Programmation Avancée (INF441)

Types abstraits

François Pottier

12 avril 2016

Rappel : les projets

Quatre sujets de projet, disponibles en ligne dès maintenant.

Réfléchissez-y et travaillez-y **dès maintenant !**

Faites connaître votre choix avant le 20 avril 2016.

- par email à `francois.pottier@inria.fr`
- (depuis un email `@polytechnique.edu`)

Architecture modulaire

Un grand système logiciel ne peut être conçu que de manière modulaire.

Un système petit ou grand sera construit plus vite s'il est modulaire.

- Réutilisation de composants pré-existants ;
- Réalisation de différents composants par différents groupes.

Voir par exemple Parnas (1972).

Objectifs

Nos composants doivent être faciles à

- ré-utiliser dans de nouveaux scénarios,
- remplacer par d'autres composants équivalents,
- faire évoluer sans affecter le reste du système,
- tester ou prouver indépendamment les uns des autres.

Abstraction ($\exists$)

Cacher les détails concrets, pour exposer une description de haut niveau.

Cela concerne données et comportements.

- p. ex, une « date » est... un objet censé représenter une date
  - pas un triplet d'entiers,
  - ni un unique entier,
  - ni une chaîne de caractères `DDMMYY` (cf. Y2K problem !)...
  - on interdit d'inspecter directement son contenu.
- p. ex., l'opération « compareDates »... compare deux dates
  - on ne révèle pas comment elle le fait,
  - et on interdit de comparer deux dates par un autre moyen.

Cela s'appelle aussi encapsulation ou information hiding.

Il y a un lien avec la quantification existentielle (Mitchell et Plotkin, 1988).

Objectifs atteints ?

L'abstraction sert-elle les objectifs énoncés ?

Ré-utiliser un composant dans de nouveaux scénarios :

- plus on cache d'informations, moins le composant est utile... !
- mais si sa description est simple, il a plus de chances d'être (ré)utilisé

Remplacer ou faire évoluer un composant :

- il faut que le service rendu, décrit abstraitement, reste le même.
- pour cela, il faut restreindre ce que le client peut observer ou faire.

Tester ou prouver un composant isolément :

- peut-on en déduire qu'il fonctionnera, quels que soient les clients ?
- pour cela, il faut restreindre ce que les clients peuvent faire.

Quelques exemples suivent...

Ré-utiliser – Exemple

Quelle description me donnera envie d'utiliser ce composant ?

« Le composant *LH* propose des files de priorités, avec `empty`, `insert`, `extract`, `merge` ».

« Le composant *LH* propose des arbres binaires, où chaque nœud binaire porte un élément et son rang. Le rang d'un sous-arbre est la longueur de sa branche la plus à droite. Ces arbres sont organisés en tas. Le rang d'un fils gauche est toujours supérieur ou égal au rang de son frère. On fournit des fonctions `empty`, `insert`, `extract`, `merge` qui... »

La plus abstraite est la meilleure pour la plupart des utilisateurs.

Remplacer – Exemple

Supposons que je veuille plutôt essayer des « pairing heaps ».

La spécification est la même :

« Le composant *PH* propose des files de priorités, avec `empty`, `insert`, `extract`, `merge` ».

Mon application peut utiliser *LH* ou *PH* indifféremment (Mitchell, 1986).

Prouver – Exemple

J'implémente le composant *LH*, qui exploite les leftist heaps.

Je pense avoir deux invariants :

- mes arbres sont des tas ;
- le rang d'un fils gauche est supérieur ou égal au rang de son frère.

Pour prouver cela, il faut :

- prouver que les arbres que je construis (ou modifie) satisfont cela ;
- argumenter que je suis le seul à pouvoir les construire ou les modifier.

La protection joue un rôle fondamental (Morris, 1973).

Besoin de protection

Une abstraction a besoin d'être protégée.

La situation rappelle un peu celle d'un système d'exploitation, qui :

- offre une abstraction – il offre à chaque processus l'illusion d'être seul à disposer de la mémoire et du processeur ;
- doit interdire à chaque processus d'interférer avec le noyau ou avec d'autres processus ;
- utilise typiquement la MMU comme mécanisme de protection.

Pous nous, les types vont offrir la protection recherchée.

Les types sont descriptifs

Nous avons vu que les types décrivent les données.

Le type T de nos arbres de Huffman est décrit ainsi :

$$T = (Int \times Char) + (Int \times T \times T)$$

Types primitifs, types produits, types sommes, types récursifs.

Les types sont descriptifs

Ils décrivent non seulement les données, mais aussi les comportements.

Un compteur doté de méthodes `next` et `reset` est décrit ainsi :

$$T = (1 \to \mathit{Int}) \times (1 \to 1)$$

ou bien en Java :

```
public interface ICount {
  int next ();
  void reset ();
}
```

Types flèches (OCaml) et types interfaces (Java).

Je les appelle des types de service.

Sont-ils seulement descriptifs ?

En résumé, les types :

- décrivent les données,
  - c'est-à-dire la structure du tas ;
- décrivent les composants logiciels,
  - c'est-à-dire la façon dont ils sont prêts à dialoguer,
  - ou encore les services qu'ils proposent.

Est-ce tout ?

Les types sont prescriptifs

Non ! Les types constituent aussi un mécanisme de protection.

Les types nous empêchent de commettre certaines erreurs.

Ces erreurs sont détectées :

- pendant la compilation, si le langage est typé statiquement ;
- pendant l'exécution, si le langage est typé dynamiquement.

Typage statique

Par exemple, en Java :

```
public static int foo () {
  return 2 + true;
}
```

Ce code provoque une erreur lors de la compilation :

```
E.java:3: error: bad operand types for binary operator '+'
  return 2 + true;
           ^
first type:  int
second type: boolean
```

L'addition exige deux arguments de type « numérique ». (JLS)

Typage statique

Par exemple, en OCaml :

```
2 + true
```

Ce code provoque une erreur lors de la compilation :

```
Error: This expression has type bool
       but an expression was expected of type int
```

L'addition exige deux arguments de type `int`.

Typage dynamique

Par exemple, en Python :

```
def f():
  2 + '2'
```

Ce code provoque une erreur lors de l'exécution par l'interprète Python :

```
>>> f
<function f at 0x7fa7c6b606e0>
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

L'opération d'addition vérifie les étiquettes de ses arguments.

Deux mécanismes distincts

Deux mécanismes différents peuvent imposer la discipline des types :

- pendant la compilation, vérification des types ;
- pendant l'exécution, vérification des étiquettes.

On peut d'ailleurs combiner ces deux mécanismes. Java le fait (un peu).

Types et abstraction

Parce que les types offrent un mécanisme de protection, ils permettent de construire et faire respecter des abstractions (Reynolds, 1983).

Je distingue deux manières d'obtenir ce résultat :

- grâce aux types abstraits ;
  - exige le typage statique
  - séance 2, aujourd'hui
- grâce aux types de service.
  - objets, clôtures
  - typage statique ou dynamique
  - séance 3, la semaine prochaine

Abstraction procédurale (rappel)

Nous avons déjà parlé d'abstraction procédurale (Wheeler, 1952) :

- Le client connaît le nom des fonctions `sin`, `cos`, etc.
- Il peut les appeler, mais ne sait pas comment elles sont définies.
- Elles se comportent comme de nouvelles opérations primitives.

Abstraction de types

L'abstraction de types, apparue dans CLU (Liskov et Zilles, 1974), est analogue, mais s'applique à un type et non à une fonction.

Reprenons l'exemple des compteurs, évoqué la semaine dernière :

- Le client connaît le nom du type `Count` (Java) ou `count` (OCaml).
- Il peut en parler, mais ne sait pas comment il est défini.
- Le constructeur et les opérations `next` et `reset` sont donc les seuls moyens de construire, observer, manipuler des compteurs.
- Les compteurs se comportent comme de nouveaux objets primitifs.

Types abstraits en Java

Un seul fichier, par exemple `Count.java`.

```
public class Count implements ICount {
  // Fields. (Data.)
  // Invariant: step divides (next - init).
  private int next;
  private final int init, step;
  // Constructor.
  public Count (int init, int step) {
    this.next = this.init = init; this.step = step;
  }
  // Methods. (Code.)
  public int next() { int n = next; next += step; return n; }
  public void reset() { next = init; }
}
```

Le modificateur `private` interdit l'accès à certains champs ou méthodes.

Ainsi, le respect de l'invariant est garanti.

Types abstraits en OCaml

Deux fichiers, par exemple `Count.mli` et `Count.ml`.

La signature prend place dans le premier :

- déclarations ou définitions de types ;
- déclarations de valeurs.

L'implémentation se trouve dans le second. Les clients n'y ont pas accès.

Types abstraits en OCaml

La signature déclare un type abstrait et trois opérations :

```
type count
val make: int -> int -> count
val next: count -> int
val reset: count -> unit
```

Le client a accès à cette information et à ces opérations ; rien de plus.

Types abstraits en OCaml

L'implémentation définit le type et les opérations :

```
type count =
  { mutable next: int; init: int; step: int }
  (* Invariant: step divides (next - init). *)

let make init step =
  { next = init; init = init; step = step }

let next c =
  let n = c.next in
  c.next <- n + c.step;
  n

let reset c =
  c.next <- c.init
```

À nouveau, le respect de l'invariant est garanti, puisque le client ne peut ni modifier un objet de type `count` existant, ni en construire un faux.

Passons à un exemple plus ambitieux

Une abstraction simple :

- les files de priorité ;

implémentée à l'aide d'une structure de données particulière :

- les leftist heaps (Crane 1972 ; Knuth, 1973 ; Okasaki, 1999).

Il faut distinguer la fin et les moyens, abstraction et structure de données :

- une même abstraction peut être implémentée de différentes façons ;
- une même structure peut servir à réaliser différentes abstractions.

Interlude

Spécification des files de priorité
et implémentation à base de leftist heaps
en OCaml.
(Code en ligne)
Poly §2.1.

Spécification

Comment décrire le service rendu par les files de priorité ?

Il faut donner une spécification précise.

Sous quelle forme ?

Prédicat de représentation

On ne veut pas révéler ce qu'est une file de priorités (arbre, tableau, ...).

On parlera donc de ce qu'elle représente abstraitement.

Une file `q` représente un certain multi-ensemble d'éléments S.

- (multi-ensemble : un élément peut apparaître plusieurs fois)
- Pour simplifier, supposons les éléments entiers,
- et supposons qu'entier plus petit signifie priorité plus forte.

L'idée remonte à Hoare (1972). Voir aussi Liskov et Guttag, §5.5.

Spécification des opérations

On spécifie chaque opération à l'aide d'un prédicat « `q` représente S ».

Par exemple,

- La constante `empty` représente le multi-ensemble $\emptyset$.

Spécification des opérations

Ensuite,

- Si `q` représente le multi-ensemble S,
- alors `insert x q` représente le multi-ensemble $S \cup \{x\}$.

L'hypothèse est appelée précondition, la conclusion postcondition.

Ensemble, elles forment la spécification de la fonction `insert`.

(Je décris ici des files persistantes. `insert` ne modifie pas son argument.)

Spécification des opérations

De même,

- Si `q1` et `q2` représentent respectivement S_1 et S_2,
- alors `merge q1 q2` représente le multi-ensemble $S_1 \cup S_2$.

Spécification des opérations

Enfin,

- Si `q` représente le multi-ensemble S, alors :
- si S est non vide,
  alors `extract q` est de la forme `Some (x, q')`,
  où $x = \min S$ et `q'` représente $S \setminus \{x\}$;
- et si S est vide,
  alors `extract q` est `None`.

Le prédicat de représentation

Le prédicat de représentation est lui-même abstrait !

Le client ne sait pas comment il est défini.

Dans le code, on peut le définir (en commentaire).

Par exemple, si les files sont implémentées comme des arbres :

- « `q` représente S ssi le multi-ensemble des éléments contenus dans l'arbre `q` est S ».

Pourquoi une spécification ?

Aujourd'hui, la spécification n'est pas lue par le compilateur.

- en Java ou en OCaml, la spécification est un commentaire

À l'avenir, elle pourrait être vérifiée ; voir par exemple SPARK.

Il faut une spécification pour tester ou prouver un composant isolément.

La spécification est un contrat entre fournisseur et client du composant.

- elle est nécessaire pour la décomposition du travail

À retenir

Abstraire, c'est :

- cacher certains détails concrets ;
- exposer une spécification de haut niveau.

Il est nécessaire de :

- restreindre ce que peut faire le client
- pour protéger les invariants que le fournisseur veut maintenir
- et lui donner plus de liberté dans le choix des détails.

En TD aujourd'hui :

- types algébriques (= arbres) et types abstraits en OCaml.