SLPC

Mid-term Exam Wednesday 7th of November 2012

Duration : 1h30 - **All documents authorized**. - Exercises are **independents**.

Exercise 1 : pointers

We consider the **While** language seen in the course (without nested blocks nor procedures). We add the type `ref t` which means "pointer to a type `t`". intuitively, a variable of type `ref t` can take as value a memory address of a variable of type `t`.
We also consider two new operators, inspired from the C programming language:

- the operator `*e`, which allows to access to the value of type `t` referred by the expression `e` of type `ref t`;
- the operator `&x`, which allows to access the address of a variable `x` of type `t`, the result is then of type `ref t`.

We thus obtain a new syntax for the **While** language:

$$
\begin{array}{lcl}
P & ::= & D \; ; \; S \\
T & ::= & \texttt{integer} \mid \texttt{boolean} \mid \texttt{ref}\; T \\
D & ::= & \texttt{var}\; x : T ; D \mid \epsilon \\
G & ::= & x \mid *G \\
E & ::= & n \mid G \mid \&x \mid E + E \mid \texttt{true} \mid E = E \mid \texttt{not}\; E \mid E \;\texttt{and}\; E \mid E + E \mid E - E \mid E < E \\
S & ::= & G := E \mid \texttt{skip} \mid S ; S \mid \texttt{if}\; E \;\texttt{then}\; S \;\texttt{else}\; S \mid \texttt{while}\; E \;\texttt{do}\; S
\end{array}
$$

In this syntax, x designates an identifier ($x \in \texttt{Var}$) and n a integer constant positive or negative ($n \in \mathbb{Z}$). The non-terminal G designates the expressions that can occur in the left-hand side of an assignment, to which the operator `*` can be applied. We give below two examples of programs, one correct the other not w.r.t. the above syntax:

```
var x : integer ;
var y : ref integer ;
var z : ref ref integer ;
x := 3 ;
y := &x ;
z := &y ;
**z := *y + x ;
```

```
var x : integer ;
var y : ref integer ;
var z : ref ref integer ;
&x := 3 ; /* syntax error on the left-hand part */
y := &(x +1) ; /* syntax error on the right-hand part */
z := &&x ; /* syntax error on the right-hand part */
**z := *&y ; /* syntax error on the right-hand part */
```

Part 1 : static semantics : type checking

We define the set Type as follows:

$$\mathsf{Type} \quad = \quad \{\mathsf{Integer}, \mathsf{Boolean}\} \cup (\{\mathsf{ref}\} \times \mathsf{Type})$$

Thus, an element of this set is either a basic type or a pair (ref, t). We then define a function $\mathcal{T}$ that transforms any syntactic element T into an element of the set `Type` :

$$
\begin{array}{rcl}
\mathcal{T}(\texttt{integer}) & = & \mathsf{Integer} \\
\mathcal{T}(\texttt{boolean}) & = & \mathsf{Boolean} \\
\mathcal{T}(\texttt{ref t}) & = & (\mathsf{ref}, t)
\end{array}
$$

We recall that the type checking rule of the While language are defined as follows:

- An *environment* Env is a partial function Var $\rightarrow$ Type, initially empty for the program. A program is correctly typed, if by constructing an environment from declarations, the statement part is correctly typed:

$$\frac{\emptyset \vdash D \mid \Gamma \qquad \Gamma \vdash S}{\emptyset \vdash D\ S}$$

- The set of declarations is used to construct the environment.

$$\Gamma \vdash \epsilon \mid \Gamma \qquad \frac{\Gamma[x \mapsto \mathcal{T}(T)] \vdash D \mid \Gamma'}{\Gamma \vdash \texttt{var}\ x\ :\ T; D \mid \Gamma'}$$

- We compute the type of an expression, when it is possible, in the environment Γ: $\Gamma \vdash E : t$. Arithmetical expressions $\{+, -\}$ are defined on integers. Logical operators $\{\texttt{and}, \texttt{not}\}$ are defined on booleans. Comparison operators $\{=, <\}$ are defined on integers and booleans.

- For a statement S, we indicate if it is correctly typed in the environment $\Gamma : \Gamma \vdash S$.

Q1.

1. Propose a typing rule for the expression &x.
2. Give an example of incorrect program w.r.t. this rule.
3. Propose a typing rule for the expression *G.
4. Give an example of incorrect program w.r.t. this rule.
5. Propose a typing rule for the expression $E_1 + E_2$
6. Give an example of incorrect program w.r.t. this rule.
7. Propose a typing rule for the expression $E_1 = E_2$
8. Give an example of incorrect program w.r.t. this rule.

Q2.

1. We suppose in this part that an assignment is correct (from a typing point of view) when the left and right operands are correct, and of the *same type*. Propose a typing rule for the statement G := E.
2. Give an example of incorrect program w.r.t. this rule.

Part 2 : natural operational semantics

We note Adr the set of memory *addresses* mémoires, and we note Val the set of *values* $\mathbb{Z} \cup \{\texttt{tt}, \texttt{ff}\} \cup \texttt{Adr}$. Moreover:

- An *environment* Env is a partial function Var $\rightarrow$ Adr ;
- A *memory* Mem is a partial function Adr $\rightarrow$ Val.

Then, the result of an expression (and thus the content of a memory slot) can be either an integer or a boolean (for a basic type), or an *address* (for a pointer type).

Semantics rule for this language are then defined as follows:

- the semantics of a *declaration* D is defined by a relation $\xrightarrow{\text{d}} = \mathcal{C}_D \times \mathcal{C}_D$ with $\mathcal{C}_D = \mathtt{Decl} \cup \mathtt{Env}$, where $\mathtt{Decl} \ni D$ is the set of declarations.
- the semantics of an *expression* E is defined by a relation $\xrightarrow{\text{e}} \subseteq \mathcal{C}_E \times \mathcal{C}_E$ with $\mathcal{C}_E = (\mathtt{Exp} \times \mathtt{Env} \times \mathtt{Mem}) \cup \mathtt{Val}$. where $\mathtt{Exp} \ni E$ is the set of expressions.
- the semantics of a *statement* S is defined by a relation $\xrightarrow{\text{s}} \subseteq \mathcal{C}_S \times \mathcal{C}_S$ with $\mathcal{C}_S = (\mathtt{Stm} \times \mathtt{Env} \times \mathtt{Mem}) \cup \mathtt{Mem}$. where $\mathtt{Stm} \ni S$ is the set of statements.

The semantics rules associated to declarations are not changed compared to those seen in the course.

Q3.

1. Propose a semantics rule for the expression `&x`
2. Propose a semantics rule for the expression `*G`
3. Propose a semantics rule for the statement `G:=E`.
   Here it will be possible to distinguish two cases (and then write two rules) according to whether `G` is of the form "`x`" or "`*G`".

Part 3 : Typing with implicit dereferencing

We now suppose that the typing rules are more flexible, and that in particular the operator "`*`" (dereferencing) is not mandatory anymore. These operators are implicitly added by the compiler, *in a minimal way*, wherever it is necessary for the program to be correctly typed. The, the following program is now correct:

```
var x : integer ;
var y : ref integer ;
var z : ref ref integer ;
x := 3 ;
y := &x ;
z := &y ;
y := z + 1 ;      /* z will be dereferred twice to allow the addition */
                  /* y will be dereferred once to allow the assignment */
x := x + y ;      /* y will be dereferred twice to allow the addition */
x := z + 1 ;      /* z will be dereferred twice to allow the addition */
if (y = z) ... ; /* z will be dereferred once to allow the comparison */
```

Q4.

1. Write the new typing rule for the expression `E1 + E2`.
2. Write the new typing rule for the expression `E1 = E2`.
3. Write the new typing rule for the statement `G := E`.

Part 4 : Extension 2, arithmetic on pointers

Regarding arithmetic on pointers, we inspire from the rules in `C`, that is

- we can add a pointer and an integer (or an integer and a pointer), the result will be of type reference,

- substract two pointers or an integer from a pointer, the result will be in the first case of type pointer, and of type reference in the second case,
- we can compare two pointers.

We modify the syntax as follows:

$$G \quad ::= \quad x \mid *G \mid *(G+n) \mid *(G+x)$$

Q5.

1. Modify the semantics rule for the assignment `G:=E`.
2. Give the semantics rule for the expression $\mathtt{E}_1 - \mathtt{E}_2$.

Exercise 2 : structural operational semantics

Q1.
We want to add the following statement to the **While** language:

$$\mathsf{repeat}\ S\ \mathsf{until}\ b$$

The informal semantics of this construct is that the statement S should be executed until the Boolean condition b becomes true.
Provide the structural operational semantics rules in order to define $\mathsf{repeat}\ S\ \mathsf{until}\ b$ without using the $\mathsf{while\ b\ do} \cdots \mathsf{od}$ construction.

Q2.
We want to add another iterative construct to the **While** language. Consider the statement

$$\mathsf{for}\ x\ \mathsf{from}\ a_1\ \mathsf{to}\ a_2\ \mathsf{do}\ S.$$

where the first expression a_1 is the initial value that x is assigned to, the second expression is the limit that x should be assigned to. Moreover, the “step” of the loop is fixed.
The purpose of this exercise is to extend the semantics of the **While** language by providing appropriate rule(s) for this construct (and without using the $\mathsf{while} \cdots\ \mathsf{do} \cdots \mathsf{od}$ construct).
Some constraints/guidelines:

- Evaluation of a_1 and a_2 are done each time the loop body is executed.
- S is allowed to modify x.
- x has been declared previously in the program
- x may have a value before entering the for loop