Programmation 1

TD n°11

1er décembre 2020

Exercise 1 :

Consider the following PCF expression u

```
letrec f (x) = 3 in
letrec g (x) = g (x) in
f (g 0)
```

1. This is not a valid expression because the type annotations are missing. Add them.
2. Calculate the denotational semantics of u.

Exercise 2 :

For each OCaml expression below, give the type of the expression, if it exists. Justify.

1. `let f x = x in (f 3, f "trois")`
2. `(fun f -> (f 3, f "trois")) (fun x -> x)`
3. `let f x = x in let g = ref f in (!g 3, !g "trois")`

Exercise 3 :

We consider the following language

$$M := x \mid \lambda x : \tau.M \mid MN \mid \textbf{let } x : \tau = M \textbf{ in } N \mid \textbf{ff} \mid \textbf{tt} \mid \textbf{if } M \textbf{ then } N \textbf{ else } P$$

1. Propose an adapted typing system.
2. Give a derivation of $\vdash (\lambda x.\textbf{if } x \textbf{ then ff else } x)\textbf{tt} : \textbf{bool}$
3. Which element of the programming language syntax is crucial to guarantee typing determinism ? Explain with an example.
4. Show that the `let` is encoded using the other constructs in a well-typed way.
5. Propose small-step semantics for this language.
6. Show that there is a theorem of *subject reduction*, that is, small-step semantics preserves typing.
7. We add to the syntax the following two constructions

   $$\textbf{try } M \textbf{ with } N \mid \textbf{abort}$$

   Propose an extension of the typing system.
8. Propose an extension of the small step semantics.

Exercise 4 :

We add exceptional constructors that we denote as $C_1, \ldots, C_n$. These are for example exceptions like `KeyboardInterrupt`. For each C_i, we consider a type τ_i of fixed argument and we add the rules of deductions

$$\frac{}{C_i : \tau_i \to \textbf{exn}}$$

1. Adapt the syntax. What are the values ? What are the contexts ?
2. Adapt the small-step semantics.
3. Use it to reduce the next term assuming that $M \to^* V$.

$$\textbf{try } (\lambda x.\lambda y.y)(\textbf{abort } M) \textbf{ with } C_i(x) \mapsto x$$

4. OCaml language prohibits building exceptions possessing a polymorphic type. Explain.