Appendix C

Corrigés des exercices du chapitre 4

Exercice 4.1 La projection $\texttt{proj}_{i,n}$ se définit comme:

$$\begin{array}{rcl} \texttt{proj}_{n,n} & = & \texttt{fun}\ x \rightarrow \texttt{snd}^{n-1}(x) \\ \texttt{proj}_{i,n} & = & \texttt{fun}\ x \rightarrow \texttt{fst}(\texttt{snd}^{i-1}(x)) \quad \text{si } i < n \end{array}$$

La traduction T de ML_t dans ML_p:

$$\begin{array}{rcl} T(x) & = & x \\ T(c) & = & c \\ T(\texttt{proj}_{n,n}) & = & \texttt{fun}\ x \rightarrow \texttt{snd}^{n-1}(x) \\ T(\texttt{proj}_{i,n}) & = & \texttt{fun}\ x \rightarrow \texttt{fst}(\texttt{snd}^{i-1}(x)) \quad \text{si } i < n \\ T(op) & = & op \quad \text{pour les autres opérateurs } op \\ T((v_1, \ldots, v_n)) & = & (T(v_1), (T(v_2), \ldots (T(v_{n-1}), T(v_n)))) \\ T(\texttt{fun}\ x \rightarrow a) & = & \texttt{fun}\ x \rightarrow T(a) \\ T(a_1(a_2)) & = & T(a_1)(T(a_2)) \\ T(\texttt{let}\ x = a_1\ \texttt{in}\ a_2) & = & \texttt{let}\ x = T(a_1)\ \texttt{in}\ T(a_2) \end{array}$$

La commutation entre T et la réduction se montre d'abord pour chaque règle de réduction en tête. La plus intéressante est bien sûr la δ-règle pour les tuples: $\texttt{proj}_{i,n}(v_1, \ldots, v_n) \xrightarrow{\varepsilon} v_i$. Si $i = n$, on a:

$$\begin{array}{l} T(\texttt{proj}_{n,n}(v_1, \ldots, v_n)) \\ \quad = \quad (\texttt{fun}\ x \rightarrow \texttt{snd}^{n-1}(x))(T(v_1), (T(v_2), \ldots (T(v_{n-1}), T(v_n)))) \\ \quad \rightarrow \quad \texttt{snd}^{n-1}(T(v_1), (T(v_2), \ldots (T(v_{n-1}), T(v_n)))) \text{ par } \beta_{fun} \\ \quad \rightarrow \quad \texttt{snd}^{n-2}(T(v_2), \ldots (T(v_{n-1}), T(v_n))) \text{ par } \delta_{snd} \\ \quad \xrightarrow{*} \quad T(v_n) \text{ par } \delta_{snd} \end{array}$$

De même, si $i < n$, on a:

$$\begin{array}{l} T(\texttt{proj}_{n,n}(v_1, \ldots, v_n)) \\ \quad = \quad (\texttt{fun}\ x \rightarrow \texttt{fst}(\texttt{snd}^{i-1}(x)))(T(v_1), (T(v_2), \ldots (T(v_{n-1}), T(v_n)))) \end{array}$$

$$\begin{array}{ll}
\rightarrow & \texttt{fst}(\texttt{snd}^{i-1}(T(v_1),(T(v_2),\ldots(T(v_{n-1}),T(v_n))))) \text{ par } \beta_{fun} \\
\rightarrow & \texttt{fst}(\texttt{snd}^{i-2}(T(v_2),\ldots(T(v_{n-1}),T(v_n)))) \text{ par } \delta_{snd} \\
\stackrel{*}{\rightarrow} & \texttt{fst}(T(v_i),\ldots) \text{ par } \delta_{snd} \\
\rightarrow & T(v_i) \text{ par } \delta_{fst}
\end{array}$$

Le résultat est immédiat pour β_{fun} et β_{let} car T commute avec la substitution: $T(a[x \leftarrow b]) = T(a)[x \leftarrow T(b)]$. Enfin, pour la règle (contexte), on utilise la commutation de T avec l'application de contexte: $T(\Gamma(a)) = T(\Gamma)(T(a))$. D'où le résultat: si $a \rightarrow a'$ dans ML_t, alors $T(a) \stackrel{*}{\rightarrow} T(a')$ dans ML_p.

Réciproquement, si $T(a) \rightarrow a''$, a'' n'est pas nécessairement la traduction d'un terme a' tel que $a \rightarrow a'$. Exemple:

$$(\texttt{fun } x \rightarrow \texttt{snd}(\texttt{snd}(x)))(1,(2,3)) \rightarrow \texttt{snd}(\texttt{snd}(1,(2,3))) \rightarrow \texttt{snd}(2,3) \rightarrow 3$$

Le terme de gauche est la traduction de $\texttt{proj}_{3,3}(1,2,3)$, mais les deux étapes suivantes de la réduction ne sont pas des traductions de termes de ML_t. Si l'on ignore les projections non appliquées et que l'on prend une traduction plus directe des applications de projections:

$$\begin{array}{rcl}
T(\texttt{proj}_{n,n}(a)) & = & \texttt{snd}^{n-1}(a) \\
T(\texttt{proj}_{i,n}(a)) & = & \texttt{fst}(\texttt{snd}^{i-1}(a)) \quad \text{si } i < n
\end{array}$$

l'exemple devient plus intéressant:

$$\begin{array}{rcccc}
\texttt{proj}_{3,3}(1,2,3) & & \texttt{proj}_{2,2}(2,3) & & 3 \\
\downarrow T & & \downarrow T & & \downarrow T \\
\texttt{snd}(\texttt{snd}(1,(2,3))) & \rightarrow & \texttt{snd}(2,3) & \rightarrow & 3
\end{array}$$

Les étapes intermédiaires de la réduction dans ML_p sont maintenant des traductions de termes de ML_t, mais la réduction intermédiaire

$$\texttt{proj}_{3,3}(1,2,3) \rightarrow \texttt{proj}_{2,2}(2,3)$$

n'est pas une réduction valide de ML_t.

Exercice 4.2 Il suffit de définir `type bool = false of unit | true of unit`. L'opérateur de filtrage $F_{\texttt{bool}}$ permet de définir la conditionnelle comme suit:

$$\texttt{if } a \texttt{ then } b \texttt{ else } c = F_{\texttt{bool}}(a, (\texttt{fun } x \rightarrow b), (\texttt{fun } x \rightarrow c))$$

où x est une variable non libre dans b et c.

Exercice 4.3 Supposons qu'on laisse passer la déclaration suivante:

```
type bug = B of 'a
```

Si l'on donne à `B` le type $\forall \alpha.\alpha \rightarrow \texttt{bug}$, alors le code suivant est bien typé:

```
match B(true) with B(x) -> x+1
```

et pourtant il finit par évaluer $\texttt{true} + 1$.

Exercice 4.4 Voici quelques valeurs du type `t`:

```
C(fun x → x)
C(fun y → C(fun x → x))
C(fun y → C(fun x → y))
```

Les fonctions `enrouler` et `dérouler`:

```
let enrouler = fun f → C f
let dérouler = fun t → match t with C f → f
```

Le codage du λ-calcul pur:

$$\begin{array}{rcl} [\![x]\!] & = & x \\ [\![\lambda x.M]\!] & = & \texttt{enrouler}(\texttt{fun}\ x \to [\![M]\!]) \\ [\![M\ N]\!] & = & \texttt{dérouler}[\![M]\!]\ [\![N]\!] \end{array}$$

En corollaire, on voit qu'il existe des expressions de type `t` (la traduction de $(\lambda f.f\ f)(\lambda f.f\ f)$ p.ex.) dont l'évaluation ne termine pas, bien qu'elles soient construites sans un seul `let rec`. Un exemple qui ne passe même pas par le codage du λ-calcul pur est:

```
let delta t = dérouler t t in delta(enrouler delta)
```

Exercice 4.5
1) On définit la substitution $F(p, v)$ comme suit:

$$\begin{array}{rcl} F(\_, v) & = & id \\ F(x, v) & = & [x \leftarrow v] \\ F(c, c) & = & id \\ F(C(p), C(v)) & = & F(p, v) \\ F((p_1, \dots, p_n), (v_1, \dots, v_n)) & = & F(p_1, v_1) + \dots + F(p_n, v_n) \end{array}$$

(Pour le dernier cas, on suppose qu'une même variable x n'apparait jamais deux fois dans un motif p.) Si aucun des cas ci-dessus ne s'applique, $F(p, v)$ est indéfini. La réduction du `match` est alors:

$$\begin{array}{rcl} (\texttt{match}\ v\ \texttt{with}\ p \to a_2 \mid \_ \to a_3) & \stackrel{\varepsilon}{\to} & \sigma(a_2) \quad \text{si } \sigma = F(p, v) \text{ est défini} \\ (\texttt{match}\ v\ \texttt{with}\ p \to a_2 \mid \_ \to a_3) & \stackrel{\varepsilon}{\to} & a_3 \quad \text{si } F(p, v) \text{ est indéfini} \end{array}$$

2) On définit l'énoncé de typage auxiliaire $\vdash p : \tau, E$ ("le motif p filtre des valeurs de type τ, et ce faisant il lie les variables $x \in \mathrm{Dom}(E)$ à des valeurs de type $E(x)$").

$$\vdash \_ : \tau, \emptyset \qquad \vdash x : \tau, \{x : \tau\} \qquad \frac{\tau' \to \tau \leq TC(C) \qquad \vdash p : \tau', E}{\vdash C(p) : \tau, E}$$

$$\frac{\vdash p_1 : \tau_1, E_1 \qquad \dots \qquad \vdash p_n : \tau_n, E_n}{\vdash (p_1, \dots, p_n) : \tau_1 \times \dots \times \tau_n, E_1 + \dots + E_n}$$

La règle de typage du `match` est alors:

$$\frac{E \vdash a_1 : \tau' \qquad \vdash p : \tau', E' \qquad E + E' \vdash a_2 : \tau \qquad E \vdash a_3 : \tau}{E \vdash (\texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid \_ \to a_3) : \tau}$$

3) On traduit ($\texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid \_ \to a_3$) en $\texttt{let}\ x = a_1\ \texttt{in}\ C(x, p, a_2, a_3)$, où le schéma de compilation C est défini comme suit:

$$\begin{array}{rcl}
C(a, _, b, c) & = & b \\
C(a, x, b, c) & = & \texttt{let}\ x = a\ \texttt{in}\ b \\
C(a, C(p), b, c) & = & F_t(a, \underbrace{f_n, f_n, \ldots, f_n}_{i-1 \text{ fois}}, f_o, f_n, \ldots, f_n) \\
 & & \text{si } C \text{ est le } i^{\text{e}} \text{ constructeur du type } t \\
 & & \text{et } f_n = \texttt{fun}\ x \to c \ \ x \text{ non libre dans } c \\
 & & \text{et } f_o = \texttt{fun}\ x \to C(x, p, b, c) \ \ x \text{ non libre dans } b \text{ et } c \\
C(a, (p_1, \ldots, p_n), b, c) & = & C(\texttt{proj}_{1,n}(a), p_1, C(\texttt{proj}_{2,n}(a), p_2, \ldots, C(\texttt{proj}_{n,n}(a), p_n, b, c)))
\end{array}$$

Pour plus d'explications, on se reportera aux chapitres 4 et 5 du livre de S.L.Peyton-Jones, *The implementation of functional programming languages*, Prentice-Hall, 1987.