Verification of Recursive Programs

Andreas Podelski
February 8, 2012
\[ m(x) = \begin{cases} 
  x - 10 & \text{if } x > 100 \\
  m(m(x + 11)) & \text{if } x \leq 100 
\end{cases} \]
procedure m(x) returns (res)

\[\begin{align*}
\ell_0: & \quad \text{if } x > 100 \\
\ell_1: & \quad \text{res} := x - 10 \\
\text{else} & \\
\ell_2: & \quad x_m := x + 11 \\
\ell_3: & \quad \text{res}_m := \text{call } m(x_m) \\
\ell_4: & \quad x_m := \text{res}_m \\
\ell_5: & \quad \text{res}_m := \text{call } m(x_m) \\
\ell_6: & \quad \text{res} := \text{res}_m \\
\ell_7: & \quad \text{assert } (x \leq 101 \rightarrow \text{res} = 91)
\end{align*}\]

return m
restrictions and conventions

- no pointers, no global variables and only call-by-value procedure calls
- forbidden to write to the input variables of a procedure
- procedure \( p \) has one input variable \( x \) ("formal argument") and one output variable \( res \)
- procedure calls appear in the following form:
  \[ res_p := \text{call } p(x_p) \]
- actual argument (actual parameter) is always \( x_p \) (special variable)
- on return, value of output variable \( res \) is stored into variable \( res_p \) of the caller
procedure m(x) returns (res)

ℓ0: if x>100

ℓ1: res:=x-10

else

ℓ2: x_m := x+11

ℓ3: res_m := call m(x_m)

ℓ4: x_m := res_m

ℓ5: res_m := call m(x_m)

ℓ6: res := res_m

ℓ7: assert (x<=101 -> res=91)

return m
procedure m(x) returns (res)

$l_0$: if x>100

$l_1$: res:=x-10

else

$l_2$: $x_m := x+11$

$l_3$: call m

$l_4$: $x_m := res_m$

$l_5$: call m

$l_6$: res := res$_m$

$l_7$: assert (x<=$101 \Rightarrow res=91$)
return m
impose a number of restrictions and conventions in order to simulate a programming language in a BoogiePL-like syntax. As usual, we do not necessarily inductively present.

2. Preliminaries

2.1 Recursive Programs

Informal Presentation. To summarize, the first method does not account for recursion. The second method, as used, e.g., in generating a sequence of interpolants from the spurious error trace of a recursive program. The interpolants are used.

For each procedure, a control flow graph is constructed. The edges between internal nodes are labeled with statements. Each node is a procedure location. Before calling procedure, its control flow graph is called from procedure. To implement McCarthy’s function together with a Floyd-Hoare assertion, the initial location is called from procedure. For each procedure, its control flow graph is implemented.

Formal Presentation: Recursive Control Flow Graph.

The node $p$'s control flow graph is implemented. The edges between internal nodes are labeled with statements. Each node is a procedure location. Before calling procedure, its control flow graph is called from procedure. To implement McCarthy’s function together with a Floyd-Hoare assertion, the initial location is called from procedure. For each procedure, its control flow graph is implemented.

Each node is a procedure location. Before calling procedure, its control flow graph is called from procedure. To implement McCarthy’s function together with a Floyd-Hoare assertion, the initial location is called from procedure. For each procedure, its control flow graph is implemented.

The result value is returned to the caller through the variable. The variable is either an assignment or a return statement. Each node is a procedure location. Before calling procedure, its control flow graph is called from procedure. To implement McCarthy’s function together with a Floyd-Hoare assertion, the initial location is called from procedure. For each procedure, its control flow graph is implemented.

Each node is a procedure location. Before calling procedure, its control flow graph is called from procedure. To implement McCarthy’s function together with a Floyd-Hoare assertion, the initial location is called from procedure. For each procedure, its control flow graph is implemented.

Each node is a procedure location. Before calling procedure, its control flow graph is called from procedure. To implement McCarthy’s function together with a Floyd-Hoare assertion, the initial location is called from procedure. For each procedure, its control flow graph is implemented.
four kinds of statements

- **call statement** \(\text{call } p\)
  creates a new calling context where (unique) input variable \(x\) gets assigned value of variable \(x_p\) from previous calling context 
  new calling context is added on top of stack

- **return statement** \(\text{return } p\)
  result value \(r_s\) is taken from the current calling context 
  then current calling context is removed from stack ("pop")
  result value \(r_s\) is written to variable \(r_s_p\) (in context of caller)

- **assignment statement** \(y := t\)
  value of term \(t\) is assigned to variable \(y\) in top-most calling context

- **assume statement** \(\phi\)
  only executed if variable valuation of top-most calling context satisfies Boolean expression \(\phi\)
To model that procedure $p$ is called from procedure $q$ at program location $\ell_j^q$ (in $q$’s control flow graph),

- an edge labeled $\text{call } p$ goes from $\ell_j^q$ to $\ell_0^p$, the entry location of $p$,
- an edge labeled $\text{return } p \uparrow \ell_j^q$ goes from $\ell_n^p$, the exit location of $p$, to the location $\ell_j^q$ + 1 in $q$’s control flow graph, where $\ell_j^q$ + 1 is the successor location of $\ell_j^q$ with respect to the call of procedure $p$. 

semantics
• *valuation* \( \nu = \) function that maps program variables to values
• (local) state of a procedure = pair \((\ell, \nu)\) of program location and valuation
• (global) state \(S\) of the program = stack of local states
  \[
  S = (\ell_0, \nu_0). (\ell_1, \nu_1) \ldots (\ell_n, \nu_n)
  \]
• stack element = called (and not yet returned) procedure
• topmost/rightmost element represents current calling context
<table>
<thead>
<tr>
<th>state</th>
<th>label of edge ((\ell, \ell'))</th>
<th>successor state</th>
<th>side condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S(\ell, \nu))</td>
<td>(\text{y:=t} )</td>
<td>(S(\ell', \nu'))</td>
<td>(\nu' = \nu \oplus {y \mapsto \nu(t)})</td>
</tr>
<tr>
<td>(S(\ell, \nu))</td>
<td>(\phi)</td>
<td>(S(\ell', \nu))</td>
<td>(\nu \models \phi)</td>
</tr>
<tr>
<td>(S(\ell, \nu))</td>
<td>(\text{call } p)</td>
<td>(S(\ell, \nu).(\ell', \nu'))</td>
<td>(\nu'(x) = \nu(x_p))</td>
</tr>
<tr>
<td>(S(\ell_&gt;, \nu_&gt;.)(\ell, \nu))</td>
<td>(\text{return } p \uparrow \ell_&gt;)</td>
<td>(S(\ell', \nu'))</td>
<td>(\nu' = \nu_&gt; \oplus {\text{res}_p \mapsto \nu(\text{res})})</td>
</tr>
</tbody>
</table>
transition from state \( S \) to successor state \( S' \) under statement \( \tau \)
\[
S \xrightarrow{\tau} S'
\]

transition under sequence of statements \( \pi = \tau_0 \ldots \tau_{n-1} \)
\[
S \xrightarrow{\pi} S'
\]
Hoare rule for recursion

\[
\begin{align*}
\{ \phi(x, res) \} & \text{ body}_p \ \{ \theta(x, res) \} \\
\{ \phi(x_p, res_p) \} & \text{ res}_p := p(x_p) \ \{ \exists res_p. \phi(x_p, res_p) \land \theta(x_p, res_p) \} 
\end{align*}
\]
procedure m(x) returns (res)
if x>100
    res := x-10
else
    x_m := x + 11
    res_m := call m(x_m)
    x_m := res_m
    res_m := call m(x_m)
    res := res_m
procedure m(x) returns (res)
{true}
if x>100
{x ≥ 101}
res:=x-10
else
{x ≤ 100}
x_m := x+11
{x_m ≤ 111}
res_m :=call m(x_m)
{res_m ≤ 101}
x_m := res_m
{x_m ≤ 101}
res_m :=call m(x_m)
{res_m = 91}
res := res_m
{res = 91 ∨ (x ≥ 101 ∧ res = x - 10)}
Hoare rule for recursion

\[
\{\phi(x, res)\} \text{ body}_p \{\theta(x, res)\}
\]

\[
\{\phi(x_p, res_p)\} \text{ res}_p := p(x_p) \{\exists res_p. \phi(x_p, res_p) \land \theta(x_p, res_p)\}
\]
procedure m(x) returns (res)
{\top}:
\l_0: \text{if } x > 100
\{x \geq 101\}
\l_1: \text{res} := x - 10
\text{else}
\text{res} := x - 10
\l_2: x_m := x + 11
\{x_m \leq 111\}
\l_3: \text{call m}
\{res_m \leq 101\}
\l_4: x_m := res_m
\{x_m \leq 101\}
\l_5: \text{call m}
\{res_m = 91\}
\l_6: \text{res} := res_m
\{res = 91 \lor (x \geq 101 \land res = x - 10)\}
\l_7: \text{assert } (x \leq 101 \rightarrow res = 91)