Compilation et langages

Examen, 16 décembre 2016, durée 3h.
Documents autorisés : une feuille A4 manuscrite recto/verso.

Ce sujet étudie le langage Àlacarte présenté en préambule, et comporte 5 parties indépendantes, concernant 5 aspects de la compilation de ce langage. Chaque partie contient des questions de niveaux de difficulté variés. Les parties peuvent être traitées dans un ordre arbitraire.

Prémbule. Àlacarte : un langage de programmation avec des options

On s'intéresse à un petit langage impératif avec des valeurs optionnelles similaires au type `option` de Caml. Ainsi, `Some e` désigne une valeur optionnelle calculée par `e`, et `None` désigne une valeur optionnelle absente. Pour accéder à la valeur d'une option `e`, on se donne une construction `?e:d`, qui évalue l'expression `e` et teste la valeur obtenue :

- si `e` s'évalue en `Some v` alors le résultat est `v`,
- si `e` s'évalue en `None` alors le résultat est la valeur de l'expression `d` (qu'on appelle le résultat par défaut).

Le langage comporte des instructions `while e { b }` qui exécute le bloc d'instructions `b` tant que le résultat de l'expression `e` est différent de `0` et `ifz e { b }` qui exécute le bloc `b` si le résultat de l'expression `e` est `0`. Un programme `p` est un bloc d'instructions, suivi par une expression dont l'évaluation donne le résultat final. L'instruction d'affectation `x := { b e }` exécute le bloc d'instructions `b` puis affecte à `x` la valeur de `e`.

Par exemple, le programme

```
{ x := { 3 };
  y := { None };
  while x { y := { x := { 0 };
                   Some ((?y:42) + 1)
                 };
          };
  y
}
```

affecte `0` à `x`, affecte `Some 43` à `y`, et a pour résultat `Some 43`.

Partie I. Analyse syntaxique

On se donne la grammaire Menhir suivante :

```
%token OB CB EOF SEMI ASSIGN IFZ WHILE
%token IDENT INT PLUS LPAR RPAR
%token NONE SOME QM COLON
%start <unit> file
%%

file:
| prog EOF          {}
;

prog:
| OB block expr CB  {}
;

block:
| (* empty *)       {}
| block instr SEMI  {}
;
```

```
instr:
| IDENT ASSIGN prog       {}
| IFZ   expr OB block CB  {}
| WHILE expr OB block CB  {}
;

expr:
| INT                     {}
| IDENT                   {}
| expr PLUS expr          {}
| LPAR expr RPAR          {}
| NONE                    {}
| SOME expr               {}
| QM expr COLON expr      {}
;
```

Question 1. Donner les étapes de l'analyse ascendante du programme suivant en précisant pour chaque étape l'état de la pile, le fragment de l'entrée encore à lire, et l'action effectuée.

```
{ x := { Some 1 }; x }
```

Cette grammaire contient trois conflits *shift/reduce*. La figure 1 contient des extraits du fichier `.conflicts` généré par Menhir.

Question 2. Pour chacun de ces trois conflits, donner
- une entrée aboutissant au conflit, et
- des arbres de dérivation justifiant les différentes possibilités.

On veut résoudre l'ambiguïté de la grammaire de sorte que l'expression

```
Some 1 + ?x:2 + 3
```

soit analysée comme

```
Some ((1 + ?x:2) + 3)
```

Question 3. Quel est la construction la plus prioritaire ? La moins prioritaire ? Donner des déclarations de précédence et de priorité pour compléter la grammaire Menhir.

Partie II. Typage

Le jugement de typage $\Gamma \vdash e : \tau$ signifie que l'expression e est de type τ dans l'environnement Γ.

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

$$\frac{}{\Gamma \vdash \texttt{None} : \tau \texttt{ option}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{Some } e : \tau \texttt{ option}}$$

On note également $\Gamma \vdash p : \tau$ pour signifier que le programme p est bien typé et produit une valeur de type τ. Le jugement $\Gamma \vdash i$ décrit le bon typage des instructions et des blocs.

$$\frac{\Gamma \vdash b \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \texttt{\{ } b\ e \texttt{ \}} : \tau} \qquad \frac{}{\Gamma \vdash \emptyset} \qquad \frac{\Gamma \vdash b \qquad \Gamma \vdash i}{\Gamma \vdash b\ i\texttt{;}}$$

$$\frac{\Gamma(x) = \tau \qquad \Gamma \vdash p : \tau}{\Gamma \vdash x \texttt{ := } p} \qquad \frac{\Gamma \vdash e : \texttt{int} \qquad \Gamma \vdash b}{\Gamma \vdash \texttt{ifz } e \texttt{ \{ } b \texttt{ \}}} \qquad \frac{\Gamma \vdash e : \texttt{int} \qquad \Gamma \vdash b}{\Gamma \vdash \texttt{while } e \texttt{ \{ } b \texttt{ \}}}$$

Question 4. Donner une règle de typage pour les expressions de la forme $\texttt{?}e_1\texttt{:}e_2$.

Question 5. Soit un environnement Γ tel que $\Gamma(\texttt{x}) = \texttt{int}$ et $\Gamma(\texttt{y}) = \texttt{int option}$. Les fragments de programmes suivants sont-ils bien typés dans l'environnement Γ ? Donner si c'est possible une dérivation de typage.

```
while x { y := { Some (x+1) }; x := { ?y:0 }; };
```

```
y := { Some 1 }; 1 + y
```

Voici un type Caml pour la syntaxe abstraite des programmes Àlacarte.

```
type expr =
  | Int  of int
  | Var  of string
  | Plus of expr * expr
  | None
  | Some of expr
  | Get  of expr * expr
```

```
and instr =
  | Assign of string * prog
  | Ifz    of expr   * block
  | While  of expr   * block
and block = instr list
and prog  =
  | Prog   of block  * expr
```

```
** Conflict (shift/reduce) in state 17.
** Token involved: PLUS
** This state is reached from file after reading:

OB block SOME expr

** The derivations that appear below have the following common factor:
** (The question mark symbol (?) represents the spot where the derivations
** begin to differ.)

file
prog EOF
OB block expr CB
         (?)

** In state 17, looking ahead at PLUS, shifting is permitted
** because of the following sub-derivation:

SOME expr
     expr . PLUS expr

** In state 17, looking ahead at PLUS, reducing production
** expr -> SOME expr
** is permitted because of the following sub-derivation:

expr PLUS expr // lookahead token appears
SOME expr .

************************************************************

** Conflict (shift/reduce) in state 16.
** Token involved: PLUS
** This state is reached from file after reading:

OB block QM expr COLON expr

************************************************************

** Conflict (shift/reduce) in state 13.
** Token involved: PLUS
** This state is reached from file after reading:

OB block expr PLUS expr
```

FIGURE 1 – Extraits du fichier `.conflicts`

Question 6. Définir un type Caml `ty` pour représenter les types des programmes Àlacarte.

Question 7. On veut écrire une fonction Caml récursive `type_expr : env -> expr -> ty` telle que `type_prog env e` renvoie le type de l'expression `e` dans l'environnement `env` s'il existe, ou lève une exception `Type_error` sinon. Écrire les cas de cette fonction correspondant aux constructeurs `Plus`, `Some` et `Get`. On ne se souciera pas de la manière dont l'environnement est défini.

Partie III. Sémantique à grands pas et aplatissement de programmes

Une valeur calculée par un programme Àlacarte est soit un entier, soit l'ensemble vide, soit un ensemble contenant une valeur [1] :

$$v \ ::= \ n \mid \emptyset \mid \{v\}$$

Un état S est une fonction des variables vers les valeurs.

La valeur d'une expression e dans un état S est notée $[\![e]\!]_S$ et est définie par les équations suivantes :

$$\begin{array}{rcll} [\![n]\!]_S & = & n & \\ [\![x]\!]_S & = & S(x) & \\ [\![\texttt{None}]\!]_S & = & \emptyset & \\ [\![\texttt{Some } e]\!]_S & = & \{[\![e]\!]_S\} & \\ [\![e_1 \texttt{ + } e_2]\!]_S & = & [\![e_1]\!]_S + [\![e_2]\!]_S & \\ [\![?e_1 : e_2]\!]_S & = & v_1 & \text{si } [\![e_1]\!]_S = \{v_1\} \\ [\![?e_1 : e_2]\!]_S & = & [\![e_2]\!]_S & \text{si } [\![e_1]\!]_S = \emptyset \end{array}$$

On donne pour les programmes Àlacarte une sémantique à grands pas basée sur le jugement $S \xrightarrow{p} S' \mid v$ qui signifie que, partant de l'état S, l'exécution du programme p amène au nouvel état S' et produit la valeur v. On utilise également une version simplifiée $S \xrightarrow{b} S'$ pour la sémantique des instructions et des blocs d'instructions (qui ne produisent pas de valeur). On note ϵ le bloc d'instructions vide.

Dans les règles ci-dessous, la notation $S[x \mapsto v]$ désigne l'état S' tel que $S'(x) = v$ et pour tout $y \neq x$, $S'(y) = S(y)$.

$$\frac{S \xrightarrow{b} S_b \qquad [\![e]\!]_{S_b} = v}{S \xrightarrow{\texttt{\{ } b\ e \texttt{ \}}} S_b \mid v} \qquad \frac{}{S \xrightarrow{\epsilon} S} \qquad \frac{S \xrightarrow{b} S_b \qquad S_b \xrightarrow{i} S_i}{S \xrightarrow{b\ i\texttt{;}} S_i} \qquad \frac{S \xrightarrow{p} S_p \mid v}{S \xrightarrow{x\ \texttt{:=}\ p} S_p[x \mapsto v]}$$

$$\frac{[\![e]\!]_S = 0}{S \xrightarrow{\texttt{while } e\ \texttt{\{ } b\ \texttt{\}}} S} \qquad \frac{[\![e]\!]_S \neq 0 \qquad S \xrightarrow{b} S_b \qquad S_b \xrightarrow{\texttt{while } e\ \texttt{\{ } b\ \texttt{\}}} S_w}{S \xrightarrow{\texttt{while } e\ \texttt{\{ } b\ \texttt{\}}} S_w}$$

Question 8. Donner la ou les règles pour l'exécution des instructions de la forme `ifz e { b }`.

Pour faciliter l'analyse des programmes Àlacarte, on veut aplatir les instructions d'affectation, de sorte à transformer par exemple le programme

```
{ x := { y := { Some 2 };
         ?y:0 + 1
       };
  x }
```

en

```
{ y := { Some 2 };
  x := { ?y:0 + 1 };
  x }
```

1. Remarque : les accolades utilisées dans $\{v\}$ sont une notation mathématique pour les ensembles. Elles ne doivent pas être confondues avec les accolades présente dans la syntaxe de Àlacarte, par exemple dans le programme `{ x }`.

Question 9. Démontrer que si $S \xrightarrow{x := \{ b\ e \}} S'$, alors $S \xrightarrow{b\ x := \{ e \};} S'$.

Indice : Raisonner sur la manière dont on peut dériver $S \xrightarrow{x := \{ b\ e \}} S'$, et en déduire une dérivation de $S \xrightarrow{b\ x := \{ e \};} S'$.

Question 10. Écrire une fonction Caml récursive `flatten_block : block -> block` telle que `flatten_block b` renvoie le bloc obtenu en appliquant cette transformation d'aplatissement à toutes les instructions d'affectation contenues dans le bloc `b`.

Question 11 (Bonus)**.** Déduire de la propriété énoncée à la question 9 que tout bloc b est aplati par la transformation de la question 10 en un bloc $b^\flat$ tel que si $S \xrightarrow{b} S'$ alors $S \xrightarrow{b^\flat} S'$.

Partie IV. Génération de code MIPS

Pour représenter en mémoire les valeurs optionnelles, on propose de procéder ainsi :

- La valeur $\emptyset$ est représentée comme l'entier `0`.
- La valeur $\{v\}$ est représentée par un pointeur vers un bloc alloué dans le tas, les blocs étant structurés comme dans les TP. Ce bloc est donc constitué d'un entête qui contient l'entier `1` suivi d'un champ qui contient la valeur v.

Question 12. Décrire l'état des registres et du tas après l'exécution du code suivant, et préciser la valeur contenue dans le registre `$v0`.

```
li   $a0, 8
li   $v0, 9
syscall
li   $t0, 1
sw   $t0, 0($v0)
li   $t0, 2
sw   $t0, 4($v0)
move $t0, $v0
li   $v0, 9
syscall
sw   $t0, 4($v0)
li   $t0, 1
sw   $t0, 0($v0)
```

Question 13. Supposons que le registre `$a0` contienne la valeur {{{3}}} (c'est-à-dire la valeur de l'expression `Some (Some (Some 3))`). Donner une configuration possible du tas, en précisant les adresses des blocs représentés et leur contenu, sachant que la première adresse du tas est `0x10040000`.

Question 14. On veut écrire une fonction récursive `generate_expr : expr -> unit` telle que `generate_expr e` affiche un fragment de code MIPS qui calcule la valeur de l'expression `e` et la place dans le registre `$v0`. Écrire les cas correspondant aux constructions `Plus`, `Some` et `Get`.

Question 15 (Bonus)**.** Supposons que les registres `$a0` et `$a1` contiennent chacun une valeur de type `int option`. Écrire un fragment de code MIPS qui écrit `1` dans le registre `$v0` si les deux valeurs sont égales, et `0` sinon.

Partie V. Analyse de flot de données et optimisation

En l'état actuel, toute expression `?x:e` génère un test, pour déterminer si la valeur de `x` est de la forme $\{v\}$. On veut pouvoir se passer de ce test dans les cas où on peut prédire que la valeur de `x` a la bonne forme, et on va utiliser pour ceci une analyse du flot de données des programmes Àlacarte.

On dit qu'une variable `x` de type `t option` est *définie* à un point de programme donné si sur tout chemin d'exécution menant à ce point de programme, la dernière affectation à `x` est de la forme `x := { Some e }`.
Pour garder une analyse simple, on considérera que les affectations de la forme `x := { e' }` où `e'` n'est pas directement sous la forme `Some e` ne rendent pas `x` définie, même si l'on est capable par ailleurs de prédire que `e'` va produire une valeur $\{v\}$.

Question 16. Donner les équations de flot de données permettant de déterminer, en entrée et en sortie de chaque nœud du graphe de flot de contrôle d'un programme, l'ensemble des variables définies. Préciser les définitions des ensembles $\boldsymbol{Gen}[n]$ et $\boldsymbol{Kill}[n]$.

Question 17. Donner le graphe de flot de contrôle du programme Àlacarte suivant, résoudre les équations et déterminer les expressions pour lesquelles la génération de code peut être optimisée.

```
{ x := { Some 1 };
  y := { Some x };
  ifz 1 {
    x := { None };
    z := { Some 0 };
  };
  while ?x:0 {
    x := { Some 2 };
    y := { Some ?y:None };
    z := { Some ?x:1 };
  };
  ?x:0 + ??y:Some 0:1 + ?z:0
}
```

Question 18 (Bonus)**.** Donner des programmes Àlacarte dans lequel on est en mesure de prédire qu'une variable `x` va produire une valeur de la forme $\{v\}$, sans que cette variable ne rentre dans le critère de définition donné ci-dessus. Proposer un raffinement de l'analyse pour prendre en compte ces situations.

Aide mémoire MIPS

Voici quelques instructions MIPS susceptibles de vous être utiles (vous avez le droit d'en utiliser d'autres) :

`li` r`,` n	charge l'entier n dans le registre r
`move` r_1`,` r_2	copie le registre r_2 dans le registre r_1
`add` r_1`,` r_2`,` r_3	calcule la somme de r_2 et r_3 et la place dans r_1
`lw` r_1`,` n`(`r_2`)`	charge dans r_1 la valeur contenue 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
`beqz` r`,` L	saute à l'adresse désignée par l'étiquette L si le registre r contient `0`
`syscall`	effectue un appel système, dont la nature est donnée par le registre `$v0` ; par exemple, si `$v0` contient `9`, alors l'appel déclenché est `sbrk`, qui alloue sur le tas un nombre d'octets donné par `$a0`, et qui place dans `$v0` l'adresse de début du bloc ainsi alloué