Gradually Verified Language with Recursive Predicates

Henry Blanchette

Contents

1 Weakest Liberal Predonditions

1.1 Concrete Weakest Liberal Precondition (WLP) Rules

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

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

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

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 $\mathsf{assert}\ \cdot$ is executed imperitively, raising an error if the argument is false.

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

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

1.2 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:

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

1.2.1 Handling Method Calls

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

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

The following definition reflects the above descriptions, in order:

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

1.2.2 Handling While Loops

The $\mathsf{handleWhileLoop}$ 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:

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

1.2.3 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.

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

1.2.4 Taints

The 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 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:

1.3 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.

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

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.

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

1.4 Gradual Weakest Liberal Precondition (WLP) Rules

$$\widetilde{\mathsf{WLP}} : \textsc{Statement} \times \textsc{FrmSat}\widetilde{\textsc{Formula}} \rightarrow \textsc{FrmSat}\widetilde{\textsc{Formula}}$$

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

$$\left|\begin{array}{lll}
s_1\mathtt{;}\ s_2 & \mapsto & \widetilde{\mathsf{WLP}}(s_1,\ \widetilde{\mathsf{WLP}}(s_2,\ \widetilde{\phi})) \\
y\ \mathtt{:=}\ z.m_C(\overline{e}) & \mapsto & \mathsf{required}\,(\overline{e})\ \wedge\ z\ \mathtt{!=}\ \mathtt{null} \wedge \\
 & & \mathsf{pre}(z.m_C(\overline{e})) * \\
 & & \mathsf{handleMethodCall}(z.m_C(\overline{e}), \phi) \\
\mathtt{if}\ (e)\ \{s_{\text{the}}\}\ \mathtt{else}\ \{s_{\text{els}}\} & \mapsto & \mathsf{required}\,(e)\ \wedge \\
 & & \text{let } \widetilde{\phi}_{\text{the}},\ \widetilde{\phi}_{\text{els}}\ := \widetilde{\mathsf{WLP}}(s_{\text{the}}, \phi),\ \widetilde{\mathsf{WLP}}(s_{\text{els}}, \phi) \text{ in} \\
 & & \text{match } \widetilde{\phi}_{\text{the}},\ \widetilde{\phi}_{\text{els}} \text{ with} \\
 & & \left|\begin{array}{lll}
 \phi_{\text{the}} * ?,\ \phi_{\text{els}} * ? & & \\
 \phi_{\text{the}},\ \phi_{\text{els}} * ? & & \\
 \phi_{\text{the}} * ?,\ \phi_{\text{els}} & \mapsto & (\mathtt{if}\ (e)\ \mathtt{then}\ \phi_{\text{the}}\ \mathtt{else}\ \phi_{\text{els}}) * ? \\
 \phi_{\text{the}},\ \phi_{\text{els}} & \mapsto & \mathtt{if}\ (e)\ \mathtt{then}\ \phi_{\text{the}}\ \mathtt{else}\ \phi_{\text{els}}
 \end{array}\right. \\
\mathtt{while}\ (e)\ \mathtt{invariant}\ \phi_{\text{inv}}\ \{s_{\text{bod}}\} & \mapsto & \mathsf{required}\,(e)\ \wedge\ \phi_{\text{inv}}\ \wedge \\
 & & (\mathtt{if}\ (e)\ \mathtt{then}\ \mathsf{WLP}(s_{\text{bod}}, \phi_{\text{inv}})\ \mathtt{else}\ \mathtt{true}) * \\
 & & \mathsf{handleWhileLoop}(e, \phi_{\text{inv}}) \\
\mathtt{assert}\ \phi_{\text{ass}} & \mapsto & \mathsf{required}\,(\phi_{\text{ass}})\ \wedge\ \phi_{\text{ass}}\ \wedge\ \phi \\
\mathtt{hold}\ \phi_{\text{hol}}\ \{s_{\text{bod}}\} & \mapsto & \text{(unimplemented)} \\
\mathtt{release}\ \phi_{\text{rel}} & \mapsto & \text{(unimplemented)} \\
\mathtt{unfold}\ \alpha_C(\overline{e}) & \mapsto & \mathsf{required}\,(\alpha_C(\overline{e}))\ \wedge \\
 & & [\mathsf{body}(\alpha_C(\overline{e}))/\alpha_C(\overline{e}),\ \phi'/\mathtt{unfolding}\ \alpha_C(\overline{e})\ \mathtt{in}\ \phi']\phi \\
\mathtt{fold}\ \alpha_C(\overline{e}) & \mapsto & \mathsf{required}\,(\overline{e})\ \wedge\ [\alpha_C(\overline{e})/\mathsf{body}(\alpha_C(\overline{e}))]\phi \\
s & \mapsto & \text{match } \widetilde{\phi} \text{ with} \\
 & & \left|\begin{array}{lll}
 \phi * ? & \mapsto & \mathsf{WLP}(s, \phi) * ? \\
 \phi & \mapsto & \mathsf{WLP}(s, \phi)
 \end{array}\right.
\end{array}\right.$$

Note that $*$ binds tighter than $\wedge$.