Compilation et langages

Examen, 21 décembre 2018, durée 3h, tous documents autorisés.

Ce sujet étudie le langage `10` présenté en préambule, et comporte 4 parties indépendantes, concernant 4 aspects de ce langage. Chaque partie contient des questions de niveaux de difficulté variés. Les parties peuvent être traitées dans un ordre arbitraire.

Prémbule. `10` : un langage avec fonctions pointées

Le langage `10` est un petit langage avec un noyau impératif (avec affectation `:=`, branchement `if`/`else` et boucle `while`), dans lequel on peut manipuler des pointeurs de fonctions : la spécificité du langage est que l'on peut passer en paramètre à une fonction f un pointeur vers une autre fonction g qui sera appelée par f. Ainsi dans ce langage, une fonction f n'est plus un identifiant d'un type particulier, mais une simple variable dont la valeur est l'adresse de la fonction correspondante. Une définition

```
int f(int x, int y) := x + y
```

définit donc une variable `f` de type `(int, int)` → `int` susceptible d'être appliquée à une paire de paramètres. Notez que le corps de la fonction f est simplement constitué d'une expression, et en particulier ne contient pas d'instruction impérative.

Partie I. Analyse syntaxique

Voici un fragment de définition en Menhir de la grammaire du langage `10` (on ne montre pas la définition des symboles non terminaux `instr`, `formals`, `types` et `parameters`).

```
%token ID CST ADD LP RP SET INT ARROW
%right ADD
%start <unit>prog
%%

prog:
| decls; instr; EOF                    {}

decls:
| (* empty *)                          {}
| fun_decl; decls                      {}
| var_decl; decls                      {}

fun_decl:
| typed_id; LP; formals; RP; SET; expr  {}
```

```
var_decl:
| typed_id; SET; expr        {}

typed_id:
| typ; ID                    {}

typ:
| INT                        {}
| LP; types; RP; ARROW; typ  {}

expr:
| ID                         {}
| CST                        {}
| expr; ADD; expr            {}
| expr; LP; parameters; RP   {}
```

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

```
    int x := 1 + y + z
```

Cette grammaire génère quelques conflits. La figure 1 présente un extrait du fichier `.conflicts` produit par Menhir.

Question 2. Combien de conflits sont-ils mentionnés dans l'extrait du fichier `.conflicts` ? Pour chacun de ces conflits, donner

- sa nature,
- une entrée en syntaxe concrète aboutissant au conflit,
- des arbres de dérivation justifiant les différentes possibilités,
- des priorités sur les opérateurs ou une modification de la grammaire permettant d'éliminer le conflit.

```
** Conflict (shift/reduce) in state 19.
** Token involved: LP
** This state is reached from prog after reading:

typed_id SET expr ADD expr

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

prog
decls instr EOF
(?)

** In state 19, looking ahead at LP, reducing production
** expr -> expr ADD expr
** is permitted because of the following sub-derivation:

var_decl decls // lookahead token appears because decls can begin with LP
typed_id SET expr // lookahead token is inherited
             expr ADD expr .

** In state 19, looking ahead at LP, shifting is permitted
** because of the following sub-derivation:

var_decl decls
typed_id SET expr
             expr ADD expr
                      expr . LP parameters RP

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

typed_id SET expr

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

prog
decls instr EOF
(?)

** In state 13, looking ahead at LP, shifting is permitted
** because of the following sub-derivation:

var_decl decls
typed_id SET expr
             expr . LP parameters RP

** In state 13, looking ahead at LP, reducing production
** var_decl -> typed_id SET expr
** is permitted because of the following sub-derivation:

var_decl decls // lookahead token appears because decls can begin with LP
typed_id SET expr .
```

FIGURE 1 – Extrait du fichier `.conflicts`

Partie II. Typage

Dans le langage `10` on manipule deux formes de types : le type des entiers et les types des fonctions. On note $(T_1, \ldots, T_n) \to T$ le type des fonctions prenant n paramètres de types T_1 à T_n et produisant un résultat de type T.

Question 3. On se donne les deux définitions suivantes, dans lesquelles les types ont été omis :

```
... f(... x, ... y) := x + y
... g(... x, ... y) := f(x(y), 1)
```

Donner des types possibles pour `f` et `g`.

On décrit formellement le typage d'une expression par un jugement de typage $\Gamma \vdash e : T$ signifiant que l'expression e est de type T dans l'environnement Γ, l'environnement Γ étant une fonction qui à chaque variable associe un type. On peut justifier un jugement de typage à l'aide des règles de typage suivantes :

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

Question 4. Proposer une règle de typage pour une expression d'application de fonction de la forme $e_0(e_1, \ldots, e_n)$.

Question 5. On se place dans un environnement Γ associant à la variable `x` le type `int` et à la variable `f` le type `(int, int) → ((int) → int)`. Les expressions suivantes sont-elles bien typées ? Donner une dérivation de typage ou expliquer le problème.

1. `1 + f(x, 2)`
2. `1 + (f(x, 2))(3)`

Pour manipuler les expressions et les types du langage `10` en Caml, on se donne les deux définitions suivantes :

```
type expression =
  | Id of string
  | Cst of int
  | Add of expression * expression
  | Call of expression * expression list
```

```
type typ =
  | Int
  | Function of typ * typ list
```

Question 6. On souhaite produire une fonction `type_expression: expression -> env -> typ` calculant le type de l'expression donnée en paramètre, ou levant une exception dans le cas où l'expression est incohérente. Écrire les cas correspondant aux constructeurs `Add` et `Call`. On ne se préoccupera pas de la manière dont le type `env` est défini.

La transformation d'*extension inline* modifie le code source du programme en remplaçant un appel de fonction par le code de la fonction. Plus précisément, si la fonction f a une définition de la forme $f(T_1\ x_1, \ldots, T_n\ x_n) = e$, alors on remplacera un appel $f(e_1, \ldots, e_n)$ par l'expression e^σ, dans laquelle σ est la substitution remplaçant la variable x_i par l'expression e_i, pour tout i entre 1 et n. Formellement, la définition de l'application à une expression d'une substitution σ remplaçant x_i par e_i est :

$$\begin{aligned}
(x_i)^\sigma &= e_i \\
x^\sigma &= x \qquad\qquad x \notin \{x_1, \ldots, x_n\} \\
n^\sigma &= n \\
(e_1 + e_2)^\sigma &= e_1^\sigma + e_2^\sigma \\
(e_0(e_1, \ldots, e_n))^\sigma &= e_0^\sigma(e_1^\sigma, \ldots, e_n^\sigma)
\end{aligned}$$

On souhaite démontrer que cette optimisation respecte le bon typage d'un programme.

Question 7. Supposons que f est définie par $f(T_1\ x_1, \ldots, T_n\ x_n) = e$ et que σ est la substitution remplaçant x_i par e_i. Démontrer que si $\Gamma \vdash f(e_1, \ldots, e_n) : T$, alors $\Gamma \vdash e^\sigma : T$.

Partie III. Génération de code MIPS

Les conventions générales pour la compilation des expressions reprennent celles des TP, avec des modifications dans les mécanismes relatifs aux appels de fonctions. Ainsi :

- Le code produit pour le calcul d'une expression e a pour effet de placer la valeur de e au sommet de la pile.
- La valeur d'une fonction est un pointeur vers son code.
- Lors d'un appel de fonction $e_0(e_1, \ldots, e_n)$, l'appelant place au sommet de la pile les valeurs des expressions e_n à e_0 (dans cet ordre), puis passe la main à la fonction dont l'adresse est donnée par la valeur de e_0. Après l'appel, l'appelant retire de la pile les valeurs e_0 à e_n.
- Comme dans les TP, l'appelé doit sauvegarder les registres `$fp` et `$ra` avant de commencer son calcul, puis restaurer ces mêmes registres avant de rendre la main à l'appelant.

Question 8. On se donne les trois définitions de fonctions `10` suivantes (où l'expression e n'est pas détaillée) :

```
int f(int x, int y) = e
int g(int z) = 3 + f(z, 2) + 3
int h(int t) = t + 1
```

On exécute ensuite l'instruction `print(g(h(5)))`. Dessiner la pile et préciser le contenu de chaque case au moment où l'expression e vient d'être évaluée (c'est-à-dire avant la fin de l'appel à la fonction `f`).

Question 9. On souhaite définir une fonction `compile_expression: expression -> unit` affichant sur la sortie standard le code Mips calculant le résultat de l'expression fournie en paramètre. Écrire les cas correspondant aux constructeurs `Add` et `Call`.

Partie IV. Analyse de flot de données et optimisation

On souhaite détecter des situations dans lesquelles le même appel de fonction est effectué plusieurs fois, afin de n'effectuer qu'une seule fois le calcul correspondant. Ainsi par exemple, le code ci-dessous à gauche pourrait être simplifié en le code de droite.

```
x := 1 + f(2, 3);
y := 2;
z := f(2, 3);
```

```
w := f(2, 3);
x := 1 + w;
y := 2;
z := w;
```

Question 10. À quelles conditions une telle transformation est-elle légitime ? Quelle particularité de notre langage garantit que ces conditions sont réunies ?

Pour systématiser cette transformation nous devons caractériser les appels de fonctions qui sont effectués à coup sûr. On appelle *expression disponible* en un point de programme p une expression e dont la valeur est calculée avant le point de programme p dans tout chemin d'exécution menant à p.

Question 11.

1. Donner un exemple dans lequel l'égalité syntaxique entre deux expressions calculées en deux point de programmes p_1 et p_2 ne garantit pas que les valeurs calculées soient les mêmes.
2. Donner une condition suffisante simple pour que l'égalité syntaxique entre deux appels de fonction garantisse l'égalité des valeurs calculées.

Question 12. Donner des équations de flot de données permettant de déterminer les appels de fonction disponibles en entrée et en sortie de chaque point de programme.

Question 13. Donner le graphe de flot de contrôle du programme suivant et y résoudre les équations de flot de données. Préciser quels appels sont déjà disponibles au moment où il sont effectués et donner une version optimisée de ce programme.

```
(* On suppose a, b, c définies avec des valeurs inconnues. *)
int f(int x) := x + a
int g(int x, int y) := x + y + a
int h(int x) := x + a + 1

b := c + f(2);
while (b) {
  if (c) { b := h(1) + f(2); }
  else { b := h(1); }
  print(h(1));
}
print(h(1));
print(f(2));
```

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
`b` L	saute à l'adresse désignée par l'étiquette L
`jr` r	saute à l'adresse contenue dans le registre r
`jalr` r	saute à l'adresse contenue dans le registre r et définit le registre `$ra`
`syscall`	effectue un appel système, dont la nature est donnée par le registre `$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
`.word` n `.word` L	à utiliser dans la zone `.data` indiquent que l'entier n et l'adresse de l'étiquette L sont stockées dans les prochains mots libres de la mémoire statique