Compilation - TD6

Grammaires à attributs

1 Présentation du langage Alpha

ALPHA est un langage fonctionnel fortement typé. Sa caractéristique essentielle est que les objets de base qu'il manipule sont des *polyèdres convexes de valeurs*. Un polyèdre convexe est une généralisation utile de la notion de tableau multidimensionnel. On le représentera par une intersection de demi-espaces, chaque demi-espace étant défini par une inéquation affine. Voici une syntaxe typique pour un polyèdre :

```
{i,j,k | 1<=i; i<=10; 1<=j; j<=10; 1<=k; 2k+j<=i}
```

L'intérêt de cette représentation est son caractère affine, qui permet l'analyse et la transformation des domaines (donc des programmes) par des outils mathématiques bien définis (algèbre linéaire, programmation linéaire en nombre entiers). Mais ce n'est pas l'objet de ce TD.

On manipulera en fait des *ensembles finis de polyèdres convexes*, que l'on appelera désormais des *domaines*. Techniquement, l'ensemble des domaines forme un treillis pour l'inclusion, et est stable par intersection, par union, par préimage par une fonction affine et (presque) par image par une fonction affine.

Voici un exemple de programme ALPHA qui décrit un produit matrice-vecteur.

```
system matvect
                (M : {i,j | 1<=i; i<=42; 1<=j; j<=42} of real;
                 V : {j | 1<=j; j<=42} of real)
        returns (R : {i | 1<=i; i<=42} of real);
var
  C : {i,j | 1<=i; i<=42; 0<=j; j<=42} of real;
let
  C = case
        {i,j| j=0} : 0.(i,j->);
        {i,j| j>0} : C.(i,j->i,j-1) + M * V.(i,j->j);
      esac;
  R = C.(i->i,42);
tel;
```

Le case réalise l'union des domaines de ses sous-expressions. Les deux points signifient la *restriction* de l'expression en partie droite au domaine en partie gauche. Le point applique la *fonction affine* en partie droite à son expression en partie gauche (il est le plus prioritaire des opérateurs). Les opérateurs arithmétiques ont une sémantique point-à-point (en particulier le `*` est une multiplication point-à-point).

Question 1-1 Dessinez l'arbre syntaxique abstrait (AST) de ce programme.

Question 1-2 Donnez la grammaire d'une fonction affine, la grammaire d'un domaine, la grammaire d'une expression ALPHA que l'on supposera toujours sous la forme *case-restriction-dépendance*, enfin la grammaire d'un programme.

2 Domaine d'une expression

Voici les règles définissant le calcul du domaine d'une expression (qui proviennent de la sémantique du langage).

Constantes	$\boxed{Dom(c) = \mathbf{Z}^0}$
Variables	$\boxed{Dom(V) \text{ est déclaré dans l'en-tête}}$
Opérateurs unaires	$\boxed{Dom(-e) = Dom(e)}$
Opérateurs binaires	$\boxed{Dom(e_1 \oplus e_2) = Dom(e_1) \cap Dom(e_2)}$
Dépendance affine	$\boxed{Dom(e.f) = f^{-1}(Dom(e))}$
Opérateur case	$\boxed{Dom(\textbf{case}\ e_1; ..; e_n; \textbf{esac}) = \bigcup_{i=1}^{n} Dom(e_i)}$

Question 2-1 Formalisez le calcul de l'attribut "domaine" dans votre grammaire. Quel type de grammaire à attributs obtenez-vous ?

Question 2-2 Définissez précisément les cas d'erreur avec les messages correspondants : incompatibilité de dimension, surdéfinition de certaines valeurs, domaines vides... Essayez d'éviter les erreurs cascadées.

3 Forme compositionnelle et forme tableau

Pour vendre ALPHA au grand public, il a été mis au point une forme tableau qui est plus lisible. Voici le même programme sous forme tableau, jugez vous-même.

```
system matvect
               (M : {i,j | 1<=i; i<=42; 1<=j; j<=42} of real;
                V : {j | 1<=j; j<=42} of real)
       returns (R : {i | 1<=i; i<=42} of real);
var
  C : {i,j | 1<=i<=42; 0<=j<=42} of real;
let
  C[i,j] =
      case
        {| j=0} : 0[];
        {| j>0} : C[i,j-1] + M[i,j] * V[j];
      esac;
  R[i] = C[i,42];
tel;
```

Question 3-1
Quels sont les attributs à ajouter à la grammaire pour que l'analyseur syntaxique gère cette forme tableau ? Quel type de grammaire à attributs est-ce à présent ?

4 Contexte d'une expression

Le *domaine de contexte*, ou plus simplement *contexte* d'une expression, est une notion qui a une définition intuitive simple : c'est le domaine sur lequel cette expression va servir à quelque chose. Cette notion dépend donc évidemment de l'environnement dans lequel on considère l'expression, d'où son nom.

Par exemple, si l'expression considérée est la partie droite d'une équation, son contexte est le domaine de la variable en partie gauche : en effet si l'expression possède un domaine plus grand que celui de cette variable en partie gauche, elle y est inutile.

A quoi sert cette notion de contexte ? Elle est utilisée principalement par toute transformation introduisant une *variable auxiliaire* pour remplacer une expression : une telle transformation doit déterminer automatiquement le domaine de cette nouvelle variable. Ce domaine est précisément le domaine de contexte de l'expression à remplacer : ainsi on ajoute le strict minimum de points de calculs au programme. L'introduction de variables auxiliaires est une transformation de programme fréquente.

Voici la liste des règles qui définissent le contexte $Cxt(e)$ d'une expression ALPHA e valide.

Racine Si e est partie droite d'une équation $V = e$
alors $\boxed{Cxt(e) = Dom(V)}$.
Sinon (expression flottante)
alors $\boxed{Cxt(e) = \mathbf{Z}^n}$ où n est la dimension de l'expression.

Restriction Si e apparaît dans le contexte $D : e$
alors $\boxed{Cxt(e) = D \cap Cxt(D : e)}$

Fonction affine Si e apparaît dans le contexte $e.f$
alors $\boxed{Cxt(e) = domImage(Cxt(e.f), f)}$

Opérateurs unaires Si e apparaît dans le contexte $-e$
alors $\boxed{Cxt(e) = Cxt(-e)}$

Opérateurs binaires Si e apparaît dans le contexte $e' + e$
alors $\boxed{Cxt(e) = Dom(e') \cap Cxt(e' + e)}$

Question 4-1 Modifiez votre grammaire pour y ajouter l'attribut Contexte.

5 Moralité

Avant l'invention des grammaires à attributs, l'analyseur syntaxique donnait un arbre abstrait non étiqueté, et on programmait des parcours d'arbres ad-hocs pour l'étiqueter avec les attributs. Le parcours d'arbre était un programme dont la structure ressemblait d'ailleurs beaucoup à une grammaire à attributs.

Question 5-1 Le parcours d'arbre ad-hoc est même en général plus simple, pourquoi ?

Question 5-2 Quel gros avantage ont tout de même les grammaires à attributs pour simplifier le boulot du programmeur ?

Question 5-3 Discutez les avantages et les inconvénients de ces deux approches pour les trois attributs considérés ici.