Master mention informatique
Spécialité STL

MI030 – Analyse des programmes et sémantique (APS)
Premier examen réparti
Lundi 14 mars 2010, 13h30 – 15h30

Directives

1. Le contrôle dure 2h00.
2. Le total des points des questions est de 30, mais la note obtenue sera ramenée sur 20 (soit n la note sur 30, la note de l'examen sera $e = si\ \mathrm{n} > 20\ alors\ 20\ sinon\ n$.
3. Tous les documents sont autorisés.
4. Tous les appareils électroniques sont **prohibés** (y compris les téléphones portables, les assistants numériques personnels et les agendas électroniques).

Les ZF-expressions

Les ZF-expressions sont utilisées dans les langages fonctionnels comme Miranda et Haskell pour construire des listes en compréhension, un peu à la manière des définitions d'ensembles en compréhension. Souvent appelées « *list comprehensions* », David Turner les a nommées ZF-expressions en l'honneur de la théorie des ensembles telle que définie par Zermelo Fränkel.

L'idée générale est de mimer les définitions d'ensembles en compréhension telles que $\{x \mid x \in \mathbb{N} \wedge pair(x)\}$ pour définir les entiers naturels pairs. Une ZF-expression remplace les ensembles par des listes. Par exemple, pour construire la liste des entiers pairs appartenant à la liste `[1, 2, 7, 12, 16]`, on écrira :

```
[ x | x <- [1,2,7,12,16], x mod 2 = 0 ]
```

La liste résultant de cette expression est `[2,12,16]`.

Décortiquons un peu cette expressions. Elle est formée de deux parties séparées par la barre verticale. La première partie, ici la variable `x`, exprime les résultats qui devront être inclus dans la liste résultante. La seconde partie, à droite de la barre verticale, sert à produire les valeurs pour la première partie, à gauche.

Dans la ZF-expression ci-dessus, la partie gauche se résume à une variable, mais en fait cela pourrait être une expression comme dans l'exemple suivant :

```
[ x * x | x <- [1,2,7,12,16]; x mod 2 = 0 ]
```

dont le résultat est `[4,144,256]`.

Dans la partie droite, qui produit les valeurs, on distingue deux types d'expressions : les générateurs et les qualificateurs. Un générateur est une expression qui produit des valeurs pour une variable. Elle a la forme *var* `<-` [*exp**] où chaque expression *exp* doit donner une valeur de la liste à partir de laquelle on générera des affectations de la variable v. Toutes ces valeurs seront candidates pour apparaître dans la liste résultante. Les qualificateurs sont des expressions booléennes

sur les variables générées qui vont filtrer celles qui vont apparaître dans le résultat : les valeurs pour lesquelles *tous* les qualificateurs sont vrais seront insérées dans le résultat, les autres seront rejetées.

Notons finalement que la partie gauche est une expression dans laquelle on ne peut utiliser que les variables qui ont un générateur dans la partie droite. Mais bien sûr, cette expression peut aussi comporter des constantes. Par exemple,

```
[ x * y + 1 | x <- [1,2,3,4]; x mod 2 = 0; y <- [5,6,7,8]; y mod 2 = 1; x + y > 7 ]
```

aura pour résultat `[15,21,29]`.

Algorithmiquement, on pourrait traduire la ZF-expression précédente en :

```
resultat := []
pour x ∈ [1, 2, 3, 4] faire
   si x mod 2 = 0 alors
      pour y ∈ [5, 6, 7, 8] faire
         si y mod 2 = 1 alors
            si x + y > 7 alors
               ajouter x * y + 1 à la fin de resultat
retourner resultat
```

En prenant la syntaxe abstraite suivante :

n	$\in$	*Numéraux*	zf	$::=$	$\texttt{zf}\ e\ p$
v	$\in$	*Variables*	e	$::=$	$n \mid v \mid e + e \mid e - e \mid e * e \mid e/e \mid e\ \texttt{mod}\ e$
e	$\in$	*Expressions*	p	$::=$	$g \mid q \mid g\ ;\ p \mid q\ ;\ p$
p	$\in$	*Productions*	g	$::=$	$v\ \texttt{<-}\ [\ e^*\ ]$
g	$\in$	*Générateurs*	q	$::=$	$e < e \mid e = e \mid q\ \texttt{and}\ q \mid q\ \texttt{or}\ q$
q	$\in$	*Qualificateurs*			

Question 1. Définir les éléments de l'état de l'évaluation des ZF-expressions et la signature des relations $\rightarrow$ qu'il faut pour donner la sémantique de celles-ci. Précisez bien la nature (les types) de chaque élément. (6 points)

Solution :

Pour traiter les expressions, il faut connaître les liaisons des variables, ce qui nous amène à utiliser la relation habituelle pour les expressions arithmétiques :

$$\rho \vdash e \rightarrow_e \mathbb{Z}$$

Pour les qualificateurs, c'est la même chose, d'où :

$$\rho \vdash q \rightarrow_q T$$

Pour les ZF-expressions globalement, il s'agit de produire une liste d'entiers. On a donc tout simplement :

$$zf \ \rightarrow_{zf} \mathbb{Z}^*$$

Pour la partie droite de la ZF-expressions, c'est-à-dire p dans la syntaxe abstraite, il s'agit clairement de produire une liste. Par contre, il faut accumuler les liaisons de variables, donc il faudra un environnement dans le contexte. D'autre part, l'expression de gauche dans la ZF-expression ne peut être évaluée qu'à la fin, quand la partie droite a attribué les valeurs aux variables. On choisit de « passer » cette expression également dans le contexte. Ceci nous donne donc :

$$e, \rho \vdash p \ \rightarrow_p \ \mathbb{Z}^*$$

Question 2. Définir la sémantique opérationnelle structurelle pour ces expressions.

(14 points)

Indication : lorsqu'on doit évaluer une séquence g ; p, on peut d'abord evaluer p en liant la variable du générateur à sa première valeur, ce qui va donner une première liste de résultats, puis réévaluer g ; p mais en enlevant la première valeur du générateur, ce qui va donner une seonde liste de résultst. Le résultat combiné sera alors la concaténation des deux listes.

Solution :

$$\rho \vdash n \ \rightarrow_e \ n$$

$$\rho \vdash v \ \rightarrow_e \ \rho(v)$$

$$\frac{\rho \vdash e_1 \ \rightarrow_e \ n_1 \qquad \rho \vdash e_2 \ \rightarrow_e \ n_2}{\rho \vdash e_1 + e_2 \ \rightarrow_e \ n_1 + n_2}$$

$$\frac{\rho \vdash e_1 \ \rightarrow_e \ n_1 \qquad \rho \vdash e_2 \ \rightarrow_e \ n_2}{\rho \vdash e_1 - e_2 \ \rightarrow_e \ n_1 - n_2}$$

$$\frac{\rho \vdash e_1 \ \rightarrow_e \ n_1 \qquad \rho \vdash e_2 \ \rightarrow_e \ n_2}{\rho \vdash e_1 * e_2 \ \rightarrow_e \ n_1 \times n_2}$$

$$\frac{\rho \vdash e_1 \ \rightarrow_e \ n_1 \qquad \rho \vdash e_2 \ \rightarrow_e \ n_2}{\rho \vdash e_1 / e_2 \ \rightarrow_e \ n_1 \div n_2}$$

$$\frac{\rho \vdash e_1 \ \rightarrow_e \ n_1 \qquad \rho \vdash e_2 \ \rightarrow_e \ n_2}{\rho \vdash e_1 \ \texttt{mod} \ e_2 \ \rightarrow_e \ n_1 - ((n_1 \div n_2) \times n_2)}$$

$$\frac{\rho \vdash e_1 \ \rightarrow_e \ n_1 \qquad \rho \vdash e_2 \ \rightarrow_e \ n_2}{\rho \vdash e_1 < e_2 \ \rightarrow_q \ n_1 < n_2}$$

$$\frac{\rho \vdash e_1 \ \rightarrow_e \ n_1 \qquad \rho \vdash e_2 \ \rightarrow_e \ n_2}{\rho \vdash e_1 = e_2 \ \rightarrow_q \ n_1 = n_2}$$

$$\frac{\rho \vdash q_1 \ \rightarrow_q \ b_1 \qquad \rho \vdash q_2 \ \rightarrow_q \ b_2}{\rho \vdash q_1 \ \texttt{and} \ q_2 \ \rightarrow_q \ b_1 \wedge b_2}$$

$$\frac{\rho \vdash q_1 \ \rightarrow_q \ b_1 \qquad \rho \vdash q_2 \ \rightarrow_q \ b_2}{\rho \vdash q_1 \ \texttt{or} \ q_2 \ \rightarrow_q \ b_1 \vee b_2}$$

$$\frac{e, \varnothing \vdash p \ \rightarrow_p \ l}{\texttt{zf}\ e\ p \ \rightarrow_{zf} \ l}$$

$$\frac{\rho \vdash \uparrow e^* \ \rightarrow_e \ n_1 \qquad \rho[n_1/v] \vdash \ e \ \rightarrow_e \ n \qquad e, \rho \vdash v \texttt{ <- } [\ \downarrow e^*\ ] \ \rightarrow_p \ l}{e, \rho \vdash v \texttt{ <- } [\ e^*\ ] \ \rightarrow_p \ n \S l} \ \#e^*>0$$

$$e, \rho \vdash \ v \texttt{ <- } [\ e^*\ ] \ \rightarrow_p \ [] \quad \#e^*=0$$

$$\frac{\rho \vdash \ q \ \rightarrow_q \ true \qquad \rho \vdash \ e \ \rightarrow_e \ n}{e, \rho \vdash \ q \ \rightarrow_p \ n \S []}$$

$$\frac{\rho \vdash \ q \ \rightarrow_q \ false}{e, \rho \vdash \ q \ \rightarrow_p \ []}$$

$$\frac{\rho \vdash \uparrow e^* \ \rightarrow_e \ n_1 \qquad e, \rho[n_1/v] \vdash \ p \ \rightarrow_p \ l_1 \qquad e, \rho \vdash v \texttt{ <- } [\ \downarrow e^*\ ] \ \rightarrow_p \ l_2}{e, \rho \vdash v \texttt{ <- } [\ e^*\ ]\ ; p \ \rightarrow_p \ l_1 + l_2} \ \#e^*>0$$

$$e, \rho \vdash \ v \texttt{ <- } [\ e^*\ ]\ ; p \ \rightarrow_p \ [] \quad \#e^*=0$$

$$\frac{\rho \vdash \ q \ \rightarrow_q \ true \qquad e, \rho \vdash \ p \ \rightarrow_p \ l}{e, \rho \vdash \ q\ ; p \ \rightarrow \ l}$$

$$\frac{\rho \vdash \ q \ \rightarrow_q \ false}{e, \rho \vdash \ q\ ; p \ \rightarrow_p \ []}$$

Question 3. Écrire un programme Prolog implantant la sémantique opérationnelle structurelle donnée à la question précédente. *Attention, il ne s'agit pas d'écrire n'importe quel programme Prolog qui fonctionne, mais bien celui qui correspond (ou correspondrait) à des règles en sémantique opérationnelle structurelle.* (10 points)

Solution :

```
evalExp(N, _, N) :- integer(N).
evalExp(V, Rho, N) :- atom(V), member((V, N), Rho).
evalExp(E1+E2, Rho, N) :-
  evalExp(E1, Rho, N1),
  evalExp(E2, Rho, N2),
  N is N1 + N2.
evalExp(E1-E2, Rho, N) :-
  evalExp(E1, Rho, N1),
  evalExp(E2, Rho, N2),
  N is N1 - N2.
evalExp(E1*E2, Rho, N) :-
  evalExp(E1, Rho, N1),
  evalExp(E2, Rho, N2),
  N is N1 * N2.
evalExp(E1 / E2, Rho, N) :-
  evalExp(E1, Rho, N1),
  evalExp(E2, Rho, N2),
```

```
  N is N1 // N2.
evalExp(mod(E1, E2), Rho, N) :-
  evalExp(E1, Rho, N1),
  evalExp(E2, Rho, N2),
  mod(N1, N2, N).

evalQualifier(E1 < E2, Rho, B) :-
  evalExp(E1, Rho, N1),
  evalExp(E2, Rho, N2),
  (N1 @< N2 -> B = true ; B = false).
evalQualifier(E1 = E2, Rho, B) :-
  evalExp(E1, Rho, N1),
  evalExp(E2, Rho, N2),
  (N1 = N2 -> B = true ; B = false).
evalQualifier(and(E1, E2), Rho, B) :-
  evalQualifier(E1, Rho, B1),
  evalQualifier(E2, Rho, B2),
  (B1 = true, B2 = true -> B = true ; B = false).
evalQualifier(or(E1, E2), Rho, B) :-
  evalQualifier(E1, Rho, B1),
  evalQualifier(E2, Rho, B2),
  (B1 = true ; B2 = true -> B = true ; B = false).

evalZF(zf(E, P), L) :-
  evalP(P, E, [], L).

evalP(gen(V, [E1|Es]), E, Rho, [N|L]) :-
  evalExp(E1, Rho, N1),
  evalExp(E, [(V, N1)|Rho], N),
  evalP(gen(V, Es), E, Rho, L).
evalP(gen(_, []), _, _, []).
evalP(Q, E, Rho, [N]) :-
  evalQualifier(Q, Rho, true),
  evalExp(E, Rho, N).
evalP(Q, _, Rho, []) :-
  evalQualifier(Q, Rho, false).
evalP(seq(gen(V, [E1|Es]), P), E, Rho, L) :-
  evalExp(E1, Rho, N1),
  evalP(P, E, [(V, N1)|Rho], L1),
  evalP(seq(gen(V, Es), P), E, Rho, L2),
  append(L1, L2, L).
evalP(seq(gen(_, []), _), _, _, []).
evalP(seq(Q, P), E, Rho, L) :-
  evalQualifier(Q, Rho, true),
  evalP(P, E, Rho, L).
evalP(seq(Q, _), _, Rho, []) :-
  evalQualifier(Q, Rho, false).
```

Fin du contrôle.