# Encoding.

We demonstrate the principles of how our invariant selection method work by giving a corresponding encoding for a simple imperative language IMP.

## 1 IMP structure.

We denote $(x_1, ..., x_m)$, $(y_1, ..., y_n)$ vectors of variable names and $(e_1, ..., e_k)$ vector of expressions. The language structure is as follows.

Procedures:
$proc := procedure\ m(x_1, ..., x_m)\ returns(y_1, ..., y_n)\{s\}$

Statements:
$s := skip \mid x := e \mid$
$\quad (x_1, ..., x_m) := call\ m(e_1, ..., e_k) \mid$
$\quad s; s \mid if(b)\ then\ s\ else\ s \mid while(e)\ do\{s\} \mid$
$\quad assume\ e \mid assert\ e \mid x := havoc$

Expressions:
For $c \in \mathbb{Z}$ (or $c \in \mathbb{R}$), $\oplus = \{+, -, *, /, \%, ==>, <==, <==>, \vee, \wedge, \neg\}$
$e := c \mid x \mid e \oplus e$

## 2 Encoding.

Let $I$ be the candidate invariant set.
Let $k := |I|$.
Let $I'$ be the ordered set $I$, i.e. $I' = (inv_1, ..., inv_k)$ such that $\forall inv \in I'.inv \in I$ and $\forall inv \in I.inv \in I'$.

$[[\bullet]]_I$ denotes the transformation function, which translates its inputs to the target encoding taking into account the candidate invariant set.

$[[\bullet]]_{I\_L}(targets)$ denotes the transformation function, which translates its inputs to the target encoding inside a loop taking into account the candidate invariant set. It takes a set of target variable names as its argument.

The difference between $[[\bullet]]_I$ and $[[\bullet]]_{I\_L}(targets)$ is that the latter does not duplicate the statements. I.e. if some loop target variables are used in a statement, $[[\bullet]]_{I\_L}(targets)$ replaces this variable names with corresponding duplicate variable names, whereas $[[\bullet]]_I$ retains the original statement and adds another one with replaced variable names.

In case of nested loop statements $[[\bullet]]_{I\_L}(targets)$ is also supposed to provide a differentiation between loops of different levels with the help of the unique loop identifier, passed as an argument.

Let *proc* be the method under transformation.

Let $loop := while(e)\ do\{s\}$.

Let *get_id*(*loop*) be the function, which returns the unique identifier of the *loop*, given as an argument.

Let *get_nested_id*(*outer_loop*) be the function, which returns the set of identifiers ot the loops inside the *outer_loop*.

We use *var_name_id*, where *id* should be replaced by an actual loop identifier for the variable names, which depends on the loop they are used in.

Let $T(loop)$ be the function, which gives a set of names of all targets of a given loop.

Let $locals := \cup_{loop \in proc} T(loop)$.

Let $vars := \{x_1, x_2, ...\}$, where $x_1, x_2, ...$ are variable names.

Let $fresh(vars)$ denote that variable names $x_1, x_2, ...$ are fresh, i.e. do not occur in the procedure under transformation.

Let $M(vars) := vars'$ be a bijective function, such that $fresh(vars')$ holds.

Let $locals' := M(locals)$.

Let $\odot_{i=1}^{n} s_i$ be the sequential composition of $n$ statements $s_1, ..., s_n$.

Let $xs = (x_1, ..., x_m)$ be the vector of parameter names, $ys = (y_1, ..., y_n)$ be the vector of return variable names.

Let $xs\_s = \{x_1, ..., x_m\}$ and $ys\_s = \{y_1, ..., y_n\}$.

$e[set_1/set_2]$ means we replace the variable names in $set_1$ with variable names in $set_2$ in expression $e$.

We further make the following assumptions.

Input variables are immutable.

$$\begin{array}{c} call \text{ m}(e_1, ..., e_n) \equiv \\ assume\ [preconditions] \\ \text{[method body]} \\ assume\ [postconditions] \end{array}$$

We assume that expressions cannot change the program state. I.e. $x := 1$, for example, is not an expression.

Together with our interpretation of the *call* statement, this implies, that there is no need to transform the *call* statement.

The translation looks as follows.

## 2.1 Procedure.

Since we define our encoding on the granularity of statements, i.e. procedure cannot be inside a loop, only $[[\bullet]]_I$ is defined for the procedure.

$$\begin{array}{l} [[procedure\ m(xs)\ returns(ys)\{s\}]]_I = \\ \quad procedure\ m(xs) returns(ys)\{\odot_{x' \in locals'} x' := havoc\ ;\ [[s]]_I\} \end{array}$$

Same names for duplicated target variables are used through the whole procedure, so we declare them right at the beginning using $x' := havoc$ statement.

## 2.2 Skip.

$$[[skip]]_I = skip$$

$$[[skip]]_{I\_L}(targets) = skip$$

Since *skip* statement does not affect the program state, our transformation functions do nothing in this case.

## 2.3 Sequential composition.

$$[[s;s]]_I = [[s]]_I \ ; \ [[s]]_I$$

$$[[s;s]]_{I\_L}(targets) = [[s]]_{I\_L}(targets) \ ; \ [[s]]_{I\_L}(targets)$$

## 2.4 If.

$$[[if(e) \ then \ s \ else \ s]]_I = if(e) \ then \ [[s]]_I \ else \ [[s]]_I$$

$$\begin{aligned}&[[if(e) \ then \ s \ else \ s]]_{I\_L}(targets) = \\ &if(e[vars/targets]) \ then \ [[s]]_{I\_L}(targets) \ else \ [[s]]_{I\_L}(targets) \ , \\ &\qquad\qquad\qquad\qquad \text{for } vars \text{ such that } targets = M[vars]\end{aligned}$$

## 2.5 Assume.

$$[[assume \ e]]_I = assume \ e$$

$$\begin{aligned}&[[assume \ e]]_{I\_L}(targets) = assume \ e[vars/targets] \ , \\ &\qquad\qquad \text{for } vars \text{ such that } targets = M[vars]\end{aligned}$$

## 2.6 Assert.

$$[[assert \ e]]_I = assert \ e$$

$$[[assert \ e]]_{I\_L}(targets) = skip$$

In case of $[[\bullet]]_{I\_L}(targets)$ function we cannot transform the statement into assertion, since it may rely on invariants, yet have to be proven. We also cannot use *assume* statement instead, since it may allow us to prove something, which we should not be able to prove. E.g. is we have *assert false* statement.

## 2.7 Havoc.

$$[[x := havoc]]_I = x := havoc$$

$$\begin{aligned}&[[x := havoc]]_{I\_L}(targets) = x' := havoc \ , \\ &\qquad\qquad \text{for } x' = M(\{x\})\end{aligned}$$

## 2.8 Assignment.

$$[[x := e]]_I = x := e$$

$$[[x := e]]_{I\_L}(targets) = x' := e[vars/targets] \ ,$$
$$\text{for } x' = M(\{x\}) \text{ and } vars \text{ such that } targets = M(vars)$$

## 2.9 Method Call.

$$[[(x_1, ..., x_m) := call\ m(e_1, ..., e_k)]]_I = (x_1, ..., x_m) := call\ m(e_1, ..., e_k)$$

$$[[(x_1, ..., x_m) := call\ m(e_1, ..., e_k)]]_{I\_L}(targets) =$$
$$(x'_1, ..., x'_m) := call\ m(e_1[vars/targets], ..., e_k[vars/targets]) \ ,$$
$$\text{for } \{x'_1, ..., x'_m\} = M(\{x_1, ..., x_m\}) \text{ and } vars \text{ such that } targets = M(vars)$$

## 2.10 While.

We first define the transformation function $[[\bullet]]_{N\_L}$, which takes IMP statements as arguments. For each statement, except the $while(e)\ do\{s\}$, it is an identity transformation. For $while(e)\ do\{s\}$ it is defined as follows.

$$[[while(e)\ do\{s\}]]_{N\_L} =$$
$$id := get\_id(while(e)\ do\{s\});$$
$$star\_id := havoc;$$
$$\bigodot_{x \in T(while(e)\ do\{s\})} x := havoc \ ;$$
$$\bigodot_{i=1, inv_j \in I'}^{k} assume\ on\_id_i ==> inv_i \ ;$$
$$if(star\_id)\ then$$

// we need this part because of possible assert statements,

// which were replaced by skip in the simulation

$$assume\ e \ ;$$
$$[[s]]_{N\_L} \ ;$$
$$assume\ false \ ;$$
$$else$$
$$assume\ \neg e \ ;$$

where $fresh(star\_id)$ and $(on\_id_1, ..., on\_id_k)$ should be declared earlier

$$[[while(e)\ do\{s\}]]_{I\_L}(targets) =$$

$$id := get\_id(while(e)\ do\{s\})\ ;$$

// check, whether an invariant holds before the loop

$$\odot_{i=1,inv_j\in I'}^{k}\ assume\ inv_i[targets/M(targets)] <==> on\_b\_id_i\ ;$$

// check the inner loop condition

$$lc\_id := havoc\ ;$$

$$assume\ e[targets/M(targets)] <==> lc\_id$$

// simulate an arbitrary iteration

$$\odot_{x'=M(x),x\in T(while(e)\ do\{s\})}\ x' := havoc\ ;$$

$$\odot_{i=1,inv_j\in I'}^{k}\ assume\ on\_a\_id_i ==> inv_i[targets/M(targets)]\ ;$$

// restrictions on invariant values due to invariants, which hold

$$\odot_{i=1,inv_j\in I'}^{k}\ assume\ on\_b\_id_i ==> inv_i[targets/M(targets)]\ ;$$

// we only simulate the inner loop, if we can ever enter it

$$if(lc\_id)\ then$$

$$assume\ e[targets/M(targets)]\ ;$$

// transformed loop body

$$[[s]]_{I\_L}(targets)\ ;$$

// infer, which invariants hold

$$\odot_{i=1,inv_j\in I'}^{k}\ assume\ (on\_b\_id_i \wedge (on\_a\_id_i ==> inv_i[targets/M(targets)])) ==> on\_id_i\ ;$$

$$else$$

// we still have to set inavriant flags

//in order to restrict the havoced variable values while verifying the original loop

$$\odot_{i=1,inv_j\in I'}^{k}\ on\_b\_id_i <==> on\_id_i\ ;$$

// we are out of the loop here

$$assume\ \neg e\ ;$$

where $fresh(id)$, $fresh(lc\_id)$ and

$(on\_id_1, ..., on\_id_k)$, $(on\_b\_id_1, ..., on\_b\_id_k)$ should be declared earlier

$[[while(e)\ do\{s\}]]_I =$ // boolean variables to infer, whether an invariant holds

$$\odot_{i=1}^{k}\ on_i := havoc\ ;$$
$$\odot_{i=1}^{k}\ on\_b_i := havoc\ ;$$
$$\odot_{i=1}^{k}\ on\_a_i := havoc\ ;$$

// boolean variables to infer, which invariants hold in nested loops

$$\odot_{id \in get\_nested\_id(while(e)\ do\{s\})}\ \odot_{i=1}^{k} on\_id_i := havoc\ ;$$
$$\odot_{id \in get\_nested\_id(while(e)\ do\{s\})}\ \odot_{i=1}^{k} on\_b\_id_i := havoc\ ;$$
$$\odot_{id \in get\_nested\_id(while(e)\ do\{s\})}\ \odot_{i=1}^{k} on\_a\_id_i := havoc\ ;$$

// check, whether an invariant holds before the loop

// no variable names replacement necessary here

$$\odot_{i=1, inv_j \in I'}^{k}\ assume\ inv_i <==> on\_b_i\ ;$$

// simulate an arbitrary iteration

$$\odot_{x'=M(x), x \in T(while(e)\ do\{s\})}\ x' := havoc\ ;$$
$$\odot_{i=1, inv_j \in I'}^{k}\ assume\ on\_a_i ==> inv_i[T(while(e)\ do\{s\})/M(T(while(e)\ do\{s\}))]\ ;$$

// restrictions on variable values due to invariants, which hold

$$\odot_{i=1, inv_j \in I'}^{k}\ assume\ on\_b_i ==> inv_i[T(while(e)\ do\{s\})/M(T(while(e)\ do\{s\}))]\ ;$$

// we only simulate the outer loop, if we can ever enter it

// we do not need the duplicates in the if clause for the outer loop

$$if(e)\ then$$
$$assume\ e[T(while(e)\ do\{s\})/M(T(while(e)\ do\{s\}))]\ ;$$

// transformed loop body

$$[[s]]_{I\_L}(T(while(e)\ do\{s\}))\ ;$$

// infer, which invariants hold

$$\odot_{i=1, inv_j \in I'}^{k}\ assume\ (on\_b_i \wedge (on\_a_i ==> inv_i[T(while(e)\ do\{s\})/M(T(while(e)\ do\{s\}))])) ==> on_i\ ;$$

$$else$$

// we still have to set inavriant flags

//in order to restrict the havoced variable values while verifying the original loop

$$\odot_{i=1, inv_j \in I'}^{k}\ on\_b_i <==> inv_i\ ;$$

// actual loop

$star := havoc;$

$\bigodot_{x \in T(while(e)\ do\{s\})} x := havoc\ ;$

$\bigodot_{i=1, inv_j \in I'}^{k} assume\ on_i ==> inv_i\ ;$

$if(star)\ then$

// we need this part because of possible assert statements,

// which were replaced by skip in the simulation

$assume\ e\ ;$

$[[s]]_{N\_L}\ ;$

$assume\ false\ ;$

$else$

$assume\ \neg e\ ;$

$$where\ fresh(on_1, ..., on_k) \wedge fresh(on\_b_1, ..., on\_b_k) \wedge fresh(on\_a_1, ..., on\_a_k) \wedge$$
$$\bigwedge_{id \in get\_nested\_id(while(e)\ do\{s\})} fresh(on\_id_1, ..., on\_id_k) \wedge$$
$$\bigwedge_{id \in get\_nested\_id(while(e)\ do\{s\})} fresh(on\_b\_id_1, ..., on\_b\_id_k) \wedge$$
$$\bigwedge_{id \in get\_nested\_id(while(e)\ do\{s\})} fresh(on\_a\_id_1, ..., on\_a\_id_k) \wedge$$
$$fresh(star)$$