

Examen du 16 décembre 2013

Les notes de TD manuscrites ainsi que les transparents de cours et le polycopié de cette année sont les seuls documents autorisés. Veuillez lire attentivement les questions. Veuillez rédiger proprement, clairement et de manière concise et rigoureuse.

1 Questions de cours

Question 1 La fonction suivante est-elle bien ou mal typées en OCaml ? Justifier votre réponse soit en donnant le type, soit en précisant l'erreur de typage.

```
let f x y = if x y then [x] else [y]
```

Question 2 Étant donné le programme Pascal suivant, dessiner les tableaux d'activation sur la pile jusqu'à l'appel à `Q(1)` (inclus).

```
procedure P();  
var x : integer;  
  procedure Q(y:integer);  
    procedure R();  
      begin Q(1) end  
    begin  
      if y=0 then R() else writeln(x)  
    end  
  begin  
    x:=10; Q(0)  
  end;  
  
begin P() end
```

2 Langage de programmation avec manipulation de tableaux

Dans cet exercice, on étudie un petit langage qui permet de manipuler des entiers et des tableaux d'entiers. Il dispose des opérations arithmétiques et de comparaison classiques (+, ≤, etc.) et celles-ci sont applicables directement sur les tableaux. Par exemple, si `t1` et `t2` désignent deux tableaux de taille n , l'expression `t1 + t2` est un nouveau tableau de taille n obtenu en additionnant les éléments de même indice de `t1` et `t2`. De même, l'expression `t1 ≤ t2` est vraie si et seulement si `t1` et `t2` sont de même taille n , et si pour tout indice $0 \leq i < n$, on a `t1[i] ≤ t2[i]` (où `t[i]` désigne l'élément i du tableau `t`).

Le but de cet exercice est de vérifier que ces opérations sont toujours possibles, c'est-à-dire que lorsqu'elles sont appliquées sur des tableaux, alors ceux-ci sont de même taille.

La syntaxe complète des expressions e de ce petit langage est donnée par la grammaire suivante :

$$\begin{aligned} e \quad := \quad & \text{let } x = e \text{ in } e \mid \text{if } e \leq e \text{ then } e \text{ else } e \mid e[i] \mid e + e \\ & \mid e[i < -e] \mid e[i; i] \mid [e, \dots, e] \mid i \mid x \end{aligned}$$

Une expression e est donc soit :

- une déclaration locale de la forme **let** $x = e_1$ **in** e_2
- une conditionnelle **if** $e_1 \leq e_2$ **then** e_3 **else** e_4 où $e_1 \leq e_2$ est applicable à des entiers ou des tableaux ; dans le cas où e_1 et e_2 sont des tableaux, ceux-ci doivent être de même taille
- l'accès $e[i]$ à la case i d'un tableau e de taille n , où i est une constante entière telle que $0 \leq i < n$
- une addition $e_1 + e_2$ applicable à deux entiers ou deux tableaux ; dans le cas où e_1 et e_2 sont des tableaux, ceux-ci doivent être de même taille n
- une opération de modification avec recopie $e_1[i \leftarrow e_2]$ qui renvoie un *nouveau* tableau égal au tableau e_1 (de taille n), mais dont l'élément d'indice i est égal à la valeur de e_2 ; i doit être une constante entière telle que $0 \leq i < n$ et e_2 une expression entière
- une création de tranche $e[i1; i2]$ où e désigne un tableau de taille n , et $i1$ et $i2$ deux constantes entières telles que $0 \leq i1 \leq i2 < n$; cette opération renvoie un *nouveau* tableau t de taille $i2 - i1 + 1$ tel que $t[i] = e[i1 + i]$, pour $0 \leq i < i2 - i1 + 1$
- un nouveau tableau d'entiers $[|e_0, e_1, \dots, e_{n-1}|]$ de taille n
- une constante entière i ou une variable x

Question 3 Indiquer (en justifiant votre réponse), parmi les deux programmes suivants, ceux qui sont bien typés et le cas échéant le résultat du calcul.

```
let x = [|1;2;3;4|] in
let z = (x + x) + [|1;1;1;1|] in
if z[2] <= 0 then z[2;3] + [|1;2|] else z[0;1]
```

```
let x = [|1;2;3;4|] in
let z = (x + x) + [|1;1;1;1|] in
if z[2] <= 0 then z[2;3] + [|1;2|] else x[1<-4]
```

Pour typer ces expressions, on définit un système de type qui introduit les types `int` et `int[n]`. Le type `int` désigne les valeurs entières. Le type `int[n]` est celui des tableaux d'entiers de taille n et d'indices 0 à $n - 1$. Les jugements de typage, notés $\Gamma \vdash e : \tau$, signifient que l'expression e est de type τ dans l'environnement de typage Γ . On note $\Gamma(x)$ le type associé à la variable x dans Γ , et $\Gamma + x : \tau$ l'environnement Γ augmenté de la liaison $x : \tau$. Le jeu (incomplet) des règles de typage pour ce langage est donné ci-dessous.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad \frac{}{\Gamma \vdash i : \text{int}} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1 \quad \Gamma \vdash e_3 : \tau_2 \quad \Gamma \vdash e_4 : \tau_2}{\Gamma \vdash \text{if } e_1 \leq e_2 \text{ then } e_3 \text{ else } e_4 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 + e_2 : \tau} \quad \frac{\Gamma \vdash e : \text{int}[n] \quad 0 \leq i1 \leq i2 < n}{\Gamma \vdash e[i1; i2] : \text{int}[i2 - i1 + 1]}$$

Question 4 Définir les règles de typage pour les constructions $e_1[i \leftarrow e_2]$, $[|e_0, e_1, \dots, e_{n-1}|]$ et $e[i]$.

On souhaite maintenant réaliser une implémentation de ce système de type en OCaml. Pour cela, on définit le type `exp` (incomplet) suivant pour représenter les arbres de syntaxe abstraite des expressions :

```

type exp =
| ELet of string * exp * exp
| EItte of exp * exp * exp * exp
| EGet of exp * int
| EAdd of exp * exp
| ECopy of exp * int * exp
| EInt of int
| Evar of string

```

Question 5 Il manque deux constructeurs au type `exp`. Lesquels ? Ajouter leur déclaration au type `exp`.

On représente les types `int` et `int[n]` de ce petit langage à l'aide du type `typ` suivant :

```

type typ = Int | Array of int

```

L'environnement de typage Γ est simplement représenté à l'aide d'une liste d'associations de type `(string * typ) list`.

Question 6 En suivant les règles d'inférence, écrire une fonction `type_expr : (string * typ) list -> exp -> typ` telle que `type_expr gamma e` renvoie le type d'une expression `e` sous l'environnement `gamma`.

3 Génération de code : `setjmp/longjmp`

Dans ce problème, on considère des programmes écrits en langage C ne manipulant que les types `int` et `void`. La bibliothèque standard de C propose deux fonctions, `setjmp` et `longjmp`, dont le fonctionnement est le suivant.

La fonction `int setjmp()`¹ ne prend pas d'argument et sauvegarde, dans une variable globale `jmp_env` (un tableau de mots mémoire), une partie du tableau d'activation courant de la fonction contenant l'appel à `setjmp`. La valeur de retour de `setjmp` lorsqu'elle est appelée directement est 0.

La fonction `void longjmp(int valeur)` prend en argument un entier `valeur`, différent de 0, et restaure le tableau d'activation sauvegardé et revient à l'endroit où `setjmp` a été appelé la première fois. L'exécution reprend de ce point là, et `setjmp` renvoie alors `valeur`. La fonction `longjmp` ne retourne jamais. Si on appelle `longjmp` alors que la fonction qui a appelé `setjmp` est déjà terminée, le comportement est indéfini. De même, `setjmp` ne peut pas être appelée dans une sous-expression (par exemple `1+2+setjmp()`) mais doit être obligatoirement la seule expression apparaissant dans la condition d'un `if` ou un `switch`. Voici le comportement illustré par un exemple :

```

#include <stdio.h>
#include <setjmp.h>

void call_fact(int i)
{
    if (setjmp()) {
        // Set jmp a renvoyé autre chose que 0, donc on revient d'ailleurs
        printf("J'ai trouvé 42 en évaluant fact(%d) ???", i);
    } else {
        // L'appel de set_jmp a renvoyé 0, c'est donc la première fois qu'on l'appelle
        printf("Tout s'est bien passé : fact(%d) = %d", i, fact(i));
    }
}

```

1. Les types et comportement de ces fonctions ont été légèrement simplifiés pour éviter d'introduire trop de concepts nouveaux.

```

    }
    return;
}

int fact(int n) {
    if (n == 0) {
        return 1;
    } else if (n != 42) {
        return n * fact (n-1);
    } else {
        longjmp(123);
        return 456; // jamais exécuté
    }
}

int main() {
    call_fact(10);
    call_fact(45);
    return 0;
}

```

L'exécution de ce programme affiche les messages suivants :

```

Tout s'est bien passé : fact(10) = 3628800
J'ai trouvé 42 en évaluant fact(45) ???

```

Explications. La fonction `fact(int n)` est une factorielle qui appelle `longjmp(123)` si jamais elle rencontre la valeur 42 (sinon elle s'exécute normalement). Lors du premier appel `call_fact(10)` dans le `main`, la fonction `call_fact` exécute `setjmp`, puis appelle la fonction `fact` avec la valeur 10. Cette dernière s'exécute normalement et la fonction `call_fact` affiche "Tout s'est bien passé ...". Lors du deuxième appel `call_fact(45)`, `setjmp` est appelé puis l'appel `fact(45)` est exécuté (car `setjmp` renvoie 0). Après quelques appels récursifs, `fact` exécute le dernier `else` et appelle `longjmp(123)`. Le cours normal de l'exécution est interrompu et on « revient » à l'appel à `setjmp()` dans la fonction `call_fact`. Mais cette fois, `setjmp` renvoie 123 (la valeur passée par `longjmp`), et on affiche alors "J'ai trouvé 42 en évaluant ...".

On se propose d'implanter `setjmp` et `longjmp` en assembleur MIPS. On suppose que le code généré passe tous les paramètres sur la pile (les registres ne sont utilisés que pour des calculs intermédiaires et les résultats des expressions toujours posés sur la pile) et on suppose que les tableaux d'activations ont la forme suivante :

	⋮	
	résultat	
	o	
	e_1	
	⋮	
appelant	e_n	
appelé	ancien \$fp	←\$fp
	ancien \$ra	
	locale 1	
	⋮	
	locale m	
	calculs	
	⋮	
	calculs	←\$sp
	⋮	

Question 7 Dessiner l'état global de la pile juste avant l'appel à `longjmp(123)`. Faire apparaître clairement les tableaux d'activation de `main()` , `call_fact(45)` et des appels récurifs à la fonction `fact`.

Question 8 Dessiner l'état global de la pile juste après l'appel à `longjmp(123)`. Quels registres sont modifiés entre ces deux états.

Question 9 Écrire une routine assembleur qui sauvegarde les registres qui ont besoin de l'être au moment du `setjmp` dans la variable globale `jmp_env` et qui renvoie 0. Il faut *aussi* penser à sauvegarder l'adresse de retour, car c'est à cette adresse que l'appel à `longjmp` devra sauter pour revenir à l'appel de `setjmp`.

Question 10 Écrire une routine assembleur qui restaure le flot d'exécution à l'appel de `setjmp` et renvoie la valeur passée en argument à `longjmp`.