Fondements de la programmation

Examen final - 20 décembre 2017 - 9-12h - Amphi D

Les notes de cours sont autorisées.

Machines RAM Parallèles: CRAM

On se propose de décrire une CRAM qui calcule la valeur maximale d'un tableau d'entiers donné en entrée, en temps parallèle constant. Les valeurs entières du tableau sont données par les compteurs $G_1, \cdots, G_n$. Le compteur d'entrée G_0 contient initialement la valeur n, c'est-à-dire la taille du tableau.

Pour simplifier l'exercice, on supposera que le PID d'un processeur est donné par une unique paire d'entiers (i, j) non nuls, et que l'on dispose des deux instructions suivantes dans le langage:

$$\begin{aligned} X_n &:= PID_i \\ X_n &:= PID_j \end{aligned}$$

qui donnent la valeur de chacune des deux composantes du PID d'un processeur.

L'idée de l'algorithme est la suivante:

1. Les compteurs $X_1, \cdots, X_n$ contiennent initialement la valeur 0
2. Les processeurs de PID (i, j) comparent en parallèle les compteurs d'entrée G_i et G_j, en écrivent 1 dans le compteur X_i si et seulement si $G_i < G_j$.
3. Enfin, les processeurs de PID $(i, 1)$ recopient le compteur G_i dans le compteur G_0 si et seulement si le compteur X_i contient la valeur 0.

Question 1 (2 points)

Expliquer pourquoi l'algorithme proposé réalise la tâche prévue (renvoyer la valeur maximale du tableau), et pourquoi le temps de calcul parallèle est constant.

Corrigé. Les compteurs X_i sont supposés globaux.

L'étape de la ligne 2 assure qu'un compteur global X_i reçoit la valeur 1 si et seulement si il existe j tel que $G_i < G_j$. Par conséquent, X_i conserve la valeur 0 si et seulement si G_i est maximal. L'étape de la ligne 3 recopie donc cette valeur maximale dans G_0. Le temps de calcul de chaque processeur est constant: l'algorithme ne contient aucune boucle. Les processeurs travaillent en parallèle, le temps d'exécution global de l'algorithme est constant. □

Question 2 (2 points)

Ecrire le programme CRAM qui réalise cet algorithme.

Corrigé. Pour simplifier l'écriture du programme, on se donne, en plus des registres X_i, un ensemble de registres locaux Y_i, $i \in \mathbb{N}$. Le programme est alors

$$
\begin{aligned}
P \quad = \quad & 1: Y_0 := 0, \\
& 2: Y_0 :=< Y_0 >^G, \ (Y_0 \textit{ reçoit la valeur } n \textit{ de } G_0) \\
& 3: Y_0 := Y_0 + 1, \\
& 4: Y_1 := PID_i, (Y_1 \textit{ reçoit } i) \\
& 5: Y_2 := PID_j, (Y_2 \textit{ reçoit } j) \\
& 6: Y_3 :=< Y_1 >^G, (Y_3 \textit{ reçoit } G_i) \\
& 7: Y_4 :=< Y_2 >^G, (Y_3 \textit{ reçoit } G_j) \\
& 8: \textit{if } Y_3 < Y_4 \textit{ goto } 9 \textit{ else } 10, \\
& 9: < Y_1 >^X := 1, (\textit{le compteur global } X_i \textit{ reçoit 1}) \\
& 10: \textit{if } Y_2 = 1 \textit{ goto } 11 \textit{ else } 16, \\
& 11: \textit{if } Y_1 < Y_0 \textit{ goto } 12 \textit{ else } 16, (\textit{on effectue les opérations suivantes que si } i < n+1) \\
& 12: Y_5 :=< Y_1 >^X, (Y_5 \textit{ reçoit } X_i) \\
& 13: \textit{if } Y_5 = 0 \textit{ goto } 12 \textit{ else } 16, \\
& 14: Y_6 := 0, \\
& 15: < Y_6 >^G :=< Y_1 >^G, (G_0 \textit{ reçoit } G_i) \\
& 16: \ \text{fin}
\end{aligned}
$$

□

Fonctions récursives primitives sur les notations: Palindrome

On rappelle la définition des fonctions récursives primitives sur les notations:

Pour un alphabet fini $A = \{a_1, \cdots, a_l\}$, on prend les fonctions de base suivantes:

- la fonction constante ϵ (le mot vide)
- Π, l'ensemble de fonctions de projection π_i^k, où $\pi_i^k(x_1, \cdots, x_k) = x_i$,
- S l'ensemble des fonctions successeur sur les mots de l'alphabet: Pour $i = 1, \cdots, l$, $s_i(x) = a_i.x$, le mot obtenu en plaçant la lettre a_i en tête du mot x.

Les fonctions récursives primitives sur les notations sont alors l'ensemble $PR_{not} = [\epsilon, \Pi, S; Comp, PR_{not}]$, où PR_{not} est le schéma de récursion primitive sur les notations suivant:

$$
\begin{aligned}
f(\epsilon, x_1, \cdots, x_n) &= g(x_1, \cdots, x_n) \\
f(s_1(x), x_1, \cdots, x_n) &= h_1(x, x_1, \cdots, x_n, f(x, x_1, \cdots, x_n)) \\
&\vdots \\
f(s_l(x), x_1, \cdots, x_n) &= h_l(x, x_1, \cdots, x_n, f(x, x_1, \cdots, x_n))
\end{aligned}
$$

On se propose dans ce problème d'écrire une fonction primitive récursive sur les notations, qui vérifie si son argument est un palindrome. On rappelle que mot sur un alphabet fini est un palindrome si et seulement si il est identique à son image renversée, c'est-à-dire que la suite des lettres de ce mot est la même, qu'on

le lise de gauche à droite ou de droite à gauche. Le fonction demandée doit renvoyer 1 si le mot donné en argument est un palindrome, et 0 sinon.

On se fixe pour ce problème l'alphabet fini binaire $A = \{0, 1\}$.

Le problème est décomposé en les questions suivantes.

Question 3 (1 point)

Ecrire une fonction récursive primitive sur les notations `concat`, sur l'alphabet A, telle que $\texttt{concat}(x, y)$ renvoie le mot $x.y$, c'est-à-dire la concaténation des mots x et y.

Corrigé.

$$\begin{aligned} \texttt{concat}(\varepsilon, y) &= y \\ \texttt{concat}(s_0(x), y) &= s_0(\texttt{concat}(x, y)) \\ \texttt{concat}(s_1(x), y) &= s_1(\texttt{concat}(x, y)) \end{aligned}$$

□

Question 4 (1 point)

Ecrire une fonction récursive primitive sur les notations `renverse`, sur l'alphabet A, qui renvoie l'image renversée de son argument; par exemple, $\texttt{renverse}(0.1.0.0)$ renvoie 0.0.1.0.

Corrigé.

$$\begin{aligned} \texttt{renverse}(\varepsilon) &= \varepsilon \\ \texttt{renverse}(s_0(x)) &= \texttt{concat}(\texttt{renverse}(x), s_0(\varepsilon)) \\ \texttt{renverse}(s_1(x)) &= \texttt{concat}(\texttt{renverse}(x), s_1(\varepsilon)) \end{aligned}$$

□

Question 5 (1 point)

Ecrire une fonction récursive primitive sur les notations `queue`, sur l'alphabet A, qui, sur un mot $x.M$, renvoie le mot M, et qui sur le mot vide ε, renvoie le mot vide ε.

Ecrire une fonction récursive primitive sur les notations `estvide`, sur l'alphabet A, qui, sur un mot $x.M$, renvoie 0, et qui sur le mot vide ε, renvoie 1.

Corrigé.

$$\begin{aligned} \texttt{queue}(\varepsilon) &= \varepsilon \\ \texttt{queue}(s_0(x)) &= x \\ \texttt{queue}(s_1(x)) &= x \end{aligned}$$

$$\begin{aligned} \texttt{estvide}(\varepsilon) &= s_1(\varepsilon) \\ \texttt{estvide}(s_0(x)) &= s_0(\varepsilon) \\ \texttt{estvide}(s_1(x)) &= s_0(\varepsilon) \end{aligned}$$

□

Question 6 (1 point)

Ecrire une fonction récursive primitive sur les notations $\texttt{if}_0$, sur l'alphabet A, telle que $\texttt{if}_0(x, y, z)$ renvoie y si la première lettre de x est 0, et z sinon.

Pour la suite, on supposera également avoir écrit la fonction similaire $\texttt{if}_1$, qui teste la première lettre de x à 1.

Corrigé.

$$\begin{aligned} \texttt{if}_0(\varepsilon, y, z) &= z \\ \texttt{if}_0(s_0(x), y, z) &= y \\ \texttt{if}_0(s_1(x), y, z) &= z \end{aligned}$$

□

Question 7 (1 point)

Ecrire une fonction récursive primitive sur les notations `compare`, telle que $\texttt{compare}(x, y)$ renvoie 1 si les mots x et y sont identiques, et 0 sinon.

Corrigé. La manière intuitive d'écrire cette fonction est

$$\begin{aligned} \texttt{compare}(\varepsilon, y) &= \texttt{estvide}(y) \\ \texttt{compare}(s_0(x), y) &= \texttt{if}_0(y, \texttt{compare}(x, \texttt{queue}(y)), s_0(\varepsilon))) \\ \texttt{compare}(s_1(x), y) &= \texttt{if}_1(y, \texttt{compare}(x, \texttt{queue}(y)), s_0(\varepsilon))) \end{aligned}$$

Mais cette fonction ne respecte pas exactement le schéma de récursion primitive. Pour écrire cette fonction sous forme récursive primitive, on utilise alors les fonctions auxiliaires suivantes, primitives récursives:

$$\begin{aligned} \texttt{extract}(\varepsilon, y) &= y \\ \texttt{extract}(s_0(x), y) &= \texttt{queue}(\texttt{extract}(x, y)) \\ \texttt{extract}(s_1(x), y) &= \texttt{queue}(\texttt{extract}(x, y)) \end{aligned}$$

$$\begin{aligned} \texttt{compare} - \texttt{rev}(\varepsilon, y) &= s_1(\varepsilon) \\ \texttt{compare} - \texttt{rev}(s_0(x), y) &= \texttt{if}_0(\texttt{extract}(x, y), \texttt{compare} - \texttt{rev}(x, y), s_0(\varepsilon))) \\ \texttt{compare} - \texttt{rev}(s_1(x), y) &= \texttt{if}_1(\texttt{extract}(x, y), \texttt{compare} - \texttt{rev}(x, y), s_0(\varepsilon))) \end{aligned}$$

$$\texttt{compare}(x, y) = \texttt{if}_1(\texttt{estvide}(\texttt{extract}(x, y)), \texttt{compare} - \texttt{rev}(\texttt{renverse}(x), y), s_0(\varepsilon))$$

Explication: $\texttt{extract}(x, y)$ retire de y ses n première lettres, où n est la longueur du mot x. $\texttt{compare} - \texttt{rev}(s_0(x), y)$ compare les n premières lettres de y avec celles de x, dans l'ordre inverse, où n est la longueur du mot x. Enfin, $\texttt{compare}(x, y)$ vérifie que les deux mots ont la même longueur, et, dans le cas positif, fait appel à $\texttt{compare} - \texttt{rev}$, en rétablissant le bon ordre de parcours des lettres de x.

□

Question 8 (1 point)

En déduire la fonction récursive primitive sur les notations `palindrome`, telle que `palindrome`(x) renvoie 1 si le mot x est un palindrome, et 0 sinon.

Corrigé.

$$\texttt{palindrome}(x) = \texttt{compare}(x, \texttt{renverse}(x))$$

□

λ-calcul pur

Question 9 Opérateur de point fixe (2 points)

On rappelle la définition de l'opérateur de point fixe de Curry:

$$\texttt{Y} = \lambda f.\,((\lambda x.(f(x\ x)))\ (\lambda x.(f(x\ x))))$$

Démontrer que pour tout λ-terme M, il existe un λ-terme M_1 tel que $\texttt{Y}\ M \to_\beta^* M1$ et $M\texttt{Y}M \to_\beta^* M1$. Donnez ce terme M_1.

Corrigé. Soit M un λ-terme quelconque. On a alors les réductions suivantes (on indique en rouge le redex):

$$\begin{array}{rcl}
\texttt{Y}M & = & \lambda f.\,((\lambda x.(f(x\ x)))\ (\lambda x.(f(x\ x))))\ M \\
& \to_\beta & ((\lambda x.(M(x\ x)))\ (\lambda x.(M(x\ x)))) \\
& \to_\beta & M\ ((\lambda x.(f(x\ x)))\ (\lambda x.(f(x\ x)))) \\
\text{et} & & \\
M\texttt{Y}M & = & M\lambda f.\,((\lambda x.(f(x\ x)))\ (\lambda x.(f(x\ x))))\ M \\
& \to_\beta & M\ ((\lambda x.(M(x\ x)))\ (\lambda x.(M(x\ x))))
\end{array}$$

d'où:

$$M_1 = M\ ((\lambda x.(f(x\ x)))\ (\lambda x.(f(x\ x))))\,.$$

Remarque: M_1 n'est pas en forme normale, et tout terme obtenu à partir d'une ou plusieurs réductions depuis M_1 est également une réponse valide. □

Question 10: Entiers de Church. (4 points)

Rappel de cours:

Pour deux termes t et x, on note $t^n\ x = t \cdots t\ x$ le terme obtenu en composant n fois le terme t sur l'argument x.

Entiers de Church: Un entier n est codé par le nombre de compositions d'une fonction f sur un argument x, comme suit.

$$\begin{array}{rcl}
\underline{0} & = & \lambda f.\lambda x.x \\
\underline{n} & = & \lambda f.\lambda x.(f^n\ x)
\end{array}$$

On donne le terme $\texttt{pred} = \lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(g\ f))\ (\lambda u.x)\ (\lambda u.u)$

- Démontrer que $(\lambda t.\lambda y.y)\ (\lambda g.\lambda h.h(g\ f))\ (\lambda u.x) \to_\beta^* \lambda u.x$. En déduire $\texttt{pred}\ \underline{0} \to_\beta^* \underline{0}$.
- Démonter par récurrence sur $n > 0$ que $(\lambda g.\lambda h.h(g\ f))^n\ (\lambda u.x) \to_\beta^* \lambda h.h\ (f^{n-1}x)$ pour tout $n > 0$. En déduire $\texttt{pred}\ \underline{n+1} \to_\beta^* \underline{n}$.

Ces deux points montrent que la fonction `pred` encode le prédécesseur sur les entiers de Church.

Corrigé. •

$$\begin{array}{rcl}(\lambda t.\lambda y.y)\ (\lambda g.\lambda h.h(g\ f))\ (\lambda u.x) & \to_\beta & (\lambda y.y\ \lambda u.x)\\ & \to_\beta & \lambda u.x\end{array}$$

$$\begin{array}{rcl}\texttt{pred}\ \underline{0} & = & (\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(g\ f))\ (\lambda u.x)\ (\lambda u.u))\ (\lambda t.\lambda y.y)\\ & \to_\beta & \lambda f.\lambda x.(\lambda t.\lambda y.y)(\lambda g.\lambda h.h(g\ f))\ (\lambda u.x)\ (\lambda u.u)\\ & \to_\beta^* & \lambda f.\lambda x.(\lambda u.x\ (\lambda u.u)) \text{ d'après ce qui précède}\\ & \to_\beta^* & \lambda f.\lambda x.x\end{array}$$

- Pour $n = 1$:

$$\begin{array}{rcl}(\lambda g.\lambda h.h(g\ f))\ (\lambda u.x) & \to_\beta & \lambda h.h\ ((\lambda u.x\ f)\\ & \to_\beta & \lambda h.h\ x\end{array}$$

Pour $n > 1$:

$$\begin{array}{rcl}(\lambda g.\lambda h.h(g\ f))^n\ (\lambda u.x) & = & (\lambda g.\lambda h.h(g\ f))(\lambda g.\lambda h.h(g\ f))^{n-1}\ (\lambda u.x)\\ & \to_\beta^* & (\lambda g.\lambda h.h(g\ f))\ (\lambda h.h\ (f^{n-2}x)) \text{ par récurrence}\\ & \to_\beta & \lambda h.h((\lambda t.t\ (f^{n-2}x))\ f)\\ & \to_\beta & \lambda h.h\ (f^{n-1}x)\end{array}$$

On déduit:

$$\begin{array}{rcl}\texttt{pred}\ \underline{n+1} & = & (\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(g\ f))\ (\lambda u.x)\ (\lambda u.u))\ (\lambda f.\lambda x.(f^{n+1}\ x))\\ & \to_\beta & \lambda f.\lambda x.((\lambda t.\lambda z.(t^{n+1}\ z))\ (\lambda g.\lambda h.h(g\ f))\ (\lambda u.x)\ (\lambda u.u))\\ & \to_\beta & \lambda f.\lambda x.(\lambda z.(\lambda g.\lambda h.h(g\ f))^{n+1}\ z)\ (\lambda u.x)\ (\lambda u.u)\\ & \to_\beta & \lambda f.\lambda x.((\lambda g.\lambda h.h(g\ f))^{n+1}\ \lambda u.x)\ (\lambda u.u)\\ & \to_\beta^* & \lambda f.\lambda x.((\lambda h.h\ (f^{n-1}x)\ (\lambda u.u)) \text{ d'après ce qui précède}\\ & \to_\beta & \lambda f.\lambda x.(\lambda u.u\ (f^{n-1}x))\\ & \to_\beta & \underline{n}\end{array}$$

□

Question 11: Entiers de Church. (1 point)

Déduire de la fonction `pred` un encodage `sub` de la soustraction sur les entiers de Church, tel que $\texttt{sub}\ \underline{n}\ \underline{m} \to_\beta^* \underline{n-m}$ si $n \geq m$, et $\texttt{sub}\ \underline{n}\ \underline{m} \to_\beta^* \underline{0}$ sinon.

Corrigé.

$$\texttt{sub} = \lambda m.\lambda n.n\ \texttt{pred}\ m$$

□

λ-calcul simplement typé

Question 12: (3 points)

Les entiers de Church du λ-calcul pur sont-ils simplement typables? si oui, proposer un type, et une dérivation de type simple pour un entier non-nul de votre choix. Faire de même pour l'addition $\texttt{plus} = \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$, en prenant garde de proposer un type compatible avec celui choisi pour les entiers.

Corrigé. Prenons l'entier $\underline{2} = \lambda f.\lambda x.(f\ f\ x)$. La dérivation de type est la suivante:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x:A, f:A\to A \vdash f:A\to A}\text{ id.} \quad \dfrac{\dfrac{}{x:A, f:A\to A \vdash f:A\to A}\text{ id.} \quad \dfrac{}{x:A, f:A\to A \vdash x:A}\text{ id.}}{x:A, f:A\to A \vdash f\ x:A}\text{ app.}}{x:A, f:A\to A \vdash f\ f\ x:A}\text{ app.}}{f:A\to A \vdash \lambda x.(f\ f\ x):A\to A}\text{ abs.}}{\vdash \lambda f.\lambda x.(f\ f\ x):(A\to A)\to(A\to A)}\text{ abs.}$$

Le même type peut aussi être donné à tous les autres entiers de Church, et en particulier à $\underline{0}$ et $\underline{1}$. On note donc dans la suite $\texttt{nat} = (A\to A)\to(A\to A)$.

Pour l'addition:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{T_1 \qquad T_2}{m:\texttt{nat}, n:\ \texttt{nat}, f:A\to A, x:A \vdash m\ f\ (n\ f\ x):A}\text{ app.}}{m:\texttt{nat}, n:\ \texttt{nat}, f:A\to A \vdash \lambda x.m\ f\ (n\ f\ x):A\to A}\text{ abs.}}{m:\texttt{nat}, n:\ \texttt{nat} \vdash \lambda f.\lambda x.m\ f\ (n\ f\ x):\texttt{nat}}\text{ abs.}}{m:\texttt{nat} \vdash \lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x):\texttt{nat}\to\texttt{nat}}\text{ abs.}}{\vdash \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x):\texttt{nat}\to(\texttt{nat}\to\texttt{nat})}\text{ abs.}$$

Le sous-arbre gauche T_1, en notant $\Gamma = m:\texttt{nat}, n:\ \texttt{nat}, f:A\to A, x:A$, est :

$$\dfrac{\dfrac{}{\Gamma\vdash m:\texttt{nat}}\text{ id.} \quad \dfrac{}{\Gamma\vdash f:A\to A}\text{ id.}}{\Gamma\vdash m\ f:A\to A}\text{ app.}$$

et le sous-arbre droit T_2 est:

$$\dfrac{\dfrac{\dfrac{}{\Gamma\vdash n:\texttt{nat}}\text{ id.} \quad \dfrac{}{\Gamma\vdash f:A\to A}\text{ id.}}{\Gamma\vdash nf:A\to A}\text{ app.} \quad \dfrac{}{\Gamma\vdash x:A}\text{ id.}}{\Gamma\vdash n\ f\ x:A}\text{ app.}$$

L'addition est donc typable, avec le type $\texttt{nat}\to(\texttt{nat}\to\texttt{nat})$. On remarque qu'il s'agit de la forme curryfiée d'une fonction à deux arguments entiers, qui renvoie un résultat entier.

□

Extensions du λ-calcul simplement typé

Soit `nat` le type simple que vous avez proposé pour les entiers de Church à la question 12 précédente. On suppose donnés un type de base simple `bool`, pour représenter les booléens, et les fonctions suivantes

- `IfThenElse`, de type `bool` $\to$ `nat` $\to$ `nat` $\to$ `nat`, qui encode le test "if then else" sur un booléen et deux entiers de Church, et
- `IsZero`, de type `nat` $\to$ `bool`, qui encode le test à zéro d'un entier de Church.
- `mult`, de type `nat` $\to$ `nat` $\to$ `nat`, qui encode la multiplication sur les entiers de Church, et
- `pred`, de type `nat` $\to$ `nat`, qui encode le prédessesseur.

On se donne alors la fonction récursive `fact` suivante, qui calcule la fonction factorielle sur les entiers de Church:

$$\texttt{letrec fact} = \lambda n.\texttt{IfThenElse}\ (\texttt{IsZero}\ n)\ \underline{1}\ (\texttt{mult}\ n\ (\texttt{fact}\ (\texttt{pred}\ n)))\ \texttt{in fact}.$$

Question 13 (2 points):

Cette fonction est-elle typable avec les types proposés ci-dessus et les extensions vues en cours? Si oui, proposer un type, et une dérivation de type, compatibles avec les types proposés ci-dessus.

Corrigé. On utilise la forme `letrec` vue en cours:

$$\texttt{let fact} = (\texttt{fix}\ \lambda f.\lambda n.\texttt{IfThenElse}\ (\texttt{IsZero}\ n)\ \underline{1}\ (\texttt{mult}\ n\ (f\ (\texttt{pred}\ n))))\ \texttt{in fact}.$$

On note également $\Gamma = \texttt{IfThenElse} : \texttt{bool} \to \texttt{nat} \to \texttt{nat} \to \texttt{nat}, \cdots, \texttt{pred} : \texttt{nat} \to \texttt{nat}$, l'environnement de typage contenant tous les types donnés ci-dessus dans l'énoncé, et $\Delta = \Gamma, f : \texttt{nat} \to \texttt{nat}, n : \texttt{nat}$.

Le type de la fonction `fact` est alors $\texttt{nat} \to \texttt{nat}$, et la dérivation de type est donnée à la page suivante.
$\therefore$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{T_1 \qquad T_2}{\Delta \vdash \texttt{IfThenElse}\ (\texttt{IsZero}\ n)\ \underline{1}\ (\texttt{mult}\ n\ (f\ (\texttt{pred}\ n))) : \texttt{nat}}\ \text{app.}}{\Gamma, f : \texttt{nat} \to \texttt{nat} \vdash \lambda n.\texttt{IfThenElse}\ (\texttt{IsZero}\ n)\ \underline{1}\ (\texttt{mult}\ n\ (f\ (\texttt{pred}\ n))) : \texttt{nat} \to \texttt{nat}}\ \text{abs.}}{\Gamma \vdash \lambda f.\lambda n.\texttt{IfThenElse}\ (\texttt{IsZero}\ n)\ \underline{1}\ (\texttt{mult}\ n\ (f\ (\texttt{pred}\ n))) : (\texttt{nat} \to \texttt{nat}) \to (\texttt{nat} \to \texttt{nat})}\ \text{abs.}}{\Gamma \vdash \texttt{fix}\ \lambda f.\lambda n.\texttt{IfThenElse}\ (\texttt{IsZero}\ n)\ \underline{1}\ (\texttt{mult}\ n\ (f\ (\texttt{pred}\ n))) : \texttt{nat} \to \texttt{nat}}\ \text{fix.} \qquad \dfrac{\textcircled{6}}{\Gamma, \texttt{fact} : \texttt{nat} \to \texttt{nat} \vdash \texttt{fact} : \texttt{nat} \to \texttt{nat}}\ \text{id.}}{\Gamma \vdash \texttt{let fact} = (\texttt{fix}\ \lambda f.\lambda n.\texttt{IfThenElse}\ (\texttt{IsZero}\ n)\ \underline{1}\ (\texttt{mult}\ n\ (f\ (\texttt{pred}\ n)))) \texttt{ in fact} : \texttt{nat} \to \texttt{nat}}\ \text{let.}$$

Le sous-arbre gauche T_1 est:

$$\dfrac{\dfrac{}{\Delta \vdash \texttt{IfThenElse} : \texttt{bool} \to (\texttt{nat} \to (\texttt{nat} \to \texttt{nat}))}\ \text{id.} \qquad \dfrac{\dfrac{}{\Delta \vdash \texttt{IsZero}\ n : \texttt{nat} \to \texttt{bool}}\ \text{id.} \qquad \dfrac{}{\Delta \vdash n : \texttt{nat}}\ \text{id.}}{\Delta \vdash (\texttt{IsZero}\ n) : \texttt{bool}}\ \text{app.}}{\Delta \vdash \texttt{IfThenElse}\ (\texttt{IsZero}\ n) : \texttt{nat} \to (\texttt{nat} \to \texttt{nat})}\ \text{app.} \qquad \dfrac{\vdots}{\Delta \vdash \underline{1} : \texttt{nat}}$$
$$\Delta \vdash \texttt{IfThenElse}\ (\texttt{IsZero}\ n)\ \underline{1} : \texttt{nat} \to \texttt{nat}\ \text{app.}$$

Le sous-arbre droit T_2 est:

$$\dfrac{\dfrac{\dfrac{}{\Delta \vdash \texttt{mult} : \texttt{nat} \to (\texttt{nat} \to \texttt{nat})}\ \text{id.} \qquad \dfrac{}{\Delta \vdash n : \texttt{nat}}\ \text{id.}}{\Delta \vdash \texttt{mult}\ n\ : \texttt{nat} \to \texttt{nat}}\ \text{app.} \qquad \dfrac{\dfrac{}{\Delta \vdash f : \texttt{nat} \to \texttt{nat}}\ \text{id.} \qquad \dfrac{\dfrac{}{\Delta \vdash \texttt{pred} : \texttt{nat} \to \texttt{nat}}\ \text{id.} \qquad \dfrac{}{\Delta \vdash n : \texttt{nat}}\ \text{id.}}{\Delta \vdash \texttt{pred}\ n : \texttt{nat}}\ \text{app.}}{\Delta \vdash f\ (\texttt{pred}\ n) : \texttt{nat}}\ \text{app.}}{\Delta \vdash \texttt{mult}\ n\ (f\ (\texttt{pred}\ n)) : \texttt{nat}}\ \text{app.}$$

Le sous-arbre de conclusion $\Delta \vdash \underline{1} : \texttt{nat}$ n'est pas donné ici: il est donné par la question 12.

□

Question Bonus (λ-calcul pur) - (3 points)

En vous inspirant de la méthode montrée en cours pour encoder le schéma de récursion primitive des algèbres de fonctions récursives en λ-calcul à l'aide d'un opérateur de point fixe, proposez un encodage du schéma de minimisation des fonctions récursives, là encore à l'aide d'un point fixe, en λ-calcul.

On pourra exprimer le schéma de minimisation comme suit. Soit f une fonction à un seul argument:

$$\begin{array}{lcl} \texttt{mu}\ \underline{n}\ f & = & \underline{n} \text{ si } (f\ \underline{n}) \rightarrow_\beta^* \underline{0} \\ \texttt{mu}\ \underline{n}\ f & = & \texttt{mu}\ (\texttt{succ}\ \underline{n})\ f \text{ sinon} \end{array}$$

Le schéma de minimisation sur f est alors donné par $\texttt{mu}\ \underline{0}\ f$.

Corrigé. Le schéma de minimisation sur une fonction f un un argument est donné par le terme suivant, appliqué à f.

$$\lambda g.(\texttt{fix}(\lambda t.\lambda n.\lambda f.(\texttt{IfThenElse}(\texttt{IsZero}(f\ n))\ n\ (t\ (\texttt{succ}\ n)\ f)))\ \underline{0}\ g)$$

□