PG104 - Programmation Fonctionnelle

D. Renault

Enseirb-Matmeca

24 février 2025, v.1.2.5

https://www.labri.fr/perso/renault/working/teaching/functional/functional.php

Techniques de programmation

Dans ce chapitre, nous allons présenter des techniques “avancées” liées à la programmation fonctionnelle :

- la programmation typée avec des fonctions ;
- la programmation dirigée par les fonctions ;
- les techniques de contrôle de l'évaluation.

Ces techniques ne font pas partie des fondamentaux de la programmation fonctionnelle, mais illustrent des manières supplémentaires de l'utiliser.

Programmation typée

Quelles interactions entre prog. typée et prog. fonctionnelle ?

Principes de la programmation typée

- Classer les expressions apparaissant dans un programme selon des **types**,
- Vérifier que le composition de ces **types** au sein du programme vérifie un ensemble de règles de cohérence.

Contre-exemple illustratif

```
locomotive + flower
```

- Ecmascript est un langage à typage **dynamique** : les types existent, mais ne sont vérifiés qu'à l'exécution.

Typescript

- Typescript est un langage développé par Microsoft et se présentant comme un sur-ensemble d'Ecmascript ajoutant une couche de **typage**.
- Le compilateur `tsc` transforme un fichier `.ts` en `.js`, tout en réalisant un ensemble de vérifications **statiques** de ces règles de cohérence.
- Les indications de types y sont optionnelles et peuvent être ajoutées de manière **graduelle**.

Typescript

- Typescript est un langage développé par Microsoft et se présentant comme un sur-ensemble d'Ecmascript ajoutant une couche de **typage**.
- Le compilateur `tsc` transforme un fichier `.ts` en `.js`, tout en réalisant un ensemble de vérifications **statiques** de ces règles de cohérence.
- Les indications de types y sont optionnelles et peuvent être ajoutées de manière **graduelle**.

```
function reverse(s) {                                   // Javascript version
  return s.split('').reverse().join('');
}
reverse('hello world'); // → 'dlrow olleh'
reverse(456);           // → ???
```

```
TypeError: s.split is not a function                    // Execution with node
```

Typescript

- Typescript est un langage développé par Microsoft et se présentant comme un sur-ensemble d'Ecmascript ajoutant une couche de **typage**.
- Le compilateur `tsc` transforme un fichier `.ts` en `.js`, tout en réalisant un ensemble de vérifications **statiques** de ces règles de cohérence.
- Les indications de types y sont optionnelles et peuvent être ajoutées de manière **graduelle**.

```
function reverse(s : any) : any {                    // Typescript 'any' version
  return s.split('').reverse().join('');
}
reverse('hello world'); // → 'dlrow olleh'
reverse(456);           // → ???
```

```
TypeError: s.split is not a function                 // Execution with node
```

Typescript

- Typescript est un langage développé par Microsoft et se présentant comme un sur-ensemble d'Ecmascript ajoutant une couche de **typage**.
- Le compilateur `tsc` transforme un fichier `.ts` en `.js`, tout en réalisant un ensemble de vérifications **statiques** de ces règles de cohérence.
- Les indications de types y sont optionnelles et peuvent être ajoutées de manière **graduelle**.

```
function reverse(s : string) : string {          // Typescript 'typed' version
  return s.split('').reverse().join('');
}
reverse('hello world'); // → 'dlrow olleh'
reverse(456);           // → ???
```

```
error TS: type 'number' not assignable to type 'string'.    // Compilation
```

Les types en Typescript

Pour référence, les types des valeurs de base en Typescript sont :

- les valeurs non initialisées
  et les valeurs nulles, vides ou inexistantes ; `undefined` `null`
- les valeurs booléennes ; `boolean`
- les chaînes de caractères immutables `string`
  et les types des clés des dictionnaires ; `symbol`
- les nombres flottants `number`
  et les grands nombres ; `bigint`
- et les **objets** :
  - les dictionnaires clé-valeurs ; `{ age: number; name: string }`
  - les tableaux ; `number[]` / `string[]`
  - les fonctions. `(param: string) ⇒ number`

Fonctions et polymorphisme

- Programmer avec des fonctions, en particulier des paramètres fonctionnels a tendance à produire du code **générique**.
- En terme de types, cette généricité est une forme de **polymorphisme** : le code peut s'appliquer à des valeurs de types différents.
- Concrètement, il s'agit de polymorphisme paramétrique, se traduisant par la présence de **variables de types**.

```
function countMatches   (coll,
                         matchFun                    ) {
    let res = 0;
    for (let i = 0; i < coll.length; i++) {
        if (matchFun(coll[i]))
          res += 1;
    }
    return res;                                          }
```

Fonctions et polymorphisme

- Programmer avec des fonctions, en particulier des paramètres fonctionnels a tendance à produire du code **générique**.
- En terme de types, cette généricité est une forme de **polymorphisme** : le code peut s'appliquer à des valeurs de types différents.
- Concrètement, il s'agit de polymorphisme paramétrique, se traduisant par la présence de **variables de types**.

```
function countMatches<T>(coll : ArrayLike<T>,
                         matchFun : (T) ⇒ boolean) : number {
    let res : number = 0;
    for (let i : number = 0; i < coll.length; i++) {
        if (matchFun(coll[i]))
          res += 1;
    }
    return res;                                          }
```

Fonctions et polymorphisme

- Programmer avec des fonctions, en particulier des paramètres fonctionnels a tendance à produire du code **générique**.
- En terme de types, cette généricité est une forme de **polymorphisme** : le code peut s'appliquer à des valeurs de types différents.
- Concrètement, il s'agit de polymorphisme paramétrique, se traduisant par la présence de **variables de types**.

```
function countMatches<T>(coll : ArrayLike<T>,
                         matchFun : (T) ⇒ boolean) : number {
    let res : number = 0;
    for (let i : number = 0; i < coll.length; i++) {
        if (matchFun(coll[i]))
          res += 1;
    }
    return res;                                          }
```

`T` est une variable de type, et peut être
- un caractère (si `coll : string`)
- un nombre (si `coll : number[]`)

Faire de la programmation fonctionnelle dans un cadre typé implique forcément de rencontrer des types paramétriques :

- En Haskell : Lien

```haskell
map :: (a → b) → [a] → [b]
zip :: [a] → [b] → [(a, b)]
```

- En Java (ici sur les `Stream`) : Lien

```java
public interface Stream<T>
void        forEach(Consumer<? super T> action)
Stream<R>   map(Function<? super T, ? extends R> mapper)
```

- Voire même en Ecmascript, par ex. avec la bibliothèque `Ramda` : Lien

```haskell
map :: Functor f ⇒ (a → b) → f a → f b
zip :: [a] → [b] → [[a,b]]
```

Exemple de typage : types sommes

Dans cet exemple en Typescript, le type `Shape` est un **type somme**.
Les valeurs de ce type peuvent prendre 4 formes distinctes.

```
interface Circle    {  kind: "circle";     radius: number;  };
interface Square    {  kind: "square";     sideLength: number;  };
interface Rectangle {  kind: "rectangle";  h: number;  w: number; };
interface Triangle  {  kind: "triangle";   sides: number[]; };

// Shape is either one of Circle, Square, Rectangle or Triangle (discriminated union)
type Shape = Circle | Square | Rectangle | Triangle;
```

Il est possible de traiter les valeurs de type `Shape` au cas par cas :

```
function getArea(shape: Shape) {
    switch (shape.kind) {
        case "circle":     return Math.PI * shape.radius ** 2;
        case "square":     return shape.sideLength ** 2;
        case "rectangle":  return shape.h * shape.w;
  } }
```

Exemple de typage : types sommes

Dans cet exemple en Typescript, le type `Shape` est un **type somme**.
Les valeurs de ce type peuvent prendre 4 formes distinctes.

```
interface Circle    {  kind: "circle";     radius: number;  };
interface Square    {  kind: "square";     sideLength: number;  };
interface Rectangle {  kind: "rectangle";  h: number;  w: number; };
interface Triangle  {  kind: "triangle";   sides: number[]; };

// Shape is either one of Circle, Square, Rectangle or Triangle (discriminated union)
type Shape = Circle | Square | Rectangle | Triangle;
```

Il est aussi possible de vérifier ici si les fonctions sont **exhaustives** :

```
function getArea(shape: Shape) {
    switch (shape.kind) {
        case "circle":     return Math.PI * shape.radius ** 2;
        case "square":     return shape.sideLength ** 2;
        default:           const unused : never = shape;  // Never reached ?
  } }
```

Exemple de typage : types sommes

Dans cet exemple en Typescript, le type `Shape` est un **type somme**.
Les valeurs de ce type peuvent prendre 4 formes distinctes.

```
interface Circle    {  kind: "circle";     radius: number;  };
interface Square    {  kind: "square";     sideLength: number;  };
interface Rectangle {  kind: "rectangle";  h: number;  w: number; };
interface Triangle  {  kind: "triangle";   sides: number[]; };

// Shape is either one of Circle, Square, Rectangle or Triangle (discriminated union)
type Shape = Circle | Square | Rectangle | Triangle;
```

Le compilateur `tsc` vérifie alors et propose les cas manquants :

```
error TS: Type 'Rectangle␣|␣Triangle' is not assignable to type 'never'.
```

Avantages et inconvénients de la prog. typée

Typer ses programmes aide à rendre leur comportement explicite.

- Des outils de **vérification** (analyse statique) peuvent être utilisés pour assurer des propriétés de **sûreté** sans exécuter le code.
- Les types servent aussi de **documentation** (minimale) pour les programmeurs, en le spécifiant de manière systématique et succinte.

Ils peuvent être aussi vus comme des freins au développement :

- La programmation sans les types est certes fragile, mais permet d'écrire du code très **rapidement** (e.g prototypage).
- Dans certains cas, forcer à typer un programme peut contraindre, voire **empêcher** d'écrire un code correct.

Programmation dirigée par les fonctions

Deux styles de programmation centrés autour des ensembles de fonctions :

Définition (Programmation dirigée par les données)

La **programmmation dirigée par les données** (*data-driven programmming*) est un style de programmation construit sur des familles de fonctions que l'on compose pour organiser dynamiquement le flot de contrôle.

Définition (Programmation événementielle)

La **programmation événementielle** (*event-driven programming*) est un style de programmation manipulant le flot de contrôle autour des événements.

- Tous deux considèrent les traitements comme des données à part entière.
- Les fonctions, profitant de la 1ère classe, s'insèrent naturellement dans de tels styles de programmation.

Exemple : compteur (1/2)

Considérons l'exemple d'une page permettant de manipuler un compteur :

```
import { h, text, app } from "https://unpkg.com/hyperapp";

const initialCpt = { count: 0 };                                  // Initial state

const actionsCpt = {                                              // Possible actions
    down: (state, event) ⇒ ({ ...state, count: state.count - 1 }),
    up:   (state, event) ⇒ ({ ...state, count: state.count + 1 }),
};

const viewCpt = (state) ⇒                                         // Page view
    h("main", {}, [
        h("h1", {}, text(‘Counter␣:␣${state.count}‘)),
        h("button", { onclick: actionsCpt.up },   text("+1")),
        h("button", { onclick: actionsCpt.down }, text("-1")),
    ]);

app({ init: initialCpt,                                           // Tying everything together
      view: viewCpt,
      node: document.getElementById("app") });
```

(exemple utilisant la bibliothèque hyperapp et adapté depuis cette page)

Exemple : compteur (2/2)

Il s'agit d'un exemple de programmation **événementielle** :

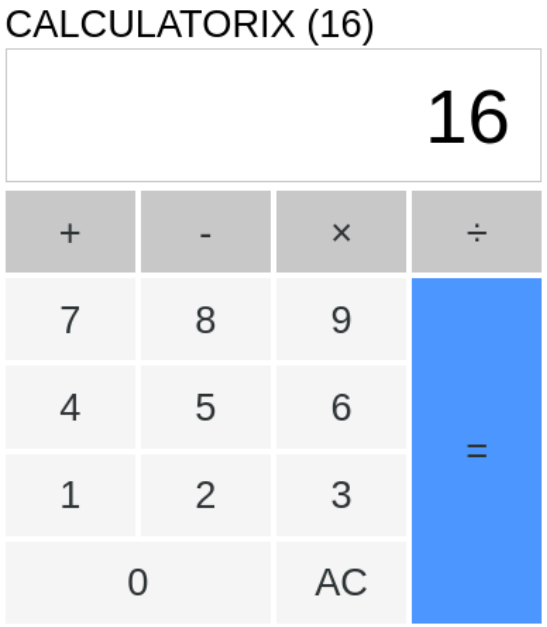
- Le modèle de la page est décrit sous la forme d'un **état** global.
- Les traitements sont organisés dans une liste `actionsCpt` :
  - Chaque traitement est rattaché à un événement, et opère une **transformation** de l'état.
  - Étendre ou contraindre les actions possibles peut se faire dynamiquement en modifiant `actionsCpt`.
- La vue est **composée** (par une fonction) à partir de l'état global et associe les événements aux actions.

Exemple : application web

Hyperapp avec un exemple d'application un peu plus complexe (Lien) :

```
const computer = {
  "+": (a, b) ⇒ a + b,
  "-": (a, b) ⇒ a - b,
  "×": (a, b) ⇒ a * b,
  "÷": (a, b) ⇒ a / b
}

const initialState = {
  fn: "",
  carry: 0,
  value: 0,
  hasCarry: false
}
```



Exemple : interpréteur BASIC (1/3)

Considérons l'exemple d'un langage de programmation comme le BASIC :

```
// A program in BASIC
10 LET I = 1

20 LET S = 1

30 LET I = I + 1



40 LET S = S * I



50 IF (I = 5) THEN GOTO 70



60 GOTO 30

70 END
```

```
const prog = [  // The equivalent in Ecmascript as an array of lines
    { num: 10,
      instr: { kind: "instrLet", id: "i",
               expr: { kind: "exprInt", int: 1}} },
    { num: 20,
      instr: { kind: "instrLet", id: "s",
               expr: { kind: "exprInt", int: 1}} },
    { num: 30,
      instr: { kind: "instrLet", id: "i",
               expr: { kind: "exprBin", op: { kind: "opPlus"},
                       lhs: { kind: "exprVar", var: "i"},
                       rhs: { kind: "exprInt", int: 1} }} },
    { num: 40,
      instr: { kind: "instrLet", id: "s",
               expr: { kind: "exprBin", op: { kind: "opMult"},
                       lhs: { kind: "exprVar", var: "i"},
                       rhs: { kind: "exprVar", var: "s"} }} },
    { num: 50,
      instr: { kind: "instrJumpIf",
               cond: { kind: "exprBin", op: { kind: "opEqual" },
                       lhs: { kind: "exprVar", var: "i"},
                       rhs:  {kind: "exprInt", int: 5}},
               num: 70} },
    { num: 60,
      instr: { kind: "instrGoto", num: 30} },
    { num: 70,
      instr: { kind: "instrEnd" } }                              ]
```

Exemple : interpréteur BASIC (2/3)

Comment faire pour interpréter un tel programme ?

- Partir d'un état gérant la ligne courante et l'environnement :

```
const initialState = { env: { },     // The variables in the environment
                       num: 10   }; // The current line number
```

- Gérer une instruction `goto` :

```
const instrGoto = (state) ⇒ (line) ⇒ { env: state.env, num: line.num };
```

- Gérer une instruction `let` :

```
const instrLet = (state) ⇒ (line) ⇒ {
            env:  { ...state.env,  // Update the environment
                    line.instr.id: evalExpr(line.instr.expr, state.env) },
            num: state.num + 10 };
```

Exemple : interpréteur BASIC (3/3)

- Il s'agit ici d'un exemple de programmation **dirigée par les données**.
- Les actions du langage peuvent être stockées dans un dictionnaire :

```
const actions = {
  instrGoto:   (state) ⇒ (line) ⇒ { ... },
  instrLet:    (state) ⇒ (line) ⇒ { ... },
  instrJumpIf: (state) ⇒ (line) ⇒ { ... },
};
```

- Les données ici sont le programme à interpréter sous la forme d'une liste d'instructions, et la liste des actions appliquables pour chaque instruction.

Possibilités

- Lister les actions disponibles, en ajouter ou en enlever dynamiquement.
- Tester le comportement d'une seule instruction facilement.

Intérêts de la prog. dirigée par les données

- Penser les programmes comme des **ensembles de comportements** avec peu de dépendances, appliquables et composables à la demande.
- De tels programmmes permettent d'**agir dynamiquement** sur ces comportements : les lister, en ajouter ou en enlever ...
- Hélas, si la liste des comportements accessibles peut varier dynamiquement, la vérification du code est rendue plus difficile.

Contrôle de l'évaluation

Définition (Contrôle de l'évaluation)

Le **contrôle de l'évaluation** (*evaluation control*) est une technique de programmation consistant à maîtriser le flot de contrôle en le stoppant et le reprenant en fonction des besoins du programme.

Permet d'envisager un nombre considérable de techniques :

- Le **déboguage** de programmes profite naturellement de pouvoir arrêter et reprendre le cours d'une exécution.
- De nombreux problèmes pour lesquels la **terminaison** n'est pas évidente peuvent aussi bénéficier du contrôle de leur exécution :
  - les explorations de grands domaines (ex. : arbres de jeu),
  - les algorithmes non-garantis de terminer (ex. : calcul de limite),
  - les programmes dépendant de facteurs externes (ex. : fetch url) ...

Congélation, décongélation

Les fonctions se prêtent bien au jeu du contrôle de l'évaluation :

Définition (Congélation)

Congeler (*to freeze*) une expression `e` consiste à construire une fonction renvoyant `e` lorsqu'elle est appliquée.

Exemple

L'expression `fact(10**10)` est un exemple de calcul extrêmement long.
La fonction `() ⇒ fact(10**10)` représente sa version congelée.

Définition (Décongélation)

Décongeler (*to thaw*) une expression congelée consiste à appliquer la fonction pour récupérer la valeur enfermée à l'intérieur.

Exemples d'expressions congélées

- Une expression congelée peut être décongelée au moment voulu :

```
let longComputation = () ⇒ fact(10**10);
// Do something else more interesting
let result = longComputation();   // → a big number
```

- Il est possible de congeler n'importe quelle expression :

```
let printingComputation = () ⇒ { console.log('Now'); };
printingComputation(); // Log : Now

let failingComputation = () ⇒ { throw new Error("Yikes"); };
failingComputation();   // Uncaught Error : Yikes
```

- Une valeur congelée peut même ne jamais être décongelée si jamais l'utilité ne se présente pas.

Exemple de congélation récursive

Considérons le programme calculant la fonction factorielle :

```
function fact(n, r) {
  console.log(`fact(${n}, ${r})`);
  if (n <= 1)
    return r;
  else
    return fact(n-1, n*r);
}
```

```
fact(5, 1);
// Log : fact(5, 1)
// Log : fact(4, 5)
// Log : fact(3, 20)
// Log : fact(2, 60)
// Log : fact(1, 120)
// → 120
```

Exemple de congélation récursive

Considérons le programme calculant la fonction factorielle :

```
function frozenFact(n, r) {
  console.log(`fact(${n}, ${r})`);
  if (n <= 1)
    return r;
  else
    return () ⇒ frozenFact(n-1, n*r);
}
```

Exemple de congélation récursive

Considérons le programme calculant la fonction factorielle :

```
function frozenFact(n, r) {
  console.log(`fact(${n}, ${r})`);
  if (n <= 1)
    return r;
  else
    return () ⇒ frozenFact(n-1, n*r);
}
```

```
let s1 = frozenFact(5, 1);
// Log : fact(5, 1), → Function
let s2 = s1();
// Log : fact(4, 5), → Function
let s3 = s2();
// Log : fact(3, 20), → Function
let s4 = s3();
// Log : fact(2, 60), → Function
let s5 = s4();
// Log : fact(1, 120), → 120
```

- Permet d'obtenir une fonction réalisant ses calculs **pas à pas**.
- L'écriture sous forme récursive terminale est fondamentale ici.

Application : les trampolines

Un **trampoline** est une fonction enlevant toutes les couches de congélation d'une valeur congelée.

```
function trampoline(v) {
    let res = v;
    while (res instanceof Function) { res = res(); }
    return res;
}
```

- Les trampolines peuvent être utilisés pour évaluer les fonctions dans le tas plutôt que dans la pile.
- Ils permettent ainsi d'écrire des fonctions récursives débarrassées des limites de taille de la pile (**stack overflow**).

```
trampoline(frozenFact(11000n, 1n)) // → some big number
```

Stratégies d'évaluation

Contrôler l'évaluation permet diverses stratégies . . . d'évaluation.

```
let square     = (x) ⇒ x*x;
let sumSquares = (a, b) ⇒ square(a) + square(b);
```

```
sum_squares(3, 5 - 1)   →   ???
```

On distingue :

- l'évaluation par valeur ;
- l'évaluation par nom ;
- l'évaluation paresseuse.

Stratégies d'évaluation

Contrôler l'évaluation permet diverses stratégies . . . d'évaluation.

Définition (Évaluation par valeur)

L'**évaluation par valeur** (*call-by-value*) est une technique d'évaluation d'une expression $f(x_1, \ldots, x_n)$ consistant à évaluer d'abord récursivement les paramètres x_i avant de les transmettre à la fonction f.

```
let square     = (x) ⇒ x*x;
let sumSquares = (a, b) ⇒ square(a) + square(b);
```

```
sumSquares(3, 5 - 1)   →   sum_squares(3, 4)       // eval params
                       →   square(3) + square(4)   // eval sumSquares
                       →   3*3 + 4*4               // eval params of '+'
                       →   9 + 16                  // ...
                       →   25                      // eval '+'
```

Stratégies d'évaluation

Contrôler l'évaluation permet diverses stratégies . . . d'évaluation.

Définition (Évaluation par nom)

L'**évaluation par nom** (*call-by-name*) est une technique d'évaluation d'une expression $f(x_1, \ldots, x_n)$ consistant à substituer les x_i sans les évaluer dans le corps de la fonction f avant d'évaluer f.

```
let square     = (x) ⇒ x*x;
let sumSquares = (a, b) ⇒ square(a) + square(b);
```

```
sumSquares(3, 5 - 1)   →   square(3) + square(5-1)  // subs sumSquares
                       →   3*3 + (5-1)*(5-1)        // subs square (!)
                       →   3*3 + 4*4                // eval '-'
                       →   9 + 16                   // eval '*' twice
                       →   25
```

Stratégies d'évaluation

Contrôler l'évaluation permet diverses stratégies . . . d'évaluation.

Définition (Évaluation paresseuse)

L'**évaluation paresseuse** (*lazy evaluation* ou *call-by-need*) est une technique d'évaluation améliorant l'évaluation par nom en mémorisant les calculs réalisés afin de ne pas les dupliquer.

```
let square    = (x) ⇒ x*x;
let sumSquares = (a, b) ⇒ square(a) + square(b);
```

```
sumSquares(3, 5 - 1)   →   square(3) + square(5-1)   // eval sumSquares
                       →   3*3 + x*x                 // eval x → 5 − 1 → 4
                       →   9 + 4*4
                       →   9 + 16
                       →   25
```

Et si on était tout le temps paresseux ?

- Rappel : si une expression est **pure**, le résultat de son évaluation ne dépend pas du moment où elle est évaluée.
- La programmation paresseuse ne peut s'appliquer facilement que dans les calculs **sans** effets de bords.
  Dans les autres cas, il faut faire attention à l'ordre des calculs.
- Quand elle s'applique, elle permet de ne réaliser les calculs qu'au moment où ils servent, et même de les éviter s'ils ne servent pas.

Haskell applique une stratégie d'évaluation paresseuse par défaut.

```
isPrime n = length [ d | d ← [1..(n `div` 2)], n `rem` d == 0 ] == 1
primes = filter isPrime $ [1..] -- filter `isPrime` on the list of all integers
primes                          -- → [2,3,5,7,11,13,17,19,23...], never stops
```

```haskell
head []     = undefined -- or some kind of error
head (x:xs) = x

insert x []     = [x]
insert x (y:ys) = if (x<y) then (x:y:ys) else (y:insert x ys)

insert_sort []     = []
insert_sort [x]    = [x]
insert_sort (x:xs) = insert x (insert_sort xs)

min xs = head (insert_sort xs)    -- Compute the minimum element of a list
```

```haskell
min [3,2,1]
    ⟶ head (insert_sort [3,2,1])
    ⟶ head (insert 3 (insert_sort [2,1]))
    ⟶ head (insert 3 (insert 2 (insert_sort [1])))
    ⟶ head (insert 3 (insert 2 [1]))
    ⟶ head (insert 3 (if (2<1) then (2:...) else (1:...))) -- '...' stand for
    ⟶ head (insert 3 (1:...))                               -- unevaluated
    ⟶ head (if (3<1) then (3:...) else (1:...))             -- computations
    ⟶ head (1:...)
    ⟶ 1
```

Structures paresseuses (1/3)

Les valeurs congelées peuvent être stockées dans des structures de données.

Exemple

Une liste peut voir sa tête et sa queue congelées.
Les décongeler permet de déplier la liste au fur et à mesure des calculs.

Structures paresseuses (2/3)

Dans l'exemple suivant, la liste est congelée à partir d'un certain point :

```
function frozenCoins() {
    const aFlip = (Math.random() > 0.5) ? 'heads' : 'tails';
    return cons(aFlip, frozenCoins);          // the tail of the list is frozen
}

function flipACoin(c) {
    let pointer = c;
    while (typeof pointer.cdr === 'object')   // go to the end of the list
        pointer = pointer.cdr;
    pointer.cdr = pointer.cdr();              // replace its tail
}
```

```
let c = frozenCoins(); // { 'tails', [Function: frozenCoins] }
flipACoin(c);          // { 'tails', 'heads', [Function: frozenCoins] }
flipACoin(c);          // { 'tails', 'heads', 'heads', [Function: frozenCoins] }
flipACoin(c);          // { 'tails', 'heads', 'heads', 'tails', [Function: frozenCoins] }
```

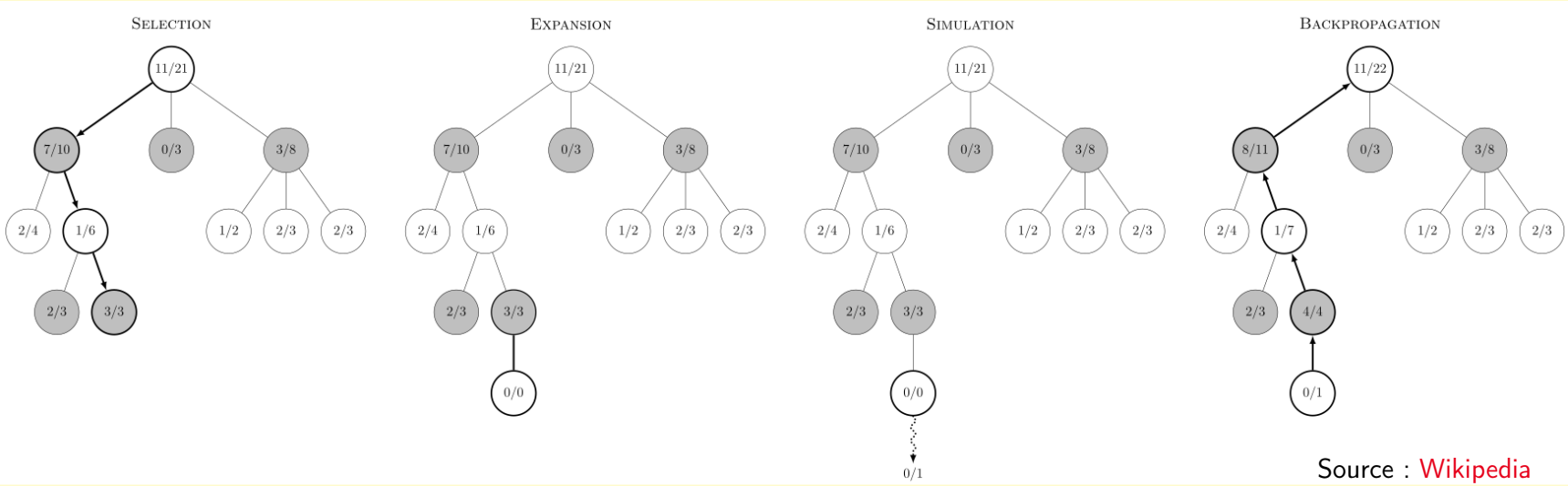
Structures paresseuses (3/3)

Les structures paresseuses permettent d'envisager :

- de construire des structures de taille quelconque, voire infinie ;
- de construire des structures qui se développent selon un algorithme.

Exemple : Monte-Carlo tree search

Exploration de l'arbre des parties d'un jeu pour évaluer les coups optimaux :



Source : Wikipedia

Promesses (1/3)

Les calculs congelés ont le bon goût de pouvoir se composer sans s'évaluer :

- Supposons disposer d'une valeur congelée : `let fznInt = () ⇒ fact(10);`
- Supposons vouloir la transformer par `Math.sqrt` sans la décongeler.

Pour cela, il suffit simplement de construire la valeur congelée suivante :

`() ⇒ Math.sqrt(fznInt())`

En généralisant, on obtient une fonction agissant sur les valeurs congelées :

```
function fznSquare(fznInt) {                    // frozen value → frozen value
    return () ⇒ Math.sqrt(fznInt());
}
```

```
let fznInt = () ⇒ fact(10);      // → [Function] (frozen value)
let result = fznSquare(fznInt);  // → [Function] (frozen value)
result();                        // → 1904.94
```

Promesses (2/3)

Définition (Promesse)

Une **promesse** (*promise*) Ecmascript est la réalisation d'un calcul asynchrone.

On la construit à partir d'une fonction à deux paramètres fonctionnels :

- `resolve`, qui est appelée sur la valeur renvoyée par la promesse,
- `reject`, qui est appelée sur la valeur d'erreur en cas d'erreur.

```
const promise = new Promise(function(resolve, reject) {
    setTimeout(function() {
        resolve('hello');
    }, 10000);
 })                              // Promise returning 'hello' after 10 seconds
```

- A la différence d'une valeur congelée, une promesse commence à être évaluée dès sa définition

Promesses (3/3)

Comme les promesses sont des calculs asynchrones, il est là aussi intéressant de pouvoir les composer :

- la méthode `then` permet de spécifier que faire à la fin du calcul :

```
const promiseCpx = promise
      .then((value) ⇒ { return value.toUpperCase(); })
      .then((value) ⇒ { return `${value} - ${value}` });
const result = await promiseCpx; // → "HELLO - HELLO"
```

- la méthode `catch` permet de spécifier que faire en cas d'erreur.

```
promise.catch((err) ⇒ { console.log(err); });
```

Idée

Composer les promesses (ou les valeurs congelées), c'est construire un calcul en composant des fonctions.

Conclusion

- L'étude de la programmation fonctionnelle peut s'aborder et se développer sur de nombreuses techniques.
- La maîtrise de chacune de ces techniques n'est pas évidente et dépasse le cadre de ce cours.
- L'idée centrale tient au fait que les fonctions sont des unités de code portables, réutilisables, appliquables à la demande, et permettant d'organiser dynamiquement le flot de contrôle d'un programme.
- La suite du cours portera sur l'architecture de ces ensembles de fonctions.

Quelques lectures . . .

Fogus, M.: Functional Javascript.
O'Reilly, 2013.

Narbel, P.: Programmation fonctionnelle, générique et objet : Une introduction avec le langage OCaml.
Vuibert, 2005.

Reade, C.: Elements of Functional Programming.
Addison-Wesley, 1989.

Bird, R. et P. Wadler: Introduction to Functional Programming.
Prentice Hall, 1988.

Okasaki, C.: Purely functional data structures.
Cambridge University Press, 1999.

Narbel, P.: Techniques Avancées de Programmation.
Cours de Master 2 à l'Université de Bordeaux.