Master Informatique, M1 — 2014–2015
Introduction à la compilation — Durée : 2h
4 pages

# Examen du 15 décembre 2014

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 (temps recommandé : moins de 30 minutes)

**Question 1** Donner le tableau d'activation pour un appel de la fonction `f` suivante. Faites apparaître explicitement les parties allouées par l'appelé et par l'appelant.

```
int f(int x, int y) {
  int z = 0;
  return (x+z);
}
```

**Question 2** Donner un arbre de dérivation de typage pour l'expression OCaml suivante :

```
let f = fun x -> x in
(f 1, f(true, 2))
```

## 2 Cocktail : un langage de programmation avec des glaçons

Dans ce problème, on s'intéresse à un petit langage arithmétique avec des « glaçons », c'est-à-dire une construction qui permet de retarder l'évaluation d'une expression. Ce langage, Cocktail, est une variante de Mini-ML, qui n'utilise pas de fonctions mais qui contient deux nouvelles formes d'expressions : `freeze` transforme une expression *expr* en un glaçon, qui sera évalué plus tard ; `eval` « dégèle » un glaçon, c'est-à-dire déclenche l'évaluation de l'expression *expr* qu'il contient.

Par exemple, l'évaluation de l'expression suivante affiche, dans l'ordre, d'abord `2` puis `3`, et enfin `4`.

```
let x = 2 in
let y = freeze( print(x+1); x*x ) in
print(x); print(eval(y))
```

Un programme Cocktail est une expression *expr* construite avec la grammaire suivante :

| | | | |
|---|---|---|---|
| *expr* | ::= | *n* | constante entière |
| | \| | *x* | variable |
| | \| | `(` *expr* `)` | parenthèses |
| | \| | *expr binop expr* | opérations arithmétiques |
| | \| | `let` *x* `=` *expr* `in` *expr* | définition locale |
| | \| | `print(` *expr* `)` | affichage |
| | \| | *expr* `;` *expr* | séquence |
| | \| | `freeze(` *expr* `)` | création d'un glaçon |
| | \| | `eval(` *expr* `)` | évaluation d'un glaçon |

*binop* ::= `+` | `-` | `*` | `/`

**Question 3** Donner l'affichage et la valeur retournée par l'expression suivante :

```
let y = let x = 2 in
        print(x); freeze( let z = x+1 in print(z); z ) in
2 * eval(y)
```

## 2.1 Typage

Pour typer les instructions de COCKTAIL, on définit un système de types similaire à Mini-ML monomorphe, qui introduit les types `int`, `unit`, `int frozen`, et `unit frozen`. L'objectif du système de type est de garantir par typage qu'une expression à exactement l'un de ces quatre types.

Le type `int` désigne les valeurs entières. Le type `unit` désigne les expressions ne retournant pas de valeur (comme `print`). Le type $\tau$ `frozen` désigne les glaçons renfermant une expression de type $\tau$. Le fait de n'avoir que les types `int frozen` et `unit frozen` permet de garantir qu'une expression bien typée ne peut contenir des glaçons qui contiennent d'autres glaçons.

Le jugement de typage $\Gamma \vdash e : \tau$ signifie que l'expression $e$ est de type $\tau$ dans l'environnement $\Gamma$. Remarque : le code de la question 3 est bien typé, de type `int`, dans un environnement vide.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{(}\ e\ \texttt{)} : \tau} \qquad \frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \texttt{print(}\ e\ \texttt{)} : \texttt{unit}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\ \textit{binop}\ e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau\ \texttt{frozen}}{\Gamma \vdash \texttt{eval(}\ e\ \texttt{)} : \tau}$$

**Question 4** Définir les règles de typage pour les constructions $e_1$ `;` $e_2$ et `freeze(` $e$ `)`.

**Question 5** Les expressions suivantes sont-elles bien typées ? Si oui, donner leur type de retour.

1. `let x = let y=2 in freeze(y+1) in eval(x)`
2. `let x = freeze(let y=2 in freeze (y+1)) in eval(x)`
3. `let x = let y=2 in freeze(print(y+1)) in eval(x)`

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

```
type expr =
  | Cst of int
  | Var of string
  | Binop of binop * expr * expr
  | Seq of expr * expr
  | Print of expr
  | Eval of expr

and binop = Add | Sub | Mul | Div
```

**Question 6** Il manque deux constructeurs au type `expr`. Lesquels ? Ajouter leur déclaration au type `expr`.

On représente les types `int`, `unit` et `frozen` à l'aide du type `typ` suivant :

```
type typ = Int
         | Unit
         | Frozen of typ
```

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

**Question 7** On veut écrire une fonction `type_expr : (string * typ) list -> expr -> typ` telle que `type_expr gamma e` renvoie le type d'une expression `e` sous l'environnement `gamma`, en suivant les règles d'inférences. Écrire les cas correspondant au typage d'une variable, d'une expression `freeze`, et d'une expression `eval`. Pour ces trois cas, proposer un traitement d'erreur qui permet de distinguer les erreurs suivantes :

- Identificateur inconnu.
- Tentative de gel d'une expression déjà gelée.
- Tentative de dégel d'une expression non gelée.

## 2.2 Génération de code

On souhaite générer du code MIPS pour le langage Cocktail. Comme il est d'usage, les variables locales sont allouées sur la pile.

Une expression `freeze(` $e$ `)` produit un glaçon, qui est représenté par un bloc alloué sur le tas, qui est similaire à une fermeture. Si l'expression $e$ fait référence à $n$ variables extérieures, ce bloc contient $n+1$ mots, à savoir un pointeur vers le code MIPS évaluant $e$ et les valeurs des $n$ variables.

Par exemple, l'expression

```
let g =
    let x = 1 in
    let y = 2 in
    freeze(print(x); x+y)
in
    eval(g) + 1
```

produit un bloc sur le tas de la forme suivante :


où `L` est une étiquette dans le code MIPS correspondant à l'évaluation du glaçon `print(x); x+y`.

**Question 8** Donner le code MIPS correspondant à l'allocation du glaçon.

**Question 9** Donner le code associé à l'étiquette `L`.

## Annexe : aide-mémoire MIPS

On donne ici un fragment du jeu d'instructions MIPS suffisant pour réaliser ce problème. (Vous êtes cependant libre d'utiliser tout autre élément de l'assembleur MIPS.) Les registres utilisés ici, et leur signification dans le schéma de compilation adopté, sont les suivants :

| | |
|---|---|
| `$v0` | résultat d'une expression |
| `$fp` | pointeur sur le tableau d'activation courant |
| `$sp` | pointeur sur le sommet de la pile |
| `$v1` | pointeur sur le gestionnaire courant |
| `$ra` | adresse de retour |
| `$a0` | registre libre d'usage |

Les instructions susceptibles d'être utiles sont les suivantes (où les $r_i$ désignent des registres, $n$ une constante entière et $L$ une étiquette de code) :

| | |
|---|---|
| `syscall` | effectue un appel système dont le numéro est spécifié dans `$v0`, les arguments éventuels sont dans les registres `$a0` - `$a3` (exemple : `$v0`=1 pour afficher un entier dont la valeur est dans `$a0`) |
| `add` $r_1$, $r_2$, $r_3$ | calcule la somme de $r_2$ et $r_3$ dans $r_1$ |
| `move` $r_1$, $r_2$ | copie le registre $r_2$ dans le registre $r_1$ |
| `la` $r$, $L$ | charge l'adresse désignée par l'étiquette $L$ dans le registre $r$ |
| `lw` $r_1$, $n(r_2)$ | charge dans $r_1$ la valeur contenue en mémoire à l'adresse $r_2 + n$ |
| `sw` $r_1$, $n(r_2)$ | écrit en mémoire à l'adresse $r_2 + n$ la valeur contenue dans $r_1$ |
| `j` $L$ | saute à l'adresse désignée par l'étiquette $L$ |
| `jr` $r$ | saute à l'adresse contenue dans le registre $r$ |
| `jal` $L$ | saute à l'adresse désignée par l'étiquette $L$, après avoir sauvegardé l'adresse de retour dans `$ra` |