Paradigmes de programmation

Cours 2 : Mutabilité, immuabilité, persistance

Benoît Montagu — benoit.montagu@inria.fr

Préparation à l'agrégation d'informatique — Automne 2022

Introduction

Mutabilité vs. immuabilité :

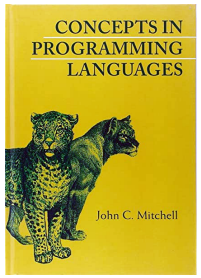
- Pour résumer en une ligne : RW vs. RO
- Notion centrale en programmation
- Peu mise en avant dans les langages impératifs ou objet
- Concept essentiel en programmation fonctionnelle
- Particulièrement important en programmation concurrente
- Mis en avant par certains langages de programmation modernes (Rust)

Persistance

- Une notion subtile
- Plusieurs formes de persistances
- Persistance ≠ immuabilité

Mutabilité, immuabilité

Transparence référentielle



> *[...] it is traditional to say that a language is referentially transparent if we may replace one expression with another of equal value anywhere in a program without changing the meaning of the program. This is a property of pure functional languages.*

Concepts in Programming Languages
John C. Mitchell
Cambridge University Press © 2003

Définition (informelle)

Une expression e est référentiellement transparente si, dans n'importe quel programme p, l'expression e peut être remplacée par n'importe quelle expression ayant la même valeur sans changer le comportement de p.

Transparence référentielle

Exemple (en C)

- `printf("Message")` n'est pas référentiellement transparente
  - ❓ Quel programme est un contre-exemple ?
- `3+4` est référentiellement transparente
- `x = 3+4` n'est pas référentiellement transparente
  - ❓ Quel programme est un contre-exemple ?
- `i++` n'est pas référentiellement transparente
  - ❓ Quel programme est un contre-exemple ?
- `i++; i--; i` est référentiellement transparente
  - ⚠ Pour des programmes séquentiels uniquement

Effets de bord

ℹ « Side effect » en anglais : devrait être traduit par « effets collatéral »

Définition

Un effet de bord est la modification d'un état global partagé (variable globale, cible d'un pointeur, mise à jour « en place », entrées/sorties, appel système...).

- Les effets de bord peuvent rendre la compréhension d'un programme difficile : une expression locale peut avoir un effet global
- Si une expression n'est pas référentiellement transparente, alors son évaluation produit nécessairement un effet de bord
- Un expression peut produire des effets de bord tout en restant référentiellement transparente. Par exemple : `i++; i--; i`
- L'immuabilité garantit l'absence d'effet de bord, et donc la transparence référentielle
- Une fonction qui n'effectue aucun effet de bord est appelée pure

Immuabilité et mutabilité

Définition

Une variable, une valeur, une structure de donnée est immuable si elle ne peut pas être modifiée. Dans le cas contraire, elle est mutable.

Exemple

Les constantes en C sont des variables immuables :

```
const int16_t INT32_MAX = 32767
```

	Python	OCaml
Variables	mutables	immuables
Tuples	immuables	immuables
Listes	mutables	immuables
Ensembles	mutables	immuables
Dictionnaires	mutables	`Map` : immuables/ `Hashtbl` : mutables
Tableaux[1]	mutables	mutables

1. En Python : tableaux extensibles = listes

⚠ Les macros sont fragiles

```c
#include <stdio.h>

#define N 2 + 0

int main() {
  printf("%i * 2 = %i\n", N, N * 2);
  return 0;
}
```

Qu'imprime le programme ci-dessus ?

Réponse :

```
2 * 2 = 2
```

Car : `N * 2 == 2 + 0 * 2 == 2 + (0 * 2) == 2`

⚠ L'expansion de macros est un simple remplacement textuel

ℹ Pour l'agrégation : n'utilisez pas `#define`, préférez les déclarations `const`

Tuples en Python

En Python, les tuples sont immuables :

```
>>> x = (1,2)
>>> y = (x, x)
>>> y
((1, 2), (1, 2))
>>> x[1] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Les tuples Python :

- Accès en temps constant
- Ne peuvent pas être « étendus » (on ne peut pas leur rajouter d'éléments)

❓ Comparaison avec les tuples en OCaml ?

Listes en Python

En Python, les listes sont mutables :

```
>>> x = [1,2]
>>> y = (x, x)
>>> y
([1, 2], [1, 2])
>>> x[1] = 4
>>> x
[1, 4]
>>> y
([1, 4], [1, 4])
```

Les « listes » Python sont en fait des tableaux extensibles (par la droite, avec la méthode **append**)

- Accès en temps constant
- Extension en temps constant *amorti*

❓ Comparaison avec les listes en OCaml ?

OCaml : variables et références

Exercice : évaluer ces deux programmes OCaml pas à pas

```
(* x vaut 42 :
   c'est définitif ! *)
let x = 42
let y = (x, x)

(* l'ancienne valeur de x est
   "masquée" par cette
   nouvelle définition *)
let x = 0

(* que vaut y ? *)
```

Code sans effet de bord

```
(* x est une référence (définitif !)
 qui "pointe" vers 42 (en ce moment) *)
let x = ref 42
let y = (x, x)

(* la valeur pointée par x
   est mise à jour *)
let () = x := 0

(* que vaut y ? *)
```

Code avec effet de bord

Variables, références et portée en OCaml

```
(* x vaudra toujours 42 *)
let x = 42

(* f retourne la valeur de x *)
let f () = x

(* l'ancienne valeur de x est
   "masquée" par cette nouvelle
   définition (shadowing) *)
let x = 0

(* Quelle est la valeur de y ? *)
let y = f ()
```

```
(* x est une référence vers 42 *)
let x = ref 42

(* f retourne la valeur pointée par x *)
let f () = !x

(* x est mis à jour: l'ancienne valeur
   pointée par x est écrasée *)
let () =
  x := 0

(* Quelle est la valeur de y ? *)
let y = f ()
```

En OCaml, la portée des variables est statique : les variables libres d'une fonction ont pour valeur celles qui sont calculées au point de définition de la fonction.

Variables et portée en Python ⚠

Le même exemple réécrit en Python :

```
# x vaut 42
>>> x = 42

# f retourne la valeur de x
>>> def f():
...     return x
...

# on change la valeur de x: que devient l'ancienne valeur ?
>>> x = 0

# que vaut y ?
>>> y = f()
```

En Python, la portée des variables est dynamique : les variables libres d'une fonction ont pour valeur celles qui sont calculées au point d'appel de la fonction.

Mutabilité en OCaml : listes mutables

Définition : listes mutables (un choix parmi d'autres)

```
type 'a cell = {
  mutable head: 'a;
  mutable tail: 'a mlist
}
and 'a mlist = 'a cell option
```

- Permet la mise à jour de la queue de la liste : `c.tail <- Some c`
- Et des éléments de la liste : `c.head <- 42`

Exercices :

- Quelles sont les signatures des fonctions `create`, `cons`, `get`, `set` ?
- Comment les implémenter ?
- Peut-on implémenter les références avec des champs mutables ? Comment ?

Interfaces pour structures de données : immuable vs. mutable

Exemple : interface pour des ensembles finis d'entiers

```
(* Ensembles immuables *)
type t

(* Test d'appartenance *)
val member: t -> int -> bool
(* Test d'inclusion *)
val subset: t -> t -> bool

(* L'ensemble vide *)
val empty: t
(* Ajout d'un élément *)
val add: t -> int -> t


(* Union de deux ensembles *)
val union: t -> t -> t

(* "copy" est gratuit *)
```

```
(* Ensembles mutables *)
type t

(* Test d'appartenance *)
val member: t -> int -> bool
(* Test d'inclusion *)
val subset: t -> t -> bool

(* Crée un nouvel ensemble vide *)
val create: unit -> t
(* [add s i] ajoute [i] à l'ensemble [s] *)
val add: t -> int -> unit
(* [union s1 s2] ajoute les éléments de [s2]
   à l'ensemble [s1].
   Précondition: [s1] et [s2] sont distincts. *)
val union: t -> t -> unit
(* [copy s] crée une nouvelle copie de [s] *)
val copy: t -> t
```


Et en C ? Variables et pointeurs

- Les variables sont mutables (sauf si déclarées `const`)
- Les pointeurs sont analogues aux références d'OCaml, avec en plus :
  - Pointeur `NULL`
  - Récupération de l'adresse d'une valeur : `&v`
  - Allocation dynamique de pointeur : `malloc(sizeof(type_t))`
  - Désallocation explicite avec `free(p)`
  - ⚠ On ne désalloue que les pointeurs créés par `malloc` !
  - Arithmétique de pointeurs
    - ℹ Hors programme MPI, mais au programme de l'agrégation ?
    - Un pointeur est une adresse, un emplacement en mémoire
    - Pour un pointeur `type_t *p` qui représente l'adresse $addr$, le pointeur `p+i` est le pointeur qui représente l'adresse (si elle est valide) : $addr + \texttt{sizeof(type\_t)} \times \texttt{i}$
    - Pour deux pointeurs `type_t *p1` et `type_t *p2` d'un même bloc, la différence de pointeurs `p2 - p1` est bien définie, et satisfait : `p1 + (p2 - p1) == p2`

Utiliser correctement les pointeurs est difficile ⚠

- Attention aux pointeurs nuls : `*p` n'est défini que si `p != NULL`
- En particulier, `malloc` peut retourner `NULL`
- Attention aux pointeurs vers des données locales

```
int* f() {
  int i = 0;
  return &i;
}
```

Le résultat retourné par `f` pointe vers une valeur indéfinie !
La variable `i` est locale à `f`, donc l'adresse `&i`
est invalide après que `f` a retourné.

- De manière similaire, après un appel à `free(p)`,
le déréférencement `*p` est illégal

Interlude : pointeurs vs tableaux en C

Tableaux et pointeurs sont des notions proches :
l'accès à un tableau `t[i]` est équivalent à `*(t+i)`

```
/* strcpy0: copy t to s;
   array subscript version */
void strcpy0(char *s, char *t) {
  int i = 0;
  while ((s[i] = t[i]) != '\0')
  {
    i++;
  }
}
```

ℹ Style recommandé pour l'agrégation

```
/* strcpy1: copy t to s; idiomatic C */
void strcpy1(char *s, char *t) {
  while ((*s++ = *t++))
    ;
}
```

⚠ Style recommandé par
The C Programming Language, K&R

Mutabilité vs. immuabilité : avantages et inconvénients

	Mutabilité	Immuabilité	Remarque
Facilité d'implémentation	☹	☺	Raisonnement délicat sur données mutables
Coût en temps	☺	☹	Mise à jour dans liste doublement chaînée vs liste fonctionnelle
Empreinte mémoire :			
- Réutilisation	☺	☹	Écraser des données est possible vs impossible
- Copies	☹	☺	Copie profonde vs superficielle
Sécurité	☹	☺	Immuabilité = garantie de non modification par client malveillant
Concurrence	☹	☺	Race conditions

Pour aller plus loin

Structures de données observationnellement immuables :
effets de bord « locaux », invisibles par un client

Exemples :

- Copy on write (cf cours OS)
- Splay trees : arbres pseudo-équilibrés qui se rééquilibrent à chaque accès
- Les strings de Python (partage de sous-parties communes)
- Hash-consing : partager un maximum de données
  Lisez le chapitre de Apprendre à programmer avec OCaml, Conchon & Filliâtre
  ❓ Besoin d'un volontaire pour présentation du hash-consing (dans 1 sem)
- Mémoïsation d'une fonction pure (voir planche suivante)

Un exemple de mutabilité non-observable

Mémoïsation

But : partager des calculs

C'est-à-dire : éviter de *re*calculer les résultats déjà calculés par une fonction

Comment : en enregistrant ce qui a déjà été calculé dans un « cache »

Une fonction de mémoïsation générique

```
(* val memo : ('a -> 'b) -> 'a -> 'b *)
let memo f =
  let h = Hashtbl.create 128 in
  fun x ->
    try Hashtbl.find h x
    with Not_found -> begin
      let y = f x in
      Hashtbl.add h x y;
      y
    end

(* une fonction qui coûte cher *)
(* val even : int -> bool *)
let even n =
  if n = min_int then true
  else let m = if n < 0 then -n else n in
    (m - 2 * (m / 2)) = 0
```

```
(* version mémoïsée de even *)
(* val even_memo : int -> bool *)
let even_memo = memo even

let b1 = even_memo 42
(* un appel à even *)

let b2 = even_memo 42
(* pas d'appel à even
   car la même table de
   hachage est utilisée *)
```


Mémoïsation : pour aller plus loin

- Comment mémoïser les appels récursifs d'une fonction récursive de manière générique ?
- Réponse dans quelques cours...

Persistance

Quelques précisions sur la notion de persistance

⚠ Persistance (français) vs Persistence (english)

C'est une notion plus subtile qu'il n'y paraît !

Plan :

Dinesh Mehta (2005). Handbook of Data Structures and Applications. Boca Raton, Fla : Chapman & Hall/CRC. ISBN : 9781584884354

- Chapitre 40 : Functional Data Structures, Chris Okasaki, US Military Academy
- Chapitre 31 : Persistent Data Structures, Haim Kaplan, Tel Aviv University

Sylvain Conchon et Jean-Christophe Filliâtre (2008). « Semi-Persistent Data Structures ». In : Programming Languages and Systems. Sous la dir. de Sophia Drossopoulou. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 322-336. ISBN : 978-3-540-78739-6

HDSA, chap. 40 : Functional Data Structures (Chris Okasaki)

> *A functional data structure is a data structure that is suitable for implementation in a functional programming language, or for coding in an ordinary language like C or Java using a functional style.* ***Functional data structures are closely related to persistent data structures and immutable data structures—in fact, the three terms are often used interchangeably. However, there are subtle differences.***
>
> ...

HDSA, chap. 40 : Functional Data Structures (Chris Okasaki) (suite)

...

> - *The term persistent data structures refers to the general class of data structures in which an update does not destroy the previous version of the data structure, but rather creates a new version that co-exists with the previous version. See Chapter 31 for more details about persistent data structures.*
> - *The term immutable data structures emphasizes a particular implementation technique for achieving persistence, in which memory devoted to a particular version of the data structure, once initialized, is never altered.*
> - *The term functional data structures emphasizes the language or coding style in which persistent data structures are implemented. Functional data structures are always immutable, except in a technical sense discussed in Section 40.4.*

HDSA, chap. 31 : Persistent Data Structures (Haim Kaplan)

> *Think of the initial configuration of a data structure as version zero, and of every subsequent update operation as generating a new version of the data structure. Then a data structure is called persistent if it supports access to all versions and it is called ephemeral otherwise. The data structure is partially persistent if all versions can be accessed but only the newest version can be modified. The structure is fully persistent if every version can be both accessed and modified. The data structure is confluently persistent if it is fully persistent and has an update operation which combines more than one version. Let the version graph be a directed graph where each node corresponds to a version and there is an edge from node V_1 to a node V_2 if and only if V_2 was created by an update operation to V_1. For partially persistent data structure the version graph is a path; for fully persistent data structure the version graph is a tree; and for confluently persistent data structure the version graph is a directed acyclic graph (DAG).* ▪▪▪

HDSA, chap. 31 : Persistent Data Structures (Haim Kaplan) (suite)

▪▪▪

> *A notion related to persistence is that of purely functional data structures. (See Chapter 40 by Okasaki in this handbook.) A purely functional data structure is a data structure that can be implemented without using an assignment operation at all (say using just the functions CAR, CDR, and CONS, of pure lisp). Such a data structure is automatically persistent.* ***The converse, however, is not true.*** *There are data structures which are persistent and perform assignments.*

Semi-Persistent Data Structures (Conchon & Filliâtre)

> *A data structure is said to be persistent when any update operation returns a new structure without altering the old version. This paper introduces a new notion of persistence, called semi-persistence, where* ***only ancestors of the most recent version can be accessed or updated****. Making a data structure semi-persistent may improve its time and space complexity. This is of particular interest in backtracking algorithms manipulating persistent data structures, where this property is usually satisfied. […]*

Persistance : rapide bilan

- ⚠ Différentes notions de persistance : notions subtiles !
- ▶ Une structure immuable est complètement persistante
- ▶ Une structure persistante n'est pas nécessairement immuable

❓ Okasaki disait que les structures de données fonctionnelles étaient toujours immuables, excepté en un sens technique. D'après vous, de quoi parlait-il ?

Conclusion

Conclusion

Bilan :

- Transparence référentielle
- Effets de bord
- Structures mutables et immuables
- Exemples de transparence référentielle avec effets de bord
- Persistance (notion tout juste effleurée)

La prochaine fois :

Tas vs. pile, appels de fonctions, appels terminaux, allocation statique/dynamique