Compilation — Examen — Automne 2024

Durée 2h. Documents de cours et notes personnelles autorisés.

JO, un langage avec des `match`

On s'intéresse à un langage basé sur des types algébriques et muni d'une opération de filtrage. L'utilisateur peut définir un type et des constructeurs par une déclaration de la forme suivante, similaire à ce qui existe en Caml. Le mot-clé `tagged` introduit un nouveau nom de type, et est suivi de la déclaration d'un certain nombre de constructeurs et des types de leurs paramètres.

```
tagged nat = Zero: () -> nat
           | Succ: (nat) -> nat

tagged btree = Leaf: (nat) -> btree
             | Node: (btree * btree) -> btree
```

On construit alors une donnée par application de constructeurs à des paramètres du bon type.

```
x = Node(Leaf(Zero()), Leaf(Succ(Zero())))
```

Enfin, on peut utiliser une opération de filtrage `match ... with` pour raisonner par cas sur la forme d'une donnée.

```
function height(t) = match t with
    Leaf(n) -> Zero()
  | Node(t1, t2) -> max(height(t1), height(t2))
```

Une telle opération de filtrage sélectionne et exécute la première branche dont la forme correspond à la forme de l'argument `t`, et échoue au cas où il n'existe aucune branche adéquate.

Partie I. Analyse syntaxique

On se donne la grammaire Menhir suivante pour les expression de JO. Le symbole non terminal `expr` désigne une expression quelconque, `params` une liste d'expressions séparées par des virgules, et `case` et `cases` des cas de filtrage. On omet ici les règles définissant le symbole non terminal `pattern`, désignant un motif de filtrage.

```
%token IDENT LPAR RPAR COMMA ADD
%token MATCH WITH ARROW BAR
%start <unit> expr
%%

expr:
| IDENT                    {}
| IDENT LPAR params RPAR   {}
| expr ADD expr            {}
| MATCH expr WITH cases    {}
```

```
params:
| expr                     {}
| expr COMMA params        {}

case:
| pattern ARROW expr       {}

cases:
| case                     {}
| case BAR cases           {}
```

Questions.

1. Donner les étapes de l'analyse ascendante de l'expression `C(x + y, z)`.
   Indication : cette expression correspond à la séquence de lexèmes `IDENT LPAR IDENT ADD IDENT COMMA IDENT RPAR`
2. Dans l'état actuel de la grammaire, l'une de ces expressions n'est pas reconnue :
   - `C(x)`
   - `C()`
   - `C(x, y, z)`

   Laquelle ? Comment modifier la grammaire pour que ces trois expressions soient reconnues ?
3. L'annexe montre un extrait du fichier `.conflict` généré par `menhir` à propos de deux conflits. Pour chacun de ces conflits, donner :
   - une entrée sur laquelle il se manifeste,
   - des arbres de dérivation justifiant les différentes possibilités,
   - une manière d'ajuster la grammaire pour le faire disparaître.

Partie II. Typage

Les seuls types admis pour une expression ou une variable dans le langage JO sont les identifiants des types introduits par l'utilisateur par une déclaration `tagged`. En interne, on utilise également des types particuliers de la forme $T_1 \times \ldots \times T_n \to T$

pour les constructeurs, et de la forme $T_1 \to T_2$ pour les cas de filtrage (notez qu'aucun de ces deux derniers éléments n'est à lui seul une expression bien formée).

Un environnement de typage Γ associe des types de la forme adaptée aux variables et aux constructeurs. On note $\Gamma[x_1 : T_1, \dots, x_n : T_n]$ l'environnement Γ' tel que $\Gamma'(x_k) = T_k$ et pour tout $y \notin \{x_1, \dots, x_n\}$, $\Gamma'(y) = \Gamma(y)$. Le jugement de typage $\Gamma \vdash e : T$ signifie que l'expression e est cohérente et de type T dans l'environnement Γ. On l'étend à un jugement $\Gamma \vdash p \texttt{ -> } e : T_1 \to T_2$ de typage des cas de filtrage.

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma(C) = T_1 \times \dots \times T_n \to T \qquad (\Gamma \vdash e_k : T_k)_{\forall 1 \le k \le n}}{\Gamma \vdash C(e_1, \dots, e_n) : T}$$

$$\frac{\Gamma \vdash e : T_1 \qquad (\Gamma \vdash c_k : T_1 \to T_2)_{\forall 1 \le k \le n}}{\Gamma \vdash \texttt{match } e \texttt{ with } c_1 \texttt{ | } \dots \texttt{ | } c_n : T_2} \qquad \frac{\Gamma(C) = T_1 \times \dots \times T_n \to T \qquad \Gamma[x_1 : T_1, \dots, x_n : T_n] \vdash e : T'}{\Gamma \vdash C(x_1, \dots, x_n) \texttt{ -> } e : T \to T'}$$

Questions.

4. On considère l'environnement $\Gamma = \{\texttt{Zero : Nat}, \texttt{Succ : Nat} \to \texttt{Nat}, \texttt{Leaf : Nat} \to \texttt{btree}, \texttt{Node : btree}\times\texttt{btree} \to \texttt{btree}\}$ correspondant aux déclarations données en introduction. Supposons que l'expression `e` est telle que l'on ait une dérivation du jugement $\Gamma \vdash$ `e : btree`. En déduire une dérivation de typage pour l'expression
   `match e with Leaf(n) -> e | Node(t1, t2) -> Node(t2, t1)`
5. Proposer une déclaration JO pour un type correspondant aux booléens, et donner un code JO correspondant à la conjonction e_1 `&&` e_2, avec e_1 et e_2 deux expressions quelconques.
6. Proposer une déclaration JO pour un type représentant une paire, et donner un code JO effectuant l'équivalent de l'opération `fst(x)` en Caml. En quoi le type des paires JO est-il différent du type des paires en Caml ?
7. Expliquer quel peut être l'intérêt d'un type de la forme

```
tagged flag = Flag of bool
```

5 lignes max, plus éventuel exemple en JO ou en Caml.

Partie III. Simplification de programmes

Les valeurs calculées par les expressions JO sont des constructeurs appliqués à des valeurs :

$v ::= \texttt{C}(v_1, \dots, v_n)$

Un environnement ρ est une fonction des variables vers les valeurs. On définit l'évaluation d'une expression JO avec deux fonctions :

— `eval`(e, ρ) définit la valeur de l'expression e dans l'environnement ρ

— `mw`(v, cs, ρ) définit le résultat du filtrage de la valeur v par la liste de cas cs, dans l'environnement ρ.

Ces deux fonctions sont caractérisées par les équations suivantes :

$$\begin{aligned}
\texttt{eval}(x, \rho) &= \rho(x) \\
\texttt{eval}(\texttt{C}(e_1, \dots, e_n), \rho) &= \texttt{C}(\texttt{eval}(e_1, \rho), \dots, \texttt{eval}(e_n, \rho)) \\
\texttt{eval}(\texttt{match } e \texttt{ with } cs, \rho) &= \texttt{mw}(\texttt{eval}(e, \rho), cs, \rho) \\
\\
\texttt{mw}(\texttt{C}(v_1, \dots, v_n), \texttt{C}(x_1, \dots, x_n) \texttt{ -> } e \texttt{ | } cs, \rho) &= \texttt{eval}(e, \rho[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]) \\
\texttt{mw}(\texttt{C}(v_1, \dots, v_n), \texttt{D}(x_1, \dots, x_k) \texttt{ -> } e \texttt{ | } cs, \rho) &= \texttt{mw}(\texttt{C}(v_1, \dots, v_n), cs, \rho) \qquad \texttt{C} \neq \texttt{D}
\end{aligned}$$

où $\rho[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]$ désigne l'environnement ρ étendu par les associations aux variables x_k des valeurs respectives v_k.

Pour simplifier les séquences de cas, on introduit une opération $cs \setminus \texttt{C}$ qui retire d'une séquence cs les branches associées au constructeur `C`, et $\sigma(cs)$ qui ne garde de la séquence cs que la première branche associée à chaque constructeur. Ces opérations sont définies par les équations suivantes, où ε désigne une séquence vide.

$$\begin{aligned}
\varepsilon \setminus \texttt{C} &= \varepsilon \\
(\texttt{C}(x_1, \dots, x_n) \texttt{ -> } e \texttt{ | } cs) \setminus \texttt{C} &= cs \setminus \texttt{C} \\
(\texttt{D}(x_1, \dots, x_n) \texttt{ -> } e \texttt{ | } cs) \setminus \texttt{C} &= \texttt{D}(x_1, \dots, x_n) \texttt{ -> } e \texttt{ | } (\texttt{cs}\setminus\texttt{C}) \qquad \texttt{C} \neq \texttt{D} \\
\\
\sigma(\varepsilon) &= \varepsilon \\
\sigma(\texttt{C}(x_1, \dots, x_n) \texttt{ -> } e \texttt{ | } cs) &= \texttt{C}(x_1, \dots, x_n) \texttt{ -> } e \texttt{ | } \sigma(cs \setminus \texttt{C})
\end{aligned}$$

Questions.

8. Montrer que pour tout environnement ρ et tous constructeurs $\texttt{C} \neq \texttt{D}$, on a
   $\texttt{mw}(\texttt{C}(v_1, \dots, v_n), cs, \rho) = \texttt{mw}(\texttt{C}(v_1, \dots, v_n), (cs \setminus \texttt{D}), \rho)$
9. En déduire que $\texttt{eval}(\texttt{match } e \texttt{ with } cs, \rho) = \texttt{eval}(\texttt{match } e \texttt{ with } \sigma(cs), \rho)$

Partie IV. Génération de code assembleur

Pour chaque définition de type `tagged` T `=` $\texttt{C}_1$`: ... | ... |` $\texttt{C}_n$`: ...` on associe à chaque constructeur un entier, dans l'ordre et en partant de 0. Pour représenter en mémoire une donnée $\texttt{C}(v_1, \ldots, v_n)$, on utilise un bloc à $n+1$ champs alloué dans le tas. Le premier champ contient l'entier désignant le constructeur `C`, et les champs à partir du deuxième contiennent les valeurs v_1 à v_n. La valeur de cette donnée est l'adresse du bloc.

Ainsi, avec les exemples donnés dans les pages précédentes, la valeur `Zero()` est représentée par un bloc contenant un unique champ avec l'entier 0, et la valeur `Node(Leaf(Zero()), Leaf(Succ(Zero())))` est représentée par un bloc contenant trois champs, dont le premier contient l'entier 1 et les suivants les adresses des deux blocs alloués pour les paramètres `Leaf(Zero())` et `Leaf(Succ(Zero()))`.

Questions.

10. Supposons que le registre `$a0` contienne la valeur `Node(Leaf(Succ(Zero())), Node(Leaf(Zero()), Leaf(Zero())))`. Donner une configuration possible du tas, en précisant les adresses des blocs représentés et leur contenu, sachant que la première adresse du tas est `0x10040000` et qu'un mot fait 4 octets.
11. Décrire le contenu des registres et du tas à chaque étape de l'exécution du code assembleur suivant, et donner une valeur JO pouvant correspondre au contenu final de `$t0` (en utilisant les types déjà introduits).

```
  li    $a0, 4
  li    $v0, 9
  syscall
  li    $t0, 0
  sw    $t0, 0($v0)
  move  $t0, $v0
  li    $a0, 8
  li    $v0, 9
  syscall
  li    $t1, 1
  sw    $t1, 0($v0)
  sw    $t0, 4($v0)
  move  $t0, $v0
```

On propose de compiler une instruction `match` e `with` ... en utilisant une table de sauts, c'est-à-dire un bloc en mémoire ayant un champ pour chaque constructeur du type de e, le champ numéro k contenant l'adresse du code à exécuter si e est de la forme $\texttt{C}_k(v_1, \ldots, v_n)$. La table de saut sera stockée dans la partie données (`.data`) de la mémoire. Considérons l'expression

```
match x with
    Leaf(n) -> x
  | Node(t1, t2) -> Node(t2, t1)
```

Questions.

12. Décrire la forme de la table de saut de l'opération `match` ci-dessus, et donner un code MIPS permettant de la créer et de l'initialiser. Au besoin, vous pouvez supposer que certaines étiquettes ont été introduites dans le code, à des endroits que vous décrirez.
13. Donner trois fragments de code MIPS, respectivement pour l'opération elle-même et pour les deux branches, sachant que l'on suppose que la valeur de la variable `x` est stockée dans le registre `$t0` et que l'on souhaite placer le résultat dans le registre `$v0`.

Annexe. 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
`la` r, L	charge l'adresse désignée par l'étiquette L 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
`jr` r	saute à l'adresse donnée par le registre r
`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

Annexe. Extrait de fichier `.conflicts`

```
** Conflict (shift/reduce) in state 22.
** Token involved: ADD
** This state is reached from main after reading:

MATCH expr WITH pattern ARROW expr

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

main
expr EOF
(?)

** In state 22, looking ahead at ADD, reducing production
** case -> pattern ARROW expr
** is permitted because of the following sub-derivation:

expr ADD expr // lookahead token appears
MATCH expr WITH cases // lookahead token is inherited
                case // lookahead token is inherited
                pattern ARROW expr .

** In state 22, looking ahead at ADD, shifting is permitted
** because of the following sub-derivation:

MATCH expr WITH cases
                case
                pattern ARROW expr
                              expr . ADD expr

** Conflict (shift/reduce) in state 24.
** Token involved: BAR
** This state is reached from main after reading:

MATCH expr WITH pattern ARROW MATCH expr WITH case

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

main
expr EOF
MATCH expr WITH cases
                (?)

** In state 24, looking ahead at BAR, reducing production
** cases -> case
** is permitted because of the following sub-derivation:

case BAR cases // lookahead token appears
pattern ARROW expr // lookahead token is inherited
              MATCH expr WITH cases // lookahead token is inherited
                              case .

** In state 24, looking ahead at BAR, shifting is permitted
** because of the following sub-derivation:

case
pattern ARROW expr
              MATCH expr WITH cases
                              case . BAR cases
```