Series 2

Typing

Exercise 15 (Program correctly typed or not?) Consider the environment $\Gamma = [x_1 \mapsto \texttt{Int}, x_2 \mapsto \texttt{Int}, x_3 \mapsto \texttt{Bool}]$. Indicate whether the following programs are correctly typed or not.

1. Program 1:
   $x_1 := 3;$
   $\texttt{while not} x_3 \texttt{ do}$
   $\quad x_1 := x_2 + 1;$
   $\quad x_3 := x_3 \texttt{ and true}$
   $\texttt{od}$
2. Program 2:
   $x_1 := 3 * x_1 + 1;$
   $\texttt{if } x_2 \texttt{ and } \neg x_3 \texttt{ then}$
   $\quad x_1 := x_2 + 1$
   $\texttt{else}$
   $\quad x_1 := x_2;$
   $\texttt{fi}$

Exercise 16 (Sequential or Collateral evaluation in declarations) Consider the sequence of variable declarations $D_V = \mathsf{var}\ x_1 := 3; \mathsf{var}\ x_2 := 2 * x_1 + 1; \mathsf{var}\ x_3 := \mathsf{true}$ and the initial environment $\Gamma_V = [\,]$.

1. Compute the resulting environment by updating Γ_V with D_V using *sequential* evaluation.
2. Compute the resulting environment by updating Γ_V with D_V using *collateral* evaluation.

Exercise 17 (Adding a typing rule for a new construct) We are interested in the construct/expression "$a_1 \ ?\ a_2 : a_3$" which is available in C or Java. The informal semantics of this construct is as follows: if a_1 is true then the value of this expression is a_2 else the value is a_3.

1. Complete the abstract syntax of expressions to support this construct.
2. Give typing rules for this construct.

Exercise 18 (Introducing floats and type conversion) We want to add the type `Float` to the **While** language.

1. Complete the abstract syntax and the type system to support the type `Float` where no conversion is allowed between `Int` and `Float`.
2. Complete the type system to allow *implicit* conversion from `Int` to `Float`.
3. Complete the abstract syntax and the type system to allow the *explicit* conversion from `Int` to `Float` through an appropriate type conversion operator.

Exercise 19 (Typing rules for the for and repeat constructs) We add two new statements to the **While** language (introduced in the lecture session):

- A "repeat" statement: `repeat` S `until` b
- A "for" statement: for x from e_1 to e_2 do S

1. Give the typing rule(s) associated to the "repeat" statement.
2. Give the typing rule(s) associated to the "for" statement. You will distinguish between two cases:
   - the "for" statement *declares* the variable x (like in Ada or Java), the scope of this new variable is S ;
   - the "for" statement *does not declare* the variable x (like in C), and therefore x has to exist in the current environment.

Exercise 20 (Other forms of variable declarations) Modify the type system seen in the course when variable declarations can take the following additional forms.

1. `var` $x\ :\ t$
2. `var` $x\ :=\ e\ :\ t$

Exercise 21 (Type-checking a program) We consider the type system seen in the course and the following program.

```
begin
  var x := 3
  proc p is x := x + 1
  proc q is call p
  begin
    proc p is x := x + 5
    call q
    call p
  end
  call p
end
```

1. Determine whether this program is correctly type in the case of *static* binding for variables and procedures.
2. Determine whether this program is correctly type in the case of *dynamic* binding for variables and procedures.

Exercise 22 (Mutually recursive procedures) We consider the program below.

```
begin
   proc p1 is
        call p2 ;
   proc p2 is
        call p1 ;
   call p1 ;
end
```

1. Show that, with the type system defined so far for the **Proc** language, the program is *incorrect*.
2. Modify this type system to take into account such *mutually recursive* procedures. Verify that this program is now correct with the new type system.
   Clue. Each sequence of procedure declaration should be analyzed twice: a first time to build its associated local environment, and a second time to check its correctness with respect to this local environment.

Exercise 23 (Correctly initialized variables) A variable is said to be *correctly initialized* if it is never *used* before being assigned with an expression containing only correctly initialized variables. Let us consider for instance the following program:

```
x := 0 ; y := 2 + x ; z := y + t ; u := 1 ; u := w ; v := v+1 ;
```

In this program:

- `x` and `y` are correctly initialized ;
- `z` is not correctly initialized (because `t` is not correctly initialized); `u` is not correctly initialized (because `w` is not correctly initialized); and `v` is not correctly initialized (because `v` is not correctly initialized).

Some compilers, such as `javac`, reject programs that contain non correctly initialized variables. We want to define in this exercise a type system which formalizes this check. To do so, we consider the following judgments:

- an environment is simply a set V of correctly initialized variables;
- $V \vdash e$ means that "in the environment V, expression e is correct (it does not contain non correctly initialized variables)";
- $V \vdash S \mid V'$ means that "in the environment V, statement S is correct and produces the new environment V' ".

1. Give the corresponding type system for the **While** language (without blocks nor procedures).
2. Apply the type system to the following code snippet, using $\Gamma = \emptyset$:
   a) $x := 1$; $\texttt{if}\ x = 0\ \texttt{then}\ y := x + 1\ \texttt{else}\ y := x - 1$,
   b) $x := 1$; $\texttt{if}\ x = 0\ \texttt{then}\ x := x + 1\ \texttt{else}\ y := x - 1$,
   c) $x := 1$; $\texttt{while}\ x \leq 10\ \texttt{do}\ y := x + y\ ;\ x := x + 1$.
3. Show (on an example) that, similarly to `javac`, your type system may reject programs that would be correct at run-time.

Exercise 24 (Procedures with one parameters) We consider the following modified abstract syntax where procedures can have one parameter:

$$
\begin{array}{lcl}
Dp & ::= & \texttt{proc}\ p\ (y\ :\ t)\ \texttt{is}\ S\ ; Dp \mid \ldots \\
S & ::= & \ldots \mid \texttt{call}\ x\ (e)
\end{array}
$$

1. Modify the type system to handle procedures with one parameter.
2. Use the extended type system to prove that the following program is correctly typed.

```
begin
  var x := 3
  proc p (u : int) is x := u + 1
  begin
    var x := true
    proc p (u : bool) is not u
    call p (x)
  end
  call p (x)
end
```

Exercise 25 (Considering functions) We extend language **Proc** to handle procedures that return value, aka functions. This entails that functions can be called within expressions.

1. Extend the abstract grammar of **Proc**.
2. Extend the type system of **Proc** accordingly.

Exercise 26 (Adding parameters to procedures in the type system) We aim at extending the **While** language to add *parameters* to procedures. We shall proceed in several steps.

1. Consider only `in` parameters;
2. Consider both `in` and `out` parameters;
3. Take into account the extra rule (inspired from the Ada language), stating that:
   - `out` parameters cannot appear in right-hand side of an assignment;
   - `in` parameters cannot appear in left-hand side of an assignment.
4. Show that, in this last case, your type system may *reject* correct programs because of this rule. How could you solve this problem?

Exercise 27 (Sub-typing and dynamic types) We extend the **While** language by introducing the notion of *sub-typing* through the following syntax for blocks, where t is a **type identifier** and `extends` means "is a sub-type of" (like in Java):

$$\begin{array}{rcl} S & ::= & \cdots \mid \texttt{begin}\ D_T\ ;\ D_V\ ;\ S\ \texttt{end} \\ D_T & ::= & \texttt{type}\ t\ \texttt{extends}\ B_T; D_T \mid \varepsilon \\ B_T & ::= & \texttt{Top}\ \mid \texttt{Int}\ \mid \texttt{Bool}\ \mid t \end{array}$$

We aim to define a type system for this language which reflects the usual notion of sub-typing, namely:

- The sub-typing relation is a partial order $\sqsubseteq$ whose greatest element is `Top`. It can be formalized by a *type hierarchy* $(X, \sqsubseteq)$, where X is a set of declared types (including the predefined types `Top`, `Int` and `Bool`).
- A value of type t_2 can be assigned to a variable of type t_1 whenever $t_2 \sqsubseteq t_1$. The converse is false.

1. Propose a type system which takes these rules into account. Judgments could be of the form:
   - $(X, \sqsubseteq), \Gamma \vdash S$, meaning that "in the environment Γ and with the type hierarchy $(X, \sqsubseteq)$, the statement S is well-typed" ;
   - $(X, \sqsubseteq), \Gamma \vdash e : t$, meaning that "in the environment Γ and with the type hierarchy $(X, \sqsubseteq)$, the expression e is well-typed and of type t" ;

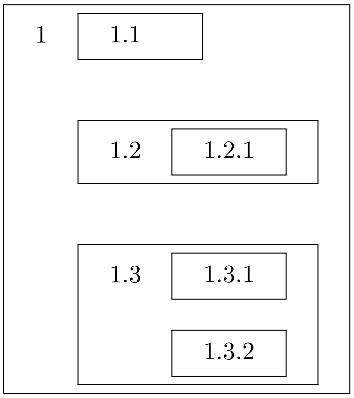
- $(X, \sqsubseteq) \vdash D_T \mid (X', \sqsubseteq')$, meaning that "type declaration D_T is correct within the type hierarchy $(X, \sqsubseteq)$ and produces the type hierarchy $(X', \sqsubseteq')$" ;
- $(X, \sqsubseteq), \Gamma \vdash D_V \mid \Gamma_l$, meaning that "in the environment Γ and with the type hierarchy $(X, \sqsubseteq)$, the variable declaration D_V is correct and produces the environment Γ_l".

2. Show that the following program is rejected by your type system:

```
begin
    type t extends Int ;
    var x1 : Int ;
    var x2 : t ;
    var x3 : Int ;
    x1 := x2 ;
    x3 := x1 ;
    x2 := x3
end
```

3. Although rejected by your type system, the previous program is perfectly safe (it does not violate the informal sub-typing rules). However, its correctness can only be ensured at run-time, by introducing a notion of *dynamic type* to each identifier. This dynamic type corresponds to the actual type value held by this identifier at each program step (contrarily to the *static type*, the one *declared* for this variable).
   Rewrite the (natural) operational semantics of the **While** language to take into account this notion of *dynamic type* and perform the type-checking at run-time. You can extend the configurations with a (dynamic) environment ρ which associates its dynamic type to each identifier.

Exercise 28 (Nested blocks and global environment) To define the type system of **Block** (possibly with nested blocks, but without procedures), we propose a notion of *global* environment in which each identifier is *uniquely* defined. More precisely, we assume a hierarchical numbering of blocks:



An environment now associates a type to a **pair** $(\mathbf{Var}, \mathbb{N}^*)$, and a statement is type-checked **within** a given block. Define the corresponding judgments and type system[1].

Exercise 29 (Static vs Dynamic Type system) We consider the following **While** program (with a command 'write'):

[1] $\mathbb{N}^*$ denotes the set of finite words over $\mathbb{N}$.

```
begin   var  x := 2;
        var  y := 1;
        proc p is x := x + y;
        begin  var  y := true;
               call p;
               write x;
        end;
end;
```

1. According to the static semantics for variables and procedures, what does this program write?
2. Is this program well-typed in the static semantics type system?
3. According to the dynamic semantics for variables and procedures, what happens with this program?
4. Is this program well-typed in the dynamic semantics type system ? We deduce that even if a program is well-typed in the static type system, it does not matter when we want to execute it with a dynamic semantics!
5. Propose a modification of this program which is well-typed in the dynamic semantics type system, and which displays a Boolean.
6. (optional) If you master the static-dynamic semantics, you can try to exhibit a program which is well-typed in the dynamic type system but not in the static-dynamic type system. You can use a second procedure q.