Appendix D

Corrigés des exercices du chapitre 5

Exercice 5.1 La fonction `f` calcule la fonction factorielle. En effet, une fois que l'on a évalué l'affectation

```
r := fun x → if x = 0 then 1 else x * (!r)(x-1)
```

on a bien que la fonction $g = !\texttt{r}$ est égale à `fun x → if x = 0 then 1 else x * (!r)(x-1)`, c'est-à-dire `fun x → if x = 0 then 1 else x *` g`(x-1)`.

Exercice 5.2 Voici l'opérateur de point fixe pour les fonctions des entiers dans les entiers:

```
let fix = fun f →
                let r = ref(fun x → 0) in
                r := (fun x → f (!r) x);
                (!r)
```

On peut le généraliser pour qu'il opère sur toutes les fonctions à condition de se donner une expression ω de type $\forall\alpha.\alpha$ et qui bien sûr ne termine pas (p.ex. une boucle infinie ou encore une levée d'exception `raise` E):

```
let fix = fun f →
                let r = ref(fun x → ω) in
                r := (fun x → f (!r) x);
                (!r)
```

Le `fix` ci-dessus a le type $\forall\alpha,\beta.\ ((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)$.

L'opérateur `fix` sur des types non-fonctionnels (p.ex. (`int` → `int`) → `int`) n'est pas définissable avec des références.

Exercice 5.3

$$c/s \xrightarrow{v} c/s \qquad op/s \xrightarrow{v} op/s \qquad (\texttt{fun}\ x \to a)/s \xrightarrow{v} (\texttt{fun}\ x \to a)/s$$

$$\frac{a_1/s \xrightarrow{v} (\texttt{fun}\ x \to a)/s_1 \qquad a_2/s_1 \xrightarrow{v} v_2/s_2 \qquad a[x \leftarrow v_2]/s_2 \xrightarrow{v} v/s_3}{a_1\ a_2/s \xrightarrow{v} v/s_3}$$

$$\frac{a_1/s \xrightarrow{v} v_1/s_1 \qquad a_2/s_1 \xrightarrow{v} v_2/s_2}{(a_1, a_2)/s \xrightarrow{v} (v_1, v_2)/s_2} \qquad \frac{a_1/s \xrightarrow{v} v_1/s_1 \qquad a_2[x \leftarrow v_1]/s_1 \xrightarrow{v} v/s_2}{(\texttt{let } x = a_1 \texttt{ in } a_2)/s \xrightarrow{v} v/s_2}$$

$$\frac{a/s \xrightarrow{v} v/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{(\texttt{ref}(a))/s \xrightarrow{v} \ell/s_1 + [\ell \leftarrow v]} \qquad \frac{a/s \xrightarrow{v} \ell/s_1 \qquad \ell \in \mathrm{Dom}(s_1)}{!a/s \xrightarrow{v} s_1(\ell)/s_1}$$

$$\frac{a_1/s \xrightarrow{v} \ell/s_1 \qquad \ell \in \mathrm{Dom}(s_1) \qquad a_2/s_1 \xrightarrow{v} v_2/s_2}{s, (a_1 := a_2) \xrightarrow{v} ()s_2 + [\ell \leftarrow v_2]}$$

Exercice 5.4 L'eta-expansion a pour effet de changer l'ordre dans lequel une fonction entrelace passage d'arguments et calculs internes. Par exemple:

```
let f = fun x →
          print_string "f";
          fun y → (x, y) in
let g = f 1 in
(g true, g "hello")
```

La chaîne `f` est affichée une seule fois, alors que si l'on fait une eta-expansion dans la définition de `g` (afin de la rendre non-expansive), on obtient:

```
let f = fun x →
          print_string "f";
          fun y → (x, y) in
let g = fun y → f 1 y in
(g true, g "hello")
```

et maintenant `f` est affichée deux fois. Toute fonction qui effectue des effets de bords entre le passage de deux arguments permet donc d'observer une différence de comportement lorsqu'on eta-expanse une application partielle de cette fonction. De tels exemples n'apparaissent cependant que très rarement en pratique. Un peu plus plausible est l'exemple de fonctions qui font une grande quantité de calculs (sans effets de bord) entre le passage de deux arguments. Là, l'eta-expansion va changer non pas la sémantique du programme, mais son temps d'exécution. Exemple: on considère une fonction qui trie des tables (clé, valeurs) par ordre croissant des clés. On peut vouloir l'écrire sous la forme d'une fonction qui prend d'abord le tableau des clés, calcule la permutation qui trie ces clés, puis prend un tableau de valeurs et y applique la permutation:

```
let tri_table = fun table →
                  let perm = trier table in
                  fun données → appliquer_permutation perm données
```

Cette écriture est avantageuse si l'on s'attend à avoir plusieurs tables de valeurs qui partagent les mêmes clés: on peut alors calculer la permutation de tri une seule fois.

```
let fonction_de_tri = tri_table clés in
... fonction_de_tri données1 ... fonction_de_tri données2 ...
```

Avec la restriction de la généralisation aux expressions non-expansives, `fonction_de_tri` devient monomorphe, et donc l'écriture ci-dessus n'est possible que si `données1` et `données2` sont du même type. Mais sinon, il faut faire une eta-expansion sur `fonction_de_tri`:

```
let fonction_de_tri = fun d → tri_table clés d in
... fonction_de_tri données1 ... fonction_de_tri données2 ...
```

et la permutation de tri est recalculée à chaque appel de `fonction_de_tri`.

Exercice 5.5 On montre d'abord la conservation du typage par réductions. Pour $(\texttt{try}\ v\ \texttt{with}\ x \to a) \xrightarrow{\varepsilon} v$, il est clair que si $E \vdash (\texttt{try}\ v\ \texttt{with}\ x \to a) : \tau$, alors $E \vdash v : \tau$ également. Pour $(\texttt{try}\ \Delta(\texttt{raise}(v))\ \texttt{with}\ x \to a) \xrightarrow{\varepsilon} a[x \leftarrow v]$, une dérivation de typage du membre gauche est nécessairement de la forme

$$\dfrac{\dfrac{\dfrac{E \vdash \texttt{raise} : \texttt{exn} \to \tau' \qquad E \vdash v : \texttt{exn}}{E \vdash \texttt{raise}(v) : \tau'}}{\vdots} \qquad E + \{x : \texttt{exn}\} \vdash a : \tau}{E \vdash (\texttt{try}\ \Delta(\texttt{raise}(v))\ \texttt{with}\ x \to a) : \tau}$$

En appliquant le lemme de substitution (proposition 2.2) à $E \vdash v : \texttt{exn}$ et $E + \{x : \texttt{exn}\} \vdash a : \tau$, il vient $E \vdash a[x \leftarrow v] : \tau$ comme désiré.

Il faut ensuite montrer une propriété de progression similaire à la proposition 2.6. Le problème, ici, est que l'évaluation peut s'arrêter à cause d'une exception non rattrapée, c'est-à-dire lorsqu'on évalue une sous-expression `raise` v qui n'est contenue dans aucun `try...with`. La propriété de progression est donc:

> Si $\emptyset \vdash a : \tau$, alors ou bien a est une valeur, ou bien $a = \texttt{raise}(v)$ pour une certaine valeur v, ou bien il existe un terme a' tel que $a \to a'$.

La preuve de ce lemme est semblable à celle de la proposition 2.6, avec des cas supplémentaires montrant que si une sous-expression est $\texttt{raise}(v)$, alors l'expression tout entière se réduit, soit par la règle pour $\texttt{try}\ \texttt{raise}(v)\ \texttt{with}\ldots$, soit par la règle pour $\Delta(\texttt{raise}(v))$.

Finalement, on obtient qu'une expression a close et bien typée se réduit en une valeur, ou se réduit en $\texttt{raise}(v)$, ou diverge.

Exercice 5.6 Avec les exceptions, on peut écrire une fonction qui a plusieurs résultats: son résultat "normal" (la valeur du corps de la fonction) plus un autre résultat qui est renvoyé en levant une exception contenant cet autre résultat dedans. Autant le type du résultat "normal" est apparent dans le type de la fonction (et ne peut donc être plus complexe que le type de la fonction elle-même), autant le type du résultat "exceptionnel" n'est pas visible dans le type de la fonction et peut être plus complexe que ce dernier. En appliquant cette idée, on peut "cacher" une fonctionnelle de type $(\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$ à l'intérieur d'une humble fonction de type $\texttt{int} \to \texttt{int}$:

```
exception M of (int → int) → (int → int)

let cacher f = fun n → (raise (M f); 0)
```

La fonction f cachée dans le résultat de `cacher` f peut ensuite être extraite par la fonction suivante:

```
let retrouver g = try (g 0; fun z → z) with M f → f
```

Ceci permet d'exprimer l'exemple bien connu de λ-terme qui boucle $(\lambda f.f\ f)\ (\lambda f.f\ f)$ de la manière suivante:

```
let delta = fun f → (retrouver f) f in delta (cacher delta)
```

On peut aussi définir l'opérateur de point fixe `fix` $= \lambda f.\ (\lambda g.\lambda x.f\ (g\ g)\ x)\ (\lambda g.\lambda x.f\ (g\ g)\ x)$ comme suit:

```
let fix = fun f →
            let d = fun g → fun x → f (retrouver g g) x in
            d (cacher d)
```

Essayez en Caml; ça marche!

```
let fact = fix (fun f → fun n → if n = 0 then 1 else n * f(n-1))
in fact 5
```