UFR-IMAG
Université Joseph Fourier

Programming Language and Compiler Design,
2009/2010
Marion Daubignard
Yassine Lakhnech
Laurent Mounier

# Series 1

## Exercise 1

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 `in` e1 ... e2 `do` S

Give the typying rules associated to each of this statement. For the "for" statement you will distinguish between two cases:

- the "for" statement *declares* variable $x$ (like in Ada or Java), the scope of this new variable is $S$ ;
- the "for" statement *does not declare* variable $x$ (like in C or Java), and therefore $x$ has to be present in the current environment.

## Exercise 2

Show that, with the type system defined so far for the `while` language, the following program is **incorrect**:

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

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 ckeck its correctness with respect to this local environment.

## Exercise 3

A variable is said *correctly initialized* if it is never *used* before having 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 initalized) ; `u` is not correctly initialized (because `w` is not correctly initalized) ; and `v` is not correctly initialized (because `v` is not correctly initalized).

Some compilers, like `javacc` reject programs that contain non correctly initialized variables. We want to define in this exercice a type system which formalizes this check. To do so, we consider the folloging 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 initalized variables)" ;
- $V \vdash S \mid V'$ means that "in the environment $V$, statement $S$ is correct and produces the new environment V'".

Give the corresponding type system for the `while language` (without blocks nor procedures).

Show (on an example) that, like `javac`, your type system may reject programs that would be correct at run-time.

# Exercise 4

Extend the while language to add **parameters** to the procedures. You will 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), saying that:
   - `out` parameters cannot appear in right-hand side of an assignment ;
   - `in` parameters cannot appear in left-hand side of an assignment.

Show that for this last case your type system may **reject** correct programs with respect to this rule. How could you solve this problem ?

# Exercise 5

We extend the `while` language by introducing notion of **subtyping** through the following syntax for blocks, where $t$ is a **type indentifier** and `extends` means "is a subtype of" (like in Java):

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

We would like to define a type system for this language which reflects the usual notion of subtyping, namely:

- The subtyping relation is a partial order $\sqsubseteq$ those greatest element is Top. It can be formalized by a **type hirerarchy** $(X, \sqsubseteq)$, 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 $\Gamma$ and with the type hierarchy $(X, \sqsubseteq)$, statement $S$ is well-typed" ;
   - $(X, \sqsubseteq), \Gamma \vdash e : t$, meaning that "in the environment $\Gamma$ and with the type hierarchy $(X, \sqsubseteq)$, statement $e$ 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 $\Gamma$ and with the type hierarchy $(X, \sqsubseteq)$, variable declaration $D_V$ is correct and produces the environment $\Gamma_l$".
2. Show that the following program is rejected by your type system:

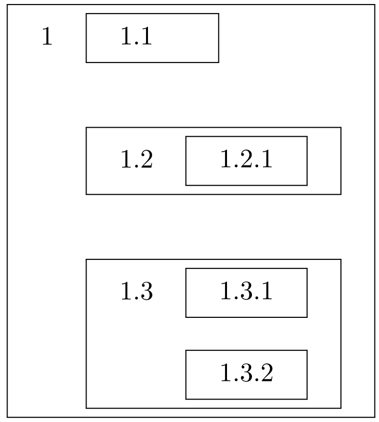
```
begin
    type t extends Int ;
    var x1 : Int ;
    var x2 : t ;
    var x3 : Int ;
    x1 := x2 ;
    x3 := x1 ;
    x2 := x3
end
```

3. Although rejected by your type system, the previous program is perfectly safe (it does not violate the informal subtyping 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 value type hold by this identifier at each program step (contrarily to the **static type**, the one *declared* for this variable).

   Rewrite the (natural) operationnal 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 $\rho$ which associates its dynamic type to each identifier.

# Exercise 6

To define the type system of the `while` language (with possible nested blocks, but without procedures) we propose a notion of **global** environment in which each identifier is **uniquely** defined. More precisely, we assume a hierarchal numbering of blocks:




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

# Exercise 7

We consider the small functional language introduced during the lecture course. Discuss the correctness of the following terms both in the F system and in the Hindley-Milner system:

1. `let` $f =$ `fun` $x.(x\ ,\ x)$ `in` $(f\ (1\ ,\ true))$
2. `let` $f =$ `fun` $x.x$ `in` $((f\ 1)\ ,\ (f\ true))$