Compléments de Programmation

Benoit Donnet

Année Académique 2023 - 2024

Agenda

Agenda

- Chapitre 5: Types Abstraits de Données
  - Principe
  - Spécification
  - Implémentation

Agenda

- Chapitre 5: Types Abstraits de Données
  - Principe
    - ✓ Généralités
    - ✓ Définition d'un TAD
  - Spécification
  - Implémentation

Généralités

- La conception d'un algorithme est indépendante de toute implémentation
- La représentation des données n'est pas fixée
  - celles-ci sont considérées de manière abstraite
- On s'intéresse à l'ensemble des opérations sur les données et aux propriétés des opérations
  - sans dire comment ces opérations sont réalisées
- Dit autrement
  - on sépare le "quoi" (la *spécification*) du "comment" (l'*implémentation*)

Généralités (2)

- Objectifs
  - clarté
    - élimination des détails de représentation
  - portabilité
    - on peut modifier la représentation sans modifier les programmes "clients"
  - définition de types généraux
    - extension du langage
  - réutilisabilité
    - dans de nombreux problèmes

Généralités (3)

- On parle de **Types Abstraits de Données** (TAD)
  - *abstract data types* (ADT)
  - B. Liskov, S. Zilles. *Programming with Abstract Data Types*. In ACM SIGPLAN Notices, 9(4), pg. 50-59. March 1974.

Définition d'un TAD

- Un TAD est
  - un ensemble de valeurs muni d'opérations sur ces valeurs
  - sans faire référence à une quelconque implémentation
- Exemples
  1. dans un algorithme qui manipule des `Integer`, on s'intéresse
     - ✓ non pas à la représentation des entiers
       - non signé, complément à 2, ...
     - ✓ mais bien aux opérations définies sur les entiers
       - +, -, ×, /
  2. le type `Boolean`
     - ✓ ensemble de deux valeurs (`True`, `False`)
     - ✓ muni des opérations `not`, `and`, `or`

Définition d'un TAD (2)

- Un TAD est caractérisé par sa **spécification**
- La spécification d'un TAD se présente en 2 parties
  - *signature*
    - ✓ définit la syntaxe du type et des opérations
  - *sémantique*
    - ✓ définit les propriétés des opérations
- Attention à ne pas confondre avec la spécification d'une fonction/procédure
  - aspect implémentation d'un TAD

Agenda

- Chapitre 5: Types Abstraits de Données
  - Principe
  - Spécification
    - ✓ Signature
    - ✓ Sémantique
    - ✓ TAD `Boolean`
    - ✓ TAD `Vector`
  - Implémentation

Signature

- La signature d'un TAD comporte
  - le nom du TAD
  - les noms des types des objets utilisés par le TAD
  - une liste d'opérations avec, pour chaque opération, l'énoncé des types des objets qu'elle reçoit et renvoie
    - ✓ formulation "fonctionnelle"
- **Sortes**
  - noms des ensembles de valeurs
  - correspond aux
    - ✓ nom du TAD
    - ✓ noms des types des objets utilisés par le TAD
- Une opération se présente comme suit

$$\text{Nom: } type_1\ arg_1 \times \ldots \times type_n\ arg_n \rightarrow type$$

Signature (2)

- La signature est décrite par 3 paragraphes
  - Type
  - Utilise
  - Opérations
- Forme générale d'une signature

```
Type:
  //Nom du type

Utilise:
  //Liste des objets utilisés

Opérations:
  //Liste des opérations (description fonctionnelle)
```

Signature (3)

- Typiquement, on aura 3 types d'opérations
  - **constructeur**
    - ✓ le TAD apparaît uniquement comme résultat
  - **observateur**
    - ✓ le TAD apparaît uniquement comme argument
  - **transformateur**
    - ✓ le TAD apparaît comme argument et comme résultat
- Il est possible de définir des **constantes**
  - opération sans argument

Signature (4)

- Il arrive aussi qu'on classifie les opérations comme suit
  - les **opérations internes**
    - ✓ constructeurs et transformateurs
  - les **opérations d'observations**
    - ✓ observateurs

Signature (5)

- La compréhension d'un TAD est basée sur l'intuition du lecteur
  - c'est la *sémantique* du type
- Comment définir la sémantique d'un type?
  - utilisation d'*axiomes*

Sémantique

- La sémantique d'un TAD précise
  - les domaines de définition (ou d'application) des opérations
  - les différentes propriétés des opérations
- **Type Abstrait Algébrique** (TAA)
  - la sémantique est définie par un système d'équations (*axiomes*)

Sémantique (2)

- La sémantique comporte 2 aspects
  - Préconditions
    - ✓ optionnelles
  - Axiomes
- Forme générale de la sémantique

```
Préconditions:
  //Domaine de définition

Axiomes:
  //Liste des axiomes
```


Sémantique (3)

- Les axiomes
  - permettent de donner la sémantique des opérations
  - correspondent aux propriétés des opérations
- Les axiomes ne sont pas définis partout
  - dépend de leurs domaines d'application
- Il peut donc être nécessaire d'ajouter des préconditions sur les variables
  - on parlera alors d'*opérations partielles*

Sémantique (4)

- Comment définir une précondition d'un TAD?
  - 1ère technique

Préconditions:
∀ *i*, `Cond(i)`, `nomOperation`($arg_1, arg_2, \ldots, i, \ldots, arg_n$)

- On souhaite limiter l'opération pour l'un de ses arguments
  - quantification universelle introduisant une variable liée
    - ✓ *i*
    - ✓ même type que l'argument à limiter
  - condition que doit satisfaire toutes les valeurs de l'argument à limiter pour que l'opération soit permise
    - ✓ *Cond(i)*

Sémantique (5)

- Comment définir une précondition d'un TAD? (cont.)
  - 2ème technique

Préconditions:
`nomOperation`($arg_1, arg_2, \ldots, i, \ldots, arg_n$) `is defined iff Cond(i)`

- Il suffit de déplacer la condition *Cond(i)*
- Quelle variante choisir?
  - préférence à la 1ère
    - ✓ plus explicite

Sémantique (6)

- Comment définir une précondition d'un TAD? (cont.)
  - la 1ère ligne des préconditions indique que tous les autres arguments peuvent prendre n'importe quelle valeur

Préconditions:
$\forall$ *arg*$_1 \in$ `Type arg`$_1$, $\forall$ *arg*$_2 \in$ `Type arg`$_2$, …, $\forall$ *arg*$_n \in$ `Type arg`$_n$

Sémantique (7)

- Comment rédiger les axiomes?
- Tous les axiomes prennent la forme suivante

Axiomes:
`Terme de gauche = Terme de droite`

- **Terme de gauche**
  - combinaison d'une
    - ✓ opération interne
      - produit une valeur de type TAD
    - ✓ avec un observateur
      - prend en argument une valeur de type TAD
  - exemple

Axiomes:
`observateur(op_interne(`*arg*$_1$, *arg*$_2$, …, *arg*$_n$`)) = Terme de droite`

Sémantique (8)

- Comment rédiger les axiomes? (cont.)
- Tous les axiomes prennent la forme suivante

```
Axiomes:
  Terme de gauche = Terme de droite
```

- **Terme de droite**
  - décrire le résultat du terme de gauche à l'aide de la récursivité!
    - ✓ cas de base
    - ✓ cas récursif(s)

Sémantique (9)

- Deux questions existentielles
  - y a-t-il des axiomes contradictoires?
    - ✓ *consistance*
  - y a-t-il suffisamment d'axiomes?
    - ✓ *complétude*
- Les axiomes doivent être consistants et complets

Sémantique (9)

- Comment assurer la consistance et la complétude?
- 2 techniques
  - s'assurer que le comportement du TAD est bien décrit
    - méthode intuitive
  - composer *tous* les observateurs avec *toutes* les opérations internes
    - tableau à double entrée
      - lignes = observateurs
      - colonnes = opérations internes
    - cocher une case du tableau quand tous les cas de la combinaison d'un observateur et opération interne

Sémantique (10)

- On peut parfois parvenir à réduire le nombre d'axiomes
  - une opération C s'exprime en fonction de 2 autres opérations A et B
  - exemple

Axiomes:

$C(args) = f(A(args), B(args))$

- f est la fonction qui combine le résultat de A et B
- Facilite la complétude et consistance

TAD `Boolean`

- Le TAD `Boolean`
  - 2 constantes
    - ✓ `True`
    - ✓ `False`
  - opérations de transformation
    - ✓ `not`
    - ✓ `and`
    - ✓ `or`

TAD `Boolean` (2)

- Définition

```
Type:
  Boolean
Utilise:
  ∅
Opérations:
  True: → Boolean
  False: → Boolean
  not: Boolean → Boolean
  and: Boolean × Boolean → Boolean
  or: Boolean × Boolean → Boolean
Préconditions:
  ∅
```

nom du TAD

n'utilise rien

nom de l'opération

2 arguments de type Boolean

type de retour

aucune précondition

TAD `Boolean` (3)

- Définition (cont.)

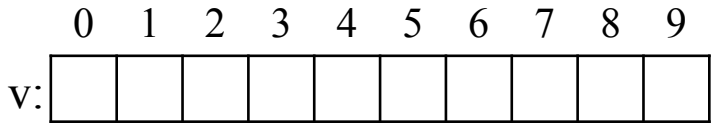
```
Axiomes:
  ∀ a,b ∈ Boolean
  not(True) = False
  not(not(a)) = a
  and(True, a) = a
  and(False, a) = False
  or(a, b) = not(and(not(a), not(b)))
```


TAD `Vector`

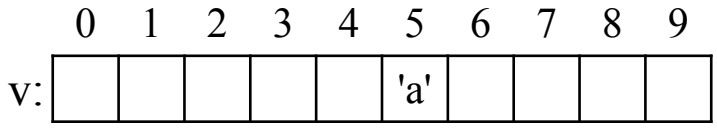
- Le TAD `Vector`
  - suite d'éléments
    - ✓ on peut accéder aux différents éléments de manière "aléatoire"
  - statique
    - ✓ taille donnée dès le début
  - opération de construction
    - ✓ `create`
  - opérations de transformation
    - ✓ `set`
  - opérations d'observation
    - ✓ `get`
    - ✓ `size`

TAD `Vector` (2)

- Opération de création
  - `create(10)` → `v`

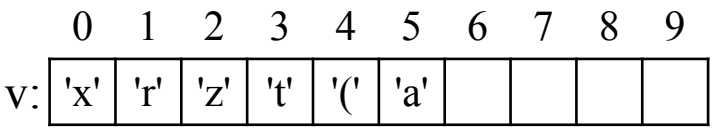


- Opération de transformation
  - `set(v, 5, 'a')` → `v`



TAD `Vector` (3)

- Opération d'observation
  - `get(v, 2)` → `'z'`
  - `size(v)` → `10`



TAD `Vector` (4)

- Définition

Type:
  Vector
Utilise:
  Integer, Element
Opérations:
  create: Integer → Vector
  set: Vector × Integer × Element → Vector
  get: Vector × Integer → Element
  size: Vector → Integer
Préconditions:
  ∀ i ∈ Integer, ∀ e ∈ Element, ∀ v ∈ Vector
  ∀ i ≥ 0, create(i)
  ∀ i, 0 ≤ i < size(v), get(v, i)
  ∀ i, 0 ≤ i < size(v), set(v, i, e)

constructeur

transformateur

observateur

opérations partielles

TAD `Vector` (5)

- Définition (cont.)

Axiomes:
  ∀ e ∈ Element, ∀ v ∈ Vector, ∀ i, j ∈ Integer :

		Opérations Internes	
		`create(·)`	`set(·)`
Observateurs	`get(·)`	∅	
	`size(·)`		

TAD Vector (6)

- Définition (cont.)

Axiomes:

$\forall e \in$ Element, $\forall v \in$ Vector, $\forall i, j \in$ Integer :

`size(create(i))` = i

`size(set(v, i, e))` = `size(v)`

		Opérations Internes	
		create(·)	set(·)
Observateurs	get(·)	∅	
	size(·)	✓	✓

TAD Vector (7)

- Définition (cont.)

Axiomes:

$\forall e \in$ Element, $\forall v \in$ Vector, $\forall i, j \in$ Integer :

`size(create(i))` = i

`size(set(v, i, e))` = `size(v)`

$$\text{get}(\text{set}(v,i,e),j) = \begin{cases} e & \text{If } i = j \\ \text{get}(v,j) & \text{Otherwise} \end{cases}$$

		Opérations Internes	
		create(·)	set(·)
Observateurs	get(·)	∅	✓
	size(·)	✓	✓

Agenda

- Chapitre 5: Types Abstraits de Données
  - Principe
  - Spécification
  - Implémentation
    - ✓ Structure de Données
    - ✓ Implémentation Générale
    - ✓ Implémentation en C
    - ✓ TAD `Boolean`
    - ✓ TAD `Vector`

Structure de Données

- L'implémentation d'un TAD correspond à une structure de données
  - *structure de données concrète*
- Chaque opération est associée à un algorithme
  - éventuellement des données spécifiques à la structure pour sa gestion
- Un même TAD peut donner lieu
  - à plusieurs structures de données
  - ayant des performances différences

Implem. Générale

- Pour implémenter un TAD
  - déclarer la structure de données retenues pour représenter le TAD
    - *interface*
  - définir les opérations primitives dans un langage particulier
    - *réalisation*
- Exigences
  - conforme à la spécification du TAD
  - efficacité
    - en terme de complexité

Implem. Générale (2)

- On utilise
  - les types élémentaires
  - les pointeurs
  - les tableaux et les enregistrements
- Rappel
  - plusieurs implémentations possibles pour un même TAD

Implémentation en C

- Utiliser la programmation modulaire
- Pour chaque TAD
  - fichier d'en-tête (.h)
    - ✓ `mon_tad.h`
    - ✓ contient l'interface du TAD
      - la structure de données est un *type opaque*
      - documentation informelle (i.e., Doxygen)
  - fichier module (.c)
    - ✓ `mon_tad.c`
    - ✓ contient l'implémentation des opérations
    - ✓ contient la définition de la structure de données concrète

Implémentation en C (2)

- Tout module ou programme ayant besoin du TAD
  - `#include "mon_tad.h"`
- Un module C implémente un TAD
  - *encapsulation*
    - ✓ détails d'implémentation cachés
      - cfr. INFO0030, Partie 1 (type opaque)
    - ✓ l'interface est la partie visible pour un utilisateur
  - *réutilisation*
    - ✓ placer les fichiers (`.h` et `.c`) dans le répertoire de l'application
    - ✓ créer une bibliothèque
      - cfr. INFO0030, Partie 2
  - *généricité*
    - ✓ cfr. INFO0030
    - ✓ cfr. INFO0062

TAD Boolean

- Interface (boolean.h)

```
#ifndef __BOOLEAN__
#define __BOOLEAN__

typedef enum{
  False,
  True
}Boolean;

Boolean and(Boolean x, Boolean y);

Boolean or(Boolean x, Boolean y);

Boolean not(Boolean x);

#endif
```


TAD Boolean (2)

- Module (boolean.c)

```
#include "boolean.h"

Boolean and(Boolean x, Boolean y){
  if(!x)
    return False;
  else
    return y;
}//end and()
```

TAD Boolean (3)

- Module (cont.)

```
Boolean or(Boolean x, Boolean y){
  if(x)
    return True;
  else
    return y;
}//end or()

Boolean not(Boolean x){
  return !x;
}//end not()
```


TAD Boolean (4)

- Utilisation du TAD Boolean
  - programme test_boolean.c

```
#include <stdio.h>
#include <stdlib.h>
#include "boolean.h"

int main(){
    Boolean x = True;
    Boolean y = False;

    printf("%d\n", not(x));
    printf("%d\n", and(x, y));
    printf("%d\n", or(x, y));

    return EXIT_SUCCESS;
}//end main()
```

TAD Vector

- Interface (vector.h)

```
#ifndef __VECTOR__
#define __VECTOR__

typedef struct Vector_t Vector;

Vector *create(int n);

Vector *set(Vector *v, int index, void *element);

void *get(Vector *v, int index);

int size(Vector *v);

#endif
```


TAD Vector (2)

- Module (vector.c)

```
#include <assert.h>
#include <stdlib.h>

#include "vector.h"

struct Vector_t{
    int size;
    void **array;
};
```

matrice car chaque indice doit contenir un pointeur vers void

TAD Vector (3)

- Module (cont.)

```
Vector *create(int n){
  assert(n>=0);

  Vector *v = malloc(sizeof(Vector));
  if(v==NULL)
    return NULL;

  v->size = n;
  v->array = malloc(n * sizeof(void *));
  if(v->array==NULL){
    free(v);
    return NULL;
  }

  return v;
}//end create()
```


TAD Vector (4)

```
Vector *set(Vector *v, int index, void *element){
  assert(v!=NULL && index>=0 && index<(size(v)));

  v->array[index] = element;

  return v;
}//end set()

void *get(Vector *v, int index){
  assert(v != NULL && index>=0 && index<size(v));

  return v->array[index];
}//end get()

int size(Vector *v){
  assert(v!=NULL);

  return v->size;
}//end size()
```

TAD `Vector` (5)

- Interface (`point.h`) des données à stocker dans un vecteur

```
#ifndef __POINT__
#define __POINT__

typedef struct Point_t Point;

Point *create_point(float x, float y);

void print_point(Point *pt);

float get_x(Point *pt);

float get_y(Point *pt);

#endif
```


TAD `Vector` (6)

- Utilisation du TAD `Vector`
  - programme `test_vector.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "vector.h"
#include "point.h"

int main(){
  Vector *v = create(10);
  printf("size(v): %d\n", size(v));

  Point *pt1 = create_point(5.4, -3.2);
  Point *pt2;
  set(v, 0, pt1);

  pt2 = get(v, 0);
  print_point(pt2);
  return EXIT_SUCCESS;
}//end main()
```