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.

Solution:

1. letrec $f_{\mathsf{int}\to\mathsf{int}}$ $(x_{\mathsf{int}}) = \dot{3}$ in
   letrec $g_{\mathsf{int}\to\mathsf{int}}$ $(x_{\mathsf{int}}) = g_{\mathsf{int}\mapsto\mathsf{int}}$ (x_{int}) in
   $f_{\mathsf{int}\to\mathsf{int}}$ $(g_{\mathsf{int}\to\mathsf{int}}$ $\dot{0})$
2. Using the rule

$$[\![\mathsf{letrec}\ f_{\sigma\to\tau}(x_\sigma) = u\ \mathsf{in}\ v]\!]\rho = [\![v]\!](\rho[f_{\sigma\to\tau} \mapsto \mathrm{lfp}(F^\rho_{f_{\sigma\to\tau},x_\sigma,u})])$$

   where $F^\rho_{f_{\sigma\to\tau},x_\sigma,u}(\varphi) = (V \in [\![\sigma]\!] \mapsto [\![u]\!](\rho[f_{\sigma\to\tau} \mapsto \varphi, x_\sigma \mapsto V]))$.
   We obtain that $[\![u]\!]\rho = 3$ for all environments ρ.

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")`

Solution:

1. The type is `int * string` .
2. This does not type, because the generalization only applies to `let` thus the function `fun x -> x` is not generalized.
3. We trigger the "value restriction". It is important because otherwise we can do things like

```
let r = (fun x -> ref x) [];;
r := [ 1 ];;
let cond = (!r = [ "foo" ]);;
```

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 $\vdash (\lambda x : \textbf{bool}.\textbf{if } x \textbf{ then ff else } x)\textbf{tt} : \textbf{bool}$
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.

Solution:

1.
$$\frac{}{\Gamma \vdash \textbf{tt} : \textbf{bool}} \qquad \frac{}{\Gamma \vdash \textbf{ff} : \textbf{bool}} \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma . M : \sigma \to \tau}$$

$$\frac{\Gamma \vdash P : \textbf{bool} \qquad \Gamma \vdash M : \tau \qquad \Gamma \vdash N : \tau}{\Gamma \vdash \textbf{if } P \textbf{ then } M \textbf{ else } N : \tau}$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma, x : \sigma \vdash N : \tau}{\Gamma \vdash \textbf{let } x = M \textbf{ in } N : \tau}$$

2. It can be shown using the rules above.
3. The fact that the types are in the syntax. That is, type inference is not deterministic, the type *erasure* loses information. For example, $\lambda x.x$.
4. We write $\textbf{let } x = M \textbf{ in } N \triangleq (\lambda x.N)M$.
5.

$$\begin{aligned} &(\lambda x : \tau . M)N \to M[N/x] \\ &\textbf{let } x : \tau = M \textbf{ in } N \to N[M/x] \\ &\textbf{if tt then } M \textbf{ else } N \to M \\ &\textbf{if ff then } M \textbf{ else } N \to N \end{aligned}$$

   and $M \to N$ implies $C[M] \to C[N]$ for all contexts C.
6. This is done by induction on the typing derivation.
7. We give to Abort the type **exn** and

$$\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash N : \tau}{\Gamma \vdash \textbf{try } M \textbf{ with } N : \tau}$$

8. We add the rules

$$\begin{aligned} \textbf{try abort with } M &\to M \\ \textbf{try } V \textbf{ with } M &\to V \end{aligned}$$

   and the following context

$$\textbf{try } C \textbf{ with } M$$

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

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

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 $M \to^* V$.

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

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

Solution:

1. Values are closed terms of the form $\lambda x.f$, **tt** ou **ff**. Exceptions are not considered as values since they will be executed in a context. The contexts are :

$$C := C \mid \mathbf{try}\ C\ \mathbf{with}\ M \mid \mathbf{abort}\ C \mid VC \mid CM \mid \mathbf{if}\ C\ \mathbf{then}\ M\ \mathbf{else}\ M$$

2. Small-step semantics adapts as follows

$$\begin{aligned} \mathbf{try}\ (\mathbf{abort}\ (C_i V))\ \mathbf{with}\ C_i(x) \mapsto N &\to N[x/M] \\ F[\mathbf{abort}\ V] &\to \mathbf{abort}\ V \\ \mathbf{try}\ V\ \mathbf{with}\ M &\to V \end{aligned}$$

3. Trivially reduces to V.
4. It is sufficient to imagine the type $\tau_i \triangleq \forall \alpha.\alpha$.
   So we lose subject reduction as shown by the following term :

$$\mathbf{try}\ \mathbf{abort}\ (C_i(\mathbf{tt})); 1\ \mathbf{with}\ C_i(x) \mapsto x + 1$$