Compilation
Master Informatique M1 2008-2009

Examen 2ième session
3 juin 2009

Les exercices sont indépendants. L'énoncé est composé de 8 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 en revanche décrire les données manipulées et les algorithmes utilisés de manière précise.

1 Allocation de registres (6 points)

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

r0 = 13	r6 = r3	r8 = r3
r1 = 2	r5 = r5<=r6	r9 = r1
r2 = 0	ifzero r5 goto lab5	r8 = r8-r9
r4 = r0	r7 = r2	r3 = r8
r3 = r4	r7 = r7+1	goto lab4
label lab4	r2 = r7	label lab5
r5 = r1		

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. Quel nombre minimal de registres faut-il pour colorier ce graphe ?
5. Parmi les affectations de la forme $r_i = r_j$, quelles sont celles où on peut espérer mettre les deux variables dans le même registre?
6. Proposer un coloriage qui prend le minimum de registres et lorsque c'est possible met deux variables pour laquelle on a une affectation $r_i = r_j$ dans le même registre.
7. Reconstruire le programme donné en utilisant les registres correspondant au programme et en supprimant les affectations triviales $r = r$.

2 Exceptions (16 points)

Dans ce problème, on considère un petit langage arithmétique avec exceptions, appelé $\mathcal{L}$ par la suite. Un programme $\mathcal{L}$ est une suite de définitions de fonctions introduites par `def`, mutuellement récursives, suivie d'un programme principal constitué d'une expression à afficher introduite par `print`. Chaque fonction a des arguments entiers et un corps qui est une expression entière. Les expressions sont formées à partir de constantes entières, de variables, des quatre opérations arithmétiques, d'une conditionnelle de la forme `ifzero then else`, d'une construction `let in` pour introduire une variable locale, d'une construction `raise` pour lever une exception et d'une construction `try with` pour la rattraper. Voici deux exemples de programmes $\mathcal{L}$ (qu'il n'est pas nécessaire de comprendre) :

```
def f(x) =
ifzero x-1 then raise(0)
else x/2
def syra(n) =
let m = f(n) in
ifzero n - 2*m then syra(m)
else syra(3*n+1)
print
try syra(42) with x -> x
```

```
def search(x) =
ifzero x then 0 else
ifzero x/17 then raise(1) else
try  1 + search(x-17)
with y ->
  ifzero x/42 then raise(y+1) else
  try 1 + search(x-42) with z -> raise(y+z)
print
try search(123) with e -> 0-e
```

La sémantique de $\mathcal{L}$ est identique à celle de Caml, aux points suivants près :

- il y a une unique exception, contenant une valeur entière, que l'on aurait par exemple déclarée en Caml avec `exception E of int` ; la construction `raise(`e`)` correspondrait alors en Caml à `raise (E `e`)`, et la construction `try `e_1` with `x` -> `e_2 correspondrait à `try `e_1` with E `x` -> `e_2 ;
- la construction `ifzero `e_1` then `e_2` else `e_3 s'écrirait en Caml `if `e_1`=0 then `e_2` else `e_3 ;
- l'évaluation se fait de la gauche vers la droite ; ainsi, si par exemple l'évaluation de e_1 lève une exception, alors l'évaluation de e_1 `+` e_2 lèvera cette exception et e_2 ne sera pas évaluée ;
- les fonctions sont mutuellement récursives.

On se donne la syntaxe abstraite suivante pour les expressions de $\mathcal{L}$:

e	::=	n	constante entière
	|	x	variable
	|	e *op* e	opération arithmétique $op \in \{+, -, \times, /\}$
	|	`let` $x = e$ `in` e	variable locale
	|	`ifzero` e `then` e `else` e	conditionnelle
	|	$f(e, \ldots, e)$	application
	|	`raise`(e)	levée d'exception
	|	`try` e `with` $x \to e$	gestionnaire d'exception

2.1 Analyse des fonctions pouvant lever une exception

Dans cette partie, on souhaite déterminer si l'évaluation d'un programme peut conduire à une exception non rattrapée (par exemple pour le rejeter). Comme il s'agit d'une propriété non décidable de manière générale, on va se contenter d'une condition suffisante. L'idée est de déterminer *pour chaque fonction* si son évaluation est susceptible de lever une exception non rattrapée. On peut alors déterminer ce qu'il en est de l'expression qui constitue le programme principal.

Question 1 Pour le programme suivant, déterminer quelles sont les fonctions dont l'évaluation peut conduire à une exception non rattrapée (on rappelle que les fonctions sont mutuellement récursives).

```
def f(x) = raise(x)
def g(x) = ifzero x then f(x+1) else i(x)
def h(x) = ifzero x then 1 else x * h(x-1)
def i(x) = try 1 + g(x) with y -> h(y)
def j(x) = x + try g(x) with y -> f(y)
print ...
```

On se donne les types Caml suivants pour représenter la syntaxe abstraite de $\mathcal{L}$. (La nature des variables n'est pas importante dans cette partie.)

```
type binop = Add | Sub | Mul | Div
type var = ...
```

```
type expr =
  | Const of int
  | Var of var
  | Binop of binop * expr * expr
  | Letin of var * expr * expr
  | Ifzero of expr * expr * expr
  | Call of string * expr list
  | Raise of expr
  | TryWith of expr * var * expr
type def = { name : string; args : var list; body : expr; }
type program = { funs : def list; print : expr; }
```

Question 2 Écrire une fonction `expr : (string * bool) list -> expr -> bool` qui détermine si l'évaluation d'une expression est susceptible de lever une exception non rattrapée. Le premier argument est une liste associant à chaque fonction du programme un booléen indiquant si son évaluation peut conduire à une exception non rattrapée.

2.2 Code intermédiaire

Dans la suite, nous nous intéressons uniquement aux programmes où toutes les exceptions sont rattrapées.

Afin de simplifier la production de code assembleur, nous allons introduire une transformation source à source (de $\mathcal{L}$ vers un sous-ensemble de $\mathcal{L}$) où dans `try` e_1 `with` $x \to e_2$ l'expression e_1 est toujours un appel de fonction. Par exemple, la fonction `i` précédente sera transformée en

```
def i(x) = try aux(x) with y -> h(y)
def aux(x) = 1 + g(x)
```

On suppose que vous disposez d'une fonction `gen_name : unit -> string` qui génère des noms de fonctions et d'une fonction `fv : expr -> var list` qui calcule l'ensemble des variables libres d'une expression.

Question 3 Écrire une fonction `expr2expr : expr -> def list * expr` qui traduit une expression de $\mathcal{L}$ dans le sous-ensemble de $\mathcal{L}$ définit précédemment. Le résultat de cette fonction est la liste des définitions de fonctions auxiliaires et l'expression traduite. Donner le code de cette fonction correspondant aux constructeurs `Const`, `Binop` et `TryWith` de la syntaxe abstraite.

2.3 Production de code

On s'intéresse enfin à la compilation de $\mathcal{L}$ vers l'assembleur vu en cours (un aide-mémoire est donné à la fin de ce sujet). On suppose que l'analyse de portée a été réalisée et que chaque variable est directement représentée par un entier qui désigne sa position par rapport au registre `fp`. La compilation s'effectue donc sur la syntaxe abstraite présentée dans la partie précédente, avec : `type var = int`.

On suppose que le code assembleur d'une expression est produit par une fonction Caml `compile_expr : expr -> instr list`. Le code assembleur produit a pour effet de stocker la valeur de l'expression au sommet de la pile. Cette fonction commence de la façon suivante :

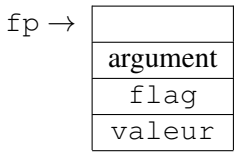
```
let rec compile_expr e =
  match e with
  | Const n -> [ PUSHI n ]
  | ...
```

Question 4 Donner la portion de code de cette fonction correspondant au filtrage des motifs `Var(x)`, `Letin(x,e1,e2)` et `Ifzero(e1,e2,e3)`.

Pour traiter la compilation des exceptions, l'exécution de chaque fonction va retourner un couple (`flag`, `valeur`) où `flag` indique si une exception a été levée ou non. Si `flag` vaut `1` alors `valeur` correspond à la valeur de l'exception et si `flag` vaut `0` alors `valeur` correspond à la valeur de retour de la fonction.

Lever une exception avec la valeur `v` correspond simplement à faire un retour de fonction avec comme valeur de retour la paire (`1`, `v`). Comme le corps d'un `try/with` est toujours un appel de fonction, la compilation de cette expression consiste à faire l'appel de fonction et de tester le `flag` de retour pour savoir s'il faut exécuter le traitement de l'exception ou s'il faut laisser la valeur de retour de la fonction (sans le flag) au sommet de la pile.

Question 5 Écrire la portion de code de la fonction `compile_expr` correspondant au filtrage des motifs `Raise(e)` et `TryWith(Call(f,[e1]),x,e2)`. Pour simplifier, nous ne considérons ici que les fonctions à un seul argument. Le tableau d'activation des fonctions est le suivant :



Pour propager la levée d'exception, la compilation d'une application correspond à faire l'appel de fonction puis de tester le `flag` de retour pour savoir si une exception a été levée. Si une exception a été levée, il faut propager l'exception en exécutant de nouveau un retour de fonction. Sinon, il faut laisser la valeur de retour de la fonction (sans le flag) au sommet de la pile.

Question 6 Écrire la portion de code de la fonction `compile_expr` correspondant au filtrage des motifs `Call(f,[e])`.

Question 7 Définir une fonction Caml `compile_def : def -> instr list` produisant le code d'une fonction de $\mathcal{L}$, en supposant que le type `def` contient, outre le nom, les arguments et le corps de la fonction, le nombre de variables locales à allouer (contenu dans le champ `locals`) :

```
type def = { name : string; args : var list; body : expr; locals : int; }
```

Question 8 Pour compiler les exceptions, il existe une technique dite du *stack cutting* qui consiste à sauter directement au gestionnaire d'exception lors de l'exécution d'un `raise`. Pourquoi ne pouvons nous pas utiliser cette technique si l'instruction `CALL` est utilisée lors de la compilation des fonctions ?

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)
`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.
`PUSHA` *label*	empile l'adresse dans le code correspondant à l'étiquette *label*
`CALL`	dépile une adresse de code a, sauvegarde *pc* et *fp* dans la pile des appels, affecte à *fp* la valeur courante de *sp* et à *pc* la valeur a.
`RETURN`	restaure les valeurs de *fp*,*sp* et *pc*.