

**UTBM**

**Automne 2004**

**IA41**

**INTELLIGENCE ARTIFICIELLE**

*représentation des connaissances et résolution de problèmes*

**Responsable: Jean César**

**Intervenants en TD: Olivier Grunder, Yassine Ruichek** en TP : **Mohamed Hariti**

**Travaux Dirigés n° 1** Semaine du 20 septembre 2004 **(corrigé)**

**SUJET: ADRESSE DANS UNE LISTE ARBORESCENTE**

**Extraction et modification d'un terme**

1) avec les primitives: tête et reste, spécifier et définir formellement la fonction: nième-reste. En déduire la fonction nième-terme, en donnant à la tête de liste l'indice 0, comme en C.

**nième-reste:**

Spécification:

- on pose que le 0° reste désigne la liste tout entière,
- pour n non nul, le nième-reste est le résultat de n applications de la fonction reste,

Optimisation: pour n supérieur ou égal à la longueur de la liste, on rend ( ), c-à-d FAUX.

Mais pratiquement, la longueur de la liste n'est pas calculée, pour éviter un nouveau parcours

- on rend donc la liste vide dès qu'elle apparaît.

Profil:  $N \times \text{List} \rightarrow \text{List}$

Jeu d'essais: (avec simulation du traitement récursif)

nième-reste 0, (a b c) = (a b c)

nième-reste 1, (a b c) = nième-reste 0, (b c) = (b c)

nième-reste 2, (a b c) = nième-reste 1, (b c) = nième-reste 0, (c) = (c)

nième-reste 5, (a b c) = nième-reste 4, (b c) = nième-reste 3, (c) = nième-reste 2, ( ) = ( )

Définition formelle récursive:

Pour tout (n, L) appartenant à  $N \times \text{List}$ :

{ n = 0 ou L = ( ) } => nième-reste n, L = L ; cas d'arrêt

{ n ≠ 0 et L ≠ ( ) } => nième-reste n, L = nième-reste (n - 1), reste L ; cas récursif

**nième-terme:**

Spécification:

Pratiquement, le nième-terme est la tête du nième-reste d'une liste.

Ainsi, on vérifie que le 0° terme de la liste est bien sa tête,

Pour n supérieur ou égal à la longueur de la liste, on rend bien ( ), c-à-d FAUX, comme conséquence de l'axiome: tête ( ) = ( ).

Une liste vide résultat peut provenir du fait que le nième-terme n'existe pas, ou qu'il est lui-même une liste vide, c'est-à-dire une valeur logique FAUX antérieure et qui est transmise:

nième-terme 5, (a b c) = ( )

nième-terme 1, (a ( ) c) = ( )

Profil:  $N \times \text{List} \rightarrow \text{Expr}$  (le résultat pouvant être un atome ou une liste)

Jeu d'essais:

nième-terme 0,  $(a\ b\ c) = \text{tête}(a\ b\ c) = a$

nième-terme 1,  $(a\ b\ c) = \text{tête}(b\ c) = b$

nième-terme 2,  $(a\ b\ c) = \text{tête}(c) = c$

nième-terme 5,  $(a\ b\ c) = \text{tête}(\ ) = (\ )$

Définition formelle:

Pour tout  $(n, L)$  appartenant à  $N \times \text{List}$ :

nième-terme  $n$ ,  $L = \text{tête}$  nième-reste  $n$ ,  $L$

2) avec les primitives: tête et reste et ajout, spécifier et définir formellement la fonction: chnième qui change le nième terme, S'IL EXISTE, d'une liste (au 1<sup>o</sup> niveau).

ajout est la primitive qui ajoute un terme en tête d'une liste:

Profil:  $\text{Expr} \times \text{List} \rightarrow \text{List}$

Exemples: ajout  $a$ ,  $(b\ c) = (a\ b\ c)$

ajout  $a$ ,  $(\ ) = (a)$  ; équivaut à une mise en liste

**chnième:**

Spécification:

- on pose qu'à l'indice 0, c'est la tête de liste qui est remplacée.
- pour  $n$  non nul, c'est le nième-terme, s'il existe, qui est remplacé.
- mais si  $n$  est supérieur ou égal à la longueur de la liste, la liste est inchangée.

Optimisation: comme pour nième-reste, on ne calcule pas la longueur, mais:

- pour une liste vide (terme d'indice  $n$  inexistant), on rend  $(\ )$ .

Profil:  $N \times \text{Expr} \times \text{List} \rightarrow \text{List}$

Jeu d'essais: (avec simulation du traitement récursif)

chnième 0, \*,  $(a\ b) = \text{ajout } *$ , reste  $(a\ b) = \text{ajout } *$ ,  $(b) = (*\ b)$

chnième 0, \*,  $(\ ) = (\ )$

chnième 2, \*,  $(a\ b\ c\ d\ e) = \text{ajout } a$ , chnième 1, \*,  $(b\ c\ d\ e) = \text{ajout } a$ , ajout  $b$ , chnième 0, \*,  $(c\ d\ e) = \text{ajout } a$ , ajout  $b$ , ajout \*, reste  $(c\ d) = \text{ajout } a$ , ajout  $b$ , ajout \*,  $(d\ e) = (a\ b\ * \ d\ e)$

NB: Le début de la liste est reconstruit terme à terme, la fin est réutilisée d'un seul bloc.

chnième 2, \*,  $(a\ b) = \text{ajout } a$ , ajout  $b$ , chnième 0, \*,  $(\ ) = \text{ajout } a$ , ajout  $b$ ,  $(\ ) = (a\ b)$

chnième 7, \*,  $(a\ b\ c\ d) = \text{ajout } a$ , ajout  $b$ , ajout  $c$ , ajout  $d$ , chnième 3, \*,  $(\ )$

$= \text{ajout } a$ , ajout  $b$ , ajout  $c$ , ajout  $d$ ,  $(\ ) = (a\ b\ c\ d)$

NB: Ces deux derniers exemples illustrent le fait que le test sur  $(\ )$  doit précéder le test sur 0.

Définition formelle récursive:

Pour tout  $(n, r, L)$  appartenant à  $N \times \text{Expr} \times \text{List}$ :

$L = (\ ) \Rightarrow$  chnième  $n$ ,  $r$ ,  $L = (\ )$

$\{ L \# (\ ) \text{ et } n = 0 \} \Rightarrow$  chnième  $n$ ,  $r$ ,  $L = \text{ajout } r$ , reste  $L$

$\{ L \# (\ ) \text{ et } n \neq 0 \} \Rightarrow$  chnième  $n$ ,  $r$ ,  $L = \text{ajout tête } L$ , chnième  $(n - 1)$ , reste  $L$

Notez comment la partition est obtenue au moyen de deux dichotomies successives.

3) Considérons l'adresse arborescente d'un terme dans une liste:

Ainsi l'adresse de y dans (a b (x y z) d) est (2 1).

La longueur de l'adresse arborescente (2 1) est égale à la profondeur du terme cherché.

Ce terme est ici accessible au moyen de deux appels de la fonction nième-terme:

nième-terme 1, nième-terme 2, (a b (x y z) d) = nième-terme 1, (x y z) = y

Pour généraliser ce processus, spécifier et définir formellement la fonction aième-terme qui extrait un terme d'une liste d'après son adresse arborescente:

**aième-terme:**

Spécification:

- pour une adresse arborescente vide, on rend l'expression donnée,
- pour une adresse arborescente non vide:
  - \_ compatible avec la donnée, on rend le terme pointé par cette adresse,
  - \_ dont au moins un indice est trop grand, on rend la liste vide, c-à-d FAUX,
  - \_ portant sur un atome, on rend la liste vide, c-à-d FAUX.

Profil étendu pour tenir compte de la spécification: ListN x Expr → Expr

Jeu d'essais: (avec simulation du traitement récursif)

aième-terme ( ), y = y

aième-terme (2 1), (a b (x y z) d) = aième-terme (1), nième-terme 2, (a b (x y z) d)  
= aième-terme ( ), nième-terme 1, nième-terme 2, (a b (x y z) d) = y

aième-terme (4 1), (a b (x y z) d) = aième-terme (1), nième-terme 4, (a b (x y z) d)  
= aième-terme (1), ( ) = ( )

aième-terme (2 4), (a b (x y z) d) = aième-terme (4), (x y z)  
= aième-terme ( ), nième-terme 4, (x y z) = aième-terme ( ), ( ) = ( )

aième-terme (2 1), (a b c d) = aième-terme (1), nième-terme 2, (a b c d)  
= aième-terme (1), c = ( )

Définition formelle:

Pour tout (Ln, e) appartenant à ListN x Expr

Ln = ( ) => aième-terme(Ln, e) = e

{ Ln # ( ) et e appartient à Atom } => aième-terme(Ln, e) = ( )

{ Ln # ( ) et e n'appartient pas à Atom } => aième-terme(Ln, e)  
= aième-terme reste Ln, nième-terme tête Ln, e

Le cas d'un indice trop grand est pris en charge par la récursivité jusqu'à ce que nième-terme rende ( ), puis l'une des deux conditions d'arrêt est exécutée. Rappel: ( ) est un atome.

4) Spécifier et définir formellement la fonction chaîme qui change dans une liste un terme EXISTANT pointé par son adresse arborescente.

### chaîme:

#### Spécification:

**NB :** On réutilise la même partition des cas que pour aième-terme, seul le résultat diffère :

- pour une adresse arborescente vide, on rend le terme de remplacement,
- pour une adresse arborescente non vide:
  - \_ compatible avec la donnée, on met à cette adresse le terme de remplacement, à la place de celui qui s'y trouve et est perdu,
  - \_ dont au moins un indice est trop grand, on rend l'expression inchangée,
  - \_ portant sur un atome, on rend l'expression inchangée.

Profil étendu pour tenir compte de la spécification: ListN x Expr x Expr → Expr

Jeu d'essais: (avec simulation du traitement récursif)

chaîme ( ), \*, y = \*

chaîme (2 1), \*, (a b (x y z) d) ...

**NB :** Prenez du recul: considérez (x \* z) comme terme de remplacement de (x y z),

Ou encore: "To write a Program, look at the Product, not at the Process" :

= chnième 2, (x \* z), (a b (x y z) d)

= chnième 2, { chaîme (1), \*, (x y z) }, (a b (x y z) d)

= chnième 2, { chnième 1, [ chaîme ( ), \*, y ], (x y z) }, (a b (x y z) d)

= chnième 2, { chnième 1, \*, (x y z) }, (a b (x y z) d)

= (a b (x \* z) d)

chaîme (4 1), \*, (a b (x y z) d) = chnième 4, { chaîme (1), \*, ( ) }, (a b (x y z) d)

= chnième 4, ( ), (a b (x y z) d) = (a b (x y z) d)

chaîme (2 3), \*, (a b (x y z) d) = chnième 2, { chaîme (3), \*, (x y z) }, (a b (x y z) d)

= chnième 2, { chnième 3, [ chaîme ( ), \*, ( ) ], (x y z) }, (a b (x y z) d)

= chnième 2, { chnième 3, \*, (x y z) }, (a b (x y z) d)

= chnième 2, (x y z), (a b (x y z) d) = (a b (x y z) d)

chaîme (2 1), \*, (a b c d) = chnième 2, { chaîme (1), \*, c }, (a b c d)

= chnième 2, c, (a b c d) = (a b c d)

#### Définition formelle:

Pour tout (Ln, r, e) appartenant à ListN x Expr x Expr

Ln = ( ) => chaîme Ln, r, e = r

{ Ln # ( ) et e appartient à Atom } => chaîme Ln, r, e = e

{ Ln # ( ) et e n'appartient pas à Atom } => chaîme Ln, r, e =

chnième tête Ln, { chaîme reste Ln, r, nth tête Ln, e }, e