A Functional Scenario for Bytecode Verification of Space Bounds

Roberto Amadio, Solange Coupet-Grimal, Silvano Dal Zilio and Line Jakubiec
LIF, Marseille (Fr)

Appeared as Research Report LIF 19-2004, January 2004
A First-Order Functional Language

- With algebraic types, e.g. the type of natural numbers is \( \text{nat} ::= \text{z} \mid \text{s} \text{ of } \text{nat} \)
- Functions and pattern-matching, e.g.
  \[
  \text{add} : (\text{nat}, \text{nat}) \to \text{nat} :
  \]
  \[
  \begin{align*}
  \text{add} \ z \ y & = y \\
  \text{add} \ s(x) \ y & = \text{add} x s(y)
  \end{align*}
  \]
- Evaluation: \( \text{add} s(s(z)) \ s(z) \Rightarrow s(s(s(z))) \)
- Compiled into bytecode instructions for a (simple) stack machine
1. **load** 1

   ![Frame 1](image1.png)

2. branch **s** 7

   ![Frame 2](image2.png)

3. load 2

   ![Frame 3](image3.png)

4. build **s**

   ![Frame 4](image4.png)

5. call **add** 2

   ![Frame 5](image5.png)

6. return

   ![Frame 6](image6.png)

7. load 2

   ![Frame 7](image7.png)

8. return

   ![Frame 8](image8.png)

**program counter**

**frame**

**returned result**
Bounding the Space Needed

- It is easy to obtain a bound on the number of values in a frame → type verification
- We need to bound the size of the values → size verification, based on quasi-interpretations.
- We need to bound the number of frames in an execution path → termination verification, based on r.p.o. (recursive path orderings)
<table>
<thead>
<tr>
<th>Step</th>
<th>Opcode</th>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>nat</td>
<td>nat</td>
</tr>
<tr>
<td>2</td>
<td>branch</td>
<td>s</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>load</td>
<td>2</td>
<td>nat</td>
</tr>
<tr>
<td>4</td>
<td>build</td>
<td>s</td>
<td>nat</td>
</tr>
<tr>
<td>5</td>
<td>call</td>
<td>add</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>return</td>
<td></td>
<td>nat</td>
</tr>
<tr>
<td>7</td>
<td>load</td>
<td>2</td>
<td>nat</td>
</tr>
<tr>
<td>8</td>
<td>return</td>
<td></td>
<td>nat</td>
</tr>
</tbody>
</table>
1. load 1
   \[
   x_1, x_2
   \]
2. branch \texttt{s} 7
   \[
   x_1, x_2, x_1
   \]
3. load 2
   \[
   s(x_3), x_2, x_3
   \]
   \[x_1 = s(x_3)\]
4. build \texttt{s}
   \[
   s(x_3), x_2, x_3, x_2
   \]
5. call \texttt{add} 2
   \[
   s(x_3), x_2, x_3, s(x_2)
   \]
6. return
   \[
   s(x_3), x_2, \text{add}(x_3, s(x_2))
   \]
7. load 2
   \[
   x_1, x_2, x_1
   \]
8. return
   \[
   x_1, x_2, x_1, x_2
   \]
Quasi-Interpretations

• The polynomial $q_{\text{add}}(x, y) = x + y$ is a valid quasi-interpretation for the function $\text{add}$.

• We can check that size information are correct (for “compiled programs”). In our example it amounts to check that:

$$q_{\text{add}}(1 + x_3, x_2) \geq q_{\text{add}}(x_3, 1 + x_2)$$
Termination

• We use termination criteria based on *recursive path ordering*.

• We can check that termination information are correct. In our example it amounts to check that:

$$\text{add}(s(x_3), x_2) >_1 \text{add}(x_3, s(x_2))$$
Result

• A combination of polynomial quasi-interpretation and r.p.o. gives a (explicit!) polynomial upper-bound on the size needed for the execution.

• In our example, in an execution starting with the frame (add, 1, x₁ x₂):
  – a stack in a frame has size at most 4
  – every value has size less than x₁ + x₂
  – the number of frames is less than x₁

  
  size needed ≤ 4 . x₁ . (x₁ + x₂)
%%% type %%%
type nat = Z | S of nat
fun nat add(nat,nat)

%%% size %%%
q_Z = 0
q_S = #1 + 1
q_add = #1 + #2

%%% termination %%%
exp, double > add

%%% code %%%
1: load 1
2: branch S 7
3: load 2
4: build S
5: call add 2
6: return
7: load 2
8: return