Gradually Verified Language with Recursive Predicates

Henry Blanchette

Contents

1 Weakest Predonditions

1.1 Concrete Weakest Liberal Precondition (WLP) Rules

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

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

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

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.

Additionally, note the following syntax rules:

- The OCaml-inspired syntax of side-effects and evaluation is defined as follows: A statement of the form $a; s$ is evaluated via "execute a, then evaluate as s".
- The meta-function $\mathsf{assert}\ \cdot$ is executed imperitively, raising an error if the argument is false.

1.2 Handling 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 body of a method call's method is inaccessible due to the (intended) modular structure of verification, so the validity of the method's definition is assumed. Additionally, the method call taints fields and predicates that it required access to, since that allows the method to change their values.
- While loops — the actual execution of a while loop's body is statically unknown since the number of times the while loop's body will exectute is not statically calculated, so references inside the while loop's body are tainted.

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, yields the following:

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

The following definition reflects the above descriptions, in order:

```
$\mathsf{handleMethodCall}(z.m_C(\overline{e}), \phi) :=$
    $\mathsf{assert}\ \mathsf{granted}(\mathsf{post}(z.m_C(\overline{e}))),$
      $\forall \pi$ such that $\mathsf{required}(\phi) \implies \pi \ \wedge\ \mathsf{granted}(\mathsf{pre}(z.m_C(\overline{e}))) \implies \pi);$
    $\mathsf{assume}\ [\mathsf{tainted}_{\mathsf{uid}(z.m_C(\overline{e}))}(e.f)/e.f,\ \mathsf{tainted}_{\mathsf{uid}(z.m_C(\overline{e}))}(\alpha_C(e.f))]\phi;$
    $[\mathsf{tainted}_{\mathsf{uid}(z.m_C(\overline{e}))}(e.f)/e.f : \mathsf{granted}(\mathsf{pre}(z.m_C(\overline{e}))) \implies \mathsf{acc}(e.f),$
     $\mathsf{tainted}_{\mathsf{uid}(z.m_C(\overline{e}))}(\alpha_C(\overline{e}))/\alpha_C(\overline{e}) : \mathsf{granted}(\mathsf{pre}(z.m_C(\overline{e}))) \implies \alpha_C(\overline{e})]\phi$
```

1.2.2 Handling While Loops

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

- assume taint-substututed ϕ_{inv}
- all references that may be set (but not defined) inside the while loop's body are tainted inside ϕ.

The following definition reflects the above descriptions, in order:

$$\begin{array}{l} \mathsf{handleWhileLoop}(z.m_C(\overline{e}), s_{\text{bod}}, \phi) := \\ \quad \mathsf{assume}\ [\mathsf{tainted}_{\mathsf{uid}(\texttt{while}\ e\ \texttt{invariant}\ \phi_{\text{inv}})}(e.f)/e.f : e.f := e' \text{ appears in } s_{\text{bod}}]\phi_{\text{inv}}; \\ \quad [\mathsf{tainted}_{\mathsf{uid}(\texttt{while}\ e\ \texttt{invariant}\ \phi_{\text{inv}})}(e.f)/e.f : e.f := e' \text{ appears in } s_{\text{bod}}]\phi \end{array}$$

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 formula has been *tainted* by a source identified by the given unique id. A *tainted* formula 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 formulas can only be asserted in some specific ways. For example, the method call could ensure that $x.f = v$, where v is some value, and this would yield the *assumption* that $\mathsf{tainted}_{\mathsf{uid}(z.m_C(\overline{e}))}(x.f) = v$.

$$\mathsf{tainted}_{uid}(\phi) := \text{ wrapped } \phi\text{, labeled with } uid.$$

The $\mathsf{uid}(\cdot)$ function generates a unique id for the given instance. This is needed because different instances of method calls and the like may be of the same method and be given the same arguments.

$\mathsf{uid}(z.m_C(\overline{e})) :=$ a unique id generated for the given instance of the method call $z.m_C(\overline{e})$.

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.

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

The footprint function, $\lfloor\cdot\rfloor$, 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.

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