

TID

Processing and analysis of robotic arm control language

Author:

Radim Luža

Structure of presentation

- Part 1: Design of the interpreter
 - The Robotic arm control language (LUA, MELFA BASIC).
 - Interpreter components and formalisms.
- Part 2: The language analysis
 - Goals of the analysis.
 - Analysis technique I used.
 - A prototype of the analyser
 - Difficulties of a LUA programming language analysis.

Part 1

Design of the interpreter

The Robotic Arm Control Language (RACL)

- An interpreted language to control the a robotic arm, a robotic manipulator and cameras.
- Compound of two languages – a LUA and a MELFA-BASIC.
 - LUA – an open-source weakly typed scripting language – common control-flow constructions, mathematical expressions, IO, supports objects ^[1].
 - MELFA-BASIC – MELFA proprietary language with a BASIC-like syntax – used as a low level language to control the arm ^[2].

Interpreter components

- The preprocessor.
- The LUA interpreter.
 - Available as OSS.
- The MELFA-BASIC interpreter.
 - Built in an arm controller or in an arm simulator.
 - Significantly simplified.
 - Limited to communication with the arm (MOVS).
 - No flow control or mathematical expressions (IF THE, operator +) of the MELFA BASIC are being used.
 - Sequence of single-line commands.

Interpreter components 2

- The LUA interpreter.
 - Third-party library to interpret LUA - used "as it is" - just wrapped into RACL interpreter code.
- The MELFA-BASIC interpreter.
 - Only splits the code into single commands that are then passed to simulator or to arm.
 - Simplified MELFA-BASIC accepted by FSM:
 - $(\text{COMMAND} \setminus n^+)^* \text{COMMAND} \setminus n^* [+ \epsilon]$
 - COMMAND is a set of commands with shape of a symbolic instruction: MOVJ 10 X1 120

Interpreter components 3

- The preprocessor
 - Main purpose - combination of MB and LUA.

Source:

```
VAR = 60
#MELFA_BASIC_BEGIN
MOVJ 10 @VAR1 20
@VAR2 :- PRINT M_SRV
MOVJ 30 @VAR1 50
#MELFA_BASIC_END
```

Preprocessed – pure LUA:

```
VAR = 60
INP_VALS["VAR1"]=VAR1 --input as a value
OUT_VARS["VAR2"]="VAR2" --output as a variable reference
OTHER_MBCall("MOVJ 10 @I["VAR1"] 20\n@O["VAR2"] :- PRINT
M_SRV\nMOVJ 30 @I["VAR1"] 50", INP_VALS,OUT_VARS)
```

Preprocessor complexity

- More complex than a Finite State Transducer.
 - Intuitive proof: FSM can't keep infinite strings.
- More complex than a Pushdown Transducer.
 - Intuitive proof: A stack can store infinite number of infinite strings but it doesn't allow to access them randomly to check appearance of variable name in a set of "remembered" variables.
- Translation can be computed by LBA (LOA).
 - Length of a list of "remembered" names and length of a generated output code is linear dependent on length of input.

Part 2

The language analysis

Goals of analysis

- Main goal: Detection of never-ending programs.
 - Undecidable → only subgoals are being analysed.
 - *(Proof: diagonalization of matrix of binary coded Turing machines and binary coded input strings.)*
- Analysable subgoals of main goal.
 - Detection of potentially infinite loops.
 - Detection of potentially infinite recursion.
- Other notes about analysis.
 - Only single-thread code.
 - Only LUA needs to be analysed.

Potentially infinite looping

- If it starts to loop it will never end.
- We can't decide easily if looping will start.
 - Loop (or recursion) might be in a conditional branch depending on unknown program inputs.
 - Condition of looping might be not satisfied.
 - Probably the easiest way of deciding if looping will start is to execute the program.

Analysis technique I used

- Syntax analysis.
 - Rejects the code with syntactic errors and constructs AST.
 - Used third-party analyser LuaFish.
- Control-flow analysis ^[3].
 - Construction of control-flow graph from AST.
 - Finding loops in control-flow graph.
- Data-flow analysis ^[4].
 - Analysis of assignments to variables and their appearance in cond-branch expressions only.

Current analyzer prototype

- Converts AST generated by LuaFish to CF.
- Gathers informations about variable assignments and about appearance of variables in conditional branch control expressions.
- Only loop analysis.
- Recursion analysis is not supported yet.
- Objective code is not supported yet.

Example of analysis

- Source code

```
a=0
```

```
for i= 3,30,3 do
```

```
  while a < 10 do  --a is not modified in the loop body
```

```
    b = b + 1
```

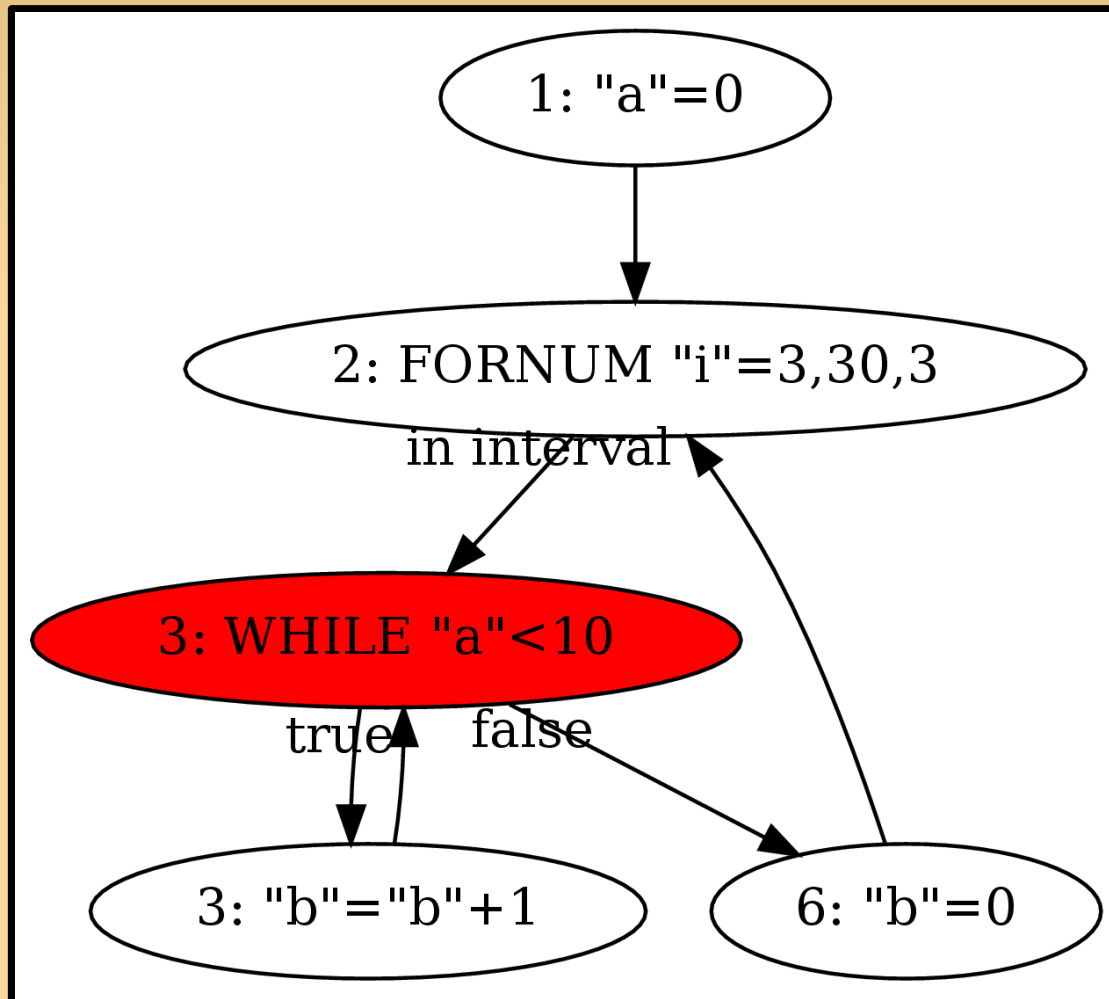
```
  end
```

```
  b = 0
```

```
end
```

Example of analysis

- CF graph with highlighted analysis result



Difficulties of LUA analysis

- Variables without prior definition.
 - They have a default value – nil.
 - nil is not like a NULL pointer – it can be casted to number, boolean or string.
 - Variables with all possible names "exist".
- Variables are global by default.

```
if x > 10 then
  a = 5
end
--a has value 5 here
```


Difficulties of LUA analysis

- Objective code.
 - Assigning correct data to object methods.
- Multiassignments.
`a,b,c = x,u --c is set to nil`
 - We have to check that all variables have right-side value. Otherwise they will be set to *nil*.

References

- [1] Ierusalimschy R. , Figueiredo L. H., Celes W.: Lua 5.1 Reference Manual, online <<http://www.lua.org/manual/5.1/>>, August 2006, cit. Dec. 2011.
- [2] Guerrero J.: COSIMIR MELFA BASIC IV, online <<http://dmi.uib.es/~jguerrero/instMelfa.pdf>>, September 2004, cit. Dec. 2011.
- [3] Kolektiv: Control flow analysis, online <http://en.wikipedia.org/wiki/Control_flow_analysis>, May 2011, cit. Dec.2011.
- [4] Nielson F, Nielson H., Hankin Ch.: Principles of Program Analysis: Data Flow Analysis, online <<http://www2.imm.dtu.dk/~riis/PPA/slides2.pdf>>, 2005, cit. Dec.2011.
- And others...

The End

Thank you for your attention.