Compilation et langages

Examen, 22 décembre 2017, durée 3h.
Documents autorisés : une feuille A4 manuscrite recto/verso.

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

Prémbule. Odyssey : un langage de *switch*

On s'intéresse à un petit langage impératif avec des données étiquetées, similaires aux types somme de Caml. L'utilisateur peut définir un type et des constructeurs par une déclaration de la forme

```
tagged btree =
    Leaf:                          (* Constructeur sans contenu          *)
  | Node: btree * bool * btree     (* Constructeur avec trois paramètres *)
```

On construit une donnée par application de son constructeur à des paramètres du bon type

```
x = Node(Leaf(), False(), Leaf())
```

Les champs 1 à n d'une donnée sont accessibles grâce aux fonctions primitives `proj_1` à `proj_n`. Enfin, on peut utiliser une instruction conditionnelle généralisée `switch` pour raisonner par cas sur la structure d'une donnée

```
int height(btree x) {
  switch x
    case Leaf: tmp = Zero()
    case Node: left_height = height(proj_1(x));
               right_height = height(proj_3(x));
               tmp = max(left_height, right_height)
    case Leaf: tmp = Succ(Zero())
    ;
  result = tmp
}
```

La variable `x` étant associée à une donnée `C(v1,...,vn)`, une conditionnelle généralisée `switch x` exécute le bloc d'instruction correspondant à la première branche `case C`. Si `C` apparaît plusieurs fois, seule la première branche est exécutée. Si `C` n'apparaît pas, il ne se passe rien. Dans tous les cas, l'exécution reprend ensuite après l'instruction `switch`. En l'occurence, l'exemple précédent calcule et stocke dans une variable `tmp` la hauteur de l'arbre passé en paramètre, puis transfère cette valeur dans la variable spéciale `result`.

Partie I. Analyse syntaxique

On se donne la grammaire Menhir suivante pour les blocs d'instructions d'Odyssey (les définitions de types et de fonctions sont supposées faites au début du fichier et ne sont pas montrées) :

```
%token SEMI SWITCH CASE COLON
%token IDENT EQ LPAR RPAR COMMA
%%

block:
| instr                          {}
| instr SEMI block               {}

instr:
| IDENT EQ expr                  {}
| SWITCH IDENT cases             {}
```

```
expr:
| IDENT                          {}
| IDENT LPAR params RPAR         {}

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

cases:
| (* empty *)                    {}
| CASE IDENT COLON block cases   {}
```

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

```
switch x; y = C(x, z)
```

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

Question 2. Pour chacun de ces conflits, donner
- une entrée aboutissant au conflit,
- des arbres de dérivation justifiant les différentes possibilités,
- des priorités des opérateurs correspondant à chaque possibilité.

Question 3. 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 ?

Partie II. Typage

Les seuls types admis pour une expression ou une variable dans le langage Odyssey sont les identifiants des types introduits par l'utilisateur par une déclaration `tagged`. En interne, on utilise aussi des produits de types $T_1 \times \ldots \times T_n$ correspondant à des n-uplets. On désignera par $\vec{T}$ un produit de types quelconque éventuellement vide, et par `unit` le produit vide. Les constructeurs enfin ont des types de la forme $\vec{T}_p \to T$. Un environnement de typage Γ associe des types aux variables et aux constructeurs. On note $\Gamma\{x : T\}$ l'environnement Γ' tel que $\Gamma'(x) = T$ et pour tout $y \neq x$, $\Gamma'(y) = \Gamma(y)$.
Le jugement de typage $\Gamma \vdash e : T$ signifie que l'expression e est de type T dans l'environnement Γ.

$$\frac{\Gamma(C) = T_1 \times \ldots \times T_n \to T \qquad \forall 1 \leq k \leq n,\ \Gamma \vdash e_k : T_k}{\Gamma \vdash C(e_1, \ldots, e_n) : T} \qquad \frac{\Gamma \vdash e : T_1 \times \ldots \times T_n}{\Gamma \vdash \texttt{proj\_k}(e) : T_k}$$

Le jugement de typage $\Gamma \vdash p$ exprime le bon typage d'un programme p, et peut aussi s'appliquer à une instruction ou un bloc d'instructions. Il s'appuie sur un jugement de typage $\Gamma \vdash_{x:T} \texttt{case}\ C\texttt{:}\ b$ exprimant que `case` C`:` b est un cas bien typé pour un *switch* sur une variable x de type T.

$$\frac{\Gamma\{C_1 : \vec{T}_1 \to T\} \ldots \{C_n : \vec{T}_n \to T\} \vdash p}{\Gamma \vdash \texttt{tagged}\ T\ \texttt{=}\ C_1 : \vec{T}_1\ \texttt{|}\ \ldots\ \texttt{|}\ C_n : \vec{T}_n\quad p} \qquad \frac{\forall k,\ \Gamma \vdash i_k}{\Gamma \vdash i_1; \ldots; i_n} \qquad \frac{\Gamma(x) = T \qquad \Gamma \vdash e : T}{\Gamma \vdash x\ \texttt{=}\ e}$$

$$\frac{\Gamma(C) = \vec{T}_p \to T \qquad \Gamma\{x : \vec{T}_p\} \vdash b}{\Gamma \vdash_{x:T} \texttt{case}\ C\texttt{:}\ b} \qquad \frac{\Gamma(x) = T \qquad \forall k,\ \Gamma \vdash_{x:T} \texttt{case}\ C_k\texttt{:}\ b_k}{\Gamma \vdash \texttt{switch}\ x\ \texttt{case}\ C_1 : b_1\ \ldots\ \texttt{case}\ C_n : b_n}$$

On propose la déclaration de type suivante pour représenter les booléens dans Odyssey :

```
tagged bool =
    True:     (* Pas de contenu *)
  | False:    (* Pas de contenu *)
```

Question 4. Dans l'environnement
$\Gamma = \{$ `Leaf : () → btree, Node : btree × bool × btree → btree, False : () → bool, x : btree` $\}$,
le bloc suivant est-il bien typé ? Pourquoi ?

```
x = Node(Leaf(), False(), Leaf());
switch x
  case Leaf: x = proj_1(x)
  case Node: x = proj_3(x)
```

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

SWITCH IDENT CASE IDENT COLON instr

** The derivations that appear below have the following common factor:
block

** In state 16, looking ahead at SEMI, reducing production
** block -> instr
** is permitted because of the following sub-derivation:
instr SEMI block // lookahead token appears
SWITCH IDENT cases // lookahead token is inherited
             CASE IDENT COLON block cases // lookahead token is inherited
                                          // because cases can vanish
                               instr .

** In state 16, looking ahead at SEMI, shifting is permitted
** because of the following sub-derivation:
instr
SWITCH IDENT cases
             CASE IDENT COLON block cases
                               instr . SEMI block

** Conflict (shift/reduce) in state 2.
** Token involved: CASE
** This state is reached from file after reading:

SWITCH IDENT CASE IDENT COLON SWITCH IDENT

** The derivations that appear below have the following common factor:
block
instr
SWITCH IDENT cases

** In state 2, looking ahead at CASE, reducing production
** cases ->
** is permitted because of the following sub-derivation:
CASE IDENT COLON block cases // lookahead token appears because cases can
                             // begin with CASE
                 instr // lookahead token is inherited
                 SWITCH IDENT cases // lookahead token is inherited
                 .

** In state 2, looking ahead at CASE, shifting is permitted
** because of the following sub-derivation:
CASE IDENT COLON block cases
                 instr
                 SWITCH IDENT cases
                               . CASE IDENT COLON block cases
```

FIGURE 1 – Extraits du fichier `.conflicts`

Question 5. Écrire un bloc d'instructions Odyssey correspondant à la construction usuelle `if (`e`) { `b_1` } else { `b_2` }`, en considérant que e est une expression quelconque, et b_1, b_2 des blocs d'instructions quelconques.
Montrer ensuite que si $\Gamma \vdash e$: `bool`, $\Gamma \vdash b_1$ et $\Gamma \vdash b_2$, avec Γ l'environnement donnant leurs types aux constructeurs `True` et `False`, alors le code que vous avez proposé est lui aussi bien typé.

Question 6. Proposer une déclaration Odyssey pour un type représentant une paire, et donner un code Odyssey effectuant l'équivalent de l'instruction `x := fst(y)` en Caml. En quoi le type des paires Odyssey est-il différent du type des paires en Caml ?

Question 7. Expliquer quel peut être l'intérêt d'un type de la forme

```
tagged flag = Flag: bool
```

5 lignes max, plus éventuel exemple en Odyssey ou Caml

Partie III. Sémantique à grands pas et simplification de programmes

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

$$v \ ::= \ C(v_1,\ldots,v_n)$$

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 :

$$\begin{array}{rcl} [\![x]\!]_S & = & S(x) \\ [\![C(e_1,\ldots,e_n)]\!]_S & = & C([\![e_1]\!]_S,\ldots,[\![e_n]\!]_S) \end{array}$$

On donne pour les programmes Odyssey une sémantique à grands pas basée sur le jugement $S, p \implies S'$ qui signifie que, partant de l'état S, l'exécution du programme (ou bloc d'instructions) p mène au nouvel état S'. Dans les règles ci-dessous, la notation $S[x \mapsto v]$ désigne l'état S' tel que $S'(x) = v$ et pour tout $y \neq x$, $S'(y) = S(y)$. On note ρ pour une séquence arbitraire de `case` et ϵ pour une séquence vide.

$$\frac{S, i \implies S' \qquad S', b \implies S''}{S, i\,;b \implies S''} \qquad \frac{[\![e]\!]_S = v}{S, x \texttt{ = } e \implies S[x \mapsto v]} \qquad \frac{}{S, \texttt{switch } x\ \epsilon \implies S}$$

$$\frac{S(x) = C(v_1,\ldots,v_n) \qquad S, b \implies S'}{S, \texttt{switch } x \texttt{ case } C\texttt{:}b\ \ \rho \implies S'} \qquad \frac{S(x) = D(v_1,\ldots,v_n) \qquad C \neq D \qquad S, \texttt{switch } x\ \rho \implies S'}{S, \texttt{switch } x \texttt{ case } C\texttt{:}b\ \ \rho \implies S'}$$

Question 8. Dériver le résultat du programme

```
x = Node(Leaf(), True(), Node(Leaf(), False(), Leaf()));
y = True();
switch x
  case Leaf: y = False()
  case Node: x = proj_3(x)
```

Pour simplifier les séquences de `case` on introduit une opération $\rho \setminus C$ qui retire d'une séquence ρ les branches associées au constructeur C, et $\sigma(\rho)$, qui ne garde de la séquence ρ que la première branche associée à chaque constructeur. Ces opérations sont définies par les équations suivantes

$$\begin{array}{rcll} \epsilon \setminus C & = & \epsilon & \\ (\texttt{case } C\texttt{:}b\ \ \rho) \setminus C & = & \rho \setminus C & \\ (\texttt{case } D\texttt{:}b\ \ \rho) \setminus C & = & \texttt{case } D\texttt{:}b\ \ (\rho \setminus C) & \text{si } D \neq C \end{array}$$

$$\begin{array}{rcl} \sigma(\epsilon) & = & \epsilon \\ \sigma(\texttt{case } C\texttt{:}b\ \ \rho) & = & \texttt{case } C\texttt{:}b\ \ \sigma(\rho \setminus C) \end{array}$$

Question 9. Montrer que pour tout état S tel que $S(x) = C(v_1,\ldots,v_n)$ et tout constructeur $D \neq C$, on a S, `switch` x $\rho \implies S'$ si et seulement si S, `switch` x $(\rho \setminus D) \implies S'$.

Question 10. En déduire que S, `switch` x $\rho \implies S'$ si et seulement si S, `switch` x $\sigma(\rho) \implies S'$.

Question 11. Déduire des questions précédentes une transformation applicable à une instruction `switch` x ρ, en supposant que le programme est bien typé et que le type T de x est défini par un unique constructeur.

Partie IV. Génération de code MIPS

Pour chaque définition de type `tagged` T `=` $C_1 : \vec{T}_1$ `|` ... `|` $C_n : \vec{T}_n$ on associe à chaque constructeur un entier, dans l'ordre et en partant de 1.

Pour représenter en mémoire une donnée `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 `False()` est représentée par un bloc content un unique champ avec l'entier 2, et la valeur `Node(Leaf(),False(),Leaf())` est représentée par un bloc contenant quatre champs, dont le premier contient également l'entier 2, et les suivants les adresses des trois blocs alloués pour les trois paramètres.

Question 12. Supposons que le registre `$a0` contienne la valeur

```
Node(Leaf(), True(), Node(Leaf(), False(), Leaf()))
```

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.

On propose de compiler une instruction `switch` x ρ en utilisant une table de sauts, c'est-à-dire un bloc en mémoire ayant un champ pour chaque constructeur du type de x, le champ numéro i contenant l'adresse du code à exécuter si x est de la forme $C_i(v_1,\ldots,v_n)$. La table de saut sera stockée dans la partie données (`.data`) de la mémoire.

Question 13. En supposant que `x` est une variable du type `t` défini par

```
tagged t = A: t * t * t
         | B: t
         | C:
```

et que le registre `$s0` est alloué à cette variable, écrire un code MIPS exécutant l'instruction

```
switch x
  case A: x = B(x)
  case B: x = proj_1(x)
  case A: x = proj_2(x)
```

ainsi qu'un code définissant sa table de sauts.
Cette question contient un certain nombre de ramifications. Des points sont alloués à chacun des aspects.

Partie V. Analyse de flot de données et optimisation

En l'état actuel, toute instruction `switch` x ρ nécessite une lecture de la table des sauts à l'exécution. Cependant, cette table n'est pas utile dans les cas où l'on peut prédire statiquement le constructeur de la valeur associée à la variable x : on pourrait alors se contenter d'exécuter le bloc de code correspondant.

On va utiliser une analyse de flot de données des programmes Odyssey pour détecter les points où cette optimisation est possible. On dit qu'un constructeur `C` est *compatible* avec une variable `x` à un point de programme donné dans les cas suivants :

— après une instruction `x = C(e1,...,en)`, le seul constructeur compatible avec `x` est `C` ;

— après une instruction `x = y`, les constructeurs compatibles avec `x` sont ceux qui étaient compatibles avec `y` ;
— après toute autre instruction `x = e`, tous les constructeurs sont compatibles avec `x`.

Question 14. 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 paires `(x,C)` telles que le constructeur `C` est compatible avec la variable `x`. Préciser les définitions des ensembles $Gen[n]$ et $Kill[n]$.

Question 15. Donner le graphe de flot de contrôle du programme Odyssey suivant

```
x = C();
switch y
  case A: x = proj_1(y)
  case B: switch x
            case C: y = A(x)
            case D: y = B()
            case E: y = A(C())
            ;
          x = D()
  ;
switch x
  case C: x = D()
  ;
result = y
```

Question 16. Pour le programme de la question précédente, résoudre les équations de flot de données et déterminer les instructions pour lesquelles l'optimisation peut être réalisée.

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
`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	à utiliser dans la zone `.data`
`.word` L	indiquent que l'entier n et l'adresse de l'étiquette L sont stockées dans les prochains mots libres de la mémoire statique