

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° 3

Semaine du 4 octobre 2004

(corrigé)

SUJET: STRUCTURES DE DONNEES AVANCEES: Listes de couples

1) Les tours de Hanoi:

- au 1<sup>o</sup> mouvement, le petit disque va-t-il sur la tige du milieu ou de l'arrivée?  
Réponse: au milieu si n est pair, à l'arrivée si n est impair.
- quel est le seul disque que l'on ne déplace qu'une fois? Réponse: le plus grand
- combien de coups dure une partie jouée avec n disques? Réponse:  $(2^n - 1)$  coups

Spécifiez et définissez formellement la fonction **hanoi**, dont l'effet de bord consiste à afficher des messages du type "Je déplace le disque ... de ... vers ..."

profil:  $N \times \text{Atom}^3$ , où  $\text{Atom}^3$  sert à conserver les noms des 3 tiges

spécification: il n'y a rien à faire pour  $n = 0$ , sauf rendre t (true)!

Dans tous les autres cas, la fonction hanoi rendra t après les affichages.

jeu d'essais (avec simulation de la récursivité):

hanoi 0 a b c = t

hanoi 3 a b c

deplacez le disque 1 de a vers c

deplacez le disque 2 de a vers b

deplacez le disque 1 de c vers b; ces 3 lignes représentent hanoi 2 a c b

deplacez le disque 3 de a vers c; déplacement unique du plus grand disque

deplacez le disque 1 de b vers a

deplacez le disque 2 de b vers c

deplacez le disque 1 de a vers c; ces 3 lignes représentent hanoi 2 b a c

= t

définition formelle:

Pour tout  $(n,x,y,z)$  appartenant à  $N \times \text{Atom}^3$ :

$n = 0 \Rightarrow \text{hanoi } n,x,y,z = t$

$n \neq 0 \Rightarrow \text{hanoi } n,x,y,z = \text{séquence: hanoi } n-1,x,z,y$

effet de bord: afficher "déplacer n de x vers z"

hanoi  $n-1,y,x,z$

2) Les listes de couples:

2-1) Spécifiez et définissez formellement la fonction **accès**, qui rend la valeur associée à une clé dans une liste de couples.

Spécification: - à partir de la liste de couples et de la clé, accès rend la valeur associée,

- si la clé ne figure pas dans la liste de couples, en particulier quand cette dernière est vide, accès rend la liste vide. Notons que ce résultat est le même que si la valeur stockée était la liste vide. Dans les deux cas, cela revient à transmettre la valeur logique Faux.

Profil: Expr x A-list  $\rightarrow$  A-list

Jeu d'essais:

accès claudé, { (jean 3076) (claudé 3028) (marina 3088) (alain 3204) } = 3028

accès philippe, { (jean 3076) (claudé 3028) (marina 3088) (alain 3204) } = ( )

### Définition formelle:

La recherche s'arrête dès que la clé -supposée unique- est trouvée.

L'asymétrie des fonctions tête et reste explique comment sont extraites la clé et la valeur:

Pour tout (clé, listecouples) appartenant à Expr x A-list:

listecouples = ( ) => accès clé, listecouples = ( )  
{ listecouples # ( ) et tête tête listecouples = clé } =>  
accès clé, listecouples = tête reste tête listecouples  
{ listecouples # ( ) et tête tête listecouples # clé } =>  
accès clé, listecouples = accès clé, reste listecouples

2-2) Spécifiez et définissez formellement la fonction **accèsinv** (accès inverse), qui rend la liste des clés associées à une même valeur dans une liste de couples.

Spécification: Accèsinv rend la liste (éventuellement vide) des clés associées à une même valeur dans une liste de couples.

Profil: Expr x A-list → Expr

Jeu d'essais:

accèsinv 3076, { (jean 3076) (claire 3028) (marina 3088) (yvan 3076) } = (jean yvan)

accèsinv 3024, { (jean 3076) (claire 3028) (marina 3088) (yvan 3076) } = ( )

Définition formelle: La recherche continue nécessairement jusqu'à la fin de la liste.

Pour tout (val, listecouples) appartenant à Expr x A-list:

listecouples = ( ) => accèsinv val, listecouples = ( )  
{ listecouples # ( ) et tête reste tête listecouples = val } =>  
accèsinv val, listecouples =  
ajout tête tête listecouples, accèsinv val, reste listecouples  
{ listecouples # ( ) et tête reste tête listecouples # val } =>  
accèsinv val, listecouples = accèsinv val, reste listecouples

2-3) Spécifiez et définissez formellement la fonction **miseàjour**, qui rend la liste de couples dans laquelle une clé est associée à une nouvelle valeur.

Spécification: - à partir de la liste de couples, de la clé et de la valeur, miseàjour rend la liste de couples dans laquelle l'ancienne valeur associée à la clé est remplacée par la nouvelle,

- si la clé n'existait pas, le nouveau couple (clé valeur) est ajouté à la liste de couples, à l'endroit le plus opportun... donc une fois la liste parcourue pour vérifier l'absence de la clé, c'est-à-dire à la fin: c'est l'exception qui confirme la règle des ajouts en tête.

Profil: Expr x Expr x A-list → A-list

Jeu d'essais:

miseàjour claire, 3030, { (jean 3076) (claire 3028) (marina 3088) (alain 3204) } =

{ (jean 3076) (claire 3030) (marina 3088) (alain 3204) }

miseàjour philippe, 3181, { (jean 3076) (claire 3028) (marina 3088) (alain 3204) } =

{ (jean 3076) (claire 3028) (marina 3088) (alain 3204) (philippe 3181) }

Définition formelle: Comme pour chnième, on doit reconstruire la partie de la liste précédant la modification, et réutiliser telle quelle celle qui suit la modification.

Pour tout (clé, val, listecouples) appartenant à Expr x Expr x A-list :

listecouples = ( ) => miseàjour clé, val, listecouples =  
mise-en-liste mise-en-liste clé val N.B.: rend ainsi une liste de couples  
{ listecouples # ( ) et tête tête listecouples = clé } =>  
miseàjour clé, val, listecouples = ajout mise-en-liste clé val reste listecouples  
{ listecouples # ( ) et tête tête listecouples # clé } =>  
miseàjour clé, val, listecouples = ajout tête listecouples  
miseàjour clé, val, reste listecouples

2-4) Spécifiez et définissez formellement la fonction **nettoie**, qui ne laisse que l'occurrence la plus récente de chaque clé, EN UN SEUL PARCOURS de la A-liste.

Spécification: La fonction nettoie ne laisse que l'occurrence la plus récente –donc la première- de chaque clé, et rend la liste de couples “propre”.

Profil: Un paramètre supplémentaire contient la liste des clés déjà rencontrées. Il est initialisé avec la liste vide. D'où le profil étendu à: A-list x List  $\rightarrow$  A-list

Jeu d'essais:

nettoie ( (jean 3076) (charles 3035) (noël 3080) (charles 3030) (marina 3088) (jean 3065) ) =  
( (jean 3076) (charles 3035) (noël 3080) (marina 3088) )

Définition formelle:

Pour tout (al, lclés) appartenant à A-list x List :

al = ( )  $\Rightarrow$  nettoie al, lclés = ( )

{ al # ( ) et app tête tête al, lclés }  $\Rightarrow$

nettoie al, lclés = nettoie reste al, lclés

; le couple réutilisant une clé déjà rencontrée est perdu

{ al # ( ) et non app tête tête al, lclés }  $\Rightarrow$  nettoie al, lclés =

nettoie al, lclés = ajout tête al, nettoie reste al, ajout tête tête al, lclés

; le couple utilisant le premier une clé est conservé

; sa clé est ajoutée à la liste des clés déjà rencontrée

2-5) Spécifiez et définissez formellement la fonction **app**, vérifiant l'appartenance d'une expression au premier niveau d'une liste, en rendant pour valeur logique Vrai la partie de la liste qui commence à la première occurrence de l'expression cherchée.

Spécification: app est un prédicat qui rend Vrai si une expression appartient à une liste, au 1<sup>o</sup> niveau, et sinon Faux -la liste vide-. Mais app est un “prédicat étendu” qui rend pour Vrai la fin de la liste, à partir de la première occurrence de l'expression donnée.

Profil: S-Expr x List  $\rightarrow$  List

Jeu d'essais:

app beethoven (ravel satie debussy) = nil

app (jean anne) ((pierre colette) (paul julie) (jean anne) (georges marie)) =  
((jean anne) (georges marie))

Définition formelle:

Pour tout (e, L) appartenant à S-Expr x List  $\rightarrow$  List

L = ( )  $\Rightarrow$  app e, L = ( )

{ L # ( ) et e = tête L }  $\Rightarrow$  app e, L = L

{ L # ( ) et e # tête L }  $\Rightarrow$  app e, L = app e, reste L

2-6) Spécifiez et définissez formellement la fonction **app2**, extension de la fonction **app** à tous les niveaux de profondeur.

Spécification: App2 est un prédicat qui rend Vrai si une expression appartient à une liste, A QUELQUE PROFONDEUR QUE CE SOIT, et sinon Faux -la liste vide-. Mais app2 rend pour Vrai la fin de la sous-liste à partir de la première occurrence de l'expression donnée, rencontrée dans le parcours en profondeur d'abord.

Profil: S-Expr x List  $\rightarrow$  List

Jeu d'essais:

app2 beethoven (ravel satie debussy) = nil

app2 w (a b (u v w x y) c) = (w x y)

Définition formelle:

Pour tout (e, L) appartenant à S-Expr x List  $\rightarrow$  List

$L = () \Rightarrow \text{app2 } e, L = ()$

$\{L \# () \text{ et } e = \text{tête } L\} \Rightarrow \text{app2 } e, L = L$

$\{L \# () \text{ et } e \# \text{tête } L \text{ et tête } L \text{ appartient à Atom}\} \Rightarrow \text{app2 } e, L = \text{app2 } e, \text{reste } L$

$\{L \# () \text{ et } e \# \text{tête } L \text{ et tête } L \text{ n'appartient pas à Atom}\} \Rightarrow$

$\text{app2 } e, L = \text{ ou } \text{app2 } e, \text{tête } L, \text{app2 } e, \text{reste } L$

2-7) Spécifiez et définissez formellement la fonction **reverse**, qui renverse le premier niveau d'une liste (fonction utilisée pour inverser le résultat dans **treeaddress**).

Spécification: Reverse rend dans l'ordre inverse (au 1<sup>o</sup> niveau seulement) la liste argument.

Profil: List  $\rightarrow$  List

Jeu d'essais: reverse2 (a (b c) d) = (d (b c) a)

Première définition formelle de reverse, calculant le résultat dans la pile:

Pour tout L appartenant à List:

$L = () \Rightarrow \text{reverse } L = ()$

$L \# () \Rightarrow \text{reverse } L = \text{concat reverse reste } L, \text{ mise-en-liste tête } L$

Quant à la fonction **concat** utilisée dans **reverse**:

Profil: List x List  $\rightarrow$  List

Définition Formelle de concat:

Pour tout (L1, L2) appartenant à List x List

$L1 = () \Rightarrow \text{concat } (L1, L2) = L2$

$L1 \# () \Rightarrow \text{concat } (L1, L2) = \text{ajout tête } L1, \text{concat reste } L1, L2$

Le coût de **concat** est celui de la recopie de la structure de L1.

Il est possible d'optimiser le cas où L2 est vide, mais sans incidence sur le cas général:

$L1 = () \Rightarrow \text{concat } (L1, L2) = L2$

$L1 \# () \text{ et } L2 = () \Rightarrow \text{concat } (L1, L2) = L1$

$L1 \# () \text{ et } L2 \# () \Rightarrow \text{concat } (L1, L2) = \text{ajout tête } L1, \text{concat reste } L1, L2$

Sur l'exemple (a (b c) d), la version "pile" de **reverse** exécute 6 ajouts, dont 3 sont empilés.

Dans la simulation suivante, "mise-en-liste x" est remplacé par sa définition "ajout x nil"

reverse (a (b c) d) = concat reverse ((b c) d), ajout a nil  
concat concat reverse (d) ajout (b c) nil, ajout a nil  
concat concat concat reverse nil, ajout d () ajout (b c) nil ajout a nil  
concat concat concat nil (d) ajout (b c) nil, ajout a nil  
concat concat (d) ajout (b c) nil ajout a nil  
concat concat (d) ((b c)) ajout a nil  
concat ajout d concat nil ((b c)) ajout a nil  
concat ajout d ((b c)) ajout a nil  
concat (d (b c)) ajout a nil  
concat (d (b c)) (a)  
ajout d concat ((b c)) (a)  
ajout d ajout (b c) concat nil (a)  
ajout d ajout (b c) (a)  
ajout d ((b c) a)  
(d (b c) a)

Généralisation: la version "pile" inverse une liste de n termes en 2n ajouts, dont n empilés.

Seconde définition formelle de **reverse**, calculant le résultat dans un paramètre:

Profil: List x List  $\rightarrow$  List

Pour tout (L,P) appartenant à List x List - le paramètre P est initialisé à ( ) -

L = ( )  $\Rightarrow$  reverse L,P = P

L # ( )  $\Rightarrow$  reverse L,P = reverse reste L, ajout tête L,P

Sur l'exemple (a (b c) d), la version "paramètre" de **reverse** exécute à la suite 3 ajouts:

reverse (a (b c) d) ( ) = reverse ((b c) d) ajout a ( )  
reverse ((b c) d) (a)  
reverse (d) ajout (b c) (a)  
reverse (d) ((b c) a)  
reverse ( ) ajout d ((b c) a)  
reverse ( ) (d (b c) a)  
(d (b c) a)

Généralisation: la version "paramètre" inverse une liste de n termes en n ajouts non empilés.

Conclusion: la version "paramètre" prend moins de temps et de mémoire que la version "pile".

2-8) Spécifiez et définissez formellement la fonction **reverse2**, extension de la fonction **reverse** à tous les niveaux de profondeur.

Spécification: Reverse2 rend dans l'ordre inverse la liste passée en argument, en inversant aussi toutes ses sous-listes, QUELLE QUE SOIT LEUR PROFONDEUR.

Profil: List x List  $\rightarrow$  List - bien entendu à partir de la version "paramètre" -

Jeu d'essais: reverse2 (a (b c) d) = (d (c b) a)

Définition formelle:

Pour tout (L,P) appartenant à List x List - le paramètre P est initialisé à ( ) -

L = ( )  $\Rightarrow$  reverse2 L,P = ( )

{L # ( ) et tête L appartient à Atom}  $\Rightarrow$

reverse2 L,P = reverse2 reste L, ajout tête L,P

{L # ( ) et tête L n'appartient pas à Atom}  $\Rightarrow$

reverse2 L,P = reverse2 reste L, ajout reverse2 tête L,( ) P

Sur l'exemple (a (b c) d), **reverse2** exécute 5 ajouts:

un par atome, plus un empilé pour la sous-liste:

reverse2 (a (b c) d) ( ) = reverse2 ((b c) d) ajout a ( )  
reverse2 ((b c) d) (a)  
reverse2 (d) ajout reverse2 (b c) ( ) (a)  
reverse2 (d) ajout reverse2 (c) ajout b ( ) (a)  
reverse2 (d) ajout reverse2 (c) (b) (a)  
reverse2 (d) ajout reverse2 ( ) ajout c (b) (a)  
reverse2 (d) ajout reverse2 ( ) (c b) (a)  
reverse2 (d) ajout (c b) (a)  
reverse2 (d) ((c b) a)  
reverse2 ( ) ajout d ((c b) a)  
reverse2 ( ) (d (c b) a)  
(d (c b) a)

Notes: -la fonction **reverse2** est souvent appelée **miroir**.

-une autre version sera étudiée au TD n°4 avec les fonctions d'application.

### 3) Retour aux tours de Hanoi

Une A-liste sera utilisée pour représenter la mini-base de connaissances que constitue l'état des tours, et à ce titre elle sera stockée de manière globale.

état-du-jeu ← '((a (1 2 3)) (b nil) (c nil)) ;initialisation de la a-liste globale

Redéfinissez formellement la fonction **hanoi**, en lui ajoutant l'effet de bord consistant à afficher, en plus des messages "Je déplace...", les états successifs des tours.

n = 0 => hanoi n,x,y,z = t

n # 0 => hanoi n,x,y,z =

hanoi n-1,x,z,y

afficher "déplacez n de x vers z"

état-du-jeu ← miseàjour arrivee

(ajout tete acces depart etat-du-jeu, acces arrivee etat-du-jeu)

etat-du-jeu

état-du-jeu ← miseàjour depart

(reste acces depart etat-du-jeu)

etat-du-jeu

afficher état-du-jeu

hanoi n-1,y,x,z

Voici l'affichage résultant d'une évaluation de cette version:

(hanoi 3 'a' 'b' 'c')

(deplacez le disque 1 de a vers c)

((a (2 3)) (b nil) (c (1)))

(deplacez le disque 2 de a vers b)

((a (3)) (b (2)) (c (1)))

(deplacez le disque 1 de c vers b)

((a (3)) (b (1 2)) (c nil))

(deplacez le disque 3 de a vers c)

((a nil) (b (1 2)) (c (3)))

(deplacez le disque 1 de b vers a)

((a (1)) (b (2)) (c (3)))

(deplacez le disque 2 de b vers c)

((a (1)) (b nil) (c (2 3)))

(deplacez le disque 1 de a vers c)

((a nil) (b nil) (c (1 2 3)))

t

Ne pas oublier de libérer la variable globale état-du-jeu pour laisser propre l'espace de travail!

Note: Il ne vous reste qu'à parfaire l'affichage des tours par colonnes... (voir TD suivant sur les fonctions d'applications)