Paradigmes de programmation

Cours 1 : Rappels sur les types de base

Benoît Montagu — benoit.montagu@inria.fr

Préparation à l'agrégation d'informatique — Automne 2022

Introduction

Chaque langage de programmation possède des types de base (entiers, booléens, flottants, chaînes de caractères…).

Mais ils peuvent être différents d'un langage à l'autre :

- Les choix d'implémentation sont différents
  - Représentation mémoire ?
  - Complexité asymptotique des opérations ?
- L'interface de programmation est différente aussi !
  - Données mutables ou immuables ?
  - Comportement en cas de dépassement de bornes ?

Plan du cours

Quelques rappels :

- Représentation en machine des entiers
- Entiers en C/OCaml/Python
- Booléens en C/OCaml/Python
- Rappels rapides sur les flottants IEEE-754
- Chaînes de caractères en C/OCaml/Python

Rappels sur les entiers

Représentation binaire des entiers

- Les entiers sont représentés sur n bits : $b_{n-1} \cdots b_1 b_0$ où $b_i \in \{0, 1\}$
- Sur les machines actuelles : $n \in \{32, 64\}$ généralement
- Vocabulaire : un octet (byte) est un entier sur 8 bits
- Interprétation non signée : $[\![\cdot]\!]_\mathrm{u} : \{0,1\}^n \to \mathbb{N}$

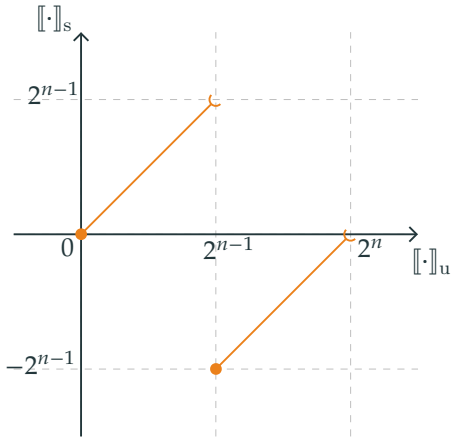
$$[\![b_{n-1} \cdots b_1 b_0]\!]_\mathrm{u} = \sum_{i=0}^{n-1} b_i \times 2^i$$

- Intervalle : $0 \leq [\![b_{n-1} \cdots b_1 b_0]\!]_\mathrm{u} < 2^n$ $\quad \mathrm{MIN}_\mathrm{u} = 0 \cdots 0$ $\quad \mathrm{MAX}_\mathrm{u} = 1 \cdots 1$
- Interprétation signée, en complément à 2 : $[\![\cdot]\!]_\mathrm{s} : \{0,1\}^n \to \mathbb{Z}$
  Le bit b_{n-1} indique le signe

$$\begin{aligned} [\![b_{n-1} \cdots b_1 b_0]\!]_\mathrm{s} &= -b_{n-1} \times 2^{n-1} + \textstyle\sum_{i=0}^{n-2} b_i \times 2^i \\ &= -b_{n-1} \times 2^n + [\![b_{n-1} \cdots b_1 b_0]\!]_\mathrm{u} \end{aligned}$$

- Intervalle : $-2^{n-1} \leq [\![b_{n-1} \cdots b_1 b_0]\!]_\mathrm{s} < 2^{n-1}$ $\quad \mathrm{MIN}_\mathrm{s} = 10 \cdots 0$ $\quad \mathrm{MAX}_\mathrm{s} = 01 \cdots 1$
  $[\![00 \cdots 0]\!]_\mathrm{s} = 0$ $\quad [\![11 \cdots 1]\!]_\mathrm{s} = -1$

Entiers binaires : non signés et signés



En complément à 2 :

- Les signés **positifs** ont la même représentation que les non signés
- Les signés **négatifs** sont représentés comme les « grands » non signés

Opérations arithmétiques

- Arithmétique non signée : les opérations se font modulo 2^n
  Exemple pour l'addition :

$$[\![a+b]\!]_\mathrm{u} = \begin{cases} [\![a]\!]_\mathrm{u} + [\![b]\!]_\mathrm{u} & \text{si } [\![a]\!]_\mathrm{u} + [\![b]\!]_\mathrm{u} < 2^n \\ [\![a]\!]_\mathrm{u} + [\![b]\!]_\mathrm{u} - 2^n & \text{sinon} \end{cases}$$

- Arithmétique signée : même implémentation que les opérations non signées !
  Exercice : démontrer que

$$[\![a+b]\!]_\mathrm{s} = \begin{cases} [\![a]\!]_\mathrm{s} + [\![b]\!]_\mathrm{s} & \text{si } -2^{n-1} \leq [\![a]\!]_\mathrm{s} + [\![b]\!]_\mathrm{s} < 2^{n-1} \\ [\![a]\!]_\mathrm{s} + [\![b]\!]_\mathrm{s} - 2^n & \text{si } 2^{n-1} \leq [\![a]\!]_\mathrm{s} + [\![b]\!]_\mathrm{s} \\ [\![a]\!]_\mathrm{s} + [\![b]\!]_\mathrm{s} + 2^n & \text{si } [\![a]\!]_\mathrm{s} + [\![b]\!]_\mathrm{s} < -2^{n-1} \end{cases}$$

- Identités remarquables :

$$\begin{aligned} 0_\mathrm{u} &= 0_\mathrm{s} & \mathrm{MAX}_\mathrm{u} &= -1_\mathrm{s} \\ \mathrm{MAX}_\mathrm{s} + 1 &= \mathrm{MIN}_\mathrm{s} & -\mathrm{MIN}_\mathrm{s} &= \mathrm{MIN}_\mathrm{s} \\ \mathrm{MAX}_\mathrm{s} \times \mathrm{MAX}_\mathrm{s} &= 1 & \mathrm{MIN}_\mathrm{s} \times \mathrm{MIN}_\mathrm{s} &= 0 \end{aligned}$$

Entiers binaires : pour aller plus loin

Chapitre 2.2 « Integer Representations »

Chapitre 2.3 « Integer Arithmetic »

Randal E. Bryant et David R. O'Hallaron (fév. 2018).
Computer Systems: A Programmer's Perspective, Global Edition.
Pearson. 1120 p. ISBN : 1292101768

- Se lit comme un roman !
- Représentation : endianness, ones' complement, ...
- Opérations : opérations logiques, xor, shifts (logique/arithmétique), ...
- Comment tester l'absence d'overflow
- Faites les exercices !

Les entiers en C : types

Type C	Intervalle typique	Intervalle minimum (standard C)
`char`	$[-2^7, 2^7-1]$	$[-(2^7-1), 2^7-1]$
`unsigned char`	$[0, 2^8-1]$	$[0, 2^8-1]$
`short`	$[-2^{15}, 2^{15}-1]$	$[-(2^{15}-1), 2^{15}-1]$
`unsigned short`	$[0, 2^{16}-1]$	$[0, 2^{16}-1]$
`int`	$[-2^{31}, 2^{31}-1]$	$[-(2^{15}-1), 2^{15}-1]$
`unsigned int`	$[0, 2^{32}-1]$	$[0, 2^{16}-1]$
`long`	$[-2^{31}, 2^{31}-1]$	$[-(2^{31}-1), 2^{31}-1]$
`unsigned long`	$[0, 2^{32}-1]$	$[0, 2^{32}-1]$
`int32_t`	$[-2^{31}, 2^{31}-1]$	$[-2^{31}, 2^{31}-1]$
`uint32_t`	$[0, 2^{32}-1]$	$[0, 2^{32}-1]$
`int64_t`	$[-2^{63}, 2^{63}-1]$	$[-2^{63}, 2^{63}-1]$
`uint64_t`	$[0, 2^{64}-1]$	$[0, 2^{64}-1]$

$2^{31} \approx 2,14 \times 10^9$ $2^{32} \approx 4,29 \times 10^9$ $2^{63} \approx 9,22 \times 10^{18}$ $2^{64} \approx 18,4 \times 10^{18}$

Les entiers en C : subtilités (1)

Entiers non signés :

- Dépassements de capacité : **définis par le standard C**
- Opérations arithmétiques sont définies sur $\mathbb{Z}/2^n\mathbb{Z}$
- **Cast** vers un type entier non signé : représentant dans le $\mathbb{Z}/2^n\mathbb{Z}$ cible
- Exemple :

```c
#include <stdint.h>
int64_t x1 = (((int64_t) 1) << 32) + 1;
// x1 = 4294967297      (= 2^32 + 1)
uint32_t y1 = (uint32_t) x1;
// y1 = 1               (= x1 - 2^32)
// c'est la troncation de x1 depuis 64 bits vers 32 bits

int64_t x2 = -x1;
// x2 = -4294967297     (= -2^32 - 1)
uint32_t y2 = (uint32_t) x2;
// y2 = 4294967295      (= 2^32 - 1)
```


Les entiers en C : subtilités (2)

Entiers signés :

- Dépassements de capacité arithmétique : **comportement indéfini**
  - Exemples : $-\text{MIN}_s$ $\text{MAX}_s + 1$
- Cast vers entier signé, sans dépassement de capacité : défini comme étant l'identité
- Cast vers entier signé, avec dépassement de capacité : **comportement indéfini**
  - Exemple : cast de 2^{31} depuis `uint32_t` vers `int32_t`
  - Exemple : cast de $-2^{31}-1$ depuis `int64_t` vers `int32_t`
- Division par zéro : exception matérielle !
- Division sur opérandes négatives : `a / b` est la **troncation vers 0** de $\frac{a}{b}$
  - Exemple : `(-5) / 3 = -1` et `5 / (-3) = -1`
- Reste sur opérandes négatives (C99) : défini tel que `a = (a / b) * b + (a % b)`
  - Exemple : `(-5) % 3 = -2` mais `5 % (-3) = 2`

ℹ Pour l'agrégation : pas de division/reste avec des négatifs !

Les entiers en OCaml

Que des entiers signés :

- `int` : entiers signés 31 bits / 63 bits
  ℹ 1 bit est réservé pour utilisation par le runtime (GC)
- `int32` : entiers signés 32 bits
- `int64` : entiers signés 64 bits
- `nativeint` : entiers signés 32 bits / 64 bits

Opérations arithmétiques :

- Définies comme les opérations en complément à 2
- Division/reste : comme en C99 (troncation vers 0 + identité d'Euclide)
- Division par zéro : `exception Division_by_zero`

Les entiers en Python

Entiers en précision arbitraire :

- Nous avons accès à tout $\mathbb{Z}$
- Pas de dépassement de capacité
- Capacité limitée par la mémoire disponible
- Pas d'entier maximal, pas d'entier minimal
- ℹ En OCaml : entiers en précision arbitraire avec `Zarith.t`

Division/reste :

- Division par zéro : `raise ZeroDivisionError()`
- « floor division » : `a // b` est la troncation vers $-\infty$ de $\frac{a}{b}$
  👉 Comportement différent de C ou OCaml sur les entiers négatifs !
- ⚠ Attention : l'expression `a / b` renvoie un flottant !

```
x = 3 // 2     # x = 1
y = 3 / 2      # y = 1.5
```

- Reste : `a % b` défini via `a = (a // b) * b + (a % b)`

Interlude : recherche dichotomique (C)

```c
/* Assumes [n >= 0] and [t] is an array of size [n] and is sorted.
   Returns an index [i] such that [0 <= i < n] and [t[i] == v],
   or returns [-1] if [t] does not contain [v]. */
int bsearch(int *t, int n, int v) {
  int l = 0; // lower bound of the searched range
  int u = n-1; // upper bound of the searched range
  while (l <= u) {
    int i = (l + u) / 2;      // middle point ⚠ possible overflow!
    if (v < t[i]) { // search after [i]
      u = i-1;
    } else if (v > t[i]) { // search before [i]
      l = i+1;
    } else { // we have found [v] at index [i]
      return i;
    }
  }
  return -1; // [v] was not found
}
```


Interlude : recherche dichotomique (C)

```c
/* Assumes [n >= 0] and [t] is an array of size [n] and is sorted.
   Returns an index [i] such that [0 <= i < n] and [t[i] == v],
   or returns [-1] if [t] does not contain [v]. */
int bsearch(int *t, int n, int v) {
  int l = 0; // lower bound of the searched range
  int u = n-1; // upper bound of the searched range
  while (l <= u) {
    int i = l + (u - l) / 2; // middle point 👍 overflow impossible
    if (v < t[i]) { // search after [i]
      u = i-1;
    } else if (v > t[i]) { // search before [i]
      l = i+1;
    } else { // we have found [v] at index [i]
      return i;
    }
  }
  return -1; // [v] was not found
}
```


Interlude : quelques notes historiques sur la recherche dichotomique

- Code publié pour la 1re fois en 1946 (John Mauchly)
- Codes incorrects publiés dans de nombreux livres
- En 1988, seulement 5 sur 20 textbooks contenaient une version correcte de l'algorithme (source)

> *Although the basic idea of binary search is comparatively straightforward, the details can be somewhat tricky, and many good programmers have done it wrong the first few times they tried.*
>
> Donald E. Knuth,
> The Art of Computer Programming, Vol. 3, p. 407

- Le bug d'overflow est resté présent dans la bibliothèque Java pendant 9 ans, corrigé seulement en 2006 (source)
- Qu'en est-il dans votre langage de programmation préféré ?

Les booléens

Booléens en C

« En C, un entier est un booléen comme un autre. »

- `0` est le booléen « faux »
- Tout entier différent de `0` est *un* booléen « vrai »

Il y a donc strictement **plus que deux valeurs booléennes**…

```
if (42) { x = 128; } else { x = -1; } // on a x = 128
```

Depuis C99 :

- `#include <stdbool.h>`
- Déclare le type `_Bool` et les constantes `true` (`== 1`) et `false` (`== 0`)
- Cast d'un type entier vers `_Bool` :
  `(_Bool) 0 == false` `(_Bool) n == true` si `n != 0`
- ⚠ `(true + true) != false` et `(true + true) != true`
  Le calcul n'est pas dans $\mathbb{Z}/2\mathbb{Z}$: l'addition promeut les booléens en entiers !
- ℹ Pour l'agrégation : autant que possible ne mélangez pas entiers et booléens

Interlude : ⚠ opérateurs logiques en C

Nom	Exemple	Remarque
Test d'égalité	`e1 == e2`	Renvoie un booléen
Affectation	`x = e`	Affecte `x` avec la valeur de `e`, et renvoie la valeur de `e`
« Ou » logique	`e1 | e2`	Opération bit à bit Les expressions `e1` et `e2` sont toujours évaluées
Expression conditionnelle	`e1 || e2`	Opérateur de contrôle, équivalent à `(e1)?true:(e2)` L'expression `e1` est toujours évaluée, mais `e2` ne l'est que si `e1` s'évalue à `false`
« Et » logique	`e1 & e2`	Opération bit à bit Les expressions `e1` et `e2` sont toujours évaluées
Expression conditionnelle	`e1 && e2`	Opérateur de contrôle, équivalent à `(e1)?(e2):false` L'expression `e1` est toujours évaluée, mais `e2` ne l'est que si `e1` s'évalue à `true`

Booléens en OCaml

- Il n'y a que **deux booléens** en OCaml : `true` et `false`
- Leur représentation est cachée : ce sont des types abstraits
- Les booléens peuvent être manipulés par :

```
if b                match b with            b1 || b2
then ...            | true -> ...           b1 && b2
else ...            | false -> ...          not b
```

- Les booléens peuvent être produits par :

```
e1 = e2          e1 <> e2    (* égalité structurelle   *)
e1 == e2         e1 != e2    (* /!\ égalité d'adresses *)
e1 < e2          e1 <= e2          e1 > e2          e1 >= e2
```

- ℹ En pratique, le compilateur OCaml *fait le choix* de représenter `false` par `0`, et de représenter `true` par `1` (choix compatible avec C)
- ❓ Exercice : écrire une fonction OCaml qui teste si deux valeurs on la même représentation (sans utiliser le module `Obj`). L'utiliser sur les booléens.

Solution de l'exercice : testeur de représentation en OCaml

```
(* Type existentiel ∃α.α implémenté à l'aide d'un GADT
   (type de données algébrique généralisé) *)
type any = Ex: 'a -> any

(* Testeur de représentation, utilisant l'égalité polymorphe *)
let same_representation a b =
  (Ex a) = (Ex b)
(* val same_representation : 'a -> 'b -> bool *)

same_representation 0 false (* résultat : true *)

same_representation 1 true (* résultat : true *)
```


Booléens en Python

- Deux constantes booléennes : `True` et `False`
- Opérateurs : `e1 and e2`, `e1 or e2`, `e1 == e2`, `e1 != e2`
  Attention : `e1 is e2` teste l'égalité d'adresses (est-ce le même objet ?)
- Les booléens sont des objets de la `<class 'bool'>`
- De manière similaire à C : on confond entiers et booléens

```
>>> if 0:
...     print("YES")
... else:
...     print("NO")
...
NO
>>> False + True
1
```

```
>>> if 42:
...     print("YES")
... else:
...     print("NO")
...
YES
>>> True + True
2
```

⚠ Mais aussi : la liste/ensemble/map/chaîne vide sont équivalents à `False`, et à `False` dans le cas non vide. Exemples : `bool([])`, `bool((0,0))`

Booléens en Python : règles de conversion

D'après la documentation :
(source : `https://docs.python.org/3/library/stdtypes.html#truth`)

By default, an object is considered true unless its class defines either a `__bool__()` method that returns `False` or a `__len__()` method that returns zero, when called with the object. Here are most of the built-in objects considered false :

- constants defined to be false : `None` and `False`.
- zero of any numeric type : `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- empty sequences and collections : `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

ℹ La conversion vers le type booléen est donc programmable, et peut être redéfinie par héritage.

Les flottants

Nombres flottants

- Nombres binaires fractionnaires (pour $x \in [0, 1)$)
  À maîtriser : l'exemple de 0.1
- Norme IEEE-754
  - s : bit de signe
  - m : mantisse
  - e : exposant
  - Valeur : $(-1)^s \times m \times 2^e$
  - Normalisés, dénormalisés, $+0$, -0, $+\infty$, $-\infty$, NaN
  - ⚠ Égalité sur les flottants : en pratique, tester l'égalité « à ε près »
- Simple précision, double précision (en C : `float`/ `double`)
- À lire : Computer Systems : A Programmer's Perspective
  Chapitre 2, Section 4 « Floating Point »
- ℹ Agrégation : les détails de la norme IEEE sont hors programme
  Mais : connaître les idées principales est essentiel !

Les chaînes de caractères

Chaînes de caractères en C : « string »

- C'est une suite de caractères situés les uns à la suite des autres...
  ... et terminée par le caractère nul `'\0'`
- Les strings en C sont mutables : elles peuvent être **modifiées en place**
- ⚠ L'accès en dehors des bornes est indéfini ! (bugs de sécurité)
- Implémenté par le type `char *` (pointeur vers un caractère)
- En C : identique à un tableau de caractères avec un `'\0'` de plus à la fin

```
1  char *s = "abcde";     // char s[] = { 'a','b','c','d','e','\0' };
2  char zeroth = s[0];    // zeroth = 'a', les indices commencent à 0
3  char zeroth = *s;      // idem: s pointe vers le 1er élément
4  char ith = s[i];       // accès au i-ème caractère
5  char ith = *(s+i);     // idem, par arithmétique de pointeurs
6  char last = s[5];      // last = '\0'
7  char unknown = s[6];   // /!\ accès hors des bornes
8  char unknown = s[-1];  // idem (comportement indéfini)
9  s[3] = 'D';            // s est maintenant la chaîne "abcDe"
```

❓ Le code de gauche produit une erreur de segmentation à la ligne 9. Pourquoi ?

Exemple : calcul de la longueur d'un chaîne en C

```
1  #include <stdio.h>
2  // Retourne le nombre de caractères de la chaîne s
3  unsigned int str_len(char *s) {
4    unsigned int n = 0;
5    for (char *tmp = s; (*tmp) != '\0'; tmp++) {
6      n++;
7    }
8    return n;
9  }
10 int main() {
11   char *s = "abcde";
12   printf("La chaîne %s contient %u caractères.\n", s, str_len(s));
13   return 0;
14 }
```

```
$ gcc -std=c99 -Wpedantic -Wall -Wextra string_test.c -o string_test
$ ./string_test
La chaîne abcde contient 5 caractères.
```

Chaînes de caractères en OCaml

- `type string` L'implémentation est cachée : c'est un type abstrait
- Les chaînes sont immuables en OCaml : elle ne peuvent pas être modifiées
- Littéraux : comme en C. Exemple : `"OCaml\tis fun.\n"`
- Module `String` (extraits) :

```
val init : int -> (int -> char) -> string
val length : string -> int
val get : string -> int -> char (* s.[i] identique à get s i *)
(* get lance l'exception Invalid_argument si accès hors bornes *)
(* Il n'y a pas de fonction "set": les strings sont immuables *)
val mapi : (int -> char -> char) -> string -> string
```

```
1 open String
2 let s = init 5 (fun i -> Char.chr (i + Char.code 'a'))
3 (* s  = "abcde" *)
4 let bad = try s.[6] with Invalid_argument _ -> '!'
5 (* bad = '!' *)
6 let s2 = mapi (fun i ci -> if i = 3 then 'D' else ci) s
7 (* s = "abcde" et s2 = "abcDe" *)
```


Chaînes de caractères en Python (1)

- Une chaîne de caractères est un object de la classe `str`
- Les chaînes sont immuables
- Les élements constituant une string Python sont des caractères UTF8 (qui peuvent tenir sur plus d'1 octet)
- Exemple : le caractère € est encodé en UTF8 sur 3 octets
- ⚠ Un « caractère » est une chaîne de taille 1

```
>>> s = 'abcde'
>>> (s[0], s[4]) # accès valide à un indice
('a', 'e')
>>> s[5] # accès invalide
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> prix = '2€99'
>>> prix[1]
'€'
```


Chaînes de caractères en Python (2)

- Indices négatifs : accès « par la fin »

```
>>> (s[-1], s[-5]) # accès valide "par la fin"
('e', 'a')
>>> s[-6] # accès invalide "par la fin"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

- Extraire des sous-chaînes à l'aide de « *slices* »

```
>>> s[2:4] # la sous-chaîne entre les indices 2 et 4 (exclu)
'cd'
>>> s[2:] # la sous-chaîne démarrant à l'indice 2
'cde'
>>> s[:4] # la sous-chaîne s'arrêtant à l'indice 4 (exclu)
'abcd'
```

- Exercice : coder l'équivalent de la fonction OCaml `String.mapi` en Python

Solution de l'exercice : écrire `mapi` en Python

```
def mapi(f, s):
    i = 0
    t = ""
    for c in s:
        t = t + f(i,c)
        i = i + 1
    return t

def f(i, c):
    if i == 3:
        return 'D'
    else:
        return c

s = "abcde"
t = mapi(f, s)
# s = "abcde" et t = "abcDe"
```

```
# Version alternative, plus "pythonesque"
def mapi(f, s):
    t = ""
    for i, c in enumerate(s):
        t = t + f(i,c)
    return t
```

Conclusion

Conclusion

- Des types de base parfois très différents suivant les langages
- Des spécifications différentes
- Des implémentations non présentées à l'utilisateur
- Premiers exemples de types abstraits
- Premier contact avec mutabilité/immuabilité
- C'est le sujet du cours suivant