Mosig M1 - PLSCD 0809 - Written exam

1 Exercise I : Operational semantics - a debugger

In this exercise we consider program execution with the help of a *debugger*. This execution will be defined on an intermediate code representation, based on a syntax given below. In this representation we do not distinguish between variables and temporary memory locations. The goal is to define the operational semantics of each debugging command and to observe debugging sequences.

1.1 Intermediate representation for this exercice

1.1.1 Syntax of the intermediate representation under consideration

– Labels : $l \in \mathbb{N}$
– Variables : $x \in \texttt{Var}$
– Statements
$S \;::=\; x := a \mid \texttt{if}\ b\ \texttt{goto}\ l \mid \texttt{goto}\ l$
– Arithmetic expressions $\mathcal{A}\texttt{Exp}$
$a \;\in\; \mathcal{A}\texttt{Exp}$
$a \;::=\; n \mid x \mid a + a \mid a - a$
– Boolean expressions $\mathcal{B}\texttt{Exp}$
$b \;\in\; \mathcal{B}\texttt{Exp}$
$b \;::=\; x < y \mid x = y \mid x > y$

1.1.2 Configurations

Configurations are triples (pc, C, σ) where pc, C, σ designate respectively the program counter, the code and the memory :

cp	$\mathbb{N}$	program counter
C	$\mathbb{N} \mapsto \texttt{Stm}$	code
σ	$\texttt{Var} \mapsto \mathbb{Z}$	memory `State`

1.1.3 Expression semantics

Arithmetic We recall the semantic function $\mathcal{A}\texttt{Exp} \mapsto (\texttt{State} \mapsto \mathbb{Z})$:

$$\begin{array}{lcl} \mathcal{A}[v]_\sigma & = & v \\ \mathcal{A}[x]_\sigma & = & \sigma(x) \\ \mathcal{A}[a_1 + a_2]_\sigma & = & \mathcal{A}[a_1]_\sigma +_I \mathcal{A}[a_2]_\sigma \\ \mathcal{A}[a_1 - a_2]_\sigma & = & \mathcal{A}[a_1]_\sigma -_I \mathcal{A}[a_2]_\sigma \end{array}$$

Boolean We recall the semantic function $\mathcal{B}\texttt{Exp} \mapsto (\texttt{State} \mapsto \{\texttt{tt}, \texttt{ff}\})$:

$$\begin{array}{lcl} \mathcal{B}[x < y]_\sigma & = & \mathcal{A}[x]_\sigma <_I \mathcal{A}[y]_\sigma \\ \mathcal{B}[x = y]_\sigma & = & \mathcal{A}[x]_\sigma =_I \mathcal{A}[y]_\sigma \\ \mathcal{B}[x > y]_\sigma & = & \mathcal{A}[x]_\sigma >_I \mathcal{A}[y]_\sigma \end{array}$$

1.1.4 Statement semantics

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

1.2 Debugging commands

We consider the following commands in this exercise :

`setbrk`(n)	:	sets a breakpoint at statement labeled by n : where the program counter is equal to n, the execution stops.
`rmbrk`(n)	:	removes the breakpoint set at statement labeled by n
`step`	:	performs a single execution step
`cont`	:	resumes program execution up to the next breakpoint.
`run`	:	similar to `cont`, but starting from the initial configuration.
`set`(x,v)	:	sets value v to variable x.
`print`(x)	:	prints the value of variable x.
`exit`	:	exits from the debugger.

1.3 Interaction between the debugger and the program

A configuration is composed by :

1. a program configuration (pc, C, σ),
2. a set of breakpoints β,
3. optionally a debugger command.

An initial configuration $((1, C, \sigma), \emptyset, \texttt{cmd})$ is defined by an initial program configuration, an empty set of breakpoints, and a command `cmd`.

The transition relation is :

$$((cp, C, \sigma), \beta, \texttt{cmd}) \rightarrow ((cp', C, \sigma'), \beta')$$

1.4 Example

```
1:x:=8
2:y:=12
3:if x=y goto 9
4:if x<y goto 7
5:x:=x-y
6:goto 3
7:x:=y-x
8:goto 3
9:
```

The previous program computes the *gcd* (greatest common divisor) between `x` and `y`, with a bug inside.

1.5 Statement semantics

For each command we want to define the associated semantic rule(s).

Let consider the previous example. Program C has two variables x et y. The memory is represented as follows :$[x \mapsto v, y \mapsto v']$ where v and v' are values associated respectively to x and y. For instance, configuration $((1, C, [x \mapsto 0, y \mapsto 0]), \emptyset, \texttt{setbrk}(3))$ produces configuration $((1, C, [x \mapsto 0, y \mapsto 0]), \{3\})$; configuration $((1, C, [x \mapsto 0, y \mapsto 0]), \{3\}, \texttt{run})$ produces configuration $((3, C, [x \mapsto 8, y \mapsto 12]), \{3\})$; configuration $((3, C, [x \mapsto 8, y \mapsto 12]), \{3\}, \texttt{cont})$ produces configuration $((3, C, [x \mapsto 4, y \mapsto 12]), \{3\})$.

Question 1 Give the operational semantics of command `setbrk`(n) by filling the rule :

$((cp, C, \sigma), \beta, \texttt{setbrk}(n)) \rightarrow (\cdots, \cdots)$.

Question 2 Give the operational semantics of command `rmbrk`(n) by filling the rule :

$((cp, C, \sigma), \beta, \texttt{rmbrk}(n)) \rightarrow (\cdots, \cdots)$.

Question 3 Give the operational semantics of command `set`(x,v) by filling the rule :

$((cp, C, \sigma), \beta, \texttt{set}(x, v)) \rightarrow (\cdots, \cdots)$.

Question 4 Give the operational semantics of command `print`(x) by filling the rule :

$((cp, C, \sigma), \beta, \texttt{print}(x)) \rightarrow (\cdots, \cdots)$.

Question 5 Give the operational semantics of command `step`.

Question 6 Give the operational semantics of command `cont`. Indication : inspire yourself from the rules defined for a repeat statement.

Question 7 Give the operational semantics of command `run`.

1.6 Debugging sequence

A *final* configuration is $((pc, C, \sigma), \beta, \texttt{exit})$.

A *debugging sequence* is a configuration sequence

$((1, C, \sigma), \emptyset, \texttt{cmd})((pc_1, C, \sigma_1), \beta_1, \texttt{cmd}_1) \cdots ((pc_i, C, \sigma_i), \beta_i, \texttt{cmd}_i)$ such that

$((pc_i, C, \sigma_i), \beta_i, \texttt{cmd}_i) \rightarrow ((pc_{i+1}, C, \sigma_{i+1}), \beta_{i+1})$

A maximal debugging sequence is
$((1, C, \sigma), \emptyset, \texttt{cmd})((pc_1, C, \sigma_1), \beta_1, \texttt{cmd}_1) \cdots ((pc_n, C, \sigma_n), \beta_n, \texttt{exit})$

Question 8 Give the debugging sequence obtained when applying the following debugging commands on the example program : `setbrk`(3), `cont`, `cont`, `cont`, `cont`.

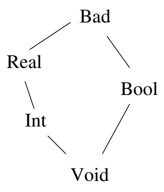
Question 9 Give the debugging sequence obtained when applying the following debugging commands on the example program : `setbrk`(3), `step`, `step`.

2 Exercice II : Static semantics

We consider a variant of the **While** language where variables are implicitely declared during their first assignment. The abstract syntax of this language is given below :

$$\begin{array}{lcl} P & ::= & S \\ S & ::= & x := E \mid S; S \mid \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \mid \texttt{while}\ E\ \texttt{do}\ S \\ E & ::= & n \mid k \mid \texttt{x} \mid \texttt{true} \mid E + E \mid E = E \mid \texttt{not}\ E \end{array}$$

In this syntax `x` designates an identifier ($\texttt{x} \in Names$), n an integer constant ($n \in \mathbb{Z}$) and k a real constant ($k \in \mathbb{R}$). The set of predefined types is the set $Types = \{\texttt{Int}, \texttt{Real}, \texttt{Bool}, \texttt{Void}, \texttt{Bad}\}$, in which `Void` designates a lack of type and `Bad` an incorrect type. We consider also a partial order relation $\leq$ on set $Types$ which defines the following lattice, where $Void$ (resp. Bad) is the minimal (resp. maximal) element :



Part 1

At a given program location the type of a variable is defined by the four informal rules given below :

R1 : A variable or an expression of type `Int` can be implicitely cast to the `Real type` (the converse being false).

R2 : The type of an expression cannot be `Void`

R3 : A variable that has not been assigned before is of type `Void`

R4 : After assignment `x := E`, the type of `x` is the least upper bound between the type of `x` before the assignment and the type of `E`.

We also assume that operator "=" can be applied to operands of same type (after possible implicit cast), operator "+" to integer and/or real operands (after possible implicit cast), and operator `not` to a boolean operand. We give below some program examples to illustrate these rules :

```
-- Example 1
x := 2 ;
y := x=2 ;
x := not y ;
  -- incorrect, x of type Int
z := t+1  ;
  -- incorrect, t of type Void
x := not y ;
```

```
-- Example 2
x := 2 ;
if x = 2 then
       y := true ;
else
       y := 2.5 ;
z := y+1 ;
  -- incorrect, y of type Bad
```

```
-- Example 3
x := 2 ;
y := x=2 ;
z := y+1  ;
  -- incorrect, y of type Bool
```

```
-- Example 4
x := 2 ;
if x = 2 then
       y := 4 ;
else
       y := 2.5 ;
y := y+1 ;
  -- correct, y of type Real
```

Question 1.

Propose a static type system for this language, inductively defined on the abstract syntax. You should use the following notations :

– An *environment* Γ is a partial function $Names \rightarrow Types$.
– The judgement used to define the static semantics of an expression is $\Gamma \vdash E : t$ meaning *within environment* Γ *expression* E *is well typed of type* t.
– The judgement used to define the static semantics of a statement is $\Gamma \vdash S \mid \Gamma'$ meaning *within environment* Γ *statement* S *is correct and produces a new environment* Γ'.

Question 2.

Propose a program example those *execution* satisfies rules **R1**, **R2**, **R3** and **R4**, but which is staticaly rejected by at least one of the semantic rules proposed at question 1 (indicate which one(s)).

Question 3. We now assume that each assignment `x := E` *re-defines* the type of `x` with the type of `E`. Thus, the following program is now correct :

```
x := 2 ; -- x of type Int
y := x+1 ; -- y of type Int
x := not (y=2) ;  -- x of type Bool
if x then
```

```
     x := 2.3 ; -- x of type Real
else
     x := 4.2 ; -- x of type Real
y := x+1 ;  -- y of type Real
```

Give the new static type system (only the rule(s) that differ(s) from the ones proposed at question 1).

Part 2

We now add to this language the notion of procedures, parameter free, with a single nesting level. The language syntax is therefore modified as follows :

P	$::=$	D_P S
D_P	$::=$	`proc p is` S `endproc ;` D_P | ε
S	$::=$	... | `call p`
E	$::=$	...

The types of variables are still defined by rules **R1** to **R4**, as illustrated by the following examples :

```
-- Example 1
proc p1 is
  x := 2 ;
endproc ;

proc p2 is
  call p1 ;
  y := x+1 ;
    -- correct, x of type Int
endproc ;

y := 0 ;
call p2 ;
end
```

```
-- Example 2
proc p1 is
  x := 2 ;
endproc ;

proc p2 is
  y := x+1 ;
    -- incorrect, x of type Void
  call p1 ;
endproc ;

y := 0 ;
call p2 ;
end
```

Question 4.

Extend the answers of question 1 to deal with this new syntax. You should assume that :
– there is no recursive call in the program ;
– each variable has a global scope ;
– a procedure declaration D_P is considered correct when :
  – procedure names are pairwise distinct ;
  – procedures are declared before used.
Indicate the new judgements introduced, if any.

Question 5.

We assumed in the previous question that there is no recursive call in the program. If a program does not satisfy this condition is it rejected by the semantic rules you proposed ? If yes, why (i.e., by which rule(s)), otherwise, modify these rules to introduce this check.

3 Exercise III : Optimization

We consider the following control flow graph (figure 1) :

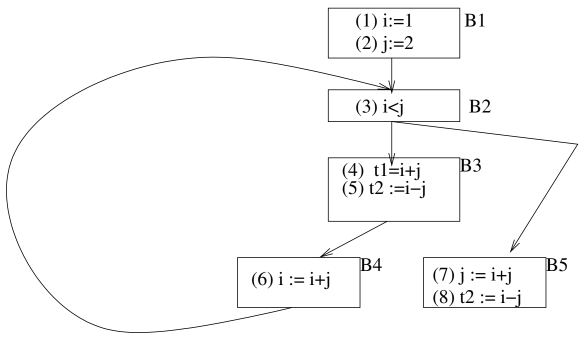


FIG. 1 – Initial control flow graph

Available expressions : questions

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

In the following part, we consider the control flow graph after modification

Active variables : questions

1. Compute sets $Gen(b)$ and $Kill(b)$ for each basic block b.
2. Compute sets $In(b)$ and $Out(b)$ for each basic block b.
3. Suppress useless statements, i.e., assignments $x := e$ such that x is inactive at the end of the block and is not used in this block after this assignment.

4 Exercise IV : Code generation

We consider the following program :

```
#include <stdio.h>
typedef int (*intprocint) ( int);
/* intprocint designates a procedure type with an integer parameter
                                             and an integer result */
typedef int (*intprocintint) ( int,int);
typedef int (*intprocvoid) ( void);

main(){
  int x1;
  int p1 () {
    int x ;
    int y ;
    int z ;
    int p2(int x,intprocint p) {
      int p3 (intprocint p){
        return p(x+x1+y);
      }
      z=11;
      z = y + p3(p);
      return z;
    }
    int g2(int x) {
      int g3 (int x){return (x-1);}
      if (x>0) y=p2(x1,g3);
    }
    y=22;
    z=g2(2);
    return z;
  }
  x1=0;
  x1=p1();
  printf("%d\n",x1);
}
```

1. Give the execution stack during execution of `g3`, drawing only static and dynamic links.
2. We consider procedure `p1`. Give the code sequence corresponding to `z = g2(2)`.
3. We consider procedure `p2`. Give the code sequence corresponding to `z = y + p3(p)`.
4. We consider procedure `p3`. Give the code sequence corresponding to `return p(x+x1+y)`.
5. We consider procedure `g2`. Give the code sequence corresponding to `y = p2(x1,g3)`.