Gradually Verified Language with Recursive Predicates

Henry Blanchette

Contents

1 Grammar

$$
\begin{array}{rcl}
x, y, z & \in & \mathit{VAR} \\
v & \in & \mathit{VAL} \\
e & \in & \mathit{EXPR} \\
s & \in & \mathit{STMT} \\
o & \in & \mathit{LOC} \\
f & \in & \mathit{FIELDNAME} \\
m & \in & \mathit{METHODNAME} \\
C, D & \in & \mathit{CLASSNAME} \\
\alpha & \in & \mathit{PREDNAME} \\
P & ::= & \overline{\mathit{cls}}\ s \\
\mathit{cls} & ::= & \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{\mathit{field}}\ \overline{\mathit{pred}}\ \overline{\mathit{method}}\} \\
\mathit{field} & ::= & T\ f; \\
\mathit{pred} & ::= & \texttt{predicate}\ \alpha_C(\overline{T\ x}) = \widetilde{\phi} \\
T & ::= & \texttt{int} \mid \texttt{bool} \mid C \mid \top \\
\mathit{method} & ::= & T\ m(\overline{T\ x})\ \texttt{dynamically}\ \mathit{contract}\ \texttt{statically}\ \mathit{contract}\ \{s\} \\
\mathit{contract} & ::= & \texttt{requires}\ \widetilde{\phi}\ \texttt{ensures}\ \widetilde{\phi} \\
\oplus & ::= & + \mid - \mid * \mid \backslash \mid \&\& \mid \,||\, \\
\odot & ::= & \neq \mid = \mid < \mid > \mid \leq \mid \geq \\
s & ::= & \texttt{skip} \mid s_1\ ;\ s_2 \mid T\ x \mid x := e \mid \texttt{if}\ (e)\ \{s_1\}\ \texttt{else}\ \{s_2\} \\
 & & \mid \texttt{while}\ (e)\ \texttt{invariant}\ \widetilde{\phi}\ \{s\} \mid x.f := y \mid x := \texttt{new}\ C \mid y := z.m(\overline{x}) \\
 & & \mid y := z.m_C(\overline{x}) \mid \texttt{assert}\ \phi \mid \texttt{release}\ \phi \mid \texttt{hold}\ \phi\ \{s\} \mid \texttt{fold}\ A \mid \texttt{unfold}\ A \\
e & ::= & v \mid x \mid e \oplus e \mid e \odot e \mid e.f \\
x & ::= & \texttt{result} \mid \mathit{id} \mid \texttt{old}(\mathit{id}) \mid \texttt{this} \\
v & ::= & n \mid o \mid \texttt{null} \mid \texttt{true} \mid \texttt{false} \\
A & ::= & \alpha(\overline{e}) \mid \alpha_C(\overline{e}) \\
\circledast & ::= & \wedge \mid * \\
\phi & ::= & e \mid A \mid \texttt{acc}(e.f) \mid \phi \circledast \phi \mid (\texttt{if}\ e\ \texttt{then}\ \phi\ \texttt{else}\ \phi) \mid (\texttt{unfolding}\ A\ \texttt{in}\ \phi) \\
\widetilde{\phi} & ::= & \phi \mid ? * \phi
\end{array}
$$

2 Well-formedness

3 Aliasing

3.1 Definitions

An **object variable** is one of the following:

- a class instance variable i.e. a variable v such that $v : C$ for some class C,
- a class instance field reference i.e. a field reference $e.f$ where $e.f : C$ for some class C,
- `null` as a value such that $\texttt{null} : C$ for some class C.

Let $\mathcal{O}$ be a set of object variables. An $O \subset \mathcal{O}$ **aliases** if and only if each $o \in O$ refers to the same memory in the heap as each other, written propositionally as

$$\forall o, o' \in O : o = o' \iff \mathsf{aliases}(O)$$

While it is possible to keep track of negated aliasings (of the form $\sim \mathsf{aliases}\,\{o_\alpha\}$), this will not be needed for either aliasing tree construction or self-framing desicions. So, it will not be tracked i.e. $x \neq y$ does not contribute anything to an aliasing context.

3.2 Aliasing Context

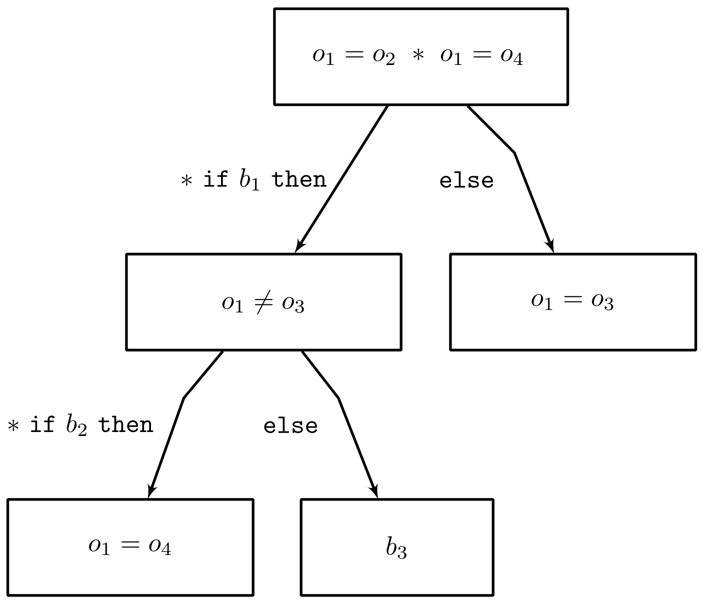
Let ϕ be a formula. The **aliasing context** $\mathcal{A}$ of ϕ is a tree of set of aliasing proposition about aliasing of object variables that appear in ϕ. $\mathcal{A}$ needs to be a tree because the conditional and unfolding sub-formulas that may appear in ϕ allow for branching aliasing contexts not expressible flatly at the top level. In the case of conditionals i.e. sub-formulas of the form $\texttt{if}\ e\ \texttt{then}\ \phi_1\ \texttt{else}\ \phi_2$, two branches sprout from the original context. In the case of unfoldings i.e. sub-formulas of the form $\texttt{unfolding}\ \alpha_C(\overline{e})\ \texttt{in}\ \phi$, one branch sprouts from the original context. Each node in the tree corresponds to a set of aliasing propositions, and each branch refers to a branch of a unique conditional in ϕ. The parts of the tree are labeled in such a way that modularly allows a specified sub-formula of ϕ to be matched to the unique aliasing sub-context that corresponds to it. For example, consider the following formula:

```
$\phi := (o_1 = o_2)\ *$
    $(\texttt{if}\ (b_1)$
      $\texttt{then}\ ($
        $(o_1 \neq o_3)\ *$
        $(\texttt{if}\ (b_2)$
          $\texttt{then}\ (o_1 = o_4)$
          $\texttt{else}\ (b_3)))$
      $\texttt{else}\ (o_1 = o_3))\ *$
    $(o_1 = o_4)$
```

where b_1, b_2 are arbitrary boolean expressions that do not assert aliasing propositions. ϕ has a formula-structure represented by the tree in figure **??**. The formula-structure tree for ϕ corresponds node-for-node and edge-for-edge to the aliasing context tree in figure **??**.

More generally, for ϕ a formula and ϕ' a sub-formula of ϕ, write $\mathcal{A}_\phi(\phi')$ as the **total**

Figure 1: Formula structure tree for ϕ.



aliasing context of ϕ' which includes aliasing propositions inherited from its ancestors in the aliasing context tree of ϕ. These aliasing contexts are combined via $\sqcup$ which will be defined in the next section. For example, the total aliasing context at the sub-formula $(o_1 = o_4)$ of ϕ is:

$$\mathcal{A}_\phi(o_1 = o_4) := \{\mathsf{aliased}\,\{o_1, o_2, o_4\}\}$$

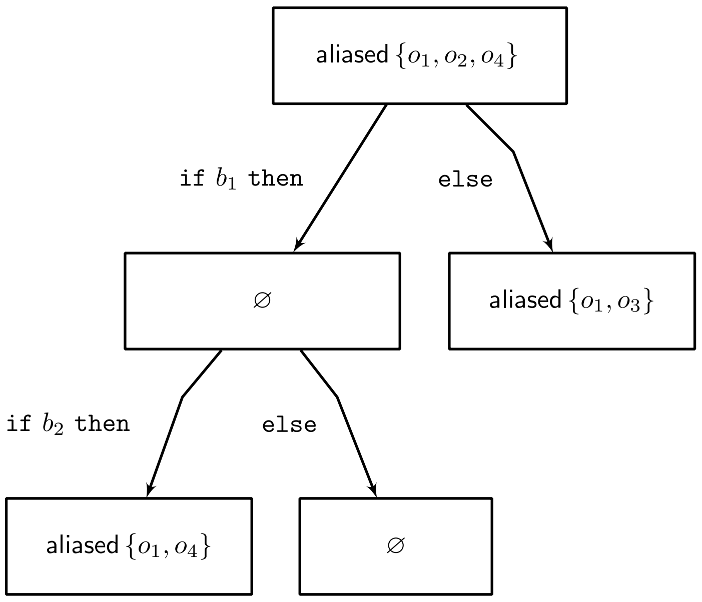
along with the fact that it has no child branches. Usually $\mathcal{A}_{\phi_{\mathsf{root}}}(\phi')$ is abbreviated to $\mathcal{A}(\phi')$ when the top level formula ϕ is implicit and ϕ' is a sub-formula of ϕ_{root}.

An aliasing context $\mathcal{A}$ may entail $\mathsf{aliased}(O)$ for some $O \subset \mathcal{O}$. Since $\mathcal{A}$ is efficiently represented as a set of propositions about sets, it may be the case that $\mathsf{aliased}(O) \notin \mathcal{A}$ yet still the previous judgement holds. For example, this is true when $\exists O' \subset \mathcal{O}$ such that $O \subset O'$ and $\mathsf{aliased}(O') \in \mathcal{A}$. So, the explicit definition for making this judgement is as follows:

$$\mathcal{A} \vdash \mathsf{aliased}(O) \iff \exists O' \subset \mathcal{O} : (O \subset O') \ \wedge \ (\mathsf{aliased}(O') \in \mathcal{A})$$

The notations $\mathsf{aliased}(O) \in \mathcal{A}$ is a little misleading because $\mathcal{A}$ is in fact a tree and not just a set. To be explicit, $\mathsf{aliased}(O) \in \mathcal{A}$ is defined to be set membership of the set of aliasing propositions in the total aliasing context at $\mathcal{A}$.

Figure 2: $\mathcal{A}(\phi)$, the aliasing context tree for ϕ.



3.3 Constructing an Aliasing Context

An aliasing context of a formula ϕ is a tree, where nodes represent local aliasing contexts and branches represent the branches of conditional sub-formulas nested in ϕ. So, an aliasing context is defined structurally as

$$\mathcal{A} ::= \langle A, \ \{l_\alpha : \mathcal{A}_\alpha\}\rangle$$

where A is a set of propositions about aliasing and the $l_\alpha : \mathcal{A}_\alpha$ are the nesting aliasing contexts that correspond to the branches of conditionals and unfoldings directly nested in ϕ, the l_α being labels for each child context.

Given a root formula ϕ_{root}, the aliasing context of ϕ_{root} is written $\mathcal{A}(\phi_{\mathsf{root}})$. With the root invariant, the following recursive algorithm constructs $\mathcal{A}(\phi)$ for any sub-formula of ϕ_{root}

(including $\mathcal{A}(\phi_{\mathsf{root}})$).

$$
\begin{array}{lll}
\mathcal{A}(\phi) \;:= & \mathsf{match}\ \phi\ \mathsf{with} & \\
& v & \mapsto \langle \varnothing,\ \varnothing \rangle \\
& x & \mapsto \langle \varnothing,\ \varnothing \rangle \\
& e_1\ \&\&\ e_2 & \mapsto \mathcal{A}(e_1) \sqcup \mathcal{A}(e_2) \\
& e_1 \mid\mid e_2 & \mapsto \mathcal{A}(\texttt{if}\ e_1\ \texttt{then}\ \texttt{true}\ \texttt{else}\ e_2) \\
& e_1 \oplus e_2 & \mapsto \langle \varnothing,\ \varnothing \rangle \\
& o_1 = o_2 & \mapsto \langle \{\mathsf{aliases}\,\{o_1, o_2\}\}\,,\ \varnothing \rangle \\
& e_1 \odot e_2 & \mapsto \langle \varnothing,\ \varnothing \rangle \\
& e.f & \mapsto \langle \varnothing,\ \varnothing \rangle \\
& \texttt{acc}(e.f) & \mapsto \langle \varnothing,\ \varnothing \rangle \\
& \phi_1 * \phi_2 & \mapsto \mathcal{A}(\phi_1) \sqcup \mathcal{A}(\phi_2) \\
& \phi_1 \wedge \phi_2 & \mapsto \mathcal{A}(\phi_1) \sqcup \mathcal{A}(\phi_2) \\
& \alpha_C(\overline{e}) & \mapsto \langle \varnothing,\ \varnothing \rangle \\
& \texttt{if}\ e\ \texttt{then}\ \phi_1\ \texttt{else}\ \phi_2 & \mapsto \langle \varnothing,\ \{e : \mathcal{A}(e) \sqcup \mathcal{A}(\phi_1),\ \sim e : (\mathcal{A}(\sim e)) \sqcup \mathcal{A}(\phi_2)\} \rangle \\
& \texttt{unfolding}\ \alpha_C(\overline{e})\ \texttt{in}\ \phi' & \mapsto \langle \varnothing,\ \{\texttt{unfolding}(\alpha_C(\overline{e})) : \mathcal{A}(\mathsf{unfold}\ \alpha_C(\overline{e})) \sqcup \mathcal{A}(\phi')\} \rangle
\end{array}
$$

Note the following:

- $\mathcal{A}(\phi_{\mathsf{root}})$ is implicitly unioned with the discrete aliasing context $\{\{o\} : o \in \mathcal{O}\}$. This convention yields that each $o \in \mathcal{O}$ is always considered an alias of itself.
- The $\sim e$ expression in the result of the rule for $\mathcal{A}(\texttt{if}\ e\ \texttt{then}\ \phi_1\ \texttt{else}\ \phi_2)$ means to negate the boolean expression of e
- The $e_1 \mid\mid e_2$ expression is translated into $\texttt{if}\ e_1\ \texttt{then}\ \texttt{true}\ \texttt{else}\ e_2$ for the purpose of aliasing. So, boolean or operations in forumals yield branching just like conditional expressions.
- The $\mathsf{unfold}\ \alpha_C(\overline{e})$ expression in the result of the rulle for $\mathcal{A}(\texttt{unfolding}\ \alpha_C(\overline{e})\ \texttt{in}\ \phi')$ is translated to a single unfolding of the body of $\alpha_C(\overline{e})$ with the arguments substituted appropriately.

As examples,

$$
\begin{aligned}
\mathcal{A}(\sim (x = y)) &= \mathcal{A}(x \neq y) = \langle \varnothing,\ \varnothing \rangle \\
\mathcal{A}(\sim (x \neq y)) &= \mathcal{A}(x = y) = \langle \{\mathsf{aliased}\,\{x, y\}\}\,,\ \varnothing \rangle
\end{aligned}
$$

Context union, $\sqcup$, and context intersection, $\sqcap$, are operations that combine aliasing contexts and are defined below.

$$
\begin{aligned}
&\langle A_1,\ \{l_\alpha : \mathcal{A}_\alpha\} \rangle \sqcup \langle A_2,\ \{l_\beta : \mathcal{A}_\beta\} \rangle := \\
&\qquad \langle \{\mathsf{aliased}\,\{o' \mid \forall o' : (A_1 \vdash \mathsf{aliased}\,\{o, o'\})\ \vee\ (A_2 \vdash \mathsf{aliased}\,\{o, o'\})\} \mid \forall o\}\,, \\
&\qquad \{l_\alpha : \mathcal{A}_\alpha\} \cup \{l_\beta : \mathcal{A}_\beta\} \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\langle A_1,\ \{l_\alpha : \mathcal{A}_\alpha\} \rangle \sqcap \langle A_2,\ \{l_\beta : \mathcal{A}_\beta\} \rangle := \\
&\qquad \langle \{\mathsf{aliased}\,\{o' \mid \forall o' : (A_1 \vdash \mathsf{aliased}\,\{o, o'\})\ \wedge\ (A_2 \vdash \mathsf{aliased}\,\{o, o'\})\} \mid \forall o\}\,, \\
&\qquad \{l_\alpha : \mathcal{A}_\alpha\} \cap \{l_\beta : \mathcal{A}_\beta\} \rangle
\end{aligned}
$$

4 Framing

4.1 Definitions

For framing, a formula is considered inside a **permission context**, a set of permissions, where a **permission** π is to do one of the following:

- to reference $e.f$, written $\mathsf{accessed}(e.f)$.
- to assume $\alpha_C(\overline{e})$, written $\mathsf{assumed}(\alpha_C(\overline{e}))$. This allows the a single unrolling of $\alpha_C(\overline{e})$. Explicitly, an instance of $\mathsf{assumed}(\alpha_C(\overline{e}))$ in a set of permissions Π may be expanded into $\Pi \cup \mathsf{granted}(\dots)$ where $\dots$ is replaced with a single unrolling of the body of $\alpha_C(\overline{e})$ with the arguments substituted appropriately[1].

Let ϕ be a formula. ϕ may **require** a permission π. For example, the formula $e.f = 1$ requires $\mathsf{accessed}(e.f)$, because it references $e.f$. The set of all permissions that ϕ requires is called the **requirements** of ϕ. ϕ may also **grant** a permission π. For example, the formula $\mathsf{acc}(e.f)$ grants the permission $\mathsf{accessed}(e.f)$.

Altogether, ϕ is **framed** by a set of permissions Π if all permissions required by ϕ are either in Π or granted by ϕ. The proposition that Π frames ϕ is written

$$\Pi \vDash_I \phi$$

Of course, ϕ may grant some of the permissions it requires but not all. The set of permissions that ϕ requires but does not grant is called the **footprint** of ϕ. The footprint of ϕ is written

$$\lfloor \phi \rfloor$$

Finally, a ϕ is called **self-framing** if and only if for any set of permissions Π, $\Pi \vDash_I \phi$. The proposition that ϕ is self-framing is written

$$\vdash_{\mathsf{frm}I} \phi$$

Note that $\vdash_{\mathsf{frm}I} \phi \iff \varnothing \vDash_I \phi$, in other words ϕ is self-framing if and only if it grants all of the permissions it requires. Or in other words still, $\lfloor \phi \rfloor = \varnothing$.

[1] As demonstrated by this description, $\mathsf{assumed}$ predicates are really just a useful shorthand and not a fundamentally new type of permission. The only kind fundamental kind of permission is $\mathsf{accessed}$.

4.2 Deciding Framing

Deciding $\Pi \vDash_I \phi$ must take into account the requirements, granteds, and aliases contained in Π and the sub-formulas of ϕ. The following recursive algorithm decides $\Pi \vDash_I \phi_{root}$, where $\mathcal{A}$ is implicitly assumed to be the top-level aliasing context (where the top-level in this context is the level that ϕ_{root} exists at in the program).

$$
\begin{array}{lcll}
\Pi \vDash_I \phi & \iff & \mathsf{match}\ \phi\ \mathsf{with} & \\
& & v & \mapsto \top \\
& & x & \mapsto \top \\
& & e_1 \oplus e_2 & \mapsto \Pi \vDash_I e_1, e_2 \\
& & e_1 \odot e_2 & \mapsto \Pi \vDash_I e_1, e_2 \\
& & e.f & \mapsto (\Pi \vDash_I e) \ \wedge\ (\Pi \vdash \mathsf{accessed}_\phi(e.f)) \\
& & \mathtt{acc}(e.f) & \mapsto (\Pi \vDash_I e) \\
& & \phi_1 \circledast \phi_2 & \mapsto (\Pi \cup \mathsf{granted}(\phi_2) \vDash_I \phi_1) \ \wedge \\
& & & \quad\ (\Pi \cup \mathsf{granted}(\phi_1) \vDash_I \phi_2) \\
& & \alpha_C(e_1, \dots, e_k) & \mapsto \Pi \vDash_I e_1, \dots, e_2 \\
& & \mathtt{if}\ e\ \mathtt{then}\ \phi_1\ \mathtt{else}\ \phi_2 & \mapsto \Pi \vDash_I e, \phi_1, \phi_2 \\
& & \mathtt{unfolding}\ \alpha_C(\overline{e})\ \mathtt{in}\ \phi' & \mapsto (\Pi \vDash_I \alpha_C(\overline{e})) \ \wedge\ (\Pi \vdash \mathsf{assumed}_\phi(\alpha_C(\overline{e}))) \ \wedge\ (\Pi \vDash_I \phi') \\
\\
\mathsf{granted}(\phi) & := & \mathsf{match}\ \phi\ \mathsf{with} & \\
& & e & \mapsto \varnothing \\
& & \mathtt{acc}(e.f) & \mapsto \{\mathsf{accessed}(e.f)\} \\
& & \phi_1 \circledast \phi_2 & \mapsto \mathsf{granted}(\phi_1) \cup \mathsf{granted}(\phi_2) \\
& & \alpha_C(\overline{e}) & \mapsto \{\mathsf{assumed}(\alpha_C(\overline{e}))\} \\
& & \mathtt{if}\ e\ \mathtt{then}\ \phi_1\ \mathtt{else}\ \phi_2 & \mapsto \mathsf{granted}(\phi_1) \cap \mathsf{granted}(\phi_2) \\
& & \mathtt{unfolding}\ \alpha_C(\overline{e})\ \mathtt{in}\ \phi' & \mapsto \mathsf{granted}(\phi')
\end{array}
$$

Where $\mathsf{accessed}_\phi$ and $\mathsf{assumed}_\phi$ indicate the respective propositions considered within the total alias context (including inherited aliasing contexts). More explicitly,

$$
\begin{array}{rl}
\Pi \vdash \mathsf{accessed}_\phi(o.f) & \iff \exists \mathsf{accessed}(o'.f) \in \Pi : \mathcal{A}(\phi) \vdash \mathsf{aliased}\ \{o, o'\} \\
\Pi \vdash \mathsf{assumed}_\phi(\alpha_C(e_1, \dots, e_k)) & \iff \exists \mathsf{assumed}(\alpha_C(e'_1, \dots, e'_k)) \in \Pi : \forall i : \mathcal{A}(\phi) \vdash \mathsf{aliased}\ \{e_i, e'_i\}
\end{array}
$$

4.3 Examples

In the following examples, assume that the considered formulas are well-formed.

Example 1

Define

$$\phi_{\mathsf{root}} := x = y * \mathsf{acc}(x.f) * \mathsf{acc}(y.f).$$

Then

$$\mathcal{A}(\phi_{\mathsf{root}}) = \langle \{\mathsf{aliased}\,\{x, y\}\}\,,\ \varnothing\rangle.$$

And so,

$$\begin{aligned}
\vdash_{\mathsf{frm}I} \phi_{\mathsf{root}} &\iff \varnothing \vDash_I \phi_{\mathsf{root}} \\
&\iff \varnothing \vDash_I x = y * \mathsf{acc}(x.f) * \mathsf{acc}(y.f) \\
&\iff (\mathsf{granted}(\mathsf{acc}(x.f) * \mathsf{acc}(y.f)) \vDash_I x = y) \ \wedge \\
&\qquad (\mathsf{granted}(x = y * \mathsf{acc}(y.f)) \vDash_I \mathsf{acc}(x.f)) \ \wedge \\
&\qquad (\mathsf{granted}(x = y * \mathsf{acc}(x.f)) \vDash_I \mathsf{acc}(y.f)) \\
&\iff \top \ \wedge \\
&\qquad (\mathsf{granted}(x = y * \mathsf{acc}(y.f)) \vDash_I x) \ \wedge \\
&\qquad (\mathsf{granted}(x = y * \mathsf{acc}(x.f)) \vDash_I y) \\
&\iff \top \ \wedge \ \top \ \wedge \ \top \\
&\iff \top
\end{aligned}$$

Example 2

Define

$$\phi_{\mathsf{root}} := \texttt{acc}(x.f) \ * \ (\texttt{if } x.f = 1 \texttt{ then } \mathit{true} \texttt{ else } \texttt{acc}(x.f))$$

Then

$$\mathcal{A}(\phi_{\mathsf{root}}) = \langle \varnothing,\ \{x.f = 1 : \langle \varnothing,\ \varnothing\rangle,\ x.f \neq 1 : \langle \varnothing,\ \varnothing\rangle\}\rangle$$

And so,

$$\begin{aligned}
\vdash_{\mathsf{frm}I} \phi_{\mathsf{root}} &\iff \varnothing \vDash_I \phi_{\mathsf{root}} \\
&\iff \varnothing \vDash_I \texttt{acc}(x.f) \ * \ (\texttt{if } x.f = 1 \texttt{ then } \mathit{true} \texttt{ else } \texttt{acc}(x.f)) \\
&\iff (\mathsf{granted}(\texttt{if } x.f = 1 \texttt{ then true else acc}(x.f)) \vDash_I \texttt{acc}(x.f)) \land \\
&\qquad (\mathsf{granted}(\texttt{acc}(x.f)) \vDash_I \texttt{if } x.f = 1 \texttt{ then true else acc}(x.f)) \\
&\iff (\mathsf{granted}(\texttt{if } x.f = 1 \texttt{ then true else acc}(x.f)) \vDash_I x) \land \\
&\qquad (\mathsf{granted}(\texttt{acc}(x.f)) \vDash_I \texttt{if } x.f = 1 \texttt{ then true else acc}(x.f)) \\
&\iff \top \ \land \ (\mathsf{granted}(\texttt{acc}(x.f)) \vDash_I \texttt{if } x.f = 1 \texttt{ then true else acc}(x.f)) \\
&\iff \top \ \land \ (\{\mathsf{accessed}(x.f)\} \vDash_I \texttt{if } x.f = 1 \texttt{ then true else acc}(x.f)) \\
&\iff \top \ \land \ (\{\mathsf{accessed}(x.f)\} \vDash_I x.f = 1) \ \land \ (\{\mathsf{accessed}(x.f)\} \vDash_I \texttt{true}) \land \\
&\qquad (\{\mathsf{accessed}(x.f)\} \vDash_I \texttt{acc}(x.f) \\
&\iff \top \ \land \ (\{\mathsf{accessed}(x.f)\} \vdash \mathsf{accessed}_{\phi_{\mathsf{root}}}(x.f)) \land \\
&\qquad \top \ \land \ (\{\mathsf{accessed}(x.f)\} \vDash_I x) \\
&\iff \top \ \land \ \top \ \land \ \top \ \land \ \top \\
&\iff \top
\end{aligned}$$

Example 3

Define

$$\phi_{\mathsf{root}} := \mathtt{acc}(x.f) \ * \ x = y \ * \ y.f = 1$$

Then

$$\mathcal{A}(\phi_{\mathsf{root}}) = \langle \{\mathsf{aliased}\, \{x, y\}\}\,,\ \varnothing \rangle$$

And so,

$$\begin{aligned}
\vdash_{\mathsf{frm}I} \phi_{\mathsf{root}} &\iff \varnothing \vDash_I \phi_{\mathsf{root}} \\
&\iff \varnothing \vDash_I \mathtt{acc}(x.f) \ * \ x = y \ * \ y.f = 1 \\
&\iff (\mathsf{granted}(x = y * y.f = 1) \vDash_I \mathtt{acc}(x.f)) \wedge \\
&\qquad (\mathsf{granted}(\mathtt{acc}(x.f) * y.f = 1) \vDash_I x = y) \wedge \\
&\qquad (\mathsf{granted}(\mathtt{acc}(x.f) * x = y) \vDash_I y.f = 1) \\
&\iff (\mathsf{granted}(x = y * y.f = 1) \vDash_I x) \wedge \\
&\qquad (\mathsf{granted}(\mathtt{acc}(x.f) * y.f = 1) \vDash_I x, y) \wedge \\
&\qquad (\mathsf{granted}(\mathtt{acc}(x.f) * x = y) \vDash_I y.f) \\
&\iff \top \ \wedge \ \top \wedge \ (\{\mathsf{accessed}(x.f)\} \vdash \mathsf{accessed}_{\phi_{\mathsf{root}}}(y.f)) \\
&\iff \top \ \wedge \ \top \wedge \ \top \\
&\iff \top
\end{aligned}$$

Example 4

Define

```
class List {
    int $head$;
    List $tail$;
    predicate List($l$) =
        $l \neq$ null $*$ acc($l.head$) $*$ acc($l.tail$) $*$
        if $l.tail =$ null then true else List($l.tail$);
}
```

$$\phi_{\mathsf{root}} := \mathsf{List}(l) \; * \; \texttt{unfolding}\ \mathsf{List}(l)\ \texttt{in}\ l.head = 1$$

Then

$$\mathcal{A}(\phi_{\mathsf{root}}) = \langle \varnothing,\ \{\texttt{unfolding}(\mathsf{List}(l)) : \{\langle \varnothing,\ \{t.tail = \texttt{null} : \langle \{\mathsf{aliased}\,\{t.tail, \texttt{null}\}\}\,,\ \varnothing\rangle,\ t.tail \neq \texttt{null} : \langle \varnothing,\ \varnothing\rangle\}\rangle\}\}\rangle$$

And so,

$$\begin{aligned}
\vdash_{\mathsf{frm}I} \phi_{\mathsf{root}} &\iff \varnothing \vDash_I \phi_{\mathsf{root}} \\
&\iff \varnothing \vDash_I \mathsf{List}(l) \; * \; \texttt{unfolding}\ \mathsf{List}(l)\ \texttt{in}\ l.head = 1 \\
&\iff (\mathsf{granted}(\texttt{unfolding}\ \mathsf{List}(l)\ \texttt{in}\ l.head = 1) \vDash_I \mathsf{List}(l)) \wedge \\
&\qquad (\mathsf{granted}(\mathsf{List}(l)) \vDash_I \texttt{unfolding}\ \mathsf{List}(l)\ \texttt{in}\ l.head = 1) \\
&\iff \top \;\wedge\; (\{\mathsf{assumed}(\mathsf{List}(l))\} \vDash_I \texttt{unfolding}\ \mathsf{List}(l)\ \texttt{in}\ l.head = 1) \\
&\iff \top \;\wedge\; (\mathsf{granted}(l \neq \texttt{null} \; * \; \texttt{acc}(l.head) \; * \; \texttt{acc}(l.tail) \; * \\
&\qquad\quad \texttt{if}\ l.tail = \texttt{null}\ \texttt{then}\ \texttt{true}\ \texttt{else}\ \mathsf{List}(l.tail)) \vDash_I \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(expansion of } \mathsf{assumed}(\mathsf{List}(l))\text{))} \\
&\qquad l.head = 1) \\
&\iff \top \;\wedge\; (\{\mathsf{accessed}(l.head), \mathsf{accessed}(l, tail)\}) \vDash_I l.head = 1) \\
&\iff \top \;\wedge\; (\{\mathsf{accessed}(l.head), \mathsf{accessed}(l, tail)\}) \vdash \mathsf{accessed}_{\phi_{\mathsf{root}}}(l.head)) \\
&\iff \top \;\wedge\; \top \\
&\iff \top
\end{aligned}$$

Example 5

Define

$$\phi_{\mathsf{root}} := \texttt{if } x = \texttt{null} \texttt{ then } \texttt{true} \texttt{ else } (\texttt{acc}(x.f) * x.f = 1)$$

Then

$$\mathcal{A}(\phi_{\mathsf{root}}) = \langle \varnothing,\ \{x = \texttt{null} : \langle \{\mathsf{aliased}\,\{x, \texttt{null}\}\}\,,\ \varnothing\rangle, x \neq \texttt{null} : \langle \varnothing,\ \varnothing\rangle\}\rangle$$

And so,

$$\begin{aligned}
\vdash_{\mathsf{frm}I} \phi_{\mathsf{root}} &\iff \varnothing \vDash_I \phi_{\mathsf{root}} \\
&\iff \varnothing \vDash_I \texttt{if } x = \texttt{null} \texttt{ then } \texttt{true} \texttt{ else } (\texttt{acc}(x.f) * x.f = 1) \\
&\iff (\varnothing \vDash_I x = \texttt{null}) \ \wedge\ (\varnothing \vDash_I \texttt{true}) \ \wedge\ (\varnothing \vDash_I \texttt{acc}(x.f) * x.f = 1) \\
&\iff \top \ \wedge\ \top \ \wedge (\mathsf{granted}(x.f = 1) \vDash_I \texttt{acc}(x.f)) \ \wedge (\mathsf{granted}(\texttt{acc}(x.f)) \vDash_I x.f = 1) \\
&\iff \top \ \wedge\ \top \ \wedge (\varnothing \vDash_I \texttt{acc}(x.f)) \ \wedge (\{\mathsf{accessed}(x.f)\} \vDash_I x.f = 1) \\
&\iff \top \ \wedge\ \top \ \wedge (\varnothing \vDash_I x) \ \wedge \\
&\qquad (\{\mathsf{accessed}(x.f)\} \vdash \mathsf{accessed}_{x.f}(x.f)) \ \wedge (\{\mathsf{accessed}(x.f)\} \vDash_I 1) \\
&\iff \top \ \wedge\ \top \ \wedge\ \top \ \wedge\ \top \ \wedge\ \top \\
&\iff \top
\end{aligned}$$

Example 6

Use the definition of List from example 4. Define

$$\begin{aligned}
\phi_{\mathsf{root}} &:= \mathtt{acc}(x.f) \;*\; \phi_1 \;*\phi_2 \\
\phi_1 &:= \mathtt{if}\ x.f = 1\ \mathtt{then}\ x = y\ \mathtt{else}\ \mathtt{true} \\
\phi_2 &:= \mathtt{if}\ x.f = 1\ \mathtt{then}\ y.f = 1\ \mathtt{else}\ \mathtt{true}
\end{aligned}$$

Then

$$\mathcal{A}(\phi_{\mathsf{root}}) = \langle \varnothing,\ \{x.f = 1 : \langle \{\mathsf{aliased}\,\{x, y\}\}\,,\ \varnothing\rangle,\ x.f \neq 1 : \langle \varnothing,\ \varnothing\rangle\}\rangle$$

And so,

$$\begin{aligned}
\vdash_{\mathsf{frm}I} \phi_{\mathsf{root}} \iff\ & \varnothing \vDash_I \mathtt{acc}(x.f) \;*\; \phi_1 \;*\phi_2 \\
\iff\ & (\mathsf{granted}(\phi_1 * \phi_2) \vDash_I \mathtt{acc}(x.f)) \ \wedge\ (\mathsf{granted}(\mathtt{acc}(x.f) * \phi_2) \vDash_I \phi_1) \wedge \\
& (\mathsf{granted}(\mathtt{acc}(x.f) * \phi_1) \vDash_I \phi_2) \\
\iff\ & (\mathsf{granted}(\phi_1 * \phi_2) \vDash_I x) \wedge \\
& (\{\mathsf{accessed}(x.f)\} \vDash_I \mathtt{if}\ x.f = 1\ \mathtt{then}\ x = y\ \mathtt{else}\ \mathtt{true}) \wedge \\
& (\{\mathsf{accessed}(x.f)\} \vDash_I \mathtt{if}\ x.f = 1\ \mathtt{then}\ y.f = 1\ \mathtt{else}\ \mathtt{true}) \\
\iff\ & \top \ \wedge\ (\{\mathsf{accessed}(x.f)\} \vDash_I (x.f = 1), (x = y), (\mathtt{true})) \wedge \\
& (\{\mathsf{accessed}(x.f)\} \vDash_I (x.f = 1), (y.f = 1), (\mathtt{true})) \\
\iff\ & \top \ \wedge\ (\{\mathsf{accessed}(x.f)\} \vDash_I x.f) \ \wedge\ (\{\mathsf{accessed}(x.f)\} \vDash_I x.f) \wedge \\
& (\{\mathsf{accessed}(x.f)\} \vDash_I y.f) \\
\iff\ & \top \ \wedge\ (\{\mathsf{accessed}(x.f)\} \vdash \mathsf{accessed}_{\phi_{\mathsf{root}}}(x.f)) \wedge \\
& (\{\mathsf{accessed}(x.f)\} \vdash \mathsf{accessed}_{y.f=1}(y.f)) \\
\iff\ & \top \wedge \top \wedge \top \qquad (\star) \\
\iff\ & \top
\end{aligned}$$

$(\star)$: $\{\mathsf{accessed}(x.f)\} \vdash \mathsf{accessed}_{y.f=1}(y.f) \iff \top$ since $\mathcal{A}(y.f = 1) \vdash \mathsf{aliased}\,\{x, y\}$ because $\mathcal{A}(y.f = 1)$ and $\mathcal{A}(x = y)$ are combined into a single branch of $\mathcal{A}(\phi_{\mathsf{root}})$, as they have the same conditions.

Example 7

Define

$$\phi_{\mathsf{root}} := \texttt{if } x = y \texttt{ then } \texttt{acc}(x.f) \texttt{ else } x.f = 2$$

Then

$$\mathcal{A}(\phi_{\mathsf{root}}) = \langle \varnothing,\ \{x = y : \langle \{\mathsf{aliased}\,\{x, y\}\}\,,\ \varnothing\rangle,\ x \neq y : \langle \varnothing,\ \varnothing\rangle\}\rangle$$

And so,

$$\begin{aligned}
\vdash_{\mathsf{frm}I}\ \phi_{\mathsf{root}} &\iff \varnothing \vDash_I \phi_{\mathsf{root}} \\
&\iff \varnothing \vDash_I \texttt{if } x = y \texttt{ then } \texttt{acc}(x.f) \texttt{ else } x.f = 2 \\
&\iff \varnothing \vDash_I (x = y), (\texttt{acc}(x.f)), (x.f = 2) \\
&\iff \top\ \wedge\ (\varnothing \vDash_I x)\ \wedge\ (\varnothing \vDash_I x.f) \\
&\iff \top\ \wedge\ \top\ \wedge\ (\varnothing \vdash \mathsf{accessed}_{x.f=2}(x.f)) \\
&\iff \top\ \wedge\ \top\ \wedge\ \bot \\
&\iff \bot
\end{aligned}$$

Example 8

Define

$$\texttt{predicate}\ \mathsf{aliasChoice}(x,y,z) :=\ \ x = y \mid\mid x = z$$

$$\begin{aligned}
\phi_{\mathsf{root}} &:= \texttt{acc}(x.f)\ *\ \mathsf{aliasChoice}(x,y,z)\ *\ \texttt{unfolding}(\mathsf{aliasChoice}(x,y,z))\ \texttt{in}\ \phi_1 \\
\phi_1 &:= y.f = 1 \mid\mid z.f = 1
\end{aligned}$$

Then

$$\begin{aligned}
\mathcal{A}(\phi_{\mathsf{root}}) = \langle \varnothing,\ \{&\texttt{unfolding}(\mathsf{aliasChoice}(x,y,z)) : \\
&\{x = y : \langle \{\mathsf{aliased}\,\{x,y\}\}\,,\ \varnothing\rangle, \\
&\ x \neq y : \langle \{\mathsf{aliased}\,\{x,z\}\}\,,\ \varnothing\rangle, \\
&\ y.f = 1 : \langle \varnothing,\ \varnothing\rangle, \\
&\ y.f \neq 1 : \langle \varnothing,\ \varnothing\rangle\}\}\rangle\}
\end{aligned}$$

Note that the $x = y \mid\mid x = z$ in the body of $\mathsf{aliasChoice}$ is translated to $\texttt{if}\ x = y\ \texttt{then}\ \texttt{true}\ \texttt{else}\ x = z$ when construction $\mathcal{A}(\phi_1)$. And so,

$$\begin{aligned}
\vdash_{\mathsf{frm}I} \phi_{\mathsf{root}} &\iff \varnothing \vDash_I \phi_{\mathsf{root}} \\
&\iff \varnothing \vDash_I \texttt{acc}(x.f)\ *\ \mathsf{aliasChoice}(x,y,z)\ *\ \texttt{unfolding}(\mathsf{aliasChoice}(x,y,z))\ \texttt{in}\ \phi_1 \\
&\iff (\mathsf{granted}(\mathsf{aliasChoice}(x,y,z)\ *\ \texttt{unfolding}(\mathsf{aliasChoice}(x,y,z))\ \texttt{in}\ \phi_1) \vDash_I \texttt{acc}(x.f)) \land \\
&\qquad (\mathsf{granted}(\texttt{acc}(x.f)\ *\ \texttt{unfolding}(\mathsf{aliasChoice}(x,y,z))\ \texttt{in}\ \phi_1) \vDash_I \mathsf{aliasChoice}(x,y,z)) \land \\
&\qquad (\mathsf{granted}(\texttt{acc}(x.f)\ *\ \mathsf{aliasChoice}(x,y,z)) \vDash_I \texttt{unfolding}(\mathsf{aliasChoice}(x,y,z))\ \texttt{in}\ \phi_1) \\
&\iff \top\ \land\ (\{\mathsf{accessed}(x.f)\} \vDash_I \mathsf{aliasChoice}(x,y,z)) \land \\
&\qquad (\{\mathsf{accessed}(x.f), \mathsf{assumed}(\mathsf{aliasChoice}(x,y,z))\} \vDash_I \texttt{unfolding}(\mathsf{aliasChoice}(x,y,z))\ \texttt{in}\ \phi_1) \\
&\iff \top\ \land\ \top\ \land\ (\{\mathsf{accessed}(x.f)\} \vDash_I y.f = 1 \mid\mid z.f = 1) \\
&\iff \top\ \land\ \top\ \land\ (\{\mathsf{accessed}(x.f)\} \vDash_I y.f)\ \land\ (\{\mathsf{accessed}(x.f)\} \vDash_I z.f) \\
&\iff \top\ \land\ \top\ \land\ (\{\mathsf{accessed}(x.f)\} \vdash \mathsf{accessed}_{y.f=1}(y.f))\ \land\ (\{\mathsf{accessed}(x.f)\} \vdash \mathsf{accessed}_{z.f=1}(z.f)) \\
&\qquad (\star) \\
&\iff \top \land \top \land \bot \land \bot \\
&\iff \bot
\end{aligned}$$

($\star$): The acccessed to $y.f, z.f$ are not framed because it is statically undetermined which branch of $x = y \mid\mid x = z$ will be taken. The case could arise that $x = z$ and then when checking the condition $y.f = 1$ there is not access to $y.f$. The idea of the original formula can be correctly captured in one of the following revisions:

$$\begin{aligned}
\phi'_{\mathsf{root}} &:= \texttt{acc}(x.f)\ *\ (x = y \mid\mid x = z)\ *\ \texttt{if}\ x = y\ \texttt{then}\ y.f = 1\ \texttt{else}\ z.f = 1 \\
\phi'_{\mathsf{root}} &:= \texttt{acc}(x.f)\ *\ \texttt{if}\ x = y\ \texttt{then}\ y.f = 1\ \texttt{else}\ (\texttt{if}\ x = z\ \texttt{then}\ z.f = 1\ \texttt{else}\ \texttt{false})
\end{aligned}$$

For example. the $z.f = 1$ will be framed because the aliasing context of the $x \neq y$ branch of $(x = y \mid\mid x = z)$ will be combined with the aliasing context of the $x \neq y$ branch of $(\texttt{if}\ x = y\ \texttt{then}\ y.f = 1\ \texttt{else}\ z.f = 1)$, yielding $\mathsf{aliased}\,\{x,z\}$ in $z.f = 1$. The similar case holds for the $x = y$ branches combining to allow the aliasing to frame $y.f = 1$.

5 Satisfiability

6 Implication

7 Weakest Predonditions

7.1 Concrete Weakest Liberal Precondition (WLP) Rules

$$\mathsf{WLP} : \textsc{Statement} \times \textsc{FrmSatFormula} \rightarrow \textsc{FrmSatFormula}$$

$$\mathsf{WLP}(s, \phi) := \text{ match } s \text{ with}$$

$$\begin{array}{|lcl}
\texttt{skip} & \mapsto & \phi \\
s_1\texttt{;}\ s_2 & \mapsto & \mathsf{WLP}(s_1,\ \mathsf{WLP}(s_2, \phi)) \\
T\ x & \mapsto & \mathsf{assert}\ x \text{ does not appear in } \phi;\ \phi \\
x\ \texttt{:=}\ e & \mapsto & \mathsf{required}\,(e)\ \wedge\ [e/x]\phi \\
x\ \texttt{:=}\ \texttt{new}\ C & \mapsto & [\mathsf{new}(C)/x]\phi \\
x.f\ \texttt{:=}\ y & \mapsto & \mathsf{required}\,(x.f)\ \wedge\ [y/x.f]\phi \\
y\ \texttt{:=}\ z.m_C(\overline{e}) & \mapsto & \mathsf{required}\,(\overline{e})\ \wedge\ z\ \texttt{!=}\ \texttt{null}\ \wedge \\
 & & [z/\texttt{this},\ \overline{e/x}]\mathsf{pre}(m_C) * \\
 & & \mathsf{handleMethodCall}(z.m_C(\overline{e}), \phi) \\
\texttt{if}\ (e)\ \{s_{\text{the}}\}\ \texttt{else}\ \{s_{\text{els}}\} & \mapsto & \mathsf{required}\,(e)\ \wedge \\
 & & \texttt{if}\ (e)\ \texttt{then}\ \mathsf{WLP}(s_{\text{the}}, \phi)\ \texttt{else}\ \mathsf{WLP}(s_{\text{els}}, \phi) \\
\texttt{while}\ (e)\ \texttt{invariant}\ \phi_{\text{inv}}\ \{s_{\text{bod}}\} & \mapsto & \mathsf{required}\,(e)\ \wedge\ \phi_{\text{inv}}\ \wedge \\
 & & (\texttt{if}\ (e)\ \texttt{then}\ \mathsf{WLP}(s_{\text{bod}}, \phi_{\text{inv}})\ \texttt{else}\ \texttt{true}) * \\
 & & \mathsf{handleWhileLoop}(e, \phi_{\text{inv}}) \\
\texttt{assert}\ \phi_{\text{ass}} & \mapsto & \mathsf{required}\,(\phi_{\text{ass}})\ \wedge\ \phi_{\text{ass}}\ \wedge\ \phi \\
\texttt{hold}\ \phi_{\text{hol}}\ \{s_{\text{bod}}\} & \mapsto & \text{(unimplemented)} \\
\texttt{release}\ \phi_{\text{rel}} & \mapsto & \text{(unimplemented)} \\
\texttt{unfold}\ \alpha_C(\overline{e}) & \mapsto & \mathsf{required}\,(\alpha_C(\overline{e}))_{\text{unf}}\ \wedge \\
 & & [\mathsf{body}(\alpha_C(\overline{e}))/\alpha_C(\overline{e}),\ \phi'/\texttt{unfolding}\ \alpha_C(\overline{e})\ \texttt{in}\ \phi']\phi \\
\texttt{fold}\ \alpha_C(\overline{e}) & \mapsto & \mathsf{required}\,(\overline{e})\ \wedge\ [\alpha_C(\overline{e})/\mathsf{body}(\alpha_C(\overline{e}))]\phi
\end{array}$$

Since WLP takes a framed, satisfiable formula and yields a framed, satisfiable formula, there is an implicit check that asserts these properties before and after WLP is computed. Note that the substitutions in the above rules do not substitute instances that appear inside of accesses (i.e. of the form $\texttt{acc}(e.f)$) or meta-predicates such as tainted, etc.

Note the following syntax rules:

- The OCaml-inspired syntax of the form $a; s$ for side-effects in evaluation is defined as "execute side-effect a, then evaluate as s."
- The meta-function $\textsf{assert}\ \cdot$ is executed imperitively, raising an error if the argument is false.

Finally, the idiom "a appears in b" is defined as follows:

$$\begin{aligned}
&e \text{ appears in } e' && \iff && \exists e'_{\text{sub}} \text{ a sub-expression of } e' : e = e'_{\text{sub}} \\
&e \text{ appears in } \texttt{acc}(e') && \iff && e = e' \\
&e \text{ appears in } \texttt{if } e' \texttt{ then } \phi_{\text{the}} \texttt{ else } \phi_{\text{els}} && \iff && e \text{ appears in at least one of } e', \phi_{\text{the}}, \phi_{\text{els}} \\
&e \text{ appears in } \alpha_C(\overline{e}) && \iff && e \text{ appears in at least one of } \overline{e}
\end{aligned}$$

7.2 Assumed and Tainted Logic

Assumed logic concerns assumed formulas that do not result direcly from statically verifying the visible code. *Tainted logic* concerns how references (variables and field references) may have their referenced values changed by sources external to the visible code. These logics are handled in the following cases:

- Method calls — The specification of a called method is visible, but the body is not visible due to the (intended) modular structure of verification. So, the validity of the called method's implementation is assumed. Additionally, a method call taints references that it requires access to.
- While loops — the actual execution of a while loop's body is statically invisible since the number of times the while loop's body will exectute is not statically calculated. So, references that are set inside the while loop's body are tainted.

Define a *reference*, r, to be an instance of x (a variable), $e.f$ or $\alpha_C(\overline{e})$. Then *access to a reference* is defined as follows:

$$\textsf{access}(r) := \begin{cases} \texttt{false} & \text{if } r = x \\ \texttt{acc}(e.f) & \text{if } r = e.f \\ \alpha_C(\overline{e}) & \text{if } r = \alpha_C(\overline{e}) \end{cases}$$

7.2.1 Handling Method Calls

The handleMethodCall helper function, for a given method call $z.m_C(\overline{e})$ and post-condition ϕ, does the following:

- assert that permissions in required (ϕ) and granted by pre$(z.m_C(\overline{e}))$ are also granted by post$(z.m_C(\overline{e}))$
- assume taint-substituted pre$(z.m_C(\overline{e}))$
- return taint-substituted ϕ

The following definition reflects the above descriptions, in order:

$$\begin{aligned}
&\mathsf{handleMethodCall}(z.m_C(\overline{e}), \phi) := \\
&\quad \mathsf{assert}\ \mathsf{granted}(\mathsf{post}(z.m_C(\overline{e}))),\ \forall \pi : \mathsf{required}\,(\phi)\,,\ \mathsf{granted}(\mathsf{pre}(z.m_C(\overline{e})))\implies \pi; \\
&\quad \mathsf{assume}\ [\mathsf{tainted}_{\mathsf{uid}(z.m_C(\overline{e}))}(r)/r : r\ \mathsf{isTaintedBy}\ z.m_C(\overline{e})]\mathsf{pre}(z.m_C(\overline{e})); \\
&\quad [\mathsf{tainted}_{\mathsf{uid}(z.m_C(\overline{e}))}(r)/r : r\ \mathsf{isTaintedBy}\ z.m_C(\overline{e})]\phi
\end{aligned}$$

7.2.2 Handling While Loops

The handleWhileLoop helper function, for a given while loop with condition e, invariant ϕ_{inv}, and post-condition ϕ, does the following:

- assume taint-substututed ϕ_{inv}
- return taint-substituted ϕ

The following definition reflects the above descriptions, in order:

$$\begin{aligned}
&\mathsf{handleWhileLoop}(z.m_C(\overline{e}), s_{\text{bod}}, \phi) := \\
&\quad \mathsf{assume}\ [\mathsf{tainted}_{\mathsf{uid}(\mathsf{while}(e,\phi_{\text{inv}}))}(r)/r : r\ \mathsf{isTaintedBy}\ s_{\text{bod}}]\phi_{\text{inv}}; \\
&\quad [\mathsf{tainted}_{\mathsf{uid}(\mathsf{while}(e,\phi_{\text{inv}}))}(r)/r : r\ \mathsf{isTaintedBy}\ s_{\text{bod}}]\phi
\end{aligned}$$

7.2.3 Assumptions

The *assumed* formula, local to the encompassing highest-level $\mathsf{WLP}(s, \phi)$ calculation, represents the truths that are assumed via references external to the direct implications of s. For example, the post-condition of a method call appearing in s may yield truths that are accepted as assumptions due to the modular structure of verification — the method call is assumed to be verified seperately (modularly).

These truths must be kept separate from $\phi_{\mathsf{WLP}} := \mathsf{WLP}(s, \phi)$ because they do not need to be implied by the pre-condition concerning ϕ_{WLP}. The $\mathsf{assume}(\phi)$ function is how these truths are accumulated during the WLP computation.

$$\mathsf{assume}\ \phi := \text{ set the } \textit{assumed} \text{ formula, } \phi_{\text{ass}}, \text{ to } \phi \wedge \phi_{\text{ass}}$$

7.2.4 Taints

The $\mathsf{tainted}$ meta-predicate indicates that the wrapped reference has been *tainted* by a source identified by the given unique identifier. A *tainted* reference is one that relies on the values of parts of the heap that may have been changed externally. For example, if a method call requires access to $x.f$, then $x.f$ is tainted because the method call could have changed the value of $x.f$.

Tainted references can only be asserted in some specific ways. For example, the previously mentioned method call could ensure that $x.f = v$, where v is some value, and this would yield the *assumption* that $\mathsf{tainted}_{\mathsf{uid}(z.m_C(\overline{e}))}(x.f) = v$. The following rules define the $\mathsf{isTaintedBy}$ relation between references (left) and statements or statement-fragments (right).

$$\begin{array}{lcl}
r\ \mathsf{isTaintedBy}\ r\ \texttt{:=}\ e & \Longleftarrow & \mathsf{true} \\
r\ \mathsf{isTaintedBy}\ y\ \texttt{:=}\ z.m_C(\overline{e}) & \Longleftrightarrow & r\ \mathsf{isTaintedBy}\ z.m_C(\overline{e}) \\
r\ \mathsf{isTaintedBy}\ z.m_C(\overline{e}) & \Longleftrightarrow & \mathsf{required}\,(\mathsf{pre}(z.m_C(\overline{e}))) \implies \mathsf{access}(r) \\
r\ \mathsf{isTaintedBy}\ s_1\,;s_2 & \Longleftrightarrow & r\ \mathsf{isTaintedBy}\ s_1\ \vee\ r\ \mathsf{isTaintedBy}\ s_2
\end{array}$$

The $\mathsf{uid}(\cdot)$ function generates a unique identifier for the given instance. This is needed because instances that contain the same arguments but appear in different parts of a program (where heap state may be different) must be treated as unique. The following function gathers all the references tainted via the arguments:

7.3 Utility Functions

The implementations of the functions in this section can be made much more efficient than the naive definition here in mathematical notation. For example, calculating the footprint of expressions and formulas can avoid redundancy by not generating permission-subformulas that are already satisfied. This can be implemented as implicit in $\wedge$ by a wrapper $\wedge_{\text{wrap}}$ operation in some way similar to this:

$$\phi \ \wedge_{\text{wrap}} \ \phi' := \begin{cases} \phi & \text{if } \phi \implies \phi' \\ \phi \wedge \phi' & \text{otherwise} \end{cases}$$

The following functions are useful abbreviations for common constructs.

$\mathsf{new}(C)$	$:=$	an object that is a new instance of class C,
		where all fields are assigned to their default values
$\mathsf{pre}(z.m_C(\overline{e}))$	$:=$	$[z/\texttt{this},\ \overline{e/x}]\mathsf{pre}(m_C)$
$\mathsf{pre}(m_C)$	$:=$	the static-contract pre-condition of m_C
$\mathsf{post}(z.m_C(\overline{e}))$	$:=$	$[z/\texttt{this},\ \overline{e/\texttt{old}(x)}]\mathsf{post}(m_C)$
$\mathsf{post}(m_C)$	$:=$	the static-contract post-condition of m_C
$\mathsf{body}(\alpha_C)$	$:=$	the body formula of α_C
$\mathsf{body}(\alpha_C(\overline{e}))$	$:=$	$[\overline{e/x}]\mathsf{body}(\alpha_C)$

The footprint function, $\mathsf{required}\,(\cdot)$, generates a formula containing all the permissions necessary to frame its argument. With efficient implementations of a wrapped $\wedge$, this can result in the smallest such formula.

$$
\begin{array}{lcl}
\mathsf{required}\,(e) & := & \text{match } e \text{ with} \\
 & & \left|\begin{array}{lcl}
 e.f & \mapsto & \mathsf{required}\,(e') \;\wedge\; e' \;\texttt{!=}\;\texttt{null} \;\wedge\; \texttt{acc}(e'.f) \\
 e_1 \oplus e_2 & \mapsto & \mathsf{required}\,(e_1) \;\wedge\; \mathsf{required}\,(e_2) \\
 e_1 \odot e_2 & \mapsto & \mathsf{required}\,(e_1) \;\wedge\; \mathsf{required}\,(e_2) \\
 e & \mapsto & \texttt{true}
 \end{array}\right. \\
\mathsf{required}\,(\overline{e}) & := & \bigwedge \mathsf{required}\,(e) \\
\mathsf{required}\,(\phi) & := & \bigwedge \{\mathsf{required}\,(e) : e \text{ appears in } \phi\} \;\wedge \\
 & & \bigwedge \{\alpha_C(\overline{e}) : \texttt{unfolding}\; \alpha_C(\overline{e}) \;\texttt{in}\; \phi' \text{ appears in } \phi\}
\end{array}
$$