C, déclaratif ?

March 12, 2024

C Déclaratif ?

1) Introduction
2) C réversible
3) C non-déterministe

1 Introduction

La programmation déclarative repose sur quelques principes et modes de programmation :

- *Réduction des effets de bord*
- *Unification*
- Raisonnement par Cas
- Récursivité
- **Réversibilité**
- **Non-Déterminisme**
- *Contraintes*

(en gras les points qui distingue particulièrement ce paradigme de programmation, en italique ce qui est partagé par d'autres paradigmes de programmation de haut niveau, par ex. la programmation fonctionnelle)

Elle favorise une programmation de haut niveau, plus algorithmique, plus rapide (pour le programmeur, mais parfois peu efficace pour la machine) et permet ainsi des phases de prototypages ou de développement et de maintenance avancée.

Si les performances sont l'objectif principal d'un projet, après un prototypage en Prolog ou Erlang, il faut pouvoir passer en C (ou dans d'autres langages plus efficaces).

Parmi les points listés précédemment, certains sont transposables (plus ou moins) facilement en C (mais plutôt plus que moins) :

- Réduction des effets de bord : ne pas en utiliser simplement (c'est évité dans la plupart des paradigmes de programmation, car cela complique, voir rend impossible, les preuves de programmes)
- Unification : traduite par une analyse à la main d'une structure attendue (juste du code en plus)
- Raisonnement par Cas : les si-alors-sinon sont déjà là (depuis longtemps)
- Récursivité : peu utilisés, mais disponibles aussi (depuis longtemps)
- Contraintes : utiliser des contraintes en C peut se faire via l'utilisation de librairies […]

Pour la réversibilité et le non-déterminisme, le passage au C n'est peut-être pas aussi évident. Notons, que ce sont (hors les contraintes) des points qui distinguent fortement ProLog (avec ces points) de Erlang (sans ces points).

La thèse de Church-Turing postule que tous les langages de programmation ont des potentiels expressifs équivalents ; ce que Prolog peut faire, C doit pouvoir le faire aussi. Voyons comment.

2 C réversible

Commençons par un programme simple (l'incrémentation) et essayons d'écrire une incrémentation réversible en C.

Objectif : avoir une fonction inc qui permette de faire l'incrémentation de 3 en 4, ou de demander l'antécédent de 4 (et d'obtenir 3).

Dans le cas de l'incrémentation, l'objectif est réalisable, parce que la fonction mathématique est (facilement) inversible (le C, pas plus que ProLog ne pourra inverser tous les programmes). En Prolog, la réversibilité de l'incrémantation reposera par exemple sur l'emploi de contraintes et permettra les exécutions suivantes :

[3]:
```
%%writefile prog.pl
/* incrementation(X,Y) vrai ssi X+1 = Y */
incrementation(X,Y):-
  {X+1=Y}.  /* merci les contraintes */

main :-
  use_module(library(clpq)),
  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- main.
```

```
Overwriting prog.pl
```

[4]:
```
!echo "incrementation(41,Y)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [incrementation(41,42)]
```

[6]:
```
!echo "incrementation(X,42)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [incrementation(41,42)]
```

En C, pour pouvoir "manipuler" entrée et sortie, (donnée et résultat, et potentiellement, en inversant les rôles !), une solution consiste à utiliser 2 paramètres passés par adresse. C'est la solution usuelle pour avoir un/plusieurs résultats (et conserver le résultat principal, celui fourni par le `return`, pour les codes d'erreurs).

Pour avoir un comportement proche de ProLog et permettre que ces paramètres soient "connus"

ou “libre”, je propose qu'un type Entier soit défini comme pointeur d'entier (pointeur d'int), avec le pointeur vide pour signifier que la variable est libre. (D'autres solutions sont possibles, par exemple, une structure entier-booléen, avec le booléen pour dire si la variable est connue ou pas).

Paramètre par adresse, variable pointeur : attention, la double étoile arrive bientôt.

Pour les entrées, sorties, et mettre en place ces Entiers, pointeur d'int (comme entrée en matière) :

```
/* saisie */
r=scanf("%d",&monInt);
if (r&&(r!=EOF)) {
    monEntier = &monInt;}
else {
    monEntier = NULL;};

/* affichage */
if (monEntier) {
  printf("%d",*monEntier);}
else {
  printf("*");}
```

Pour saisir un entier : * défini : donner le nombre * non-défini : donner un caractère non numérique, par ex. ”*”

[72]:
```
%%writefile prog.c
#include <stdio.h>

int main() {
int monInt, r, *monEntier;
char c;

/* saisie */
r=scanf("%d",&monInt);
if (r&&(r!=EOF)) {
    monEntier = &monInt;}
else {
    monEntier = NULL;
    scanf("%c",&c);};

/* affichage */
if (monEntier) {
  printf("%d",*monEntier);}
else {
  printf("*");}

return 0;}
```

```
Overwriting prog.c
```

[73]:
```
!gcc prog.c
!echo -e "42" | ./a.out
```

```
42
```

[74]:
```
!gcc prog.c
!echo -e "*" | ./a.out
```

```
*
```

Pour l'incrémentation, le code est un peu plus long, il distingue les cas où les variables sont définies (ou pas), et indique (via le résultat global) si l'incrémentation est possible ou pas.
Rem. : comme le C n'est pas à instanciation unique, en cas de passage d'une variable du statut de non-défini à défini, une allocation mémoire est ajoutée (pas de partage mémoire) :

```
int incrementation(int **x, int**y) {  // x + 1 -> y ?
if ((!*x)&&(!*y)) {  //x et y inconnus, on ne peut rien faire
    return 1;}
else if ((*x)&&(*y)) {  //x et y sont connus, juste on vérifie que x+1=y
    return ((**y)!=((**x)+1));}
else if (*x) {   // selon que x (ou y) est connu, on fait l'incrémentation
    *y=(int *)malloc(sizeof(int));
    (**y) = (**x) + 1;
    return 0;}
else {           // ou l'inverse
    *x=(int *)malloc(sizeof(int));
    (**x) = (**y) - 1;
    return 0;}}
```

Tout ensemble :

[75]:
```
%%writefile prog.c
#include <stdio.h>
#include <stdlib.h>

int incrementation(int **x, int**y) {  // x + 1 -> y ?
if ((!*x)&&(!*y)) {
    return 1;}
else if ((*x)&&(*y)) {
    return ((**y)!=((**x)+1));}
else if (*x) {
    *y=(int *)malloc(sizeof(int));
    (**y) = (**x) + 1;
    return 0;}
else {
    *x=(int *)malloc(sizeof(int));
    (**x) = (**y) - 1;
    return 0;}
}
```

```c
int main() {
int a, b, r, *_a, *_b;
char c;

r=scanf("%d",&a);
if (r&&(r!=EOF)) {
    _a = &a;}
else {
    _a = NULL;
    scanf("%c",&c);};
r=scanf("%d",&b);
if (r&&(r!=EOF)) {
    _b = &b;}
else {
    _b = NULL;
    scanf("%c",&c);};

if (!incrementation(&_a, &_b)) {
    printf("+1 ok\n");}
else {
    printf("+1 ko\n");}

if (_a) {
  printf("%d\n",*_a);}
else {
  printf("*\n");}
if (_b) {
  printf("%d\n",*_b);}
else {
  printf("*\n");}
return 0;}
```

```
Overwriting prog.c
```

Et on teste, les 4 cas de figures, et sous-cas (connus/pas connus, et surcontraint ou incohérent) :

[76]:
```
!gcc prog.c
!echo -e "42\n43" | ./a.out
```

```
+1 ok
42
43
```

[80]:
```
!gcc prog.c
!echo -e "42 1024" | ./a.out
```

```
+1 ko
```

```
42
1024
```

[82]:
```
!gcc prog.c
!echo -e "* *" | ./a.out
```

```
+1 ko
*
*
```

[83]:
```
!gcc prog.c
!echo -e "42 *" | ./a.out
```

```
+1 ok
42
43
```

[84]:
```
!gcc prog.c
!echo -e "* 1024" | ./a.out
```

```
+1 ok
1023
1024
```

Tout marche bien !

Pour une version un peu plus intégrée :

[91]:
```
%%writefile prog.c
#include <stdio.h>
#include <stdlib.h>

typedef int *Entier ;

int lireEntier(Entier *x) {
  int rep;
  Entier tmp;
  char c;
tmp=(Entier ) malloc(sizeof(int));
rep=scanf("%d",tmp);
if (rep&&(rep!=EOF)) {
    *x = tmp;}
else {
    *x = NULL;
    free(tmp);
    scanf("%c",&c);}
return 1;}

int ecrireEntier(Entier x) {
```

```
if (x) {
  printf("%d\n",*x);}
else {
  printf("*\n");}
return 1;}

int estDefiniEntier(Entier x) {
return x!=NULL;}

int valeurEntier(Entier x) {
return *x;}

int affecteEntier(Entier *x, int v) {
if ((*x)==NULL) {
  *x=(Entier )malloc(sizeof(int)); }
**x=v;
return 1;}


int plusUn(Entier *x, Entier *y) {  // x + 1 -> y ?
// en sortie 0 : ok,
//           1 : ko (soit les 2 entiers sont indéfinis,
//                   soit ils sont définis mais ne respectent pas x + 1 = y)
if ((!estDefiniEntier(*x))&&(!estDefiniEntier(*y))) {
    return 1;}
else if (estDefiniEntier(*x)&&estDefiniEntier(*y)) {
    return (valeurEntier(*y)!=(valeurEntier(*x)+1));}
else if (estDefiniEntier(*x)) {
    affecteEntier(y,valeurEntier(*x) + 1);
    return 0;}
else {
    affecteEntier(x,valeurEntier(*y) - 1);
    return 0;}}


int main() {
Entier a, b;

lireEntier(&a);
lireEntier(&b);


if (plusUn(&a, &b)) {
    printf("+1 ko\n");}
else {
    printf("+1 ok\n");}
```

```
ecrireEntier(a);
ecrireEntier(b);
return 0;}
```

```
Overwriting prog.c
```

[87]:
```
!gcc prog.c
!echo -e "42 43" | ./a.out
```

```
+1 ok
42
43
```

[88]:
```
!gcc prog.c
!echo -e "* 43" | ./a.out
```

```
+1 ok
42
43
```

[89]:
```
!gcc prog.c
!echo -e "42 *" | ./a.out
```

```
+1 ok
42
43
```

[92]:
```
!gcc prog.c
!echo -e "* *" | ./a.out
```

```
+1 ko
*
*
```

[95]:
```
!gcc prog.c
!echo -e "42 1024" | ./a.out
```

```
+1 ko
42
1024
```

Autre exemple, sur les listes : Dupliquer l'élément en queue de liste.
(À l'envers, retire un élément en queue de liste (si doublon).)

La liste sera représentée de manière contiguë dans un tableau de longueur limitée, avec longueur explicite.

[96]:
```
%%writefile prog.c
#include <stdio.h>
```

```c
#include <stdlib.h>

typedef struct {int tab[10]; int lg;} *Liste;

int lireListe(Liste *x) { //non-liste : - ; liste vide :  ; liste 1, 2, 3 : 1 2␣
 ↪3
  int rep, elt, flt, i;
  Liste tmp;
rep=scanf("%d",&elt);
if (!rep) {
  *x=NULL;
  return 1;}
else if (rep==EOF) {
  *x=(Liste ) malloc(11*sizeof(int));
  (*x)->lg=0;
  return 1;}
else {
  *x=(Liste ) malloc(11*sizeof(int));
  (*x)->lg=0;
  for(i=0;i<10;i++) {
    (*x)->tab[(*x)->lg]=elt;
    (*x)->lg=(*x)->lg+1;
    rep=scanf("%d",&elt);
    if ((!rep)||(rep==EOF)) {
        return 1;}}}
return 1;}

int ecrireListe(Liste x) {
int i;
if (x) {
  for(i=0;i<x->lg;i++) {
  printf("%d \n",x->tab[i]);}}
else {
  printf("-\n");}
return 1;}


int dupliqueDernierElement(Liste *x, Liste *y) {  // x ++ -> y ?
// en sortie 0 : ok,  1 : ko
int i;
if (((*x) != NULL)&&((*y) != NULL)) {
    return 1;}
else if ((*x == NULL)&&(*y == NULL)) {
    return (*y)->tab[(*y)->lg-1] != (*y)->tab[(*y)->lg-2];} //on vérifie juste␣
 ↪que y se termine avec un doublon
else if (*y == NULL) {
    *y=(Liste ) malloc(11*sizeof(int));
```

```
    (*y)->lg=0;
    for(i=0;i<(*x)->lg;i++) {
      (*y)->tab[(*y)->lg]=(*x)->tab[(*y)->lg];
      (*y)->lg=(*y)->lg+1;}
    (*y)->tab[(*y)->lg]=(*y)->tab[(*y)->lg-1];
    (*y)->lg=(*y)->lg+1;
    return 0;}
else {
    *x=(Liste ) malloc(11*sizeof(int));
    (*x)->lg=0;
    for(i=0;i<(*y)->lg-1;i++) {
      (*x)->tab[(*x)->lg]=(*y)->tab[(*x)->lg];
      (*x)->lg=(*x)->lg+1;}
    return (*y)->tab[(*y)->lg-1] != (*y)->tab[(*y)->lg-2];}} //on vérifie que y␣
 ↪se termine avec un doublon

int main() {
Liste a=NULL, b=NULL;

lireListe(&a);
 if (dupliqueDernierElement(&a, &b)) {
    printf("dup ko\nOn ne peut pas inverser\n");
    ecrireListe(a);}
else {
    printf("dup ok\n");
    ecrireListe(a);
    printf(" donne -> \n");
    ecrireListe(b);}
printf("\n on inverse\n\n");
b=NULL;
if (dupliqueDernierElement(&b,&a)) {
    printf("dup ko\nOn ne peut pas inverser\n");
    ecrireListe(a);}
else {
    ecrireListe(a);
    printf(" est obtenue à partir de <<-- \n");
    printf("dup ok\n");
    ecrireListe(b);}
return 0;}
```

```
Overwriting prog.c
```

[97]:
```
!gcc prog.c
!echo -e "42 1024 1024" | ./a.out
```

```
dup ok
42
```

```
1024
1024
 donne ->
42
1024
1024
1024

 on inverse

42
1024
1024
 est obtenue à partir de <<--
dup ok
42
1024
```

[98]:
```
!gcc prog.c
!echo -e "42 1024" | ./a.out
```

```
dup ok
42
1024
 donne ->
42
1024
1024

 on inverse

dup ko
On ne peut pas inverser
42
1024
```

3 C non-détermiste

Objectif : Course à 100 Non Déterministe (faire une somme égale à 100 en prenant des sous-liste d'une liste d'entiers donnée par ordre de longueur des sous-liste, la plus courte en premier).

Si l'on avait un oracle qui nous permettait de choisir les bonnes valeurs à prendre (ou rejeter), cela pourrait être résolu en quelques lignes :

```
void mainRec(int LgFinSol, int k) {
if (k<10) {
  if (LgFinSol==0) {
    somme(k);}
```

```
  else {
    env[k]=__choixEnv(k)__ //0 ou 1 ?
    mainRec(LgFinSol [-1 selon choix],k+1);}}

int main() {
  mainRec(__choixLgSol()__,0);  // 1 .. 10 ?
return 0;}
```

Pour mettre en place, les points de choix, pour un problème général, il faudrait avoir une pile des environnements définis à chaque choix, et la liste des choix possibles.

Pour le problème posé, on se satisfera d'utiliser la pile des appels récursifs avec les 2 variables LgFinSol et k, (il faudrait ajouter sur cette pile l'environnement des variables définies, en particulier le tableau env[] (mais ne n'est pas immédiat, ni efficace, et ce n'est pas utile ici, car on travaille peu dessus)) et on liste les différents choix possibles, comme du code à exécuter, les uns à la suite des autres (avec une pile des environnement, entre deux choix, il faudrait rétablir l'environnement de départ).

Le résultat est le suivant (avec la définition auxiliaire de somme qui n'est là que pour vérifier que l'on a une solution ou pas, somme est déterministe, la partie non-déterministe est dans le main/mainRec) :

[101]:
```
%%writefile prog.c
#include <stdio.h>

int env[10], val[10]={10,20,30,50,80,70,50,20,20,10};

void somme(int tailleUtileTabVal) {
int s=0;
for(int i=0;i<tailleUtileTabVal;i++) {
    if (env[i]) {s=s+val[i];}}
if (s==100) {
  for(int i=0;i<tailleUtileTabVal;i++) {
    if (env[i]) {printf("%d ",val[i]);}}
  printf("\n");}}

void mainRec(int LgFinSol, int indEC) {
if (indEC<10) {
  if (LgFinSol==0) {
    somme(indEC);}
  else {
    env[indEC]=0;
    mainRec(LgFinSol,indEC+1);
    // pt de choix
    env[indEC]=1;
    mainRec(LgFinSol-1,indEC+1);}}}

int main() {
for (int lgSol=1;lgSol<=10;lgSol++) {  //points de choix itératifs
```

```
  mainRec(lgSol,0);}
return 0;}
```

```
Overwriting prog.c
```

Ca marche très bien :

[102]:
```
!gcc prog.c
!./a.out
```

```
80 20
80 20
50 50
30 70
20 80
30 50 20
30 50 20
30 50 20
30 50 20
20 30 50
20 30 50
10 70 20
10 70 20
10 20 70
10 50 20 20
10 50 20 20
10 20 50 20
10 20 50 20
10 20 50 20
10 20 50 20
10 20 30 20 20
```

4 Conclusion

Le C aussi peut être déclaratif !

remarques :

- travailler en C, permet le plus souvent d'avoir des programmes qui vont plus vites (2 fois plus vite, 50 fois plus vites, cela dépend des langages de comparaison), mais :
- changer de langage de programmation ne change pas (en général) la classe de complexité d'un programme, et donc :
- sur des grandes données, un programme linéaire en Prolog a toutes les chances d'aller plus vite qu'un programme quadratique en C
- le premier objectif, pour gagner en temps de calcul, c'est de travailler l'algorithmique pourquoi pas avec un langage de plus haut niveau qui permet d'être plus à l'aise (sans trop se soucier de détails de programmation)