Cours 2 : Algorithmique élémentaire

January 15, 2024

Séance 3 ; Cours 2 : Algorithmique élémentaire

1) Algorithmique Prolog (rappel/1ère approche)
   - Structure de données
   - Structure de contrôle / du contrôle de l'exécution (cf. implémentation)
2) Algorithmique classique
   - Structure de données
   - Structure de contrôle
   - et Implémentation (compilation vs compilation ; gestion de la mémoire)
3) Propriétés algorithmiques (classiques, [et déclaratives])
   - Correction
   - Complétude
   - Terminaison (+Complexité)
   - [Déclarativité] ([Réversibilité], [Non-Déterminisme])
4) Exemples de programme

1 Algorithmique ProLog (rappel/1ère approche)

Sur la base de : **Prog = Data + Contrôle**

Rappel des principes (JP Delahaye) :

1. Enoncer des **Faits**
2. Donner des **règles de raisonnement** se basant sur ces faits pour produire d'autres faits
3. Savoir poser des questions (**requêtes**)

Dans tous les cas, les faits doivent être vrais ("prouvés" à partir de faits plus élémentaires), sinon ils sont réputés faux (Hypothèse de monde clos)

[9]:
```
%%writefile prog.pl
/* nbZero(L,N) est vrai ssi la liste L contient N zéros */

/* des faits : */
nbZero([],0).
nbZero([0],1). /* fait utile ? */
nbZero([1],0). /* fait utile ? */


/* des règles : */
nbZero([1|L],N):- /* règle utile ? */
```

```
    nbZero(L,N).
nbZero([0|L],NplusUn):-
    nbZero(L,N),
    plusUn(NplusUn,N).
nbZero([X|L],N):-
    dif(X,0),
    nbZero(L,N).

plusUn(NplusUn,N) :-
    NplusUn is N+1.

main :-  writeln('L ?'), read(L),
  nbZero(L,N), write('Pour L = '), write(L), write(' N : '), writeln(N).
:- main.
```

```
Overwriting prog.pl
```

des requêtes :

[10]:
```
!echo "[1,0,0,1,1,0]." | swipl -g halt -s prog.pl
```

```
L ?
Pour L = [1,0,0,1,1,0] N : 3
```

[11]:
```
!echo "[A,B,C]." | swipl -g halt -s prog.pl
```

```
L ?
Pour L = [1,1,0] N : 1
```

1.1 Structures de données

- constantes (entier, chaine, etc.) et identificateurs = commence par une minuscule
- Variables = Commence Par Une Majuscule (_ou _un _souligné "_"),
  Attention : les variables sont à instanciation unique
  (hors retour sur points de choix, les variables ont vocation à prendre une valeur et ne pas en changer ensuite au cours d'une exécution : on ne peut pas faire N = N+1, ou N++)

- Listes = liste vide [ ], liste explicite [1, 2, 3] et constructeur de liste [E | L]

Autres :

- Liste = File (pour avoir une file, prendre 2 piles ?, ou 2 listes), **Pile**, Séquence ou TeteQueue ?
- Ensemble ? Dictionnaire ?
- Arbre = Liste de liste, ex. : `[Racine,[FilsGauche,[],[]],[FilsDroit,[],[]]]`
- Graphes, liste circulaire (plus tard)
- Pointeur ? (tout est pointeur, partage naturel des données, possibilité de structures incomplètes et concaténation en temps constant)

Q ? Peut-on parcourir les nombres de 1 à N ?

[12]:
```
%%writefile prog.pl
/* parcourEtAffiche(N) est vrai ssi R parcours et affiche les entiers jusqu'à N␣
 ↪*/
parcourEtAffiche(0).
parcourEtAffiche(N):-
  moinsUn(NMoins1,N),
  parcourEtAffiche(NMoins1),
  writeln(N).

moinsUn(NMoinsUn,N) :-
  NMoinsUn is N-1.

main :-  writeln('N ?'), read(Entree),
  parcourEtAffiche(Entree).
:- main.
```

```
Overwriting prog.pl
```

[13]:
```
!echo "5." | swipl -g halt -s prog.pl
```

```
N ?
1
2
3
4
5
```

1.2 Structure de contrôle

- Unification (affectation *implicite*, opérations *structurelles* [hors arithmétique], test d'égalité, pour le test de différence : dif(X,Y) ; si besoin eq(X,X) pour test et affectation explicite)
- Groupe de règles (liées syntaxiquement par des points ".") : liées au niveau sémantique par un **OU** (ou "naturel" de la logique classique, pas le "ou" exclusif ; chaque règle doit pouvoir être *choisie* lors d'une résolution), représente une décomposition du problème par un **Raisonnement par Cas**, les cas peuvent être disjoints ou se chevaucher, l'espace recouvert est la zone de résolution du problème, il faut qu'il y ait égalité entre l'un et l'autre (que l'union des cas recouvre *complètement* la zone de résolution du problème)
- Corps d'une règle : faits liés au niveau sémantique par des **ET** (et syntaxiquement par des virgules ","), lors de la résolution d'une requête, tous les faits doivent être prouvés, il y a **enchaînement** de la recherche de ces preuves selon l'ordre d'apparition dans le fichier
- Récursivité de la recherche de faits pour un même ensemble de règles possible ! sous réserve d'avoir des règles s'appliquant sans récursivité (sinon, problème de boucle infinie). Il s'agit d'une forme de résolution par sous-problèmes isomorphes, ex. : diviser pour régner (divide and conquer) ; souvent cela ressemble à une répétition (enrichie de l'accès explicite à l'ensemble de la pile d'appel, même si cette pile d'appel n'est pas toujours utilisée au delà du haut de pile)
  - Motif de base :

```
prog(Env,Sol). %%cas de base (=sans appel récursif)
prog(Env,Sol) -: %%cas de décomposition et de propagation de la résolution
  décomposition(Env,E,SousEnv),
  prog(SousEnv,SousSol),
  recomposition(SousSol,E,Sol).
```

- Rem. : `décomposition` et `recomposition` peuvent avoir lieu dès la tête de la règle.
- Rem. : une décomposition peut aussi donner lieu à un autre problème, il n'y a pas alors nécessairement récursivité (à priori), mais tout de même résolution du sous-problème puis recomposition, ex. : coupe selon pivot choisi en début de liste. Dans ce cas, une astuce (?!) pour rester dans les résolutions récursives, consiste souvent à ne pas effectuer vraiment la décomposition, et garde l'environnement complet dans l'appel récursif.
- Rem. : une récursivité peut être *lancée* pour chaque paramètre ayant une structure récursive (pas seulement pour les paramètres pensés comme des données en entrée du problèmes ; les paramètres pensés comme les solutions du problème peuvent aussi être sujet à décomposition récursive : chaînage *avant* **vs** chaînage *arrière*)
- Rem. : les **cas de base** habituels pour les listes concernent la liste vide, mais peuvent aussi concerner les listes à 1 éléments, 2 éléments, ou plus, 1 élément ayant une propriété particulière (nul, non nul, etc.)
- Rem. : les **cas de propagation** habituels pour les listes concernent des décompositions de liste `[E|L]`, et concerne un appel récursif sur `L` mais peuvent aussi concerner des décompositions de liste avec 2 éléments, ou plus, (etc.), et peuvent concerner plus d'un appel, sur `L` ou sur `E` (etc.)
- Le zoo des formes de récursivité n'est pas limité au motif de base, il est aussi fourni (et pourquoi pas plus) que le zoo des formes de répétition dans les langages classiques
- Exemple : filtrage, recherche, ajout, concaténation

[31]:
```
%%writefile prog.pl
/* paritéLongueur(L,P) est vrai ssi la liste L est de parité P */ /* version␣
 ↪avec 2 propagations */
pariteLongueur1([],pair).
pariteLongueur1([_E],impair).  /* fait utile ? */
pariteLongueur1([_E|L],impair):-
  pariteLongueur1(L,pair).
pariteLongueur1([_E|L],pair):-
  pariteLongueur1(L,impair).

main :-  writeln('L ?'), read(L),
  pariteLongueur1(L,P), write('Pour L = '), write(L), write(' P : '),␣
 ↪writeln(P).
:- main.
```

```
Overwriting prog.pl
```

[35]:
```
%%writefile prog.pl
/* paritéLongueur(L,P) est vrai ssi la liste L est de parité P */ /* version␣
 ↪avec 2 prédicats */
pariteLongueur2([],pair).
```

```
pariteLongueur2([_E|L],Q):-
  pariteLongueur2(L,P),
  inverse(P,Q).

inverse(pair,impair).
inverse(impair,pair).

main :-  writeln('L ?'), read(L),
  pariteLongueur2(L,P), write('Pour L = '), write(L), write(' P : '),␣
 ↪writeln(P).
:- main.
```

```
Overwriting prog.pl
```

[33]:
```
%%writefile prog.pl
/* paritéLongueur(L,P) est vrai ssi la liste L est de parité P */ /* version␣
 ↪avec 2 faits */
pariteLongueur3([],pair).
pariteLongueur3([_E],impair).
pariteLongueur3([_E,_F|L],P):-
  pariteLongueur3(L,P).

main :-  writeln('L ?'), read(L),
  pariteLongueur3(L,P), write('Pour L = '), write(L), write(' P : '),␣
 ↪writeln(P).
:- main.
```

```
Overwriting prog.pl
```

[36]:
```
!echo "[1,2,3,4]." | swipl -g halt -s prog.pl
```

```
L ?
Pour L = [1,2,3,4] P : pair
```

[28]:
```
!echo "L." | swipl -g halt -s prog.pl
```

```
L ?
Pour L = [] P : pair
```

2 Algorithmique classique

ProLog permet-il de faire la même chose que les langages “classiques” ?

- Structures de données
  - similaire, sauf pour les Variables, les Variables prolog sont moins souples (mais cela facilitera les preuves de programme plus tard)
    la notion de variable classique (modifiable sur place) est plus proche de la notion de paramètre (et il y a plus de paramètre en programmation logique qu’en programmation

classique, cela permet de compenser)
Si vous prévoyez de transposer un algorithme classique avec une variable locale changeant régulièrement de valeur (par exemple un accumulateur), ajouter un paramètre à votre programme (pensez à bien l'initialiser)

- Structures de contrôle
  - Expression (structurelles), Affectation, Séquence, Test : similaire
  - Conditionnelle : raisonnement par cas, cependant dans les conditionnelles classiques le "**OU**" est exclusif, pour obtenir le même résultat (si nécessaire), il faut garantir le non recouvrement des différents cas, que les cas représentent une partition de l'espace de résolution. Ex pour `Si C alors A sinon B`, traduction :

```
prog(Env,Sol) :-
  testC(Env),
  progA(Env,Sol).
prog(Env,Sol) :-
  testNonC(Env),
  progB(Env,Sol).
```

- Répétition (boucle) : les boucles sont des conditionnelles récursives ! Traduction de la boucle `Tant que C faire A`:

```
progTantQue(Env,Sol) :-
  testC(Env),
  progA(Env,SolInterm),
  progTantQue(Env,SolInterm).
progTantQue(Env,Sol) :-
  testNonC(Env).
```

Rem. la programmation logique ne permet pas de faire plus/mieux que la programmation classique (il est "facile' de simuler la programmation logique avec la programmation classique [les algo d'unification et de résolution sont donnés])

[65]:
```
%%writefile prog.pl
/* max(A,B,M) est vrai ssi le max de A et B vaut M */   /* exemple de␣
 ↪conditionnelle simple */
max(A,A,A).
max(A,B,A):-
  A>B.
max(A,B,B):-
  B>A.

main :-  writeln('A ?'), read(A), writeln('B ?'), read(B),
  max(A,B,M), write('M : '), writeln(M).
:- main.
```

```
Overwriting prog.pl
```

[66]:
```
!echo "2. 4." | swipl -g halt -s prog.pl
```

```
A ?
```

```
B ?
M : 4
```

[67]:
```
%%writefile prog.pl
/* longueurClassique(L,N) est vrai ssi L est une liste de longueur N */   /*␣
 ↪exemple de boucle simple version classique ! */
longueurClassique(L,N) :-
  longueur(L,0,N).

longueur([_E|L],LongLoc,N) :-
  plusUn(LongLocPlusUn,LongLoc),
  longueur(L,LongLocPlusUn,N).
longueur([],LongLoc,LongLoc).

plusUn(NplusUn,N) :-
    NplusUn is N+1.

main :-  writeln('L ?'), read(L),
  longueurClassique(L,N), write('N : '), writeln(N).
:- main.
```

```
Overwriting prog.pl
```

[68]:
```
!echo "[1,2,3,4]." | swipl -g halt -s prog.pl
```

```
L ?
N : 4
```

[70]:
```
%%writefile prog.pl
/* longueur(L,N) est vrai ssi L est une liste de longueur N */   /* version␣
 ↪récursive */
longueur([_E|L],NPlusUn) :-
  longueur(L,N),
  plusUn(NPlusUn,N).
longueur([],0).

plusUn(NplusUn,N) :-
    NplusUn is N+1.

main :-  writeln('L ?'), read(L),
  longueur(L,N), write('N : '), writeln(N).
:- main.
```

```
Overwriting prog.pl
```

[71]:
```
!echo "[1,2,3,4]." | swipl -g halt -s prog.pl
```

```
L ?
```

```
N : 4
```

3 Propriétés algorithmiques

3.1 Correction

Un programme (ensemble de faits et de règles) est correct vis à vis d'une spécification quand les faits produits par ce programme correspondent aux spécifications (i.e. : sont corrects vis à vis des spécifications, vérifient ces spécifications)

Pour prouver la correction d'un programme, il faut vérifier que les faits sont correctes et que chaque règle engendre des faits corrects. Chaque preuve est réduite à la règle, avec une hypothèse de correction globale (c'est un avantage du découpage / raisonnement par cas)

Faudra-t-il faire des preuves formelles ? Ce serait faire dire des "et" à des "et" et composer des spécifications (informelle ?).

Dans une approche semi-formelle, il suffit de trouver des arguments convaincants ; cela doit reposer sur des quasi-évidences ; pour chaque fait et règle, individuellement

Conclusion : version limitée de la correction (un minimum d'argumentation sera nécessaire, mais "on" n'ira pas au delà)

3.2 Complétude

Un programme sera dit complet vis à vis d'un problème, si toutes les solutions seront produites par ce programme ; il ne sera pas complet, si des instances du problème ayant des solutions ne reçoivent pas de solutions

Pour prouver la complétude d'un programme, il faut prouver que le découpage par cas recouvre l'ensemble données et des solutions du problème. Certains cas peuvent donc ne pas apparaître dans le découpage si ils ne donnent pas lieu à solution, mais sinon, il faut qu'ils y soient.

Une analyse des têtes de règle permet souvent de prouver la complétude d'un programme (c'est le cas de problème à solution unique pour un ensemble d'entrée déterminé)

La preuve de complétude nécessite donc une analyse globale d'un ensemble de règle (souvent réduite à l'analyse des têtes de règles, mais cela reste une analyse globale)

Des contraintes dans le corps des règles peuvent invalider certains cas/tête. Dans ce cas, l'analyse doit être plus fine.

Pour faciliter l'analyse de la complétude, une analyse de la combinatoire des situations peut être utile. Ex. : cas de l'analyse d'une partie de shifumi (peut dépendre de ce que l'on veut avoir comme comportement : le nom du vainqueur, si le premier jour gagne, si la partie est nulle, etc.)

Rem. : un programme complet peut fournir plusieurs solutions (identiques ou différentes) à un problème donnée, c'est le cas des résolutions non-déterministes (voir le non-déterminisme plus loin), cela n'empêche pas le programme d'être complet.

3.3 Terminaison

L'exécution d'un programme termine si la résolution de la requête initiale se termine après un nombre fini d'étapes (pas de boucle infini).

Parmi les exécutions de programmes ne terminant pas :

- il y a ceux qui cherchent une solution indéfiniment mais ne donnent aucune solution (en un temps fini, qlq soit sa durée)
- il y a ceux qui donnent une ou une partie des solutions en un temps fini, mais qui échouent à donner certaines solutions quelque soit le temps attendu
- il y a ceux qui donneront tous les solutions attendues si l'on attends assez longtemps (c'est le cas de programmes décrivant dans un bon ordre un ensemble infini de solution)

Pour prouver la terminaison d'un programme, il suffit en général d'exhiber une fonction archimédienne bornée basée sur l'exécution du programme, par exemple la taille d'une liste, de taille finie au début et réduite d'un élément au moins à chaque appel récursif (c'est un cas courant).

La preuve doit être fait pour chaque appel récursif (individuellement).

Au delà de la terminaison, la complexité algorithmique d'un programme peut être définie, avec 2 variantes (par ex.) :

- nombre d'étapes nécessaire pour produire la première solution d'un programme
- nombre d'étapes nécessaire pour produire l'ensemble des solutions et s'arréter

Attention, les preuves de terminaison disent si les exécutions se terminent, elles ne disent pas si elles se terminent avec une solution ou pas. Attention aux programmes qui se terminent trop facilement, sans donner de solution ou en donnant des solutions qui n'en sont pas (par exemple, pour vérifier la croissance d'une liste, ou couper une liste en deux sous-listes de même longueur [ou presque], en prenant 2 éléments par 2 éléments, la croissance sera par morceau, ou le découpage peut échouer…)

3.4 Exemple de programmes à problème

Termine, correct, mais incomplet

[73]:
```
%%writefile prog.pl
/* coupe(L,P,Q) est vrai ssi L est une liste et P, Q deux sous listes (rg pair/
 ↪impair) */
coupe([],[],[]).
coupe([E,F|L],[E|P],[F|Q]) :-
  coupe(L,P,Q).

main :-  writeln('L ?'), read(L),
  coupe(L,P,Q), write('P : '), writeln(P), write('Q : '), writeln(Q).
:- main.
```

```
Overwriting prog.pl
```

[74]:
```
!echo "[1,2,3,4]." | swipl -g halt -s prog.pl
```

```
L ?
P : [1,3]
Q : [2,4]
```

Termine, complet, mais incorrect

[78]:
```
%%writefile prog.pl
/* croissant(L) est vrai ssi L est une liste croissante */
croissant([]).
croissant([E,F|L]) :-
  E<F,
  croissant(L).

main :-  writeln('L ?'), read(L),
  croissant(L), writeln('croissant.').
:- main.
```

```
Overwriting prog.pl
```

[84]:
```
!echo "[1,2,3,4]." | swipl -g halt -s prog.pl
```

```
L ?
croissant.
```

Complet, correct, mais ne termine pas.

[85]:
```
%%writefile prog.pl
/* coupe(L,P,Q) est vrai ssi L est une liste et P, Q deux sous listes (rg pair/
 ↪impair) */
coupe(L,P,Q) :-
  longueur(P,N),
  longueur(Q,N),
  concatene(P,Q,L).
coupe(L,P,Q) :-
  longueur(P,N),
  longueur(Q,NPlus1),
  plusUn(NPlus1,N),
  concatene(P,Q,L).

longueur([_E|L],NPlusUn) :-
  longueur(L,N),
  plusUn(NPlusUn,N).
longueur([],0).

plusUn(NplusUn,N) :-
    NplusUn is N+1.

concatene([],F,F).
concatene([E|L],F,[E|R]):-
```

```
  concatene(L,F,R).


main :-  writeln('L ?'), read(L),
  coupe(L,P,Q), write('P : '), writeln(P), write('Q : '), writeln(Q).
:- main.
```

```
Overwriting prog.pl
```

[86]:
```
!echo "[1,2,3,4]." | swipl -g halt -s prog.pl
```

```
L ?

ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 0.8Gb, global: 0.2Gb, trail: 38.4Mb
ERROR:   Stack depth: 5,032,945, last-call: 0%, Choice points: 5,032,923
ERROR:   Possible non-terminating recursion:
ERROR:     [5,032,945] user:longueur(_40266196, _40266198)
ERROR:     [5,032,944] user:longueur([length:1|_40266224], _40266218)
Warning: Goal (directive) failed: user:main
```

3.5 Autres propriétés (liées à la programmation déclarative)

Non-déterminisme : quand plusieurs solutions peuvent être données à une même situation. (rem. : certaines ou toutes les solutions peuvent être redondantes). C'est surtout utile pour des recherches combinatoires et de la programmation par contraintes (à voir plus tard).

Réversibilité : quand le programme permet de remonter des solutions aux données ; ou plus généralement quand la notion de donnée/sortie n'est pas prise en compte. Ajoute de la souplesse dans la programmation, mais peut être difficile à obtenir. (à voir plus tard).

3.6 Conclusion

Les propriétés correction/complétude/terminaison aident à écrire ses programmes avec un découpage local/global qui peut clarifier le travail :

- la complétude : pour trouver l'ensemble des cas de base (analyse globale souvent réduite aux têtes de règles)
- correction et terminaison : pour rédiger chaque cas individuellement (analyses locales, individuelles, pour chaque cas)

En programmation classique, (il arrive que l')on écrit/ve le programme puis on prouve la correction/complétude/terminaison et on calcul la complexité algorithmique ; en programmation déclarative, le processus est en partie inversé, on (peut) raisonner sur la complétude/correction pour écrire l'essentiel du programme en assurant la terminaison par construction et vérifier la complexité à la fin : la correction/completude/terminaison est obtenue par construction, ou à la construction, c'est peut-être plus délicat à écrire, mais au final on a les propriétés voulues sans avoir à les payer à la suite ! Le bilan est positif.

Et la programmation ressemble à la définition récursive de nos programmes, sous forme d'équations de récurrence.

4 Exemples

- premier, dernier
- ajouterEnPremier, ajouterEnDernier
- estTrié, différencesSuccessives
- concatener
- suppressionDoublonConsécutifs, inversionOrdre
- coupeEnDeux

4.1 Sur Caseine

Prévu pour TD2 :

- Ensemble 1
  - longueur
  - dérivation
  - inversion
  - palindrome
- Ensemble 2
  - somme
  - supprimer0
  - supprimerDoublonQlcq
- Ensemble 3
  - prefixe
  - coupe
  - echangeDebutFin

Prévu pour TD 3

- les 4 Tris
  - insertion et triInsertion
  - minimum et triSelectionMin
  - fusion et triFusion
  - coupe et triPivot