \Sigma), x := \text{new } C; , Q) \equiv$
let $o = \text{freshvar}$ **in**
 $Q((H + \{o.f_1 \mapsto \text{null}, \dots, o.f_n \mapsto \text{null}\}, G, \Gamma[x \mapsto o], \Sigma \cup \{o \neq \text{null}\}))$
 where $\text{fields}(C) = C_1 \ f_1 \dots C_n \ f_n$
 $\text{exec}((H, G, \Gamma, \Sigma), \text{if}(e_1 = e_2) \{ \bar{s}_1 \} \text{ else } \bar{s}_2; , Q) \equiv$
 $\text{eval}(\mathcal{P}, (H, G, \Gamma, \Sigma), e_1, (\lambda t_1 \bullet \text{eval}(\mathcal{P}, (H, G, \Gamma, \Sigma), e_2, (\lambda t_2 \bullet$
 $\Sigma \not\vdash_{Z3} t_1 \neq t_2 \Rightarrow \text{exec}((H, G, \Gamma, \Sigma \cup \{t_1 = t_2\}), \bar{s}_1, Q) \wedge$
 $\Sigma \not\vdash_{Z3} t_1 = t_2 \Rightarrow \text{exec}((H, G, \Gamma, \Sigma \cup \{t_1 \neq t_2\}), \bar{s}_2, Q))))$
 $\text{exec}((H, G, \Gamma, \Sigma), \text{assert } e_1 = e_2; , Q) \equiv$
 $\text{eval}(\mathcal{P}, (H, G, \Gamma, \Sigma), e_1, (\lambda t_1 \bullet \text{eval}(\mathcal{P}, (H, G, \Gamma, \Sigma), e_2, (\lambda t_2 \bullet$
 $\Sigma \vdash_{Z3} t_1 = t_2 \wedge Q((H, G, \Gamma, \Sigma))))))$
 $\text{exec}((H, G, \Gamma, \Sigma), \text{open } e_0.q(e_1, \dots, e_n); , Q) \equiv$
 $\text{eval}(\mathcal{P}, (H, G, \Gamma, \Sigma), e_0, (\lambda t_0 \bullet \dots \text{eval}(\mathcal{P}, (H, G, \Gamma, \Sigma), e_n, (\lambda t_n \bullet$
 $\text{consume}(\mathcal{P}, (H, G, \Gamma, \Sigma), e_0.q(e_1, \dots, e_n), (\lambda(H', -, -, \Sigma'), \text{snapshot} \bullet$
 $\text{produce}(\mathcal{P}, (H, G, [\text{this} \mapsto t_0, \dots, x_n \mapsto t_n], \Sigma), \text{snapshot}, \text{mbody}(C.q), (\lambda(H'', -, -, \Sigma'') \bullet$
 $Q((H'', G, \Gamma, \Sigma''))))))))$
 $\text{exec}((H, G, \Gamma, \Sigma), \text{close } e_0.q(e_1, \dots, e_n); , Q) \equiv$
 $\text{eval}(\mathcal{P}, (H, G, \Gamma, \Sigma), e_0, (\lambda t_0 \bullet \dots \text{eval}(\mathcal{P}, (H, G, \Gamma, \Sigma), e_n, (\lambda t_n \bullet$
 $\text{consume}(\mathcal{P}, (H, G, [\text{this} \mapsto t_0, \dots, x_n \mapsto t_n], \Sigma), \text{mbody}(C.m), (\lambda(H', -, -, \Sigma'), \text{snapshot} \bullet$
 $Q((H' + \{t_0.q[\text{snapshot}](t_1, \dots, t_n)\}, G, \Gamma, \Sigma''))))))))$
 $\text{exec}((H, G, \Gamma, \Sigma), \text{use } e_0.p(e_1, \dots, e_n); , Q) \equiv$
 $\text{eval}(\mathcal{P}, (H, G, \Gamma, \Sigma), e_0.p(e_1, \dots, e_n), (\lambda t_{\text{call}} \bullet \text{eval}(\mathcal{P}, (H, G, \Gamma, \Sigma), \text{body}, (\lambda t_{\text{body}} \bullet$
 $Q((H, G, \Gamma, \Sigma \cup \{t_{\text{call}} = t_{\text{body}}\}))))))$
 where body is $\text{mbody}(C.p)[e_0/\text{this}, e_1/x_1, \dots, e_n/x_n]$

4.8 Symbolic validity

Definition 24. A mutator

$$\text{void } m(C_1 \ t_1, \dots, C_n \ t_n) \text{ requires } \phi_{\text{pre}}; \text{ ensures } \phi_{\text{post}}; \{ \bar{s} \}$$

is valid if all of the following hold:

- The precondition is well-defined and the postcondition is well-defined, provided the precondition held in the method pre-state.

let $t_0, \dots, t_n = \text{freshvar}, \dots, \text{freshvar}$ **in**
 $\text{produce}(\mathcal{P}, (\emptyset, \emptyset, [\text{this} \mapsto t_0, x_1 \mapsto t_1, \dots, x_n \mapsto t_n], \{t_0 \neq \text{null}\}), \text{freshvar}, \phi_{\text{pre}}, (\lambda(H, -, \Gamma, \Sigma) \bullet$
 $\text{produce}(\mathcal{P}, (\emptyset, H, \Gamma, \Sigma), \text{freshvar}, \phi_{\text{post}}, (\lambda-, - \bullet \text{true}))))$

- The method body satisfies the method contract.

```

let  $t_0, \dots, t_n = \text{freshvar}, \dots, \text{freshvar}$  in
produce( $\mathcal{P}, (\emptyset, \emptyset, [\mathbf{this} \mapsto t_0, x_1 \mapsto t_1, \dots, x_n \mapsto t_n], \{t_0 \neq \text{null}\})$ ,  $\text{freshvar}, \phi_{pre}, (\lambda(H, -, \Gamma, \Sigma) \bullet$ 
   $\text{exec}((H, H, \Gamma, \Sigma), \bar{s}, (\lambda(H', G', \Gamma', \Sigma') \bullet$ 
     $\text{consume}(\mathcal{P}, (H', G', \Gamma', \Sigma'), \phi_{post}, (\lambda-, - \bullet \text{true}))))))$ 

```

Definition 25. A predicate

```

predicate  $q(C_1 t_1, \dots, C_n t_n) \{ \text{return } \phi_{body}; \}$ 

```

is valid if the body is a well-defined assertion:

```

let  $t_0, \dots, t_n = \text{freshvar}, \dots, \text{freshvar}$  in
produce( $\mathcal{P}, (\emptyset, \emptyset, [\mathbf{this} \mapsto t_0, x_1 \mapsto t_1, \dots, x_n \mapsto t_n], \{t_0 \neq \text{null}\})$ ,  $\text{freshvar}, \phi_{body}, (\lambda- \bullet \text{true}))$ 

```

Definition 26. A pure method

```

pure  $C p(C_1 t_1, \dots, C_n t_n) \text{requires } \phi_{pre}; \{ \text{return } e_{body}; \}$ 

```

is valid if the precondition is a well-defined assertion and the body is well-defined, provided the precondition holds:

```

let  $t_0, \dots, t_n = \text{freshvar}, \dots, \text{freshvar}$  in
let  $P = \{p' \in \mathcal{P} \bullet p' < p\}$  in
produce( $P, (\emptyset, \emptyset, [\mathbf{this} \mapsto t_0, x_1 \mapsto t_1, \dots, x_n \mapsto t_n], \{t_0 \neq \text{null}\})$ ,  $\text{freshvar}, \phi_{pre}, (\lambda(H, -, \Gamma, \Sigma) \bullet$ 
   $\text{eval}(P, (H, \emptyset, \Gamma, \Sigma), e_{body}, (\lambda- \bullet \text{true})))$ 

```

Notice that the precondition or the body of a pure method may call only pure methods defined earlier in the program text, unless the call is in the body of an **opening** expression or after consuming the call's precondition, at least one field chunk is left in the symbolic heap.

Definition 27. The main routine \bar{s} is valid if the following holds

```

exec(( $\emptyset, \emptyset, \emptyset, \emptyset$ ),  $\bar{s}, \lambda- \bullet \text{true}$ )

```

Definition 28. A program is valid if all methods and the main routine are valid.

4.9 Soundness

We now prove that symbolic execution simulates abstract execution, in the following sense. Consider an abstract state σ_a and a symbolic state σ that represents it. Then, if abstract execution fails, symbolic execution fails, and if abstract execution reaches an abstract state σ'_a , then symbolic execution reaches a symbolic state σ' that represents it.

Definition 29. A symbolic heap H matches an abstract heap H_a under an interpretation I , denoted $I \models H \Downarrow H_a$, iff the heap obtained from H by replacing each term by its value under I equals the heap obtained from H_a by removing the nested heaps.

Definition 30. A symbolic state represents an abstract state if there is an interpretation I of the logical symbols such that the value of the symbolic state under I matches the abstract state.

Lemma 14 (Soundness of symbolic evaluation, production, and consumption). *Since the three functions are defined by mutual recursion, we prove soundness of all three in one lemma.*

- Consider a set of pure methods P , a symbolic heap H , a symbolic heap G , a symbolic store Γ , a path condition Σ , an expression e , and a symbolic evaluation postcondition Q . Further consider an interpretation I of the symbols such that $I \models \Sigma$ and an abstract heap H_a , such that $I \models H \Downarrow H_a$, and an abstract heap G_a , such that $I \models G \Downarrow G_a$, and a store Γ_a such that $I(\Gamma) = \Gamma_a$. If $\text{eval}(P, (H, G, \Gamma, \Sigma), e, Q)$, then $\text{aeval}(P, (H_a, G_a, \Gamma_a), e, (\lambda v \bullet \exists t \bullet Q(t) \wedge I(t) = v))$.

- Consider a set of pure methods P , a symbolic heap H , a symbolic heap G , a symbolic store Γ , a path condition Σ , a term θ , an assertion ϕ , and a symbolic production postcondition Q . Further consider an interpretation I of the symbols such that $I \models \Sigma$ and an abstract heap H_a , such that $I \models H \Downarrow H_a$, and an abstract heap G_a , such that $I \models G \Downarrow G_a$, and a store Γ_a such that $I(\Gamma) = \Gamma_a$. Then, if $\text{produce}'(P, G, \Gamma, \Sigma, H, \phi, Q)$, then $\text{aproduce}'(P, G_a, \Gamma_a, H_a, \phi, (\lambda H'_a \bullet \exists H', \Sigma' \bullet I \models \Sigma' \wedge I \models H' \Downarrow H'_a \wedge Q(H', \Sigma')))$.
- Consider a set of pure methods P , a symbolic heap H , a symbolic heap G , a symbolic store Γ , a path condition Σ , a symbolic heap H_2 , an assertion ϕ , and a symbolic consumption postcondition Q . Further consider an interpretation I such that $I \vdash \Sigma$, an abstract heap H_a , such that $I \models H \Downarrow H_a$, an abstract heap G_a , such that $I \models G \Downarrow G_a$, a store $\Gamma_a = I(\Gamma)$, an abstract heap H_{2a} , such that $I \models H_2 \Downarrow H_{2a}$. Then, if $\text{consume}'(P, (H, G, \Gamma, \Sigma), H_2, \phi, Q)$, then $\text{aconsume}'(P, (H_a, G_a, \Gamma_a), H_{2a}, \phi, (\lambda H'_{2a}, \theta_a \bullet \exists H'_2, \Sigma', \theta \bullet I \models \Sigma' \wedge I \models H'_2 \Downarrow H'_{2a} \wedge I(\theta) = \theta_a \wedge Q(H'_2, \Sigma', \theta)))$.

Proof. By induction on the number of recursive calls of eval , $\text{produce}'$, and $\text{consume}'$. \square

Lemma 15 (Soundness of symbolic statement execution). *Consider a symbolic heap H , a symbolic store Γ , a path condition Σ , a statement s , and a symbolic execution postcondition Q . Further consider an interpretation I such that $I \models \Sigma$, an abstract heap H_a whose flattening is a concrete heap, such that $I \models H \Downarrow H_a$. If $\text{exec}(H, \emptyset, \Gamma, \Sigma, s, Q)$, then $\text{aexec}(H_a, I(\Gamma), s, (\lambda H'_a, \Gamma'_a \bullet \exists I', H', \Gamma', \Sigma' \bullet I \subseteq I' \wedge (I' \models \Sigma') \wedge (I' \models H' \Downarrow H'_a)))$. (In words: for each abstract state that is reached by abstract execution, there is a symbolic state that represents this abstract state and that satisfies the symbolic execution postcondition.)*

Proof. By induction on s . \square

Lemma 16. *Valid programs are abstractly valid.*

Theorem 2 (Soundness). *Valid programs do not get stuck.*

Proof. By combining the soundness of abstract validity (Theorem 1) and Lemma 16. \square