Proof-Carrying Code

Basados en la ejercitación de la materia
Language-Based Security de *The Technical University of Denmark*

Octubre de 2014

1 Hoare Logic

```
y:=0;
while (y*y<=x){
  y:=y+1;
}
y:=y-1;
```

- What does this program compute?
- How could that be expressed by pre/post-conditions?
- What would be a suitable invariant for the `while`-loop?
- Try to prove it, starting from the post-condition. Note: you may arrive at a slightly stronger pre-condition than desired. What could be done about this?

The rule for `while` is defined as follows:

$$\frac{(\!| I \wedge C |\!)\ P\ (\!| I |\!)}{(\!| I |\!)\ \texttt{while(C)\{P\}}\ (\!| I \wedge \neg C |\!)}$$

2 Project Description

2.1 General Description

Describe in general terms the problem studied in this part of the course and the approach taken to solve it. This should include a discussion what type of policies can be enforced.

2.2 A Language for PCC

2.2.1 Generating a Verification Condition

Based on the language and the example program given in Section 3 do the following:

- Describe what function the exercise program performs.
- Compute the verification condition for the program given in Figure 5, starting with the post condition $post = \texttt{true}$.
- Suppose the example program should only accept *ordered* lists as input. How would you impose that? Give the specification of your addition as well as the changed program including the precondition and invariants.

2.2.2 Language Extension [optativo]

Extend the language presented in Section 3 with some (meaning at least one) proof obligation(s) that a code consumer could impose. Specify the verification condition generator and show verification conditions for at least two (meaningful) input programs for each of the obligations.

Here you could either extend the verification condition generator by adding a new statement or change the semantics, or you could give a new program that imposes other preconditions than the original one or the one from question 2.2.1.

2.3 Proof-carrying Architectures

In this part, we are going to turn the traditional PCC scenario around. In traditional PCC systems, the exchanged messages are security policies, code, and proofs. This part instead targets a concept we call *proof-carrying*

architectures (PCAs). The exchanged messages are security policies, proofs, and *data*. Instead of a code producer in this system we have a *data provider*, and instead of a code consumer, we have a *data consumer*.

For example, the data provider might be a user that wants to submit credit card data to a web site. In this case the data consumer would be the web site that the credit card data will be sent to.

In traditional PCC system, the code consumer fixes the security policy, the code producer creates a proof, and the code consumer checks it. In PCA systems, the security policy is fixed by the *data provider* (DP). Before providing any data to a *data consumer* (DC), the DP demands to receive a proof that its policies are obeyed.

In the example above, the user might specify a policy like

> The data consumer has to
>
> - encrypt my data before sending it on
> - check that its communication partners encrypt received data before sending it on

This kind of system does not really exist, so you are allowed to dream in this part of the project. Describe, what the data-consumer side of this system could look like, and which problems have to be solved.

3 Language for Questions 2.2.1 and 2.2.2

The following is the language for questions 2.2.1 and 2.2.2.

3.1 Necessities

The **Instruction Set** is a standard one, with an additional instruction to copy a register value to another register.

- `MOV` $r_{s2} := r_{s1}$. Moves register r_{s1} to r_{s2}.
- `ADD` $r_d := r_{s1} + r_{s2}$. Adds registers r_{s1} and r_{s2} and stores the sum in r_d.
- `ADD` $r_d := r_s + c$. Adds constant c to register r_s and stores the sum in r_d.

instruction	r'	m'
MOV $r_{s2} := r_{s1}$	$upd(r'', s2, r_{s1})$	m
ADD $r_d := r_{s1} + r_{s2}$	$upd(r'', d, r_{s1} + r_{s2})$	m
ADD $r_d := r_s + c$	$upd(r'', d, r_s + c)$	m
LD $r_d := m(r_s + c)$ and $readable(r_s + c)$	$upd(r'', d, m(r_s + c))$	m
ST $m(r_d + c) := r_s$ and $writable(r_s + c)$	r''	$upd(m, r_d + c, r_s)$
BEQ $(r_{s1} = r_{s2})$ c and $r_{s1} = r_{s2}$	$upd(r, pc, r_{pc} + c)$	m
BEQ $(r_{s1} = r_{s2})$ c and $r_{s1} \neq r_{s2}$	r''	m
BGT $(r_{s1} > r_{s2})$ c and $r_{s1} > r_{s2}$	$upd(r, pc, r_{pc} + c)$	m
BGT $(r_{s1} > r_{s2})$ c and $r_{s1} \leq r_{s2}$	r''	m
JMP c	$upd(r, pc, r_{pc} + c)$	m
RET	$upd(r, pc, r_{ret})$	m
INV p	r''	m

Figure 1: Effect of instructions.

- LD $r_d := m(r_s + c)$. Loads value from the memory cell pointed to by register r_s plus offset c and stores the value in register r_d.
- ST $m(r_d + c) := r_s$. Stores the value from register r_s in the memory cell pointed to by register r_d plus offset c.
- BEQ $(r_{s1} = r_{s2})$ c. Compare values in registers r_{s1} and r_{s2} and if they are equal jump to label c.
- BGT $(r_{s1} > r_{s2})$ c. Compare values in registers r_{s1} and r_{s2} and if r_{s1} is greater than r_{s2} jump to label c.

$$\frac{v :_m intlist}{m(v) = 0 \vee m(v) = 1}$$

$$\frac{v :_m intlist \qquad m(v)=1}{m(v+1) :_m int} \qquad \frac{v :_m intlist \qquad m(v)=1}{m(v+2) :_m intlist}$$

Figure 2: Type checking rules.

- `JMP` c. Jump to label c.
- `RET`. Returns from the current method by copying the special register r_{ret} to the program counter. The returned value is the content of r_0.
- `INV` p. p is an invariant for the code.

Register banks, memory, and expressions are defined in a standard way. A register bank is either the original bank R or the result of updating register n in bank r with expression e by invoking $upd(r, n, e)$. The memory is either the original memory M or the result of updating the memory cell pointed to by an expression e_1 in memory m to the value of expression e_2 by invoking method $upd(m, e_1, e_2)$. Finally, an expression e is a variable x, a natural number n, the sum of two expressions, a standard register, one of the special registers r_{pc} or r_{ret}, or an access to the memory cell pointed to by an expression.

- register bank. $r \ ::= \ R \mid upd(r, n, e)$
- memory. $m \ ::= \ M \mid upd(m, e_1, e_2)$
- expressions. $e \ ::= \ x \mid n \mid e_1 + e_2 \mid r_n \mid r_{pc} \mid r_{ret} \mid m(e)$

We also need a way to check types (Figure 2) and to specify **safety policies**, which is done by means of the following formulas:

- Types
  $\tau \ ::= \ int \mid intlist$
- Atomic Formulas
  $A \ ::= \ e :_m \tau \mid e_1 = e_2 \mid e_1 < e_2 \mid readable(e) \mid writable(e) \mid tag(e) = n$

$$VC_i = \begin{cases} VC_{i+1}[r_{s1}/r_{s2}] & \text{if } \Pi_i = \texttt{MOV } r_{s2} := r_{s1} \\ VC_{i+1}[(r_{s1}+r_{s2})/r_d] & \text{if } \Pi_i = \texttt{ADD } r_d := r_{s1}+r_{s2} \\ VC_{i+1}[(r_{s1}+c)/r_d] & \text{if } \Pi_i = \texttt{ADD } r_d := r_{s1}+c \\ VC_{i+1}[m(r_s+c)/r_d] \wedge \mathit{readable}(r_s+c) & \\ & \text{if } \Pi_i = \texttt{LD } r_d := m(r_s+c) \\ VC_{i+1}[\mathit{upd}(m, r_d+c, r_s)/m] \wedge \mathit{writable}(r_d+c) & \\ & \text{if } \Pi_i = \texttt{ST } m(r_d+c) := r_s \\ (r_{s1} = r_{s2} \rightarrow VC_{i+c-1}) \wedge (r_{s1} \neq r_{s2} \rightarrow VC_{i+1}) & \\ & \text{if } \Pi_i = \texttt{BEQ } (r_{s1} = r_{s2})\ c \\ (r_{s1} > r_{s2} \rightarrow VC_{i+c-1}) \wedge (r_{s1} \leq r_{s2} \rightarrow VC_{i+1}) & \\ & \text{if } \Pi_i = \texttt{BGT } (r_{s1} > r_{s2})\ c \\ VC_{i+c-1} & \text{if } \Pi_i = \texttt{JMP } c \\ \mathit{post} & \text{if } \Pi_i = \texttt{RET} \\ p & \text{if } \Pi_i = \texttt{INV } p \end{cases}$$

Figure 3: The verification condition generator. Π is the vector of all instructions of the program, and Π_i is the instruction at program counter location i.

- Formulas
  $P ::= A \mid \bot \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \rightarrow P_2 \mid \neg P \mid (P)$

The instructions are executed by an abstract machine. For one computation step, we have the following relation between the register banks and memory states before and after the step: $(r, m) \mapsto (r', m')$ if $(\mathit{upd}(r, pc, r_{pc}+1), m)$ evaluates to (r', m'). The **effect of each of the instructions** is given in Figure 1, assuming $r'' = \mathit{upd}(r, pc, r_{pc}+1)$.

Finally, we need to specify the **verification condition generator**, based on the available instructions. This is given in Figure 3. Note that each jump target must be proceeded by an `INV` instruction, that provides the invariant for the jump target.

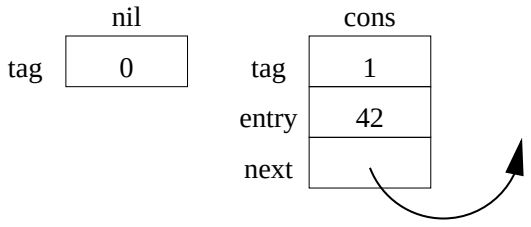

Figure 4: Layout of cells in memory.

Precondition: $r_0 :_m \mathit{intlist} \wedge r_1 :_m \mathit{intlist} \wedge \mathit{tag}(r_1) = 0 \wedge r_6 = 0$

```
1:          MOV r2 := r0
2:          MOV r3 := r1
3:          INV r2 :m intlist ∧ r3 :m intlist ∧ r6 = 0
4:    L1 :  LD r5 := m(r2 + 0)
5:          BEQ (r5 = r6) 7
6:          LD r4 := m(r2 + 2)
7:          ST m(r2 + 2) := r3
8:          MOV r3 := r2
9:          MOV r2 := r4
10:         JMP − 6
11:         INV r3 :m intlist
12:   L2 :  MOV r0 := r3
13:         RET
```

Figure 5: Exercise program.

3.2 Program

We use a list of integers consisting of two constructors, *nil* and *entry*. A nil object has the tag 0 and no further elements, a cons object has the tag 1 that is followed by an integer and a pointer to the next element of the list (Figure 4).

The program for the exercise is given in Figure 5.