LIFAP5 – Programmation fonctionnelle pour le WEB

CM3 – Ordre supérieur en λ-calcul et `javascript`

Licence informatique UCBL – Printemps 2017–2018

https://liris.cnrs.fr/~rthion/dokuwiki/doku.php?id=enseignement:lifap5:start

Plan

Motivation : *Glossary of Modern JavaScript Concepts*

https://auth0.com/blog/glossary-of-modern-javascript-concepts/

- Purity : Pure Functions, Impure Functions, Side Effects
- State : Stateful and Stateless
- Immutability and Mutability
- Imperative and Declarative Programming
- Higher-order Functions
- Functional Programming
- ...

Motivation : *Glossary of Modern JavaScript Concepts*

https://auth0.com/blog/glossary-of-modern-javascript-concepts/

- Purity : Pure Functions, Impure Functions, Side Effects
- State : Stateful and Stateless
- Immutability and Mutability
- Imperative and Declarative Programming
- Higher-order Functions
- Functional Programming
- ...

Portée des variables

Déclaration (rappel)

`var x` portée = fonction englobante

`let y` portée = bloc (ES6)

`const c` constante, portée = bloc (ES6)

Exemple : `nom` est accessible dans `afficheNom`

```
function init() {
  let nom = "Mozilla"; // variable locale
  function afficheNom() { // fonction interne
    console.log(nom); // nom est une variable libre
  }
  afficheNom();
};
init();
```

Portée des variables

Déclaration (rappel)

`var x` portée = fonction englobante

`let y` portée = bloc (ES6)

`const c` constante, portée = bloc (ES6)

Exemple : `nom` est accessible dans `afficheNom`

```
function init() {
   let nom = "Mozilla"; // variable locale
   function afficheNom() { // fonction interne
       console.log(nom); // nom est une variable libre
   }
   afficheNom();
};
init();
```

Portée des variables : `var`, `let` et `const`

Exemple, portée de `var` : `i` est accessible hors de la boucle

```
function forLoop() {
  const nb = 5;
  for(var i = 0; i < nb; ++i) {
    console.log("loop: " + i);
  }
  console.log("function: " + i);
};
forLoop();
```

```
loop: 0
...
loop: 4
function: 5
```

Portée des variables : `var`, `let` et `const`

Exemple, portée de `var` : `i` est accessible hors de la boucle

```
function forLoop() {
  const nb = 5;
  for(var i = 0; i < nb; ++i) {
    console.log("loop: " + i);
  }
  console.log("function: " + i);
};
forLoop();
```

```
loop: 0
...
loop: 4
function: 5
```

Portée des variables : `var`, `let` et `const`

Exemple, portée de `let` : `i` n'est pas accessible hors de la boucle

```
function forLoop() {
  const nb = 5;
  for(let i = 0; i < nb; ++i) {
    console.log("loop: " + i);
  }
  console.log("function: " + i);
};
forLoop();
```

```
loop: 0
...
loop: 4
ReferenceError: i is not defined [Learn More]
```

Portée des variables : `var`, `let` et `const`

Exemple, portée de `let` : `i` n'est pas accessible hors de la boucle

```
function forLoop() {
  const nb = 5;
  for(let i = 0; i < nb; ++i) {
    console.log("loop: " + i);
  }
  console.log("function: " + i);
};
forLoop();
```

```
loop: 0
...
loop: 4
ReferenceError: i is not defined [Learn More]
```

const en λ-calcul

Pas de construction native const en λ-calcul.

Nouvelle construction :

$$\texttt{const}\ x = M\ \texttt{in}\ N$$

Sucre syntaxique pour :

$$(\lambda x.N)M$$

Remarque :

$$\texttt{const}\ x = M\ \texttt{in}\ N \underset{\beta}{\rightarrow} N[x := M]$$

const : javascript $\rightsquigarrow$ λ-calcul

Si Vjs $\rightsquigarrow V_\lambda$ et Ejs $\rightsquigarrow E_\lambda$:

javascript

```
function (p) {
   const a = Vjs;
   return Ejs;
   // Ejs utilise p et a
}
```

λ-calcul

$\lambda p.$
$\quad$ (const $a = V_\lambda$ in
$\quad$ E_λ)

i.e. : $\lambda p.((\lambda a.E_\lambda)V_\lambda)$

Extension du λ-calcul : type `unit`

`unit` = type des actions

Facilite le lien `javascript` $\leftrightarrow$ λ-calcul

- Unique valeur : $\square$
- Pas de réduction
- Si utile : on suppose l'existence de certaines fonctions *builtin*, `console_log`: `string` $\to$ `unit`

Typage

$$\frac{}{\Gamma \vdash \square : \texttt{unit}}$$

`javascript`	λ-calcul
fonction sans argument	fonction : `unit` $\to \tau$
fonction sans résultat	fonction : $\tau \to$ `unit`

Fermeture

```
const creerFonction = function () {
  const nom = "Mozilla";
  return function(){
    console.log(nom);
  };
}
const maFonction = creerFonction();
maFonction();
```

Que fait ce code ?

1. Une erreur de reference
2. Une autre erreur
3. Il affiche "Mozilla"
4. Autre chose

Fermeture

```
const creerFonction = function () {
   const nom = "Mozilla";
   return function(){
      console.log(nom);
   };
}
const maFonction = creerFonction();
maFonction();
```

Que fait ce code ?

1. Une erreur de reference
2. Une autre erreur
3. Il affiche "Mozilla"
4. Autre chose

Pseudo-codage en λ-calcul et réduction

$$
\begin{array}{l}
\texttt{const } c = \lambda u_1.(\\
\qquad \texttt{const } n = \texttt{"Mozilla" in} \\
\qquad \lambda u_2.(\texttt{console\_log } n) \\
) \texttt{ in} \\
\texttt{const } m = (c\ \square) \texttt{ in} \\
(m\ \square)
\end{array}
$$

$\texttt{creerFonction} \rightsquigarrow c$

$\texttt{maFonction} \rightsquigarrow m$

$\texttt{nom} \rightsquigarrow n$

Pseudo-codage en λ-calcul et réduction

$$
\begin{array}{l}
\texttt{const } c = \lambda u_1.(\\
\qquad \texttt{const } n = \texttt{"Mozilla" in} \\
\qquad \lambda u_2.(\texttt{console\_log } n) \\
)\ \texttt{in} \\
\texttt{const } m = (c\ \square)\ \texttt{in} \\
(m\ \square)
\end{array}
\quad \rightarrow \quad
\begin{array}{l}
\texttt{const } m = (\lambda u_1.(\\
\qquad \texttt{const } n = \texttt{"Mozilla" in} \\
\qquad \lambda u_2.(\texttt{console\_log } n) \\
)\ \square)\ \texttt{in} \\
(m\ \square)
\end{array}
$$

$\texttt{creerFonction} \rightsquigarrow c$

$\texttt{maFonction} \rightsquigarrow m$

$\texttt{nom} \rightsquigarrow n$

Pseudo-codage en λ-calcul et réduction

$$
\begin{array}{l}
\texttt{const } m = (\lambda u_1.(\\
\qquad \texttt{const } n = \texttt{"Mozilla" in} \\
\qquad \lambda u_2.(\texttt{console\_log } n) \\
) \square) \texttt{ in} \\
(m \ \square)
\end{array}
\qquad \rightarrow
$$

$\texttt{creerFonction} \rightsquigarrow c$

$\texttt{maFonction} \rightsquigarrow m$

$\texttt{nom} \rightsquigarrow n$

Pseudo-codage en λ-calcul et réduction

$$
\begin{array}{l}
\texttt{const } m = (\lambda u_1.(\\
\qquad \texttt{const } n = \texttt{"Mozilla" in} \\
\qquad \lambda u_2.(\texttt{console\_log } n) \\
) \square) \texttt{ in} \\
(m \square)
\end{array}
\quad \rightarrow \quad
\begin{array}{l}
\texttt{const } m = (\\
\qquad \texttt{const } n = \texttt{"Mozilla" in} \\
\qquad \lambda u_2.(\texttt{console\_log } n) \\
) \texttt{ in} \\
(m \square)
\end{array}
$$

$\texttt{creerFonction} \rightsquigarrow c$

$\texttt{maFonction} \rightsquigarrow m$

$\texttt{nom} \rightsquigarrow n$

Pseudo-codage en λ-calcul et réduction

```
const m = (
      const n = "Mozilla" in
      λu2.(console_log n)
) in
(m □)
```

$\texttt{creerFonction} \rightsquigarrow c$

$\texttt{maFonction} \rightsquigarrow m$

$\texttt{nom} \rightsquigarrow n$

Pseudo-codage en λ-calcul et réduction

$$\texttt{const } m = (\ \texttt{const } n = \texttt{"Mozilla" in } \lambda u_2.(\texttt{console\_log } n)\ )\ \texttt{in } (m\ \square) \rightarrow (\ (\ \texttt{const } n = \texttt{"Mozilla" in } \lambda u_2.(\texttt{console\_log } n)\ )\ \square)$$

$\texttt{creerFonction} \rightsquigarrow c$

$\texttt{maFonction} \rightsquigarrow m$

$\texttt{nom} \rightsquigarrow n$

Pseudo-codage en λ-calcul et réduction

$$
\begin{array}{l}
(\,(\\
\qquad \texttt{const}\ n = \texttt{"Mozilla"}\ \texttt{in}\\
\qquad \lambda u_2.(\texttt{console\_log}\ n)\\
\,)\,\square)
\end{array}
$$

$\texttt{creerFonction} \rightsquigarrow c$

$\texttt{maFonction} \rightsquigarrow m$

$\texttt{nom} \rightsquigarrow n$

Pseudo-codage en λ-calcul et réduction

$$
\begin{array}{l}
(\,(\\
\quad \texttt{const}\ n = \texttt{"Mozilla"}\ \texttt{in}\\
\quad \lambda u_2.(\texttt{console\_log}\ n)\\
\,)\,\square)
\end{array}
\quad \rightarrow \quad
\begin{array}{l}
(\,(\\
\lambda u_2.(\texttt{console\_log}\ \texttt{"Mozilla"})\\
\,)\,\square)
\end{array}
$$

$\texttt{creerFonction} \rightsquigarrow c$

$\texttt{maFonction} \rightsquigarrow m$

$\texttt{nom} \rightsquigarrow n$

Pseudo-codage en λ-calcul et réduction

$$((\lambda u_2.(\texttt{console\_log}\ \texttt{"Mozilla"}))\ \square)$$

$\texttt{creerFonction} \rightsquigarrow c$

$\texttt{maFonction} \rightsquigarrow m$

$\texttt{nom} \rightsquigarrow n$

Pseudo-codage en λ-calcul et réduction

$$((\lambda u_2.(\texttt{console\_log "Mozilla"})\ \square)) \quad \rightarrow \quad (\texttt{console\_log "Mozilla"})$$

$\texttt{creerFonction} \rightsquigarrow c$

$\texttt{maFonction} \rightsquigarrow m$

$\texttt{nom} \rightsquigarrow n$

Fermeture

Éléments remarquables de `"creerFonction()"`

- `"creerFonction()"` renvoie une fonction
- la fonction renvoyée est anonyme
- la fonction renvoyée capture la valeur de la variable `"nom"`

Le concept de fermeture

On dit que `"maFonction()"` est une fermeture, elle combine :

- la fonction renvoyée par `"creerFonction()"`
- son environnement, c'est-à-dire l'ensemble des variables locales qui existaient quand `"maFonction()"` a été créée

- fermeture : *closure* en anglais
- `https://developer.mozilla.org/fr/docs/Web/JavaScript/Closures`
- environnement : ou contexte

Fermeture

Éléments remarquables de "creerFonction()"

- "creerFonction()" renvoie une fonction
- la fonction renvoyée est anonyme
- la fonction renvoyée capture la valeur de la variable "nom"

Le concept de fermeture

On dit que "maFonction()" est une fermeture, elle combine :

- la fonction renvoyée par "creerFonction()"
- son environnement, c'est-à-dire l'ensemble des variables locales qui existaient quand "maFonction()" a été créée

- fermeture : *closure* en anglais
- https://developer.mozilla.org/fr/docs/Web/JavaScript/Closures
- environnement : ou contexte

Fermeture

Un contexte propre à chaque fermeture est crée

```
function sayHello(person) {
  return function(str){
    console.log("Hi " + person + ", " + str);
  };
}
let helloBuddy = sayHello("Buddy");
let helloGuys = sayHello("Guys");
helloBuddy("c'mon");
helloGuys("how do you do?");
```

```
Hi Buddy,  c'mon
Hi Guys,  how do you do?
```

La variable `person` a une valeur dans `helloBuddy`
et une autre dans `helloGuys`

Fermeture

Un contexte propre à chaque fermeture est crée

```
function sayHello(person) {
  return function(str){
    console.log("Hi " + person + ", " + str);
  };
}
let helloBuddy = sayHello("Buddy");
let helloGuys = sayHello("Guys");
helloBuddy("c'mon");
helloGuys("how do you do?");
```

```
Hi Buddy,  c'mon
Hi Guys,  how do you do?
```

La variable `person` a une valeur dans `helloBuddy`
et une autre dans `helloGuys`

Fermeture

Un contexte propre à chaque fermeture est crée

```
function sayHello(person) {
  return function(str){
    console.log("Hi " + person + ", " + str);
  };
}
let helloBuddy = sayHello("Buddy");
let helloGuys = sayHello("Guys");
helloBuddy("c'mon");
helloGuys("how do you do?");
```

```
Hi Buddy,  c'mon
Hi Guys,  how do you do?
```

La variable `person` a une valeur dans `helloBuddy`
et une autre dans `helloGuys`

Fermeture

Le contexte d'une fermeture peut être partagé

```
let f, g;
function globals(){
   let num = 0;
   f = function (){console.log(num);};
   g = function (){num++;};
}

globals();

f();   // 0 is printed
g();   // num is incremented
f();   // 1 is printed
```

⚠ la fermeture d'un `let` est une référence ⚠

Fermeture

Piège : passage par référence pour les objets

```
function sayHello(person) {
  return function(str){
    console.log("Hi " + person.name + ", " + str);
  };
}
const o = {name: "Romu"}; let helloRomu = sayHello(o);
o.name = "Manu";          let helloManu = sayHello(o);
helloRomu("how do you do?");
helloManu("how do you do?");
```

```
Hi Manu, how do you do?
Hi Manu, how do you do?
```

La variable `person` a été passée *par référence* dans `helloManu` et `helloRomu`

Fermeture

Piège : passage par référence pour les objets

```
function sayHello(person) {
  return function(str){
    console.log("Hi " + person.name + ", " + str);
  };
}
const o = {name: "Romu"}; let helloRomu = sayHello(o);
o.name = "Manu";          let helloManu = sayHello(o);
helloRomu("how do you do?");
helloManu("how do you do?");
```

```
Hi Manu, how do you do?
Hi Manu, how do you do?
```

La variable `person` a été passée *par référence* dans `helloManu` et `helloRomu`

Fermetures

La fermeture est un concept clef des fonctions renvoyées comme résultat.

Construire des fonctions

De nombreux éléments `javascript` prennent des fonctions en paramètre

- fonctions génériques sur les tableaux
- *callbacks* (fonctions exécutées en cas de succès ou échec)
- *listener* sur des événements

Le changeur de taille

```
function fabriqueRedimensionneur(taille) {
  return function() {
    document.body.style.fontSize = taille + 'px';
  };
};
let taille12 = fabriqueRedimensionneur(12);
let taille14 = fabriqueRedimensionneur(14);
document.getElementById('t-12').onclick = taille12;
document.getElementById('t-14').onclick = taille14;
```

Construire des fonctions

De nombreux éléments `javascript` prennent des fonctions en paramètre

- fonctions génériques sur les tableaux
- *callbacks* (fonctions exécutées en cas de succès ou échec)
- *listener* sur des événements

Le changeur de taille

```
function fabriqueRedimensionneur(taille) {
  return function() {
    document.body.style.fontSize = taille + 'px';
  };
};
let taille12 = fabriqueRedimensionneur(12);
let taille14 = fabriqueRedimensionneur(14);
document.getElementById('t-12').onclick = taille12;
document.getElementById('t-14').onclick = taille14;
```

Immediately-Invoked Function Expression

Une *Immediately-Invoked Function Expression* (IIFE) est une fonction qui est invoquée (une seule fois) immédiatement après sa définition.

Prototype

```
(function () {
  /* code */
})();
```

Les parenthèses englobantes sont obligatoires

Un *design pattern* typique de `javascript`

- Simuler des variables privées en contrôlant la portée
- Encapsuler du code en créant un espace de nommage [a]

a. ES6 dispose d'une notion de *module* pour éviter cette pratique

Application IIFE : variable « privée »pour un compteur

Mauvaise solution 1

```
let counter = 0;
function add() {
    return counter++;
}
add();
add();
```

Mauvaise solution 2

```
function add() {
    let counter = 0;
    return counter++;
}
add();
add();
```

Application IIFE : variable « privée »pour un compteur

Mauvaise solution 1

```
let counter = 0;
function add() {
    return counter++;
}
add();
add();
```

Mauvaise solution 2

```
function add() {
    let counter = 0;
    return counter++;
}
add();
add();
```

Application IIFE : variable « privée »pour un compteur

Solution : IIFE

```
let add = (function () {
    let counter = 0;
    return () => (counter++);
})();

add();  //0
add();  //1
add();  //2
```

Extension du λ-calcul aux paires

Expressions :

- construction de paire : $\langle M, N \rangle$
- première composante : π_1
- deuxième composante : π_2

Réductions :

- $\pi_1 \langle M, N \rangle \underset{\beta}{\rightarrow} M$
- $\pi_2 \langle M, N \rangle \underset{\beta}{\rightarrow} N$

Typage :

$$\frac{\Gamma \vdash M : \tau_M \quad \Gamma \vdash N : \tau_N}{\Gamma \vdash \langle M, N \rangle : \tau_M \times \tau_N}$$

$$\pi_1 : \quad \tau_M \times \tau_N \rightarrow \tau_M$$
$$\pi_2 : \quad \tau_M \times \tau_N \rightarrow \tau_N$$

Passer deux arguments à une fonction

$$\lambda p.(\pi_1 p + \pi_2 p) : \texttt{number} \times \texttt{number} \to \texttt{number}$$

$$\lambda p.(\pi_1 p + \pi_2 p)\langle 2, 3\rangle \underset{\beta}{\to} \pi_1\langle 2, 3\rangle + \pi_2\langle 2, 3\rangle \underset{\beta}{\to} 2 + \pi_2\langle 2, 3\rangle \underset{\beta}{\to} 2 + 3 \underset{\beta}{\to} 5$$

Version **curryfiée** de $\lambda p.(\pi_1 p + \pi_2 p)$

$$(\lambda x.\lambda.y.(x + y)) : \texttt{number} \to \texttt{number} \to \texttt{number}$$

$$(\lambda x.\lambda.y.(x + y))\ 2\ 3 \underset{\beta}{\to} (\lambda y.(2 + y))\ 3 \underset{\beta}{\to} 2 + 3 \underset{\beta}{\to} 5$$

Passer deux arguments à une fonction

$$\lambda p.(\pi_1 p + \pi_2 p) : \texttt{number} \times \texttt{number} \to \texttt{number}$$

$$\lambda p.(\pi_1 p + \pi_2 p)\langle 2, 3\rangle \underset{\beta}{\to} \pi_1\langle 2, 3\rangle + \pi_2\langle 2, 3\rangle \underset{\beta}{\to} 2 + \pi_2\langle 2, 3\rangle \underset{\beta}{\to} 2 + 3 \underset{\beta}{\to} 5$$

Version **curryfiée** de $\lambda p.(\pi_1 p + \pi_2 p)$

$$(\lambda x.\lambda.y.(x + y)) : \texttt{number} \to \texttt{number} \to \texttt{number}$$

$$(\lambda x.\lambda.y.(x + y))\ 2\ 3 \underset{\beta}{\to} (\lambda y.(2 + y))\ 3 \underset{\beta}{\to} 2 + 3 \underset{\beta}{\to} 5$$

Passer deux arguments à une fonction

$$\lambda p.(\pi_1 p + \pi_2 p) : \texttt{number} \times \texttt{number} \to \texttt{number}$$

$$\lambda p.(\pi_1 p + \pi_2 p)\langle 2, 3\rangle \underset{\beta}{\to} \pi_1\langle 2, 3\rangle + \pi_2\langle 2, 3\rangle \underset{\beta}{\to} 2 + \pi_2\langle 2, 3\rangle \underset{\beta}{\to} 2 + 3 \underset{\beta}{\to} 5$$

Version **curryfiée** de $\lambda p.(\pi_1 p + \pi_2 p)$

$$(\lambda x.\lambda.y.(x + y)) : \texttt{number} \to \texttt{number} \to \texttt{number}$$

$$(\lambda x.\lambda.y.(x + y))\ 2\ 3 \underset{\beta}{\to} (\lambda y.(2 + y))\ 3 \underset{\beta}{\to} 2 + 3 \underset{\beta}{\to} 5$$

Curryfication

Curryfier [1] : transformer une fonction à deux arguments en une fonction à un argument qui renvoie une fonction à un argument

Exemple en javascript

```
function sayHello(str, age){
  console.log("Hello " + str +  ", you're " + age);
}
function sayHelloCurry(str){
  return function(age){
    console.log("Hello " + str +  ", you're " + age);
  };
}
sayHello("Romu", 35);
sayHelloCurry("Romu")(35);
```

1. Le nom est donné en hommage à Haskell Curry

Curryfication

Soit f une fonction $(x, y) \mapsto f(x, y)$ alors la fonction g définie par $x \mapsto (y \mapsto f(x, y))$ est telle que $f(x, y) = g(x)(y)$ et est unique. On appelle g le *curryfié* de f.

En javascript

```
function sayHello(str, age){
  console.log("Hello " + str +  ", you're " + age);
}

function curry(fn) {
  return (arg1) => ((arg2) => fn(arg1, arg2));
}

let sayHelloCurry = curry(sayHello);
sayHelloCurry("Romu")(35);
```

Application partielle d'une fonction

Soit f une fonction $(x, y) \mapsto f(x, y)$ et a une constante fixée, alors l'application partielle de a à f est la fonction f_a définie par $y \mapsto f(a, y)$.

En javascript

```
function sayHello(str, age){
  console.log("Hello " + str +  ", you're " + age);
}

function partial(fn, arg1) {
  return (arg2) => fn(arg1, arg2);
}

let sayHelloRomu = partial(sayHello, "Romu");
sayHelloRomu(35); // Hello Romu, you're 35
```

Composition de fonctions

Soit f et g deux fonctions, alors la composition de f par g est la fonction $g \circ f$ définie par $x \mapsto g(f(x))$.

En javascript

```
let sayHello = (str) => ("Hello " + str);
let addWelcome = (str) => (str + ", you're welcome");

function compose(fn2, fn1) {
  return ((arg) => fn2(fn1(arg)));
}

let sayHelloWelcome = compose(addWelcome, sayHello);
sayHelloWelcome("Romu"); // Hello Romu, you're welcome
```

Notion de typage

Typer, c'est associer à une expression (ou plus généralement, un programme) l'ensemble des valeurs qu'elle peut prendre.

Un langage des types : *Type*

Type est le plus petit ensemble définit par la grammaire suivante, avec $T \in \mathit{Type}$, $T' \in \mathit{Type}$ et $\tau_1, \tau_2, \ldots$ des variables de type :

$$\mathit{Type} := \texttt{Array}(T) \mid T \to T' \mid T \times T' \mid \texttt{boolean} \mid \texttt{number} \mid \ldots \mid \tau_1 \mid \tau_2 \mid \ldots$$

`number`	l'ensemble des nombres
`boolean`	l'ensemble des booléens
$T \to T'$	l'ensemble des fonctions de T dans T'
`Array`(T)	l'ensemble des tableaux (liste) d'éléments du type T
$T \times T'$	l'ensemble des paires composées d'un premier élément de type T et d'un second élément de type T'

Notion de typage

Well-typed programs cannot "go wrong". [2]

Pourquoi typer ?

- Pour s'assurer que le comportement des programmes est défini
- Pour spécifier les programmes
- Pour comprendre les programmes

Typage en `javascript`

- Pas de typage explicite (à la différence du C/C++ par exemple)
- Conversions implicites de type (par exemple, pour la concaténation)
- Pas d'inférence de type (calcul du type sans exécution de programme)
- Mais, vérifications à l'exécution

2. Robin Milner, voir https://en.wikipedia.org/wiki/Type_safety

Exemples en javascript

```
function curry2(fn) {
  return (arg1) => ((arg2) => fn(arg1, arg2));
}
function compose1(fn2, fn1) {
  return ((args) => fn2(fn1(args)));
}
function reduce_r(fn, acc, arr, idx) {
  return (idx == arr.length) ? acc :
           reduce_r(fn, fn(arr[idx],acc), arr, idx+1);
}

let reduce = (fn, zero, arr)=> reduce_r(fn, zero, arr, 0);
```

- curry2 : $((\tau_1 \times \tau_2) \to \tau_3) \to (\tau_1 \to (\tau_2 \to \tau_3))$
- compose1 : $((\tau_2 \to \tau_3) \times (\tau_1 \to \tau_2)) \to (\tau_1 \to \tau_3)$
- reduce_r : $((\tau_1 \times \tau_2) \to \tau_2) \times \tau_2 \times \mathtt{Array}(\tau_1) \times \mathtt{number}) \to \tau_2$
- reduce : $((\tau_1 \times \tau_2) \to \tau_2) \times \tau_2 \times \mathtt{Array}(\tau_1)) \to \tau_2$

Exemples en javascript

```
function curry2(fn) {
  return (arg1) => ((arg2) => fn(arg1, arg2));
}
function compose1(fn2, fn1) {
  return ((args) => fn2(fn1(args)));
}
function reduce_r(fn, acc, arr, idx) {
  return (idx == arr.length) ? acc :
           reduce_r(fn, fn(arr[idx],acc), arr, idx+1);
}

let reduce = (fn)=> (zero)=> (arr)=> reduce_r(fn, zero, arr, 0);
```

- curry2 : $((\tau_1 \times \tau_2) \to \tau_3) \to (\tau_1 \to (\tau_2 \to \tau_3))$
- compose1 : $((\tau_2 \to \tau_3) \times (\tau_1 \to \tau_2)) \to (\tau_1 \to \tau_3)$
- reduce_r : $((\tau_1 \times \tau_2) \to \tau_2) \times \tau_2 \times \mathtt{Array}(\tau_1) \times \mathtt{number}) \to \tau_2$
- reduce : $((\tau_1 \times \tau_2) \to \tau_2) \to (\tau_2 \to (\mathtt{Array}(\tau_1) \to \tau_2))$