Compilation
Master Informatique M1 2008-2009

Examen Final
15 décembre 2008

Les exercices sont indépendants. L'énoncé est composé de 6 pages. L'examen dure 3 heures. Les notes de cours et de TD manuscrites ainsi que les supports de cours distribués cette année sont les seuls documents autorisés.
Consigne Lorsqu'il vous est demandé de décrire une méthode, vous n'êtes pas obligés de donner du code dans tous ses détails (vous pouvez par exemple utiliser des fonctions auxiliaires dont l'implantation est directe en vous contentant de spécifier leur comportement). Vous devez par contre décrire les données manipulées et les algorithmes utilisés de manière précise.

1 Question de cours (4 points)

- Quand dit-on qu'un système de type est polymorphe ?
- Donner au moins deux exemples de langages et d'expressions utilisant un typage polymorphe.
- Quel est l'intérêt du typage polymorphe ?
- En quoi le typage polymorphe a-t-il une incidence sur la méthode de génération de code ?

2 Langage avec valeurs de paramètres par défaut (7 pts)

La grammaire du langage a pour symboles terminaux :

$$\{\texttt{let}, \texttt{int}, \texttt{bool}, \texttt{id}, \texttt{num}, \texttt{true}, \texttt{false}, \texttt{op}, =, \texttt{if}, \texttt{then}, \texttt{else}, \texttt{,}, \texttt{(}, \texttt{)}\}$$

`id` représente un identificateur correspondant à une variable ou une fonction, `num` représente une constante entière, `op` est un des opérateurs arithmétiques $\{+, -, *, /, <, \leq, ==\}$.

Les expressions représentent des entiers ou des booléens. Elles sont engendrées par la grammaire suivante dans laquelle F représente une suite d'expressions (éventuellement vide) séparées par des virgules :

E ::= num	E ::= E op E	F ::= ϵ
E ::= true	E ::= (E)	F ::= F_1
E ::= false	E ::= id (F)	F_1 ::= E
E ::= id	E ::= if E then E else E	F_1 ::= F_1 , E

Un programme dans ce langage est composé d'une suite de déclarations qui peuvent être soit des déclarations de fonctions soit l'association d'un nom à une expression. Par exemple :

```
let square (x:int) : int = x * x
let f (x:int,y:int) : int = square(x) + square(y)
let x = 3
let y = 4
let z = f (x,y)
```

La particularité de ce langage est de permettre de donner des valeurs par défaut à certains paramètres des fonctions. Une valeur par défaut peut être n'importe quelle expression. Au moment de l'appel, il n'est pas nécessaire de fournir une valeur pour un tel paramètre, à défaut, la valeur associée au paramètre au moment de la déclaration de la fonction sera utilisée. Ainsi on peut écrire :

```
let g (x:int,y:int,k=1,l=1) : int := k * square(x) + l * square(y)
let t := g(x,y)
let u := g(x,y,2)
let v := g(x,y,2,3)
```

Les paramètres k et l de la fonction g ont la valeur 1 par défaut. L'identificateur t aura pour valeur $1 * \texttt{square}(3) + 1 * \texttt{square}(4)$ (k et l ont leur valeur par défaut); u aura pour valeur $2 * \texttt{square}(3) + 1 * \texttt{square}(4)$ (k est donné explicitement alors que l a la valeur par défaut); finalement v aura pour valeur $2 * \texttt{square}(3) + 3 * \texttt{square}(4)$ (k et l sont donnés explicitement).

On choisit dans la déclaration des paramètres de regrouper tous les paramètres sans valeur par défaut d'abord puis de mettre les paramètres avec valeur par défaut ensuite. S'il y a plusieurs paramètres par défaut, il faut les donner dans l'ordre.

1. Compléter la grammaire des expressions du langage avec les entrées pour les déclarations et les programmes de ce langage.
2. Compléter les déclarations de types suivantes (`expr`, `decl` et `prog`) pour représenter les arbres de syntaxe abstraite de ce langage.

```
type op = Add | Sub | Mul | Inf | Infeq | Eq
type typ  = Int | Bool
type expr = Ident of string | IntCte of int
   | BoolCte of bool | Oper of op * expr * expr
   | ...
type decl = ...
type prog = ...
```

3. Proposer les informations à conserver dans la table des symboles des fonctions et des variables afin de vérifier qu'un programme est bien formé, c'est-à-dire qu'il respecte les règles usuelles de portée et de typage :
   - toute variable utilisée dans une expression doit au préalable avoir été déclarée.
   - Les paramètres d'une fonction ne sont visibles que dans le corps de la fonction, les fonctions peuvent être récursives.
   - Les opérateurs s'appliquent à des entiers, sauf == qui peut s'appliquer aussi à deux booléens.
   - Dans une expression conditionnelle, la condition a pour type un booléen et les deux branches ont le même type.
4. Proposer une méthode permettant de vérifier qu'un programme est correctement formé.
   Il n'est pas nécessaire de détailler les opérations de manipulation de la table des symboles, il suffit de donner leur spécification.
5. On souhaite engendrer du code de la machine à pile pour calculer les valeurs de chaque variable introduite dans le programme.
   (a) Donner le code correspondant au programme où on déclare les fonctions `square` et `g` ainsi que les variables `x`, `y`, `t`, `u` et `v`.
   (b) Proposer une méthode pour compiler les programmes de ce langage.

3 Optimisation (9 points)

Pour les optimisations, on utilise comme langage intermédiaire (appelé SSA) un code à trois adresses dont les instructions sont de la forme suivante avec e qui est soit un identificateur soit une constante entière.

`id = e`	**goto** L	**label** L
`id = e op e`	**ifzero** `id` **goto** L	

3.1 Compilation de la machine à pile vers du code à trois adresses

On se propose d'écrire un compilateur qui à partir d'un programme écrit dans le langage de la machine à pile engendre du code pour une machine à registres. Une telle technique est utilisée dans les compilateurs Java qui traduisent ainsi le byte-code dans un code machine efficace.

Le langage source est le langage de la machine à pile représenté par le type :

```
type op = Add | Sub | Mul | Inf | Infeq | Eq
type pinstr =  Pushi of int | Op of op | Jump of label | JZ of label
  | Pushg of int | Stroreg of int | Pushn of int | Pop of int
```

L'idée est de considérer chaque emplacement sur la pile comme une nouvelle variable (notée r_n où n est la position dans la pile). Si la valeur courante du pointeur de pile *sp* est n:

- l'instruction `ADD` de la machine à pile se traduit par l'instruction $r_{n-2} = r_{n-1} + r_{n-2}$, puisque l'effet de l'instruction est de dépiler deux valeurs (qui correspondent aux variables r_{n-1} et r_{n-2}) puis de rempiler (à l'emplacement correspondant à r_{n-2}) le résultat de la somme.
- l'instruction `PUSHG` i de la machine à pile se traduit par $r_n = r_i$
- l'instruction `STOREG` i de la machine à pile se traduit par $r_i = r_{n-1}$
- les instructions `POP` et `PUSHN` ne font que modifier la valeur courante de *sp*, elles n'engendrent pas d'instructions.

1. Soit le programme suivant dans le code de la machine à pile :

PUSHN 4	STOREG 3	PUSHG 2	SUB
PUSHI 13	LABEL lab0	PUSHI 1	STOREG 3
STOREG 0	PUSHG 1	SUB	JUMP lab0
PUSHI 2	PUSHG 3	STOREG 2	LABEL lab1
STOREG 1	INFEQ	PUSHG 3	
PUSHG 0	JZ lab1	PUSHG 1	

   Donner la traduction dans le langage SSA en suivant le schéma de traduction proposé.

2. Compléter le type suivant pour représenter les programmes du langage SSA.

   ```
   type ident = int
   type atom = Var of ident | Int of int
   type ssa_expr = At of atom | Bop of ...
   type ssa_instr = Aff of ident * ssa_expr | Lab of string
                  | Goto of string | Ifzero of ...
   ```

3. Ecrire une fonction qui étant donné un programme écrit dans le langage de la machine à pile, produit le programme correspondant dans le langage SSA. On pourra penser à utiliser la valeur courante du registre *sp* comme argument de la fonction de traduction.

3.2 Optimisation de code à trois adresses

On cherche maintenant à optimiser le code à trois adresses. On étudie le programme suivant:

r2 = 5	r4 = 1	r7 = r1	r8 = r8-r9
r1 = r2	r5 = r1	r6 = r6*r7	r1 = r8
r3 = 1	r4 = r4<=r5	r0 = r6	goto lab1
r0 = r3	ifzero r4 goto lab2	r8 = r1	label lab2
label lab1	r6 = r0	r9 = 1	

1. Construire le graphe de flot de controle de ce programme.
2. Donner sur chaque arète de ce graphe la liste des variables vivantes (possiblement utilisées avant d'être redéfinies).
3. Construire le graphe d'interférence de ce programme.
4. Proposer un coloriage à l'aide de 4 registres, on s'attachera si possible à choisir la même couleur pour r et r' s'il y a une affectation $r = r'$.
5. Peut-on trouver un coloriage avec trois registres sans modifier le programme ?
6. Reconstruire le programme donné en utilisant seulement les 4 registres. On note p ce programme.
7. On cherche maintenant à éliminer les instructions $r = k$ où k est atomique. Pour cela on fait une recherche de définitions actives:
   – chaque déclaration $r = e$ du programme est nommée d_k avec k le numéro de ligne.
   – chaque utilisation de r dans une instruction est annotée avec la liste des déclarations qui peuvent avoir déterminé sa valeur.

   Dans le programme p obtenu précédemment, donner pour chaque utilisation de variable dans une instruction les déclarations actives correspondantes.
8. Si une déclaration est de la forme $r = e$ avec e atomique et si pour toutes les utilisations de r pour lesquelles cette déclaration est active il n'y a pas d'autres déclarations actives de cette même variable, alors on peut substituer r par e dans cette utilisation et supprimer la déclaration.
   Indiquer dans le programme précédent quelles sont les déclarations qui satisfont cette propriété et construire le programme transformé.

4 Instructions de la machine à pile

`PUSHI` n	empile la constante entière n.
`ADD`/`SUB`/`MUL`/`DIV`	dépile y puis x et empile $(x + y)$/$(x - y)$/$(x * y)$/(x/y)
`INF`/`INFEQ`/`EQ`	dépile y puis x et empile 1 si $(x < y)$/$(x \leq y)$/$(x = y)$ et 0 sinon
`PUSHG` n	empile la valeur située n cases au dessus de *gp*.
`STOREG` n	dépile x et le stocke à l'emplacement situé n cases au dessus de *gp*.
`PUSHL` n	empile la valeur située n cases au dessus de *fp*.
`STOREL` n	dépile x et le stocke à l'emplacement situé n cases au dessus de *fp*.
`PUSHN` n	empile n valeurs nulles sur la pile.
`POP` n	dépile n valeurs de la pile.
`JZ` l	dépile x, si $x = 0$ le pointeur de code saute à l'instruction de label l.
`JUMP` l	saute à l'instruction de label l.
`CALL` c	sauve la valeur courante de *fp* et le pointeur d'instructions, affecte à *fp* la valeur de la première case libre de la pile, saute à l'instruction d'adresse c.
`RETURN`	restaure les valeurs de *fp*,*sp* et *pc*.