TRAVAUX PRATIQUES: série n°4

Implémentation des types abstraits ARBRE et FORET

Christoph-Samuel / Jankowiak-Matthias

I-Problématique:

Ce TP a pour but de fournir une implémentation correcte pour les types abstraits ARBRE et FORET en s'appuyant sur les spécifications CASL proposées en cours. Nous devons réaliser l'implémentation de ces deux types abstraits.

En effet, pour cela nous allons réaliser les 4 étapes du cycle de développement initié lors de la série n°1:

- étape 1: la spécification du type abstrait
- étape 2: la validation de la spécification sous hets (DOLiator: hets en ligne)
- étape 3: l'implémentation de la spécification
- étape 4: la vérification de l'implémentation

II-Réalisation:

Étape 1: Spécification du type abstrait

La spécification s'effectue en 3 parties, on distingue l'en-tête (1 à 8), la signature (9 à 27) et la sémantique (29 à 52). La signature de la spécification est la partie visible ou interface de celle-ci. La sémantique est un ensemble de propriétés (axiomes), elle consiste à énoncer les propriétés des opérations du type abstrait.

Spécification en CASL :

```
1   library libraryArbreForet
2
3   %% liste des importations (downloading)
4   from Basic/Numbers get Int
5
6   %% spécification canonique
7   spec ARBRE0[sort Elem] given Int=
8   generated type
9     Foret[Arbre[Elem]] ::= foretVide|planter(Arbre[Elem];Int;Foret[Arbre[Elem]]);
10    Arbre[Elem]::= arbreVide| construire(racine:?Elem;listeSousArbres:?Foret[Arbre[Elem]])
11  end
12
13  spec ARBRE[sort Elem] =
14        ARBRE0[sort Elem]
15  then
16    preds
17    estArbreVide: Arbre[Elem];
18    estForetVide: Foret[Arbre[Elem]]
19    ops
20    racine:Arbre[Elem] ->?Elem;
21    listeSousArbres:Arbre[Elem] ->? Foret[Arbre[Elem]];
22    nombreArbres:Foret[Arbre[Elem]] ->? Int;
23    iemeArbre: Foret[Arbre[Elem]] * Int ->? Arbre[Elem]
24
25    forall A1:Arbre[Elem]; F1:Foret[Arbre[Elem]]; i1,k1:Int; e1:Elem
26     .def racine(A1) <=> not estArbreVide(A1)
27     .def listeSousArbres(A1) <=> not estArbreVide(A1)
```

```
28
29  %% accesseur estArbreVide
30    .estArbre Vide(arbreVide)
31    .not estArbreVide(construire(e1,F1))
32
33  %%accesseur estForetVide
34    .estForetVide(foretVide)
35    .not estForetVide(planter(A1,i1,F1))
36
37  %% accesseur racine
38    .racine(construire(e1,F1)) = e1
39
40  %% accesseur listeSousArbres
41    .listeSousArbres(construire(e1,F1)) = F1
42
43  %% accesseur nombreArbres
44    .nombreArbres(foretVide) = 0
45    .nombreArbres(planter(A1,i1,F1)) = nombreArbres(F1)+1
46
47  %% accesseur iemeArbre
48    .i1=k1  => iemeArbre(planter(A1,i1,F1),k1) = A1
49    .0< k1 /\k1 < i1 => iemeArbre(planter(A1,i1,F1),k1) =iemeArbre(F1,k1)
50    .i1 < k1 /\k1 < nombreArbres(F1)+2 => iemeArbre(planter(A1,i1,F1),k1) = iemeArbre(F1,k1-1)
51
52  end
```

Dans une spécification, une opération peut avoir deux statuts possibles les constructeurs et les accesseurs. Une opération est un constructeur si elle retourne un résultat, dont le type est défini par la spécification en cours. Ici les constructeurs sont : listeSousArbres, construire et planter.

Un accesseur du type est une opération qui permet d'observer l'état d'un objet du type et sans y apporter la moindre modification. Mais cette opération est un accesseur si et seulement si elle a au moins un argument de type défini, elle rend un résultat de type importée. Dans notre cas les accesseurs sont : estArbreVide(v), estForetVide(v) racine, nombreArbres, iemeArbre.

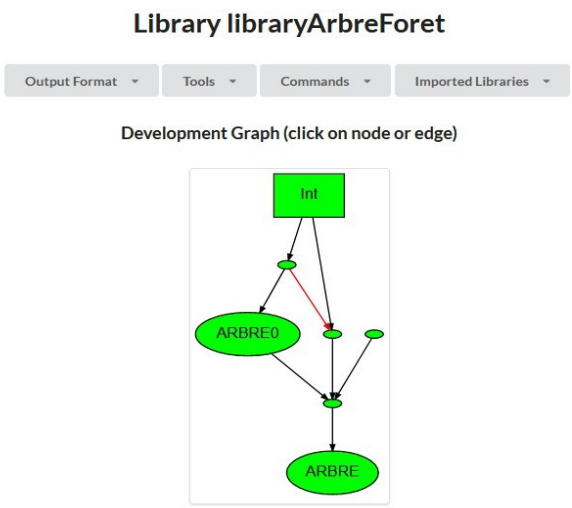
Pour dérouler le cycle de développement du type abstrait, il faut d'abord étudier le cahier des charges. Nous devons donc respecter les fonctionnalités suivantes : Pour les implémentations, on se limitera ici aux opérations suivantes (utilisées dans l'algorithme de parcours d'un arbre nécessaire pour bonne implémentation...):

-créer un arbre vide: Arbre[Elem]::= arbreVide;
-construire un arbre: construire(racine:? Elem; listeSousArbres:? Foret[Arbre[Elem]]);
-tester si un arbre est vide: estArbreVide: Arbre[Elem];
-connaître la racine d'un arbre: Arbre[Elem] → ?Elem;
-connaître le nombre de sous-arbres d'un arbre: Foret[Arbre[Elem]] → ?Int;
-connaître la forêt des sous-arbres d'un arbre: Arbre[Elem] → ?Foret[Arbre[Elem]];
-connaître le ième sous-arbre d'un arbre: Foret[Arbre[Elem]] * Int → ? Arbre[Elem];

Étape 2: Validation de la spécification du type abstrait.

Cette étape permet de prouver que la spécification satisfait la consistante (pas d'axiomes exprimant des propriétés contradictoires) et la complétude (suffisamment d'axiomes pour savoir si une certaine propriété est vraie ou fausse). Dans notre cas, la spécification est validée.

Résultat à partir de l'analyseur HETS:



L’un des outils les plus puissants permettant de valider une spécification est hets (Heterogeneous ToolSet). Il est développé par CoFi (Universität Magdeburg) Il est possible d’appeler l’analyseur HETS en ligne à partir de DOLiator (comme ci-dessus).

Étape 3: Implémentation de la spécification

Fichier.h

Pour cela on réalise un fichier.h et un fichier.c : Le fichier.h est un fichier d'interface qui est consultable par le futur utilisateur du type. Ce fichier est ici appelé libraryArbreForet.h

Voici notre fichier.h : libraryArbreForet.h

```c
/*
définition du fichier.h */
#ifndef LIBRARYARBREFORET_H
#define LIBRARYARBREFORET_H

/*
en-tête standard d'entrée/sortie en langage c */
#include <stdio.h>
/*
permet de gérer la mémoire dynamiquement (free, malloc...) */
#include <stdlib.h>
/*
propose un ensemble de fonction de traitement de caractères */
#include <ctype.h>

/*
définition de faux à la valeur 0 */
#define FAUX 0
/*
définition de vrai à la valeur 1 */
#define VRAI 1
/*
taille maximale de 100 éléments à la pile */
#define MAX 100

/*
type entier déf des booléens */
typedef int BOOLEEN;
/*
type entier déf d'un élement */
typedef int ELEMENT;

```

Comme vu lors des TP précédents la partie définition générale dans laquelle on retrouve la définition de notre fichier.h des fichiers d’en-tête, nos variables et constantes.

```c
/*
propose un type CONCRET pour implémenter le type ABSTRAIT des arbres */
/*
un arbre est défini par: sa racine et une forêt composée de ses sous-arbres */
typedef struct NOEUD_{
  ELEMENT etiquette;
  int nb_sous_arbres;
  struct NOEUD_* sous_arbres[MAX];
}*NOEUD;  /* type concret des noeuds */

/*
définition du type des noeuds: un type pointeur vers un objet de type arbre */
typedef NOEUD ARBRE;  /* type concret des arbres */

/*
propose un type CONCRET pour implémenter le type ABSTRAIT des forêt */
/*
une forêt est une liste dont les élément sont des arbres */
struct tree{
  ARBRE un_arbre;
  struct tree *suivant;
};

/*
définition du type des trees: un type pointeur vers un objet de type forêt */
typedef struct tree *FORET;  /* type concret des forêts */

```

Ensuite la définition des types concrets des arbres et des forêts avec leurs structures correspondantes.

```c

/* ---------------------------------- ABRRE ------------------------------------ */

/*
constructeur qui créer une arbre vide */
ARBRE arbreVide();
/*
constructeur construire qui construit un arbre a à partir d'un noeud o et d'une forêt f */
ARBRE construire(NOEUD o, FORET f);
/*
accesseur qui teste si un arbre a est vide */
BOOLEEN estArbreVide(ARBRE a);
/*
accesseur qui permet de donner la racine d'un arbre a */
NOEUD racine(ARBRE a) ;

```

La définition des constructeurs et accesseurs pour l'élément ARBRE.

```c

/* ---------------------------------- FORÊT ------------------------------------ */

/*
constructeur qui créer une forêt vide */
FORET foretVide();
/*
accesseur qui teste si une forêt f est vide */
BOOLEEN estForetVide(FORET f);
/*
constructeur qui donne la forêt composée des sous_arbre d'un arbre a */
FORET listeSousArbres(ARBRE a);
/*
constructeur qui ajoute un arbre a au rang i dans une forêt f */
FORET planter(ARBRE a, int i, FORET f);
/*
entier qui définit le nombre de sous arbres d'une forêt f */
int nombreArbres(FORET f);
/*
accesseur qui cherche un arbre de rang i dans la forêt f */
ARBRE iemeArbre(FORET f, int i);

```

La définition des constructeurs et accesseurs pour l'élément FORET.

```
98   /*
99   fin de la définition du fichier.h */
100  #endif
```

Et pour finir la fin de la définition de notre fichier.h

Fichier.c

Le fichier.c est un fichier d'implémentation, non consultable pour implémenter les corps de toutes les opérations déclarées dans libraryPile.h. Ce fichier est ici appelé libraryArbreForet.c

Voici notre fichier.c : libraryArbreForet.c

```
1   /*
2   en-tête standard d'entrée/sortie en langage c */
3   #include <stdio.h>
4   /*
5   permet de gérer la mémoire dynamiquement (free, malloc...) */
6   #include <stdlib.h>
7   /*
8   propose un ensemble de fonction de traitement de caractères */
9   #include <ctype.h>
10  /*
11  pour les en-têtes des constructeurs et accesseurs prédéfinis */
12  #include "libraryArbreForet.h"
13
```

La partie définition générale dans laquelle on retrouve les fichiers d'en-tête ainsi que notre fichier.h crée précédemment qui va inclure les en-têtes de nos fonctions.

```
14  /*
15  création d'un arbre vide */
16  ARBRE arbreVide() {
17    return NULL;
18  }
19
```

On retrouve les fonctions de tout les constructeur et accesseurs présentés dans notre .h, en commençant par arbreVide qui crée un élément de type ARBRE.

```
20  /*
21  construire un arbre a à partir d'un nœud o et d'une forêt f */
22  ARBRE construire(NOEUD o, FORET F)
23  {
24    FORET foret = F;
25    ARBRE A;
26    int nombre, i;
27    A = (ARBRE) malloc(sizeof(struct NOEUD_)) ;
28    A->etiquette=o->etiquette;
29    A->nb_sous_arbres = nombreArbres(F);
30    nombre = A->nb_sous_arbres;
31    for(i=1; i<=nombre; i++)
32    {
33      A->sous_arbres[i] = foret->un_arbre;
34      foret = foret->suivant;
35    }
36    return A;
37  }
38
```

Le constructeur construire qui construit un arbre a partir d'un nœud et d'une forêt.

```c
/*
tester un arbre est vide */
BOOLEEN estArbreVide(ARBRE A)
{
  return (A == NULL);
}

```

L'accesseur estArbreVide qui nous retourne NULL si un arbre a est vide.

```c
/*
donner la racine d'un arbre A */
NOEUD racine(ARBRE A)
{
  return A;
}

```

L' accesseur racine qui nous retourne la racine d'un arbre a.

```c
/*
créer une forêt vide */
FORET foretVide()
{
  return NULL;
}

```

L' accesseur foretVide qui crée un élément de type FORET.

```c
/*
tester un forêt est vide */
BOOLEEN estForetVide(FORET F )
{
 return (F == NULL);
}

```

L'accesseur estForetVide qui nous retourne NULL si une forêt f est vide.

```c
/*
donner la forêt composée des sous_arbres d'un arbre A */
FORET listeSousArbres(ARBRE A)
{
  FORET F, foret;
  int i, n;
  F = (FORET) malloc(sizeof(struct tree)) ;
  n = A->nb_sous_arbres;
  if (n == 0){
    return foretVide();
  }
  else
  {
    foret = F;
    for(i=1; i<=n; i++)
    {
      foret->un_arbre = A->sous_arbres[i];
      foret->suivant = (FORET) malloc(sizeof(struct tree));
      foret = foret->suivant;
    }
    foret = NULL;
  }
  return F;
}

```

Le constructeur listeSousArbre lui nous donne comme indiqué la liste (forêt) des sous-arbres d'un arbre a si les arbres ne sont pas vides sinon la fonction nous renvoie à l'accesseur forêtVide.

```c
/*
ajouter un arbre au rang i dans une forêt F */
FORET planter(ARBRE A,  int i, FORET F)
{
  int k;
  struct tree *la_premiere, *la_nouvelle, *actuelle, *la_precedente;
  /* cas d'insertion à la première place */
  if(i == 1)
  {
    la_premiere = (struct tree*) malloc(sizeof(struct tree));
    la_premiere->un_arbre = A;
    la_premiere->suivant = F;
    F = la_premiere;
  }
  /* cas général */
  else
  {
    actuelle = F;
    for(k=1; k<=i-1; k++)
    {
      la_precedente = actuelle;
      actuelle = actuelle->suivant;
    }
    la_nouvelle = (struct tree*) malloc(sizeof(struct tree));
    la_precedente->suivant = la_nouvelle;
    la_nouvelle->un_arbre = A;
    la_nouvelle->suivant = actuelle;
  }
  return F;
}

```

Le constructeur planter qui insère un arbre a dans une forêt f a un rang quelconque.

```c
/*
nombre de sous arbres d'une forêt F */
int nombreArbres(FORET F)
{
  FORET foret = F;
  int nombre = 0;
  while(foret != NULL)
  {
    nombre++;
    foret = foret->suivant;
  }
  return nombre;
}

```

L'accesseur nombreArbre qui nous retourne le nombre d'arbres présent dans une forêt f.

```c
/*
chercher l'arbre de rang i dans la forêt F */
ARBRE iemeArbre(FORET F,  int i)
{
  int k;
  FORET foret = F;
  if( i== 1)
  {
    return foret->un_arbre;
  }
  else{
    for(k=1; k<= i-1; k++)
    {
      foret= foret->suivant;
    }
  return foret->un_arbre;
  }
}
```

Et pour finir l'accesseur iemeArbre qui nous retourne un arbre a un certain rang dans une forêt f.

Étape 4: Vérification de l'implémentation

Afin de vérifier si l'implémentation est correcte vis à vis de sa spécification, nous allons créer un fichier preuveArbreForet.c (processus de la vérification formelle) incluant libraryArbreForet.c (permettant d'afficher les erreurs demandées en cas d'échec de compilation). Le principe de mise en œuvre consiste à vérifier que les constructeurs du type construisent correctement les objets du type. Cela signifie que ces constructeurs doivent satisfaire tous les axiomes énoncés en spécification.

La fonction main() va permettre de vérifier que:
– chacun des constructeurs implémentés
– satisfait tous les axiomes des accesseurs

Voici notre preuveArbreForet.c

```c
/*
en-tête standard d'entrée/sortie en langage c */
#include <stdio.h>
/*
permet de gérer la mémoire dynamiquement (free, malloc...) */
#include <stdlib.h>
/*
propose un ensemble de fonction de traitement de caractères */
#include <ctype.h>
/*
pour les fonctions constructeurs et accesseurs à vérifier */
#include "libraryArbreForet.c"

```

Comme précédemment, on retrouve la partie définition générale dans laquelle il y a les fichiers d'en-tête ainsi que notre fichier.c crée précédemment qui va inclure nos fonctions à vérifiées.

```c
/*
fonction principale */
int main()
{
  ARBRE a, a1;
  NOEUD e1;
  FORET F, F1;
  int i, k, success;

```

Au début du programme, on crée deux arbres (a et a1), deux forêts (F et F1) on initialise deux variables i et k nécessaire aux fonctions ainsi qu'une variable success qui va vérifier pour chaque constructeurs et accesseurs leur bon fonctionnement et on crée également un nœud e1.

```
23    /*
24    allocation mémoire et vérification */
25    a=malloc(sizeof(struct NOEUD_));
26    a1=malloc(sizeof(struct NOEUD_));
27    if(a == NULL || a1 == NULL)
28    {
29      fprintf(stderr,"Allocation impossible \n");
30      exit(EXIT_FAILURE);
31    };
32
33    /*
34    allocation mémoire et vérification */
35    F=malloc(sizeof(struct tree));
36    F1=malloc(sizeof(struct tree));
37    if(F == NULL || F1 == NULL)
38    {
39      fprintf(stderr,"Allocation impossible \n");
40      exit(EXIT_FAILURE);
41    };
42
```

On procède à une allocation mémoire pour les deux éléments ARBRE ainsi que pour les deux éléments FORET.

```
44    /*
45    vérification de l'implémentation du constructeur arbreVide */
46    a=arbreVide();
47    /*
48    initialiser l'indice success */
49    success=0;
50    /*
51    vérification avec l'accesseur estArbreVide
52    vérifier la propriété : estArbreVide(a)=True */
53    if(!(estArbreVide(a))) success=success+1;
54    /*
55    bilan de la vérification */
56    if(success != 0)
57    {
58      printf("\n Implémentation incorrecte du constructeur arbreVide");
59      printf("\n Interruption de la vérification : revoir l'implémentation du type asbtrait\n");
60      exit(EXIT_FAILURE);
61    };
62
```

On effectue un test de précondition pour le premier constructeur, si le test n'est pas fonctionnel la variable success prend pour valeur 1 rentre dans une boucle et affiche une erreur d'implémentation. On répète ensuite ce schéma pour les autres constructeurs et accesseurs, réinitialisation de success a 0, test et bilan de la vérification si erreur sinon on passe au prochain...

```
64    /*
65    vérification de l'implémentation du constructeur construire */
66    /*
67    ne pas oublier de saisir e1 */
68    printf("\n saisir un élément, e1 est le premier argument");
69    /*
70    ne pas oublier de saisir la foret F1 */
71    printf("\n saisir un élément, F1 est le second argument\n");
72    a=construire(e1,F1);
73
74
75    /*
76    réinitialiser la variable success */
77    success=0;
78    /*
79    vérification avec l'accesseur arbreVide */
80    if(estArbreVide(a)) success=success+1;
81    /*
82    bilan de la vérification */
83    if(success != 0)
84    {
85      printf("\n Implémentation incorrecte de estArbreVide(construire)");
86      printf("\n Interruption de la vérification: revoir l'implémentation du type abstrait\n");
87      exit(EXIT_FAILURE);
88    };
89
```

.
.
.
.
.

```
205   /*
206   réinitialiser la variable success */
207   success=0 ;
208   /*
209   vérification avec l'accesseur iemeArbre */
210   if(i==k)
211   {
212     if(iemeArbre(F,k) != a1) success=success+1;
213   }
214   if((0 < k) && (k < i))
215   {
216     if(iemeArbre(F,k) != iemeArbre(F1,k)) success=success+1;
217   }
218   if((i < k) && (k < nombreArbres(F1)+2))
219   {
220     if(iemeArbre(F,k) != iemeArbre(F1,k-1)) success=success+1;
221   }
222   /*
223   bilan de la vérification */
224   if(success != 0)
225   {
226     printf ("\n Implémentation incorrecte de iemeArbre(planter)");
227     printf ("\n Interruption de la vérification: revoir l'implémentation du type abstrait\n");
228     exit(EXIT_FAILURE);
229   };
230
```

```
232   /*
233   bilan de l'implémentation du type abstrait */
234   printf("\n L'implementation du type abstrait est vérifiée");
235   printf("\n Fin normale de la vérification de l'implémentation du type abstrait\n\n");
236   return EXIT_SUCCESS;
237 }
```

Pour finir si les deux dernier printf s'affichent (234 et 235) l'étape de vérification de l'implémentation est bonne et toutes les préconditions sont vérifiées.

II-Bilan/Conclusion:

Sur le plan théorique, réaliser ce TP nous a permis de comprendre le cycle de développement afin d'implémenter les types abstraits en lien avec notre problématique : le type ARBRE et FORET. L'utilisation d'un type ARBRE permet de montrer comment la hiérarchie d'objets comme par exemple de fichiers, sont organisés.

(La définition du type abstrait est ici avantageuse par rapport à la définition d'un type concret car elle est indépendante du langage de codage utilisé. Donc tout changement de langage ne remet pas cause cette définition. Il en résulte une grande stabilité des logiciels développés.)