U.F.R SCIENCES ET TECHNIQUES
Département d'Informatique
B.P. 1155
64013 PAU CEDEX
Téléphone secrétariat : 05.59.40.79.64
Télécopie : 05.59.40.76.54

TYPES ABSTRAITS DE DONNEES
Partie II

III- Sémantique d'un type abstrait
IV-Validation d'une spécification
V-Vérification formelle d'une implémentation

III- Sémantique d’un type abstrait

Position du problème

1-La **signature** fournit seulement une **interface** du type abstrait qui **déclare** des **sortes** et des **opérations**.

2-La signature n'intègre aucun élément pour exprimer les **propriétés(sémantique)** de ces sortes et opérations.

3-Elle ne suffit donc pas à définir **complètement** le type abstrait.

4-La signature doit être munie d'une **sémantique**.

5-C'est cette dernière qui confère des **propriétés** aux différentes **sortes** et **opérations** déclarées.

Approche algébrique

L'approche **algébrique** est utilisée pour munir une signature d'une **sémantique.**

Elle consiste:

-à définir les **sortes** déclarées

-en énonçant les **propriétés** des **opérations** qui manipulent les **objets** de ces sortes.

Pourquoi des axiomes ?

Les **propriétés des opérations** sont exprimées en décrivant le **comportement** de ces opérations.

Les **comportements** des opérations sont exprimés à l’aide d'**axiomes.**

Un **axiome** est :

-un **prédicat,**

-admis comme **vrai sans démonstration**.

Qu’est-ce qu’une théorie ?

L’**ensemble des axiomes** énoncés dans une spécification algébrique constitue une **théorie.**

On dit qu’une telle **théorie** confère une **sémantique** à la **signature** d’une spécification.

Validation d'une spécification

Position du problème

Lorsqu'on écrit les axiomes, pour définir un type, on est amené à se poser les deux questions :

N'y- a-t-il pas d'axiomes qui soient ***contradictoires ?***

A-t-on écrit ***suffisamment d'axiomes*** *pour établir* ***toutes*** *les propriétés des opérations du type ?*

1- Consistance

La première interrogation soulève le problème de **consistance.**

Un système d'axiomes de premier ordre est dit **consistant** si, et seulement si, il évite l'écriture des **propriétés contradictoires**.

En logique, la **consistance** est un problème **indécidable**.

En **pratique**, la consistance impose:

- l'absence de toute **contradiction**,
- parmi les **résultats** retournés par les **accesseurs** du type.

Pour s'assurer de la consistance, il faut vérifier que chaque **accesseur** du type retourne, lorsqu'il est appliqué, une **valeur et une seule**,

2- Complétude

La seconde interrogation soulève le problème de **complétude**.

Un système d'axiomes est dit **complet** si et seulement :

- il est **consistant**,
- pour toute formule P, on peut **prouver** soit **P**, soit **non P**.

D’après les travaux de A. Church, la **complétude** d’un système d’axiomes de premier ordre est un problème **indécidable**.

3- Comment construire les axiomes ?

En pratique, le critère proposé pour construire les axiomes est la **complétude suffisante**.

Qu'est-ce que la complétude suffisante ?

La **complétude suffisante** est une propriété qui, en pratique, **guide la construction** d'un système d'axiomes (théorie).

La complétude suffisante garantit que les axiomes construits sont **suffisants** pour **évaluer l'état** de **tous** les objets du type.

En d'autres termes, la complétude suffisante assure d'évaluer toute application possible :

- d'un **accesseur**
- à un **objet** du type.

Comment l'assurer **en pratique** ?

Il faut se rappeler que tout objet du type est le résultat d'application d'un **constructeur**.

Donc, assurer la **complétude suffisante** revient:

- à proposer **suffisamment** d'axiomes
- pour pouvoir **évaluer** l'état de tout objet **construit**.

On sait que tout **objet** est **construit** en appliquant un certain **constructeur.**

Soit **F**, un tel constructeur.

Par ailleurs, l'**état** de cet objet est **évalué** en appliquant les **accesseurs**.

Désignant par **G** n'importe lequel des accesseurs.

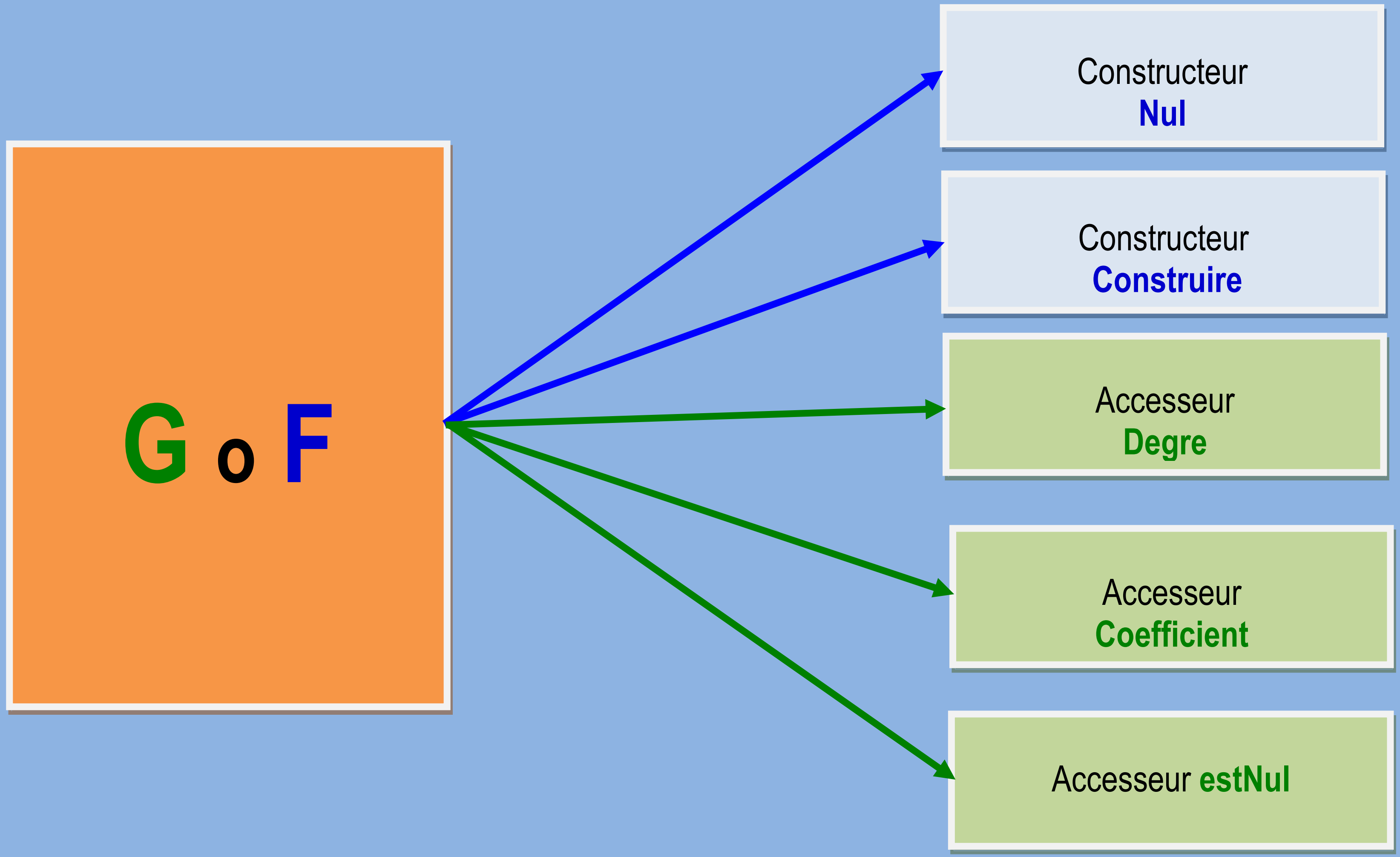
Donc, proposer **suffisamment** d'axiomes signifie:

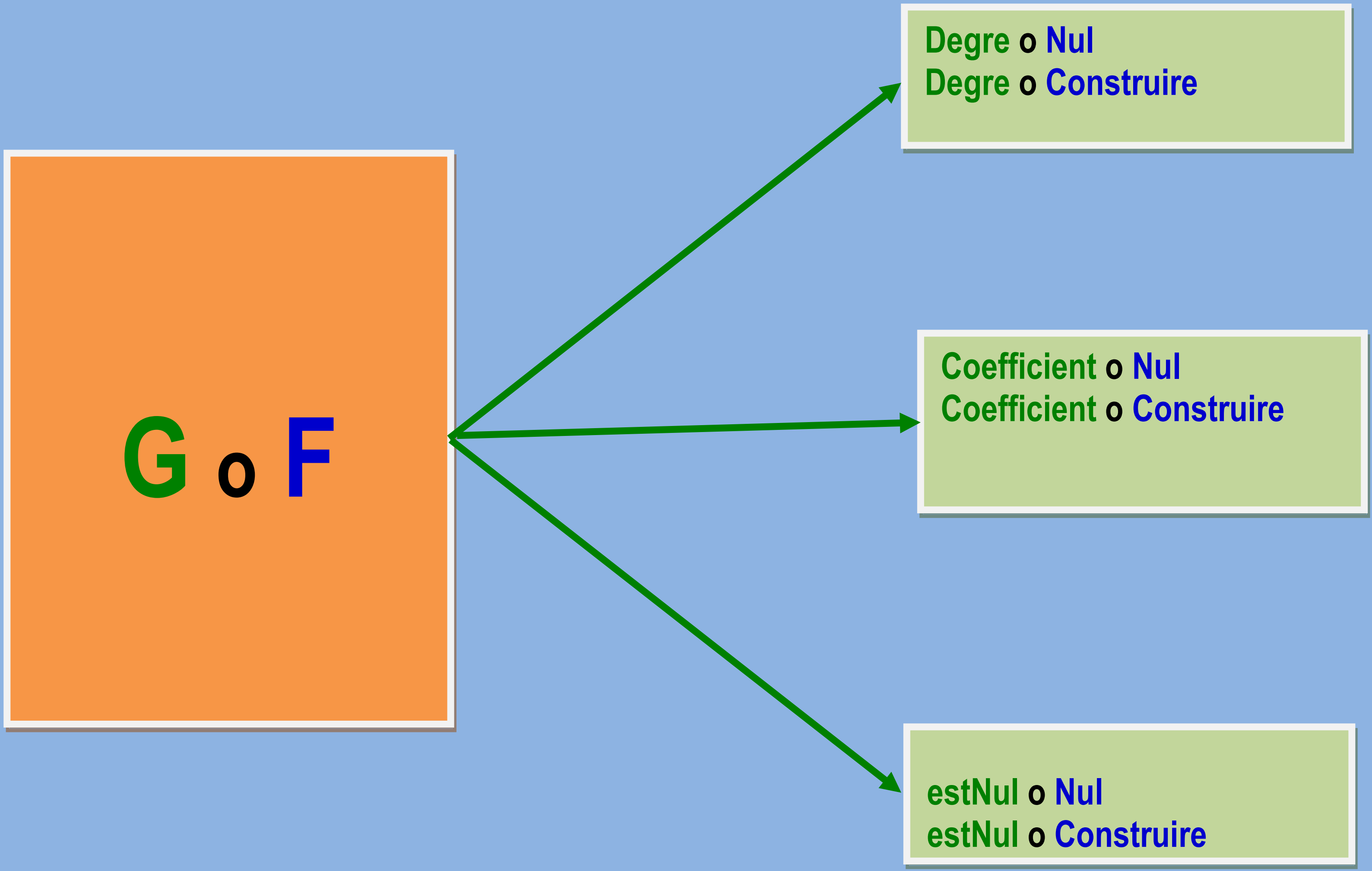
-qu'il est possible, à l'aide de **ces seuls axiomes**,

-d'**évaluer** de **façon unique** toute composition possible

G o F

Cas du type abstrait des polynômes




G o F
Degre o Nul
Degre o Construire
Coefficient o Nul
Coefficient o Construire
estNul o Nul
estNul o Construire

A retenir :

La **complétude suffisante** signifie que les axiomes proposés doivent être suffisants pour :

1- **calculer toutes les compositions G o F** possibles,
2- assurer que ce calcul produit un résultat **unique**.

La première condition satisfait la **complétude**.

La deuxième condition satisfait la **consistance**.

Exemple de mise en oeuvre

1-On part de la signature du type :

%% signature des opérations du type

Nul : Polynome

Construire : Polynome x Rat$\rightarrow$ Polynome

estNul : Polynome $\rightarrow$ Boolean;

Degre : Polynome $\rightarrow$Int;

Coefficient: Polynome x Int $\rightarrow$Real

2-On considère, d’une part, les **constructeurs** du type

%% signature des constructeurs du type

Nul : Polynome
Construire : Polynome x Rat$\rightarrow$ Polynome

3-On considère, d'autre part, les **accesseurs** du type

%% signature des accesseurs du type

estNul: Polynome $\rightarrow$ Boolean;

Degre : Polynome $\rightarrow$Int;

Coefficient: Polynome x Int $\rightarrow$Real

Sémantique du constructeur Nul()

Avec l'accesseur estNul() :

F → Nul()

p=Nul()

G → estNul()

GoF → estNul(Nul())

Pour évaluer **GoF** on génère l'axiome :

- **p = Nul() => estNul(p) = True**

Composé avec l'accesseur Degré() :

$$F \rightarrow \text{Nul()}$$

$$p = \text{Nul()}$$

$$G \rightarrow \text{Degré()}$$

$$GoF \rightarrow \text{Degre(Nul())}$$

Pour évaluer GoF on génère l'axiome :

- p = Nul() => Degre(p) = 0

Composé avec l’accesseur **Coefficient()**:

F $\rightarrow$ **Nul()**

p = Nul()

G $\rightarrow$ **Coefficient(-,i)**

GoF $\rightarrow$ **Coefficient(Nul()**,i **)**

Pour évaluer **GoF** on génère l’axiome :

forall i:Int

- **p = Nul()** => **Coefficient**((**p**,i) = 0

Sémantique du constructeur Construire()

<u>Composé avec l'accesseur **estNul()**</u> :

F $\rightarrow$ **Construire(p1,a0)**

p = Construire(p1,a0)

G $\rightarrow$ **estNul()**

GoF $\rightarrow$ **estNul(Construire(p1,a0))**

Pour évaluer **GoF** on génère l’axiome suivant:

p= Construire(p1,a0)

forall p1 : Polynome; a0 : Rat

- **estNul**(**p**) = True **<=>** estNul(p1)= True /\ a0 = 0

Composé avec l'accesseur Degré() :

$F \rightarrow$ Construire(p1,a0)

p= Construire(p1,a0)

$G \rightarrow$ Degré()

$GoF \rightarrow$ Degre(Construire(p1,a0))

Pour évaluer **GoF** on génère les 2 axiomes suivants :

p= Construire(p1,a0)

forall p1 : Polynome; a0 : Rat

- **estNul**(p1) = True => **Degre** (**p**) = 0
- **estNul**(p1) = False => **Degre** (**p**) = **Degre**(p1)+1

Composé avec l’accesseur **Coefficient ()** :

$$F \rightarrow \text{Construire}(p1,a0)$$

$$p = \text{Construire}(p1,a0)$$

$$G \rightarrow \text{Coefficient}(\;-\;,i)$$

$$GoF \rightarrow \text{Coefficient}(\text{Construire}(p1,a0),\; i)$$

Pour évaluer **GoF** on génère les 4 axiomes suivants :

$$p = \text{Construire}(p1,a0)$$

forall p1 :Polynome; a0: Rat , i :Int

- **estNul**(**p**) = True => **Coefficient**((**p**, i) = 0

- **Coefficient** (**p**,0)= a0

- i>=1 /\ i <= **Degre**(p1) +1
  => **Coefficient** (**p**, i) = **Coefficient**(p1,i-1)

- i> **Degre**(p1) +1 => **Coefficient** (**p**,i) = 0

Pour récapituler

En compilant tous les axiomes:

- on munit la signature d'une **théorie** (ensemble d'axiomes)
- laquelle théorie définit la **sémantique** de la spécification

forall p1:Polynome; a0:Rat ; i:Int

%% axiomes définissant les opérations par leurs propriétés

%%constructeur **Nul**

*%% avec l'accesseur **estNul***

- **estNul(Nul**) = True

*%%accesseur **Degre***

- **Degre**(**Nul**) = 0

*%%accesseur **Coefficient***

- **Coefficient**((**Nul**,i) = 0

%%constructeur **Construire**(p1,a0))

*%% avec l'accesseur **estNul***

- **estNul**(**Construire**(p1,a0)) = True <=> estNul(p1)= True /\ a0 = 0

*%%accesseur **Degre***

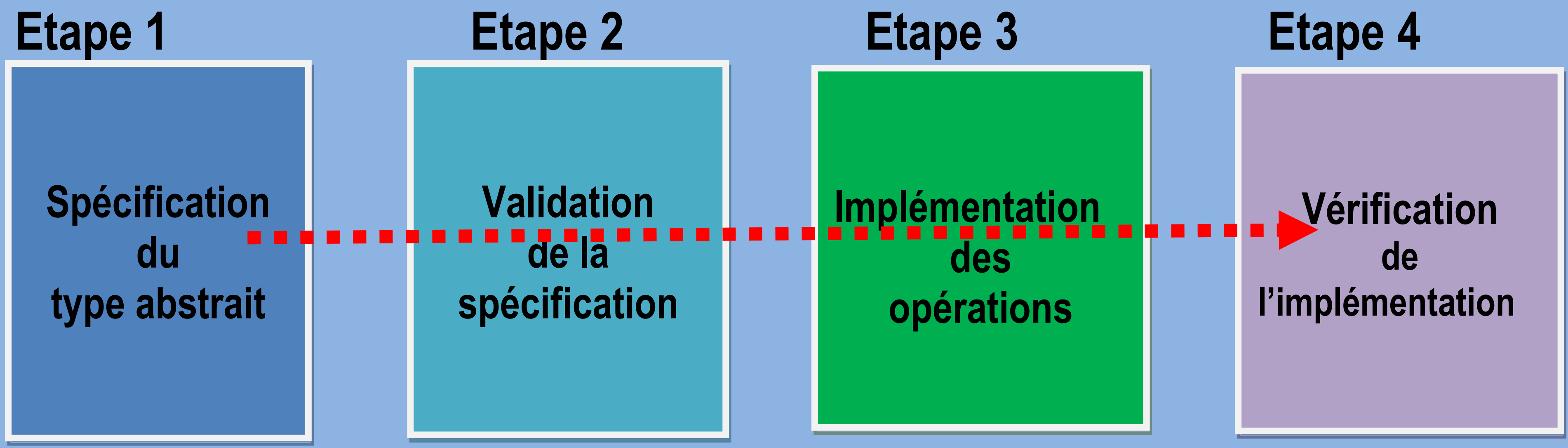
- **Degre** (**Construire**(**Nul**,a0)) = 0
- **estNul**(p1) = False => **Degre** (**Construire**(p1,a0)) = **Degre**(p1)+1

%%avec l'accesseur ***Coefficient***

- i=0 => **Coefficient** (**Construire**(p1,a0), i) = a0
- i>= 1 /\ i <= **Degre**(p1) +1 => **Coefficient** (**Construire**(p1,a0), i) = **Coefficient**(p1,i-1)
- i> **Degre**(p1) +1 => **Coefficient** (**Construire**(p1,a0),i) = 0

end

Cycle pour implémenter un TAD



IV-Validation d'une spécification

Il existe des outils pour valider les spécifications formelles.

L'un des plus puissants est le logiciel HETS (**H**eterogeneous **T**ool **S**et**)** développé par CoFi (Université de Brême)

http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/index_e.htm

On peut lancer **hets** par la commande suivante:

hets -g polynome.casl

où **polynome.casl** est le fichier contenant le composant **polynome**

Il est possible pour les spécifications de taille raisonnable d'appeler l'analyseur HETS **en ligne** à partir de **DOLiator** :

http://rest.hets.eu/

Deux modalités d'utilisation se présentent :

1-**télécharger** le fichier de spécification **polynome.casl**

Choose File

2-**séléctionner/copier** le texte de la spécification et le **coller** dans la fenêtre du DOLiator

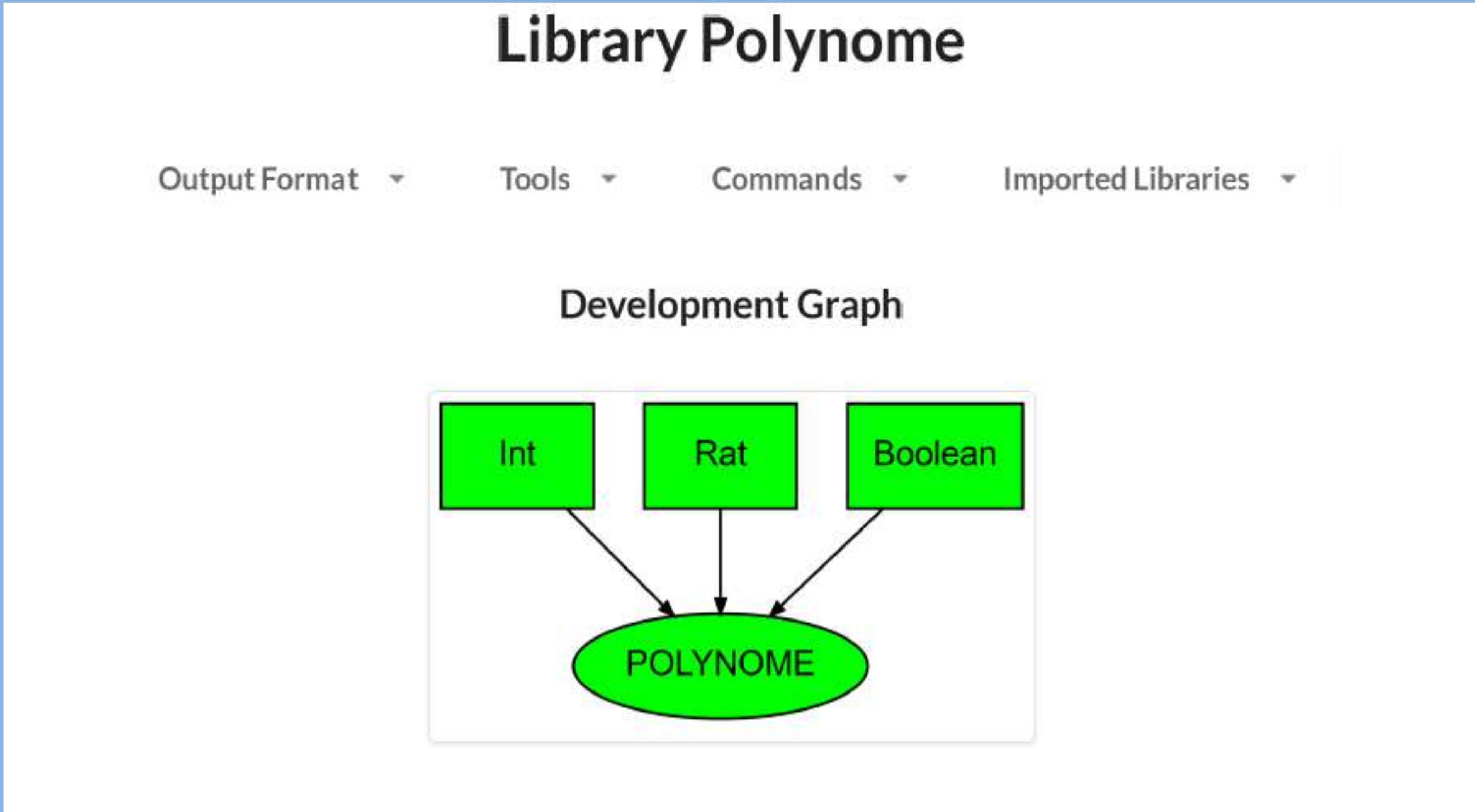
3-cliquer sur **submit**

```
library  Polynome
from Basic/Numbers                get     Int , Rat
from Basic/SimpleDatatypes     get     Boolean

spec  POLYNOME  =
     Rat and Int and Boolean
     then
     sort  Polynome
ops
    Nul : Polynome ;
    Construire : Polynome * Rat  -> Polynome ;
         estNul           : Polynome        -> Boolean;
         Degre            : Polynome        -> Int;
         Coefficient      : Polynome   * Int   -> Int

forall p1:Polynome; a0, x0:Rat; i:Int
         . estNul(Nul) = True
         . Degre(Nul) = 0
         . Coefficient (Nul,i) = 0
         . estNul(Construire(p1,a0)) = True <=>   estNul(p1)= True   /\   a0 = 0
         . Degre (Construire(p1,a0)) = 0   <=>  estNul(p1)= True
         . Degre (Construire(p1,a0)) = Degre(p1)+1   <=>  estNul(p1)= False
         . Coefficient (Construire(p1,a0), 0)  = a0
         . estNul(p1) = True  => ( i>0   => Coefficient (Construire(p1,a0), i) = 0 )
         . estNul(p1) = False  =>    ( i>= 1 /\  i <=  Degre(p1) +1  =>  Coefficient (Construire(p1,a0), i) = Coefficient(p1,i-1)  )
         . estNul(p1) = False  =>   ( i> Degre(p1) +1 => Coefficient (Construire(p1,a0),i) = 0  )
end
```

Si la validation est confirmée, l'analyseur affiche le graphe suivant:



V-Vérification d'une implémentation

Toutes les **propriétés** énoncées en spécification doivent être **satisfaites** par les implémentations futures du type abstrait.

On dit alors que l'implémentation doit être **correcte** vis-à-vis de sa spécification.

1-Qu’est-ce que la vérification formelle ?

La **vérification formelle** est le processus qui consiste à **prouver** qu’une l’implémentation est **correcte** vis à vis de sa spécification.

2-Mise en œuvre pratique

Etape 1

Spécification
du
type abstrait

Etape 2

Validation
de la
spécification

Etape 3

Implémentation
des
opérations

Etape 4

Vérification
de
l'implémentation

Cycle pour implémenter un TAD

2.1-implémentation des opérations du type

1-On commence par créer un fichier **interface** qui sera **consultable** par le futur utilisateur du type

Ce fichier est ici appelé **polynome.h**

2-On ensuite crée un second fichier d'**implémentation**, **non consultable** pour implémenter les **corps** de toutes les opérations déclarées dans **polynome.h**

Ce fichier est ici appelé **polynome.c**

1- Fichier **interface** **polynome.h**

C’est dans le fichier interface qu’il faut:

- proposer une **représentation** pour les objets de type **Polynome**,

- déclarer toutes des **opérations** du type

a-Proposer une **représentation** pour les objets du type

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define     MaxSize  10
#define     FAUX         0
#define     VRAI     1
typedef     int      BOOLEEN;
/*
Proposer un type CONCRET pour implémenter le type ABSTRAIT des polynomes
*/
typedef  struct un_polynome
     {
        int   sonDegre ;
         float  sonCoefficient[MaxSize];
     }  polynome;

/*  définition du type des polynômes: un type pointeur vers un objet de type polynôme */
typedef  struct un_polynome *  POLYNOME;
```

b-Déclaration des opérations du type

```
/*
 créer un polynôme nul*/
POLYNOME Nul();

/*
 construire le polynôme "p1*x + a0" */
POLYNOME Construire(POLYNOME p1, float a0);

/*
calculer le degré d'un polynôme */
int Degre(POLYNOME p);

/*
 calculer le coefficient de rang i d'un polynôme */
float Coefficient(POLYNOME p, int i) ;

/*
 tester si un polynôme  est un polynôme nul*/
BOOLEEN estNul(POLYNOME  p);
```

Exemple de fichier interface : polynome.h

(à copier, éditer sous **emacs** et compiler avec **gcc**)

```
#include   <stdio.h>
#include  <stdlib.h>
#include  <ctype.h>
#define        MaxSize   10
#define        FAUX              0
#define        VRAI      1
typedef     int           BOOLEEN;
/*
Proposer un type CONCRET pour implémenter le type ABSTRAIT des polynômes
*/
typedef   struct un_polynome
        {
         int    sonDegre ;
        float  sonCoefficient[MaxSize] ;
        }  polynome;

/*  définition du type des polynômes: un type pointeur vers un objet de type polynôme */
typedef  struct un_polynome *  POLYNOME;
```

```
/*
 créer un polynôme nul*/
POLYNOME Nul() ;

/*
 construire le polynôme "p1*x + a0" */
POLYNOME Construire(POLYNOME p1, float a0);

/*
calculer le degré d'un polynôme */
int Degre(POLYNOME p);

/*
 calculer le coefficient de rang i d'un polynôme */
float Coefficient(POLYNOME p, int i) ;

/*
 tester si un polynôme  est un polynôme nul*/
BOOLEEN estNul(POLYNOME  p);
```

2- Fichier **implémentation** **polynome.c**

Dans le fichier implémentation il faut:

-inclure le fichier **interface** : polynome.h

-implémentation de **toutes** des **opérations** déclarées dans le fichier **interface :**

Nul(), Construire(), estNul, Degré, Coefficient()

Implémentation des constructeurs du type

1-Spécification du constructeur Nul()

D'après la spécification, voici les propriétés (axiomes) qui **guident l'implémentation** de l'opération **Nul()**:

estNul(**Nul**) = VRAI
Degre(**Nul**) = 0
$\forall$ i$\in\mathbb{N}$ • **Coefficient**(**Nul** ,i) = 0

2- Exemple d'implémentation de Nul()

```
/*Créer un polynôme nul*/
POLYNOME Nul()
   {
    POLYNOME  p;
    int i;
    p = malloc(sizeof(struct un_polynome));
    if(p== NULL)
            {
            fprintf(stderr,"Allocation impossible \n");
            exit(EXIT_FAILURE);
             }
             else
            {
             p->sonDegre = 0;
             for(i=0;  i<=MaxSize-1;  i++)   p-> sonCoefficient[i] = 0;
          }
   return p;
   }
```

3-Spécification du constructeur **Construire( )**

D'après la spécification, voici les propriétés(axiomes) qui **guident l'implémentation** de l'opération **Construire(** p1,a0**) :**

$\textbf{estNul}(p1) = \text{True} \wedge a0 = 0 \Leftrightarrow \textbf{estNul}(\textbf{Construire(}\ p1,a0\textbf{)}) = \text{VRAI}$

$\textbf{estNul}(p1) = \text{True} \Leftrightarrow \textbf{Degre}(\textbf{Construire(}\ p1,a0\textbf{)}) = 0$

$\textbf{estNul}(p1) = \text{False} \Leftrightarrow \textbf{Degre}(\textbf{Construire(}\ p1,a0\textbf{)}) = \textbf{Degre}(p1)+1$

$\textbf{Coefficient}(\textbf{Construire(}\ p1,a0\textbf{)},0) = a0$

$\textbf{estNul}(p1) = \text{False} \Rightarrow$

$\quad (i \geq 1 \wedge i \leq \textbf{Degre}(p1)+1 \Rightarrow \textbf{Coefficient}(\textbf{Construire(}\ p1,a0\textbf{)}, i) = \textbf{Coefficient}(p1,i-1))$

$\textbf{estNul}(p1) = \text{False} \Rightarrow (i > \textbf{Degre}(p1)+1 \Rightarrow \textbf{Coefficient}(\textbf{Construire(}\ p1,a0\textbf{)}, i) = 0)$

4-Exemple d’implémentation **Construire( )**

```
POLYNOME Construire(POLYNOME p1, float a0)
 {
 POLYNOME  p;
 int i;
 p = malloc(sizeof(struct un_polynome));
 if(p== NULL)
          {
          fprintf(stderr,"Allocation impossible \n");
          exit(EXIT_FAILURE);
          }
else
        {
         if( estNul(p1))
                {p->sonDegre = 0;  p->sonCoefficient[0] = a0;
                 for(i=1;  i<=MaxSize-1;  i++)   p->sonCoefficient[i] = 0;
                  }
```

```
        else
              {p->sonDegre = p1->sonDegre +1;
              p->sonCoefficient[0] = a0 ;
              for(i=1;  i<=p1->sonDegre+1;  i++)            p->sonCoefficient[i] =p1->sonCoefficient[i-1];
              for(i=p1->sonDegre+2; i<=MaxSize-1; i++)   p->sonCoefficient[i] = 0;
             }
      }
  return p;
}
```

Implémentation des accesseurs du type

On implémente les **accesseurs** **directement** à partir de la représentation des données du type.

```
typedef   struct un_polynome
        {
         int    sonDegre ;
        float  sonCoefficient[MaxSize];
        }  polynome;

/*  définition du type des polynômes: un type pointeur vers un objet de type polynôme */
typedef  struct un_polynome *  POLYNOME;
```

1-Implémentation de l’accesseur **Degré()**

```
/*Calculer le  degré d’un polynôme */
int Degre(POLYNOME p)
       {return p->sonDegre;
      }
```

2-Implémentation de l’accesseur **Coefficient()**

```
/*Calculer le coefficient de rang i d'un polynôme */

float Coefficient(POLYNOME p, int i)
      {return p->sonCoefficient[i] ;
      }
```

3-Implémentation de l'accesseur estNul()

```
/*Tester si le polynôme  est nul*/
BOOLEEN estNul(POLYNOME  p)
        {int i;
        if (p->sonDegre != 0) return FAUX;
        for(i=0;  i<= p-> sonDegre;  i++)
                        if (p-> sonCoefficient[i] != 0) return FAUX;
        return VRAI;
      }
```

Exemple de fichier d'implémentation : **polynome.c**

```
#include "polynome.h"

/*Créer un polynôme nul*/
POLYNOME Nul()
   {
    POLYNOME  p;
    int i;
    p = malloc(sizeof(struct un_polynome));
    if(p== NULL)
            {
            fprintf(stderr,"Allocation impossible \n");
            exit(EXIT_FAILURE);
             }
             else
            {
             p->sonDegre = 0;
                for(i=0;  i<=MaxSize-1;  i++)   p-> sonCoefficient[i] = 0;
           }
   return p;
   }
```

```
/*
Construire un polynôme non nul */
POLYNOME Construire(POLYNOME p1, float a0)
  {
  POLYNOME  p;
  int i;
  p = malloc(sizeof(struct un_polynome));
  if(p== NULL)
           {
           fprintf(stderr,"Allocation impossible \n");
           exit(EXIT_FAILURE);
           }
 else
         {
          if( estNul(p1))
                    {p->sonDegre = 0;
                      p->sonCoefficient[0] = a0;
                       for(i=1;  i<=MaxSize-1;  i++)   p->sonCoefficient[i] = 0;
                     }
          else
                   {p->sonDegre = p1->sonDegre +1;
                   p->sonCoefficient[0] = a0 ;
                     for(i=1;  i<=p1->sonDegre+1;  i++)
```

```
                                p->sonCoefficient[i] =p1->sonCoefficient[i-1];
                        for(i=p1->sonDegre+2; i<=MaxSize-1; i++)
                                p->sonCoefficient[i] = 0;
                }
        }
  return p;
}
/*Calculer le  degré d'un polynôme */
int Degre(POLYNOME p)
        {return p->sonDegre;
          }
/*Calculer le coefficient de rang i d'un polynôme */
float Coefficient(POLYNOME p, int i)
        {return p->sonCoefficient[i];
          }
/*Tester si le polynôme  est nul*/
BOOLEEN estNul(POLYNOME  p)
         {int i;
         if (p->sonDegre != 0) return FAUX;
         for(i=0;  i<= p-> sonDegre;  i++)
                            if (p-> sonCoefficient[i] != 0) return FAUX;
         return VRAI;
          }
}
```

2.2-Vérification de l'implémentation

Le principe consiste à vérifier que les **constructeurs** du type **construisent correctement** les **objets** du type.

Cela signifie que ces **constructeurs** doivent satisfaire **toutes** les **propriétés** énoncées en **spécification**.

Mise en œuvre de la technique

1-**Construire les objets du type**

Les objets du type sont **construits** en appliquant les **constructeurs** du type.

Ici, deux constructeurs suffissent, à savoir :

Nul( )

Construire()

Donc, n'importe quel objet peut être construit en appelant les deux opérations suivantes :

p = Nul( )

p = Construire(p1, a0**)**

2-**Vérifer l’état des objets du type**

L’**état** des objets du type construits est évalué en appliquant les **accesseurs** du type.

Ici, trois accesseurs suffissent, à savoir :

estNul( )

Degre()

Coefficient()

Donc, l’état de n’importe quel objet **p** peut être évalué en appelant les trois opérations suivantes :

estNul(p)
Degre(p)
Coefficient(p, i**)**

Mise en pratique

A ce stade, on suppose que **toutes** les opérations du type sont implémentées.

Commencer par créer un fichier **preuvePolynome.c** incluant **polynome.c**

```
#include  "polynome.c"
```

Ecrire dans ce fichier une fonction **main().**

```
#include <stdio.h>
#include <stdlib.h>
#include "polynome.c"

int main(int argc, char *argv[])
{
POLYNOME   p, p1;
 Float a0 0;
…
 …

}
```

La fonction **main()** va permettre de **vérifier** que :

- les **constructeurs implémentées**
- **construisent correctement** les objets du type.

Ici, il y a deux constructeurs, à savoir :

Nul( ):Polynome

Construire(p1, a0**)** :Polynome

1-Vérifier l'implémentation de Nul()

Ecrire dans **main()** une instruction qui appelle la fonction **Nul ()** comme par exemple:

```
POLYNOME p
...
p = Nul()
```

Appeler successivement les accesseurs :

estNul( ) , **Degre**( ) , **Coefficient**( )

pour **vérifier** si **p** satisfait toutes les **propriétés** de **Nul()**, à savoir:

estNul(**p**) = True **Degre**(**p**) = 0 $\forall$ i • **Coefficient**(**p** ,i) = 0

1.1-Vérifier avec l'accesseur estNul

D'après la spécification on a la propriété :

estNul(**p**) = True

Exemple de vérification

```
/* Vérifier la propriété :
        estNul(p ) = True */

succes =1 ;
if ( !estNul(p)  )      succes = 0;
if (succes ==0 ) printf( "\n Implémentation de Nul () incorrecte")
```

1.2-Vérifier avec l'accesseur Degré()

D'après la spécification on a la propriété :

Degre(p) = 0

Exemple de vérification

```
/* Vérifier la propriété :
      Degre(p ) = 0
*/

if (Degre(p ) != 0)   succes = 0
if (succes ==0 ) printf( "\n Implémentation de Nul () incorrecte")
```

1.3-Vérifier avec l'accesseur Coefficient()

D'après la spécification on a la propriété :

$$\forall \; i \bullet \mathbf{Coefficient}(\mathbf{p}\;, i) = 0$$

Exemple de vérification

```
/*Vérifier la propriété :
 ∀  i • Coefficient(p ,i) = 0
*/
for (i=0; i< MaxSize; i++)
            if (Coefficient(p ,i) != 0   ) succes =0


if (succes ==0 ) printf( "\n Implémentation de Nul () incorrecte")
```

2-Vérifier l'implémentation de Construire()

1-Appeler dans **main()** la fonction **Construire** qui retourne la polynôme **p** :

```
....
POLYNOME  p , p1;
float a0 ;
......
/* on suppose que a0 est saisi et  p1 construit */
......
p = Construire( p1,a0)
```

Appeler successivement les accesseurs :

estNul( ) , **Degre**( ) , **Coefficient**( )

pour vérifier si **p** :

p = Construire(p1,a0)

satisfait **toutes les propriétés** énoncées en spécification.

2.1-Vérifier avec l'accesseur **estNul()**

D'après la spécification on a la propriété suivante :

$$\textbf{estNul}(\textbf{p1}) = \text{Vrai} \;\wedge\; \textbf{a0} = 0 \;\Leftrightarrow\; \textbf{estNul}(\textbf{p}) = \text{Vrai}$$

Exemple de vérification

```
/* vérification de la propriété :
                estNul(p) = True ⇔ estNul(p1) = True ∧ a0 = 0
*/
/* réinitialiser la variable succes */
succes=0 ;

if ( estNul(p))
                if ( estNul(p1)   &&  a0 == 0  )  succes=1;

if ( estNul(p1) && a0 == 0)
                              if (estNul(p) )   success = succes+1;

if (succes != 2 ) printf( “\n Implémentation de Construire(estNul) incorrecte”) ;
```

2.2-Vérifier avec l'accesseur **Degré()**

D'après la spécification on a les propriétés suivantes :

Degre(**Construire(** p1,a0**)**) = 0 $\Leftrightarrow$ **estNul**(p1) = True
Degre(**Construire(** p1,a0**)**) = **Degre**(p1)+1 $\Leftrightarrow$ **estNul**(p1) = False

Exemple de vérification

/* vérification de la propriété :

Degre(**p**) = 0 ⇔ **estNul**(**p1**) = True

*/

/* réinitialiser la variable « succès »

```
succes =0;

if ( Degre(p) ==0 )   if ( estNul(p1) )   succes =1;

if ( estNul(p1)   if ( Degre(p) ==0 )     succes =  succes + 1;
```

/* vérification de la propriété :

Degre(p) = Degre(p1)+1 ⇔ estNul(p1) = False */

```
if ( Degre(p) == Degre(p1) +1  )
                              if ( !estNul(p1))  succes =  succes + 1;
if ( !estNul(p1)
            if( Degre(p) == Degre(p1) +1 )  succes =  succes + 1;

if (succes !=4)
            printf( "\n Implémentation de Construire (Degré) incorrecte") ;
```

2.3-Vérifier avec l'accesseur Coefficient

D'après la spécification on a les propriétés suivantes :

Coefficient(**Construire(** p1,a0**)**,0) = a0

estNul(p1) = True $\Rightarrow$

i>0 $\Rightarrow$ **Coefficient**(**Construire(** p1,a0**)**, i) =0

estNul(p1) = False $\Rightarrow$

i$\geq$1 $\wedge$ i $\leq$**Degre**(p1)+1 $\Rightarrow$

Coefficient(**Construire(** p1,a0**)**, i) =**Coefficient**(p1,i-1)

estNul(p1) = False $\Rightarrow$

i> **Degre**(p1)+1 $\Rightarrow$ **Coefficient**(**Construire(** p1,a0**)**, i) = 0

Exemple de vérification de propriétés

```
/* vérification de la propriété :
                    Coefficient(p,0)  = a0
 */
```

```
/* réinitialiser la variable succès  */
succes=1;

if (coefficient(p, 0)  != a0) )  success = 0 ;
```

```
/* vérification de la propriété :
            estNul(p1) = True  ⇒   ( i> 0 ⇒ Coefficient(p, i) =0   )
*/

if ( estNul(p1))
        for(i=1; i< MaxSize;  i++)
                        if (  Coefficient(p,i)  !=0  ) success = 0 ;
```

/* vérification de la propriété :

estNul(p1) = False $\Rightarrow$ (i$\geq$1 $\wedge$ i $\leq$Degre(p1)+1 $\Rightarrow$ Coefficient(p, i) =Coefficient(p1,i-1))

*/

```
if (!estNul(p1 ) )
        for(i=1; i<= Degre(p1)+1 ; i++)
                if ( Coefficient(p, i) != Coefficient(p1,i-1)  ) success = 0 ;
```

```
/* vérification de la propriété :
estNul(p1) = False  ⇒
                    i> Degre(p1)+1 ⇒ Coefficient(Construire( p1,a0), i) = 0  */


if (!estNul(p1 ) )
         for(i= Degre(p1)+2;  i<= MaxSize ; i++)
                                   if ( Coefficient(p, i) != 0  ) success = 0 ;

if (succes ==0)
              printf( “\n Implémentation de Construire (Coefficient) incorrecte”) ;
```

Exemple de fichier preuvePolynome.c

```
#include <stdio.h>
#include <stdlib.h>
#include "polynome.c"

int main(int argc, char *argv[])
{
POLYNOME   p, p1;
 int i, succes;
float a0;
/* Allocation mémoire et vérification  */
p  = malloc( sizeof( struct un_polynome) );
p1 = malloc( sizeof( struct un_polynome) );
if( p == NULL || p1 == NULL   )
            {
            printf("Allocation impossible \n");
            exit(EXIT_FAILURE);
             };
```

```
/*  Vérifier l'implémentation du constructeur Nul() */
p = Nul() ;
/* Initialiser l'indice succes */
succes= 0;
/* Vérification avec l'accesseur estNul */
/* Vérifier la propriété :   estNul(p ) = True */
if ( !estNul(p) )     succes = 0;

/* Vérification avec l'accesseur Degre */
/* Vérifier la propriété :  Degre(p ) = 0   */
if (Degre(p ) != 0)   succes = 0

/* Vérification avec l'accesseur Coefficient */
/*Vérifier la propriété : ∀  i • Coefficient(p ,i) = 0    */
for (i=0; i< MaxSize; i++)
                        if (Coefficient(p ,i) != 0   ) succes =0

/* Bilan de la vérification */
if (succes =0 )
            {printf ("\n Implémentation incorrecte du constructeur Nul() ");
            printf("Interruption de la vérification: revoir l'implémentation du type abstrait \n");
            exit(EXIT_FAILURE);
             };
```

```
/*Vérifier l'implémentation du constructeur "Construire(p1,a0)" */

/* ne pas oublier de saisir le polynome p1  */
printf("\n saisir un polynome: p1 est le premier argument");

/* ne pas oublier de saisir le réel a0 */
printf("\n saisir un reel: a0 est le second argument");

/* Apres la saisie de p1 et a0,  appliquer "Construire" pour créer un polynome p  */
 p = Construire( p1,a0);
/* réinitialiser la variable succes  */
succes=0 ;

/* Vérification avec l'accesseur estNul */
/* vérification de la propriété :  estNul(p) = True ⇔  estNul(p1) = True  ∧ a0 = 0    */
if ( estNul(p))   if ( estNul(p1)   &&  a0 == 0  )  succes=1;

if ( estNul(p1) && a0 == 0)   if (estNul(p) )   success = succes+1;

/*Bilan de la vérification   */
if (succes != 2 )
                    { printf( "\n Implémentation de Construire(estNul) incorrecte") ;
                      printf("Interruption de la vérification: revoir l'implémentation du type abstrait \n");
                      exit(EXIT_FAILURE);
```

```
            };

/* Vérification avec l'accesseur Degre */
/* réinitialiser la variable « succès » */
succes =0;

/* vérification de la propriété :   Degre(p) = 0 ⇔   estNul(p1) = True    */
if ( Degre(p) ==0 )    if ( estNul(p1)  )   succes =1;
if ( estNul(p1)   if ( Degre(p) ==0 )      succes =  succes + 1;


/* vérification de la propriété :   Degre(p) = Degre(p1)+1 ⇔   estNul(p1) = False  */
if ( Degre(p) = Degre(p1) +1  )  if ( !estNul(p1) )        succes =  succes + 1;
if ( !estNul(p1))   if( Degre(p) == Degre(p1) +1 )        succes =  succes + 1;

/* Bilan de la vérification */
if (succes !=4)
        { printf( "\n Implémentation de Construire (Degré) incorrecte") ;
         printf("Interruption de la vérification: revoir l'implémentation du type abstrait \n");
         exit(EXIT_FAILURE);
         };
```

```
/* Vérification avec l'accesseur Coefficient */
/* réinitialiser la variable succès  */
succes=1;
/* vérification de la propriété : Coefficient(p,0)  = a0      */
if (coefficient(p, 0)  != a0) )  success = 0 ;

/* vérification de la propriété :   estNul(p1) = True  ⇒    ( i> 0 ⇒ Coefficient(p, i) =0    )  */
if ( estNul(p1))
            for(i=1; i< MaxSize;  i++)   if (  Coefficient(p,i)  !=0  ) success = 0 ;

/* vérification de la propriété :   estNul(p1) = False ⇒ (i≥1 ∧ i ≤Degre(p1)+1 ⇒ Coefficient(p, i) =Coefficient(p1,i-1))  */
if (!estNul(p1 ) )
          for(i=1; i<= Degre(p1)+1 ; i++)   if ( Coefficient(p, i) != Coefficient(p1,i-1)  ) success = 0 ;

/* vérification de la propriété :  estNul(p1) = False  ⇒   i> Degre(p1)+1 ⇒ Coefficient(Construire( p1,a0), i) = 0  */
 if (!estNul(p1 ) )
                   for(i= Degre(p1)+2;  i<= MaxSize ; i++)  if ( Coefficient(p, i) != 0  ) success = 0 ;

/* Bilan de la verification */
if (succes ==0)
             { printf( "\n Implémentation de Construire (Coefficient) incorrecte") ;
              printf("Interruption de la vérification: revoir l'implémentation du type abstrait \n");
              exit(EXIT_FAILURE);
              };
```

```c
printf("L'implementation du type abstrait est vérifiée");
printf("Fin normale de la vérification de l'implémentation du type abstrait");
return EXIT_SUCCESS;
}
```