Université Joseph Fourier **Grenoble INP**
Master MOSIG M1 **2009-10**

Written Exam - December 2009

1 Exercise I : Operational Semantics

In this exercise we will formalize the semantics of some primitives for process management within a multi-thread program. This semantics will be expressed on an intermediate program representation.

We first give the syntax and semantics of sequential statements, and then we describe the extensions for multi-threading.

1.1 Intermediate program representation : sequential part

1.1.1 Syntax

- Labels $l \in \mathbb{N}$
- Variables : $x \in \texttt{Var}$
- Statements
  $I \in \texttt{Stm}$
  $I ::= x := e \mid \texttt{if}\ b\ \texttt{goto}\ l \mid \texttt{goto}\ l$
- Arithmetic expressions $\mathcal{A}\texttt{Exp}$
  $a \in \mathcal{A}\texttt{Exp}$
  $a ::= n \mid x \mid a + a \mid a - a$
- Boolean expressions $\mathcal{B}\texttt{Exp}$
  $b \in \mathcal{B}\texttt{Exp}$
  $b ::= x < y \mid x = y \mid x > y$
- Expressions $\mathcal{E}\texttt{xp}$ $e ::= a \mid b$

A program consists in a sequence of names D followed by a sequence of statements C (for "code"). A sequence of statement is formalized by a partial function from the (positive) integers to the statement set. The integer gives the rank of the statement in the sequence (i.e., it coresponds to the value of a program counter).

1.1.2 Configurations

The configurations are triples (pc, C, σ) where pc, C, σ denotes respectively the program counter, the code and the memory ; the memory is a partial function from D to the set of values. A value is either a boolean or an integer. To summarize :

cp	$\mathbb{N}$	program counter
C	$\mathbb{N} \mapsto \texttt{Stm}$	the code
σ	$\texttt{Var} \mapsto \mathbb{Z} \cup \mathbb{B}$	the memory

1.1.3 Statement semantics

$$
\begin{array}{lcll}
(pc, C, \sigma) & \longrightarrow & (pc+1, C, \sigma[x \mapsto \mathcal{A}[a]_\sigma]) & \text{if } C(pc) = x := a \\
(pc, C, \sigma) & \longrightarrow & (l, C, \sigma) & \text{if } C(pc) = \texttt{goto}\ l \\
(pc, C, \sigma) & \longrightarrow & (pc+1, C, \sigma) & \text{if } C(pc) = \texttt{if}\ b\ \texttt{goto}\ l \text{ and } \mathcal{B}[b]_\sigma = \texttt{ff} \\
(pc, C, \sigma) & \longrightarrow & (l, C, \sigma) & \text{if } C(pc) = \texttt{if}\ b\ \texttt{goto}\ l \text{ and } \mathcal{B}[b]_\sigma = \texttt{tt}
\end{array}
$$

1.2 Extensions for multi-threading

We consider now new elements for process management (creation, destruction and communication).

We introduce *process variables* for storing process identification numbers (Pid). Values of type Pid are represented by positive integers. The body of a process consists in a sequence of names, followed by a sequence of statements. We introduce the following new statements :

i :=`create`(D_V C)	:	The current process creates a new process, identified by i
`kill`(i)	:	The current process removes from the set of active processes the process i.
`send` e `to` j	:	the current process sends the value of expression e to the process j.
`recv` x `to` j	:	the current process receives a value from process j and stores it in x.
`terminate`	:	(normal) termination of the current process.

The syntax of the `while` language is therefore modified as follows :

$$
\begin{array}{lcl}
P & ::= & D_V\ C \\
D_V & ::= & \texttt{var}\ x\ ;\ D_V \mid \epsilon
\end{array}
$$

In this syntax :
– P is a program,
– D_V is a sequence of process variable declarations.
Thus, a program is a particular process which is implicitely created when the program execution starts.

The following program creates 2 processes, identified by `i` and `j`. Process `i` declares 3 variables, sends `x` and `y` to `j`, and stores a received value in `z`. Process `j` receives two values from `i` and sends back their sum.

```
var i ; var j ;
i:= create (
      var x; var y ; var z;
      x:=3; y:=4;
      send x to j ;
      send y to j ;
      recv z from j ) ;
j:= create (
      var x ; var y;
      recv x from i ;
      recv y from i ;
      send x+y to i ) ;
kill(i) ; kill(j) ;
```

1.3 Semantics of the new statements

Process configurations are triples (π, Π, Φ) where :

- $\pi(i)$ associates a process identifier to each Pid variable,
- Π associates to each **active** process identifier w a triple (pc, C, σ) where :
  - pc is the program counter associated to w,
  - C is the code associated to w and
  - σ is the local memory associated to w.
- Φ is a set of triples (i, j, v) where :
  - i is the Pid of the sending process,
  - j is the Pid of the receiving process,
  - v is the value sent.
- For a program $D_V\ C$, the initial configuration is $(\emptyset, \Pi_0, \emptyset)$ where $\Pi_0 = [0 \mapsto (1, C, \sigma_{D_V})]$.

The semantics of the new statements is expressed by operational semantic rules. For instance, if the current statement of process i is `j := create`$(D_V\ C)$, the corresponding semanting rule consists in adding a new (fresh) process to Π such that is counter program is set to 1 and the definition domain of its local memory σ is the set D_V. Formally :

$$\boxed{\frac{\pi(i) = n \quad \Pi(n) = (pc, C, \sigma) \quad C(pc) = j := create(D\ C') \quad m = |\Pi| + 1}{(\pi, \Pi, \Phi) \longrightarrow (\pi[j \mapsto m], \Pi[n \mapsto (pc+1, C, \sigma)][m \mapsto (1, C', \sigma_D)], \Phi)}}$$

where $|\Pi|$ is the number of active processes.

Question 1 Assume that $\Pi(\pi(i)) = (pc, C, \sigma)$ and $C(pc) =$ `kill`(j). Give the operational semantics of statement `kill`(j) which removes the process identified by j from the list Π of active processes.

Question 2 Assume that $\Pi(\pi(i)) = (pc, C, \sigma)$ and $C(pc) =$ `send` e `to` j. Give the operational semantics of statement `send` e `to` j which sends the value of expression e to the process j by adding to Φ the triple (i, j, v) where v is the value of e in σ.

Question 3 Assume that $\Pi(\pi(i)) = (pc, C, \sigma)$ and $C(pc) =$ `recv` x `from` j. Give the operational semantics of statement `recv` x `from` j which receives a value v in x and removes (i, j, v) from Φ, if it exists.

Question 4 Assume that $\Pi(\pi(i)) = (pc, C, \sigma)$ et $C(pc) = x := e$. Give the operational semantics of statement x :=e.

Question 5 Assume that $\Pi(\pi(i)) = (pc, C, \sigma)$ et $C(pc) =$ `terminate`. Give the operational semantics of statement `terminate`.

Question 6 Assume that $\Pi(\pi(i)) = (pc, C, \sigma)$ et $C(pc) =$ `goto` l. Give the operational semantics of statement `goto` l.

Question 7 Assume that $\Pi(\pi(i)) = (pc, C, \sigma)$ et $C(pc) =$ `if` b `goto` l. Give the operational semantics of statement `if b goto l`.

Question 8 We now consider another communication primitive based on "rendez-vous". The sending process i needs to synchronize with the receiving process j, and the communication (with value exchange) occurs in an atomic way (both processes execute the statement in the same atomic step), without using Φ.

The corresponding statement are `sendsync` e `to` j and `recvsync` x `from` j. Give the operational semantics of these statements.

2 Exercice II : Static semantics

We consider here a variant of the **While** language those abstract syntax is given below :

$$
\begin{array}{lcl}
P & ::= & D \;\; S \\
D & ::= & \texttt{var x ;}\ D \mid \varepsilon \\
S & ::= & \texttt{x :=}\ E \mid S\ \texttt{;}\ S \mid \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \mid \texttt{while}\ E\ \texttt{do}\ S \\
 & & \mid \texttt{read(x)} \mid \texttt{syscall}(E) \\
E & ::= & c \mid \texttt{x} \mid E\ \texttt{op}\ E
\end{array}
$$

In this syntax :
- P denotes a program ;
- SD denotes a declaration list ;
- S denotes a statement list ;
- E denotes an expression (either boolean or arithmetic) ;
- `x` denotes a variable and `c` denotes a constant ;
- `read(x)` reads a value from the keyboard and stores it in variable `x`
- `syscall`(E) performs a system call with an actual parameter E

A variable is considered as **tainted** at a program location if its current value at this location **depends on a value that has been read from the keyboard**. It is considered as **untainted** otherwise. Similarly, an expression is tainted when one of its operand is tainted, and a constant is always untainted. We assumed that initialy all variables are untainted.

Let us consider the following program :

program location	statement	tainted variables	untainted variables
1	`var w, x, y, z ;`	$\emptyset$	{w, x, y, z}
2	`read(x) ;`	{x}	{w, y, z}
3	`read(y) ;`	{x, y}	{w, z}
4	`z := x + 2 ;`	{x, y, z}	{w}
5	`w := z ;`	{w, x, y, z}	$\emptyset$
6	`x := 3 ;`	{w, y, z}	{x}
7	`w := x + 1 ;`	{y, z}	{w, x}
8	`read(x) ;`	{x, y, z}	{w}

For security reasons, a system call `syscall`(E) is authorized only if expression E is **untainted**. The goal of this exercise is to write a **type system** for checking this security rule and forbids unsecure system calls. We introduce the following notations :

- $Vars$ is the set of variable names ;
- $Types = \{T, U\}$ is the set of types where T means "tainted" and U means "untainted" ;
- $\Gamma : Vars \rightarrow Types$ is an environment, i.e., a function assigning a type to each variable.

This type system will be defined with the following judgements :

- for a **declaration** D, $\Gamma \vdash D \mid \Gamma'$ means that "*when executed within environment* Γ*, declaration* D *produces the new environment* Γ'"
- For an **expression** E, $\Gamma \vdash E : t$ means that " *within environment* Γ *expression* E *is of type* t" (remember that in this context the type of an expression is either T or U).
- For a **statement** S, $\Gamma \vdash S \mid \Gamma'$ means that "*when executed within environment* Γ*, statement* S *produces the new environment* Γ'".

The typing rules associated to a declaration are the following :

$$\Gamma \vdash \varepsilon \mid \Gamma$$

$$\frac{\Gamma \vdash D \mid \Gamma'}{\Gamma \vdash \texttt{var x ;}\ D \mid \Gamma'[x \rightarrow U]}$$

The typing rule associated to a **program** is the following :

$$\frac{\emptyset \vdash D \mid \Gamma \quad \Gamma \vdash S \mid \Gamma'}{\vdash D\ ;\ S \mid \Gamma'}$$

Question 1. Give the typing rule corresponding to each kind of expression (constant, variable and binary operation).

Question 2. Give the typing rule for an assignement "`x := E`".

Question 3. Give the typing rule for sequential composition "`S1 ; S2`".

Question 4. Give the typing rule for the read statement "`read(x)`".

Question 5. Give the typing rule for a system call "`syscall(E)`."
Using your type system show that Program1 below is correct and Program 2 is incorrect :

```
-- Program 1
var x, y ;
x := 3 ;
y := x+1 ;
syscall(y+2) ;
```

```
-- Program 2
var x, y ;
read(x) ;
y := x+1 ;
syscall(y+2) ;
```

Question 6. For the conditionnal statement "`if E then S1 else S2`", we should consider that when the condition E is tainted then every variable assigned within `S1` or within `S2` should be tainted as well. For example :

```
var x, y, z ;
read(x) ; -- x is tainted
if (x >0) then
        y := 1 ;
else
        z := 2 ;
-- the values of y and z now depends on the value of x, so they are tainted
```

Give the typing rule for the conditionnal statement. You can use an auxilliary function *Assigned* to denote the sets of variables assigned within a statement (but you should properly define this function).

Question 7. For the iteration statement "`while E do S`" we should consider also that when the condition E is tainted every variable assigned within S is tainted (since its value at the end of the iteration depends on E).

Give the typing rule for the iteration statement and, using your rules, show that the following program is **incorrect**[1] ?

```
var x, y, z ;
read(x) ;
while (x>0)
        x := x - 1 ;
        z := y + 1 ;
        read(y) ;
syscall(z) ;
```

Question 8. The type system we defined in the previous questions may reject correct programs, namely programs in which **at execution time** there is no system calls `syscall(`E`)` such that E is tainted. Give an example of such a program, rejected by this type system, and correct when executed.

[1] since `z` is tainted if the loop is executed more than twice ...

3 Exercice III : Optimization

We consider the control flow graph depicted on Figure 1 :

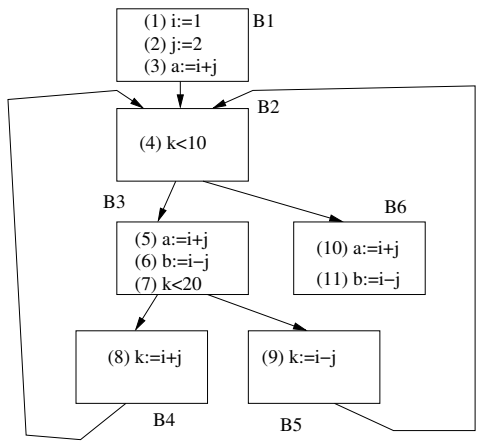

FIG. 1 – Initial control flow graph

Available expressions

1. Compute the sets $Gen(b)$ and $Kill(b)$ for each basic block b.
2. Compute the sets $In(b)$ and $Out(b)$ for each basic block b.
3. Suppress redundant computations.

In the following part we consider the new control flow graph

Active variables

1. Compute the sets $Gen(b)$ and $Kill(b)$ for each basic block b.
2. Compute the sets $In(b)$ and $Out(b)$ for each basic block b.
3. Suppress useless assignements, i.e., assignements (l) $x := e$ such that x is not active at the end of the block and not used between location l and the end of the block.

4 Exercice IV : Code generation

Part 1

We consider the following program :

```
proc P1() {

   int x1 ;

   proc Q(int a) {
      x1 := a
   }

   proc P2 (int c, int d) {
       int x2 ;
       proc P3 (int b) {
            if b>1 then
               call P3(b-1) ;
               x1:=x2+c+b
            else
               call Q(x1)
            fi
       } /* of P3 */
       x2:=1 ;
       call P3(2)
   } /* of P2 */

   x1:=0 ;
   call P2(x1+1, 7)
} /* of P1 */
```

Q1. Give a complete picture of the execution stack when procedure `Q` is executed.

Q2. Give the intermediate M code produced for procedure `P3`.

Part 2

We consider now a program with a variable P of type procedure. At the code level, such a variable is represented by a pair (l, s) where l is the address of the code of the procedure assigned to P and s is the address of the definition environment of the procedure assigned to P (its static link).

```
proc P1() {

   int x1 ;
   var P : proc ;   /* P is a variable of type procedure */

   proc P2() {
      call P()     /* P is called */
   }

   proc P3 () {
      var x2 ;
      x2 := x1 + 1 ;
   }

   P := P3 ;      /* P2 is assigned to P */
   call P2()
}
```

Q1. Give a complete picture of the execution stack when procedure `P3` is executed.

Q2. Give the intermediate M code produced for procedure `P2`.