Cours N+2 : Programmation Fonctionnelle en Erlang

February 29, 2024

Séance 8 (S+2) ; Cours N+2 : Programmation fonctionnelle en Erlang

1) Fonctions simples (rappels)
   - exemple (longueur, concaténation)
   - moteur d'exécution (rappels)
   - raisonnement par cas et garde (rappels)
   - récursivité et propriétés algos (rappels)
2) Fonctions simples (suite)
   - fonction avec plusieurs résultats
   - récursivité terminale
   - accumulateur
3) Fonctions d'ordre supérieur : Fun => Fun
   - Générateur de fonctions
   - Cloture (paramètres privés/publics)
4) Fonction d'ordre supérieur : Fun(Fun)
   - fonction auxiliaire (ex. comparaison() pour tri(), tri(comparaison))
   - map, reduce et autres
5) Autres utilisations
   - Evaluation paresseuse
   - Structure incomplète

1 Fonctions simples (rappels)

1.1 Exemples (longueur, concaténation)

[12]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

longueur([])->0;
longueur([_E|L])->1+longueur(L).

concatener([],B)->B;
concatener([E|L],B)->[E|concatener(L,B)].

main([]) ->
  io:format('~nResultat : ~p ~n',[longueur([1,2,3,4,5])]),
  io:format('~nResultat : ~p ~n',[concatener([1,2],[3,4,5])]).
```

```
Overwriting prog.erl
```

[13]:
```
!escript prog.erl
```

```
Resultat : 5

Resultat : [1,2,3,4,5]
```

1.2 Moteur d'exécution (rappels)

L'exécution se déroule en 2 temps :

- Evaluation des paramètres et/ou arguments des expressions
- Evaluation de la fonction elle-même

D'autres langages fonctionnelles font autrement, avec une évaluation dite "paresseuse" des paramètres, au besoin, seulement si c'est nécessaire. Cette évaluation passe donc en second par rapport à l'évaluation de la fonction.

On peut donner un exemple de trace d'exécution :

?> longueur(concaténation([1],[2]))
–» longueur([1,2])
–» 1 + longueur([2])
–» 1 + 1 + longueur([])
–» 1 + 1 + 0
–» 2

1.3 Raisonnement par cas et garde (rappels)

(Presque) rien de nouveau ici, juste un exemple et qlq détails sur les gardes.

[18]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

valeurAbsolue(A) when A > 0 -> A;
valeurAbsolue(A) -> -A.

main([]) ->
  io:format('~nResultat : ~p ~n',[valeurAbsolue(2)]),
  io:format('~nResultat : ~p ~n',[valeurAbsolue(-3)]).
```

```
Overwriting prog.erl
```

[19]:
```
!escript prog.erl
```

```
Resultat : 2
```

```
Resultat : 3
```

Les expressions et fonctions autorisées dans les gardes sont (limitées) :

- expressions arithmétiques et relationnelles simples (+, -, /, div, rem, >, ==, /=, etc.)
- fonction prédéfinies autorisées (BIF) :
  - is_integer, is_list, is_atom
  - abs, hd, tl, length, self

(plus de détail dans la doc erlang)

1.4 Récursivité et propriétés algos (rappels)

Rien de nouveau ici.

Récursivite et raisonnement par cas : structures de contrôle complémentaires, permet d'écrire boucles et conditionnelles.

Ne pas oublier de vérifier :

- la terminaison des récursivités,
- la complétude des raisonnements par cas,
- (et toujours) la correction de vos programmes.

2 Fonctions simples (suite)

Quelques détails supplémentaires sur les fonctions.

(il restera encore plein de détails à donner, on verra plus tard)

2.1 Fonction avec plusieurs résultats

En Pascal, en C, en ProLog, on peut mettre autant de paramètres résultats que l'on veut (en C en faisant un passage par adresse).

En Erlang, le résultat est unique, mais comme en Python, on peut "à la volée" construire une liste de plusieurs résultats.

Ex. rechercheEtSupprimeMin(Liste) -> [Min,ListeSansMin] (à faire en exercice)

Attention, pour la récurrence, il est souvent nécessaire d'expliciter les résultats des appels imbriqués.

```
rechercheEtSupprimeMin([E|L])->
  [X,LX] = rechercheEtSupprimeMin(L),
   ... travail sur E, L, X et LX pour produire le résultat final
  [R|LR].
```

2.2 Récursivité terminale

Dans l'exemple précédent, `longueur(concaténation([1],[2]))`, l'organisation de la récursivité n'est pas terminale : l'appel récursif de longueur n'est pas la dernière action à faire, (il reste à faire une addition), cela peut nécessiter d'empiler les additions (si le compilateur ne trouve pas comment optimiser).

Pour aider le compilateur, il est parfois nécessaire de modifier l'organisation du code, par exemple en ajoutant un accumulateur (cf. la suite)

Selon les compilateur Erlang, les optimisations sont bien faites, il n'est pas nécessaire de chercher les récursivités terminales.

2.3 Accumulateur (et inversion)

Le calcul ce longueur avec accumulateur :

[34]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

longueur2(L) ->  %% introduction d'un accumulateur à 0
  longueurAvecAccumulateur(L,0).

longueurAvecAccumulateur([],Accumulateur)->Accumulateur;  %% retour de la␣
 ↪valeur finale obtenue grace à l'accumulateur
longueurAvecAccumulateur([_E|L],Accumulateur)->longueurAvecAccumulateur(L,1+Accumulateur).
 ↪



main([]) ->
  io:format('~nResultat : ~p ~n',[longueur2([1,2,3])]).
```

```
Overwriting prog.erl
```

[33]:
```
!escript prog.erl
```

```
Resultat : 3
```

Ainsi, l'évaluation de l'addition est faite dés le début de chaque instant d'évaluation de la longueur (sur les longueurs connues du début de la liste), au moment de l'évaluation des paramètres (1+Accumulateur).

L'évaluation de la fonction correspond à l'appel récursif, il n'est suivi d'aucune autre évaluation, il n'est pas nécessaire d'empiler un état pour la pile d'appel, ou d'augmenter la taille de l'expression finale à évaluer, le compilateur optimise cet appel (et l'expresssion) et la pile d'appel (et l'expression) n'augmente pas de taille.

Attention, les programmes avec accumulateurs peuvent avoir des comportements inverses de ceux sans accumulateurs (et des résultats inverses).

Exemple, une mauvaise concaténation obtenue à cause d'une utilisation erronée d'un accumulateur :

[39]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).
```

```
concatener2(D,F) ->
  concatenerAvecAccumulateur(D,F,[]). %% introduction d'un accumulateur à []

concatenerAvecAccumulateur([],B,Acc)->[B|Acc];
concatenerAvecAccumulateur([E|L],B,Acc)->concatenerAvecAccumulateur(L,B,[E|Acc]).
 ↪

main([]) ->
  io:format('~nResultat : ~p ~n',[concatener2([1,2],[3,4,5])]).
```

```
Overwriting prog.erl
```

[40]:
```
!escript prog.erl
```

```
Resultat : [[3,4,5],2,1]
```

Autre essai

[42]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

concatener2(D,F) ->
  concatenerAvecAccumulateur(D,F,[]). %% introduction d'un accumulateur à []

concatenerAvecAccumulateur([],B,Acc)->[Acc|B];
concatenerAvecAccumulateur([E|L],B,Acc)->concatenerAvecAccumulateur(L,B,[E|Acc]).
 ↪

main([]) ->
  io:format('~nResultat : ~p ~n',[concatener2([1,2],[3,4,5])]).
```

```
Overwriting prog.erl
```

[43]:
```
!escript prog.erl
```

```
Resultat : [[2,1],3,4,5]
```

Dans les 2 cas, “1,2” est inversé, et le résultat n’est plus une liste simple.

Conclusion : attention aux accumulateurs, ils sont souvent utilisés pour faire comme en programmation classique (non récursive) et éviter la récursivité ou pour optimiser un code (cf. prec.). Mais même avec des Accumulateurs il faut comprendre la récursivité pour les utiliser, et quand on comprend la récursivité, ils deviennent souvent inutiles. Et pour les optimisations, Erlang dit que ce n’est pas nécessaire (que le compilateur sait optimiser).
Bref, à utiliser avec précaution.

3 Fonctions d'ordre supérieur : Fun => Fun

3.1 Générateur de fonctions

Exemple, un incrémenteur universel ?

Pour faire un incrémenteur, une fonction simple suffit :

`incrementeur(X)->X+1.`

Idem si on veut un incrémenteur qui avance de 2, de 3, de 42 etc.

Mais cela demande autant de fonction et de code pour les écrire que d'incrémenteur, alors qu'ils seront tous définis presque pareil.

Pour améliorer cela, on peut faire une fabrique d'incrémenteur, et générer les incrémenteurs à la demande :

[47]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

fabriqueIncrementeur(Increment) ->
  fun (X) -> X+Increment end.

main([]) ->
  IncrementeurNormal = fabriqueIncrementeur(1),
  IncrementeurDe5en5 = fabriqueIncrementeur(5),
  io:format('~nResultat : ~p ~n',[IncrementeurNormal(42)]),
  io:format('~nResultat : ~p ~n',[IncrementeurDe5en5(42)]).
```

```
Overwriting prog.erl
```

[48]:
```
!escript prog.erl
```

```

Resultat : 43

Resultat : 47
```

À noter au passage :

- la définition d'une fonction à la volée, fonction anonyme avec 1 paramètre `fun (X) -> ... end`
- ne pas oublier le `end` à la fin de la définition de la fonction (rem. : le point n'est pas associé au end de la fonction définie à la volée mais à la fin de la fonction fabriqueIncrementeur)
- des **V**ariables avec pour valeur des fonctions
- pour faire des fonction récursives, il faut des fonctions qui ne soient pas anonymes, c'est possible il suffit de nommer la fonction définie à la volée `fun maFonction(X) -> ... end`

3.2 Cloture (paramètres privés/publics)

Dans l'exemple précédent, l'incrément est un variable capturée par le fonction à la volée. Elle devient un paramètre privé de cette fonction. On dit qu'elle appartient à la cloture de la définition.

4 Fonction d'ordre supérieur : Fun(Fun)

4.1 Fonction auxiliaire

Il y a des cas simples où les fonctions sont des paramètres (d'autres fonctions).

Par exemple, dans le cas d'un tri que l'on veut indépendant de la structure de données. Pour faire le lien avec la structure de données, il faut tout de même pouvoir comparer les éléments. Cette fonction de comparaison peut être un paramètre du tri. C'est une fonction auxiliaire.

4.2 Continuations (fonction de rappel, post-traitement, futur, promesse, delai)

L'ordre supérieur a été beaucoup (vraiment beaucoup) utilisé pour gérer les traitements asynchrones, paralèlles ou pas, par exemple dans la programmation évènementielle. Ex. les fonctions de callback dans les xmlHttpRequest de Javascript.

Pour donner une exemple, on va simuler une fonction d'E/S avec analyse de la réponse (en fonction de rappel).

Intérêt : séparer les E/S et les analyses. Ici pour faire une E/S avec une analyse disjointe.

Rem. dans l'exemple, comme souvent, la priorité semble donnée à la première fonction exécuté (la suite se trouve dans sa fonction de rappel de cette première fonction exécutes, l'appel général est l'appel de cette première fonction exécutée, etc.). On pourrait avoir une construction plus abstraite qui enchaine des fonctions arbitraires, sans donner l'illusion qu'il y a une priorité à la première fonction exécutée (suite).

[116]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

analyseReponseALaGrandeQuestion(42) -> cEstLaSolution;
analyseReponseALaGrandeQuestion(N) when N < 42 -> peutEtrePlusGrand;
analyseReponseALaGrandeQuestion(_N) -> peutEtrePlusPetit.

sansLeSigne(N) when N < 0 -> -N;
sansLeSigne(N) -> N.

questionEtAnalyse(Analyse) ->
  {ok, Term} = io:read("Donne un nombre : "),
  Analyse(Term).


main([]) ->
```

```
  io:format('~nResultat : ~p ~n',[questionEtAnalyse(fun (X) ->␣
 ↪analyseReponseALaGrandeQuestion(X) end)]),
  io:format('~nResultat : ~p ~n',[questionEtAnalyse(fun (X) -> sansLeSigne(X)␣
 ↪end)]).
```

```
Overwriting prog.erl
```

[117]:
```
!echo -e " 42.\n42." | escript prog.erl
```

```
Donne un nombre :
Resultat : cEstLaSolution
Donne un nombre :
Resultat : 42
```

version plus abstraite (la première fonction, n'est pas la fonction globalement appelée, il y a une autre fonction générale appelée)

[171]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

analyseReponseALaGrandeQuestion(42) -> cEstLaSolution;
analyseReponseALaGrandeQuestion(N) when N < 42 -> peutEtrePlusGrand;
analyseReponseALaGrandeQuestion(_N) -> peutEtrePlusPetit.

question() ->
  {ok, Term} = io:read("Donne un nombre : "),
  Term.

enchaine(A,B) ->
  B(A()).


main([]) ->
  io:format('~nResultat : ~p ~n',[enchaine(fun () -> question() end, fun (Y) ->␣
 ↪analyseReponseALaGrandeQuestion(Y) end)]).
```

```
Overwriting prog.erl
```

[172]:
```
!echo "42." | escript prog.erl
```

```
Donne un nombre :
Resultat : cEstLaSolution
```

Version plus compacte, en compilant le fichiers, ce qui permet de nommer les fonctions (au lieu de les utiliser le biais de fonctions anonyme qui lie les fonctions locales par cloture)

[168]:
```
%%writefile prog.erl
-module(prog).
```

```
-compile([export_all,nowarn_export_all]).

analyseReponseALaGrandeQuestion(42) -> cEstLaSolution;
analyseReponseALaGrandeQuestion(N) when N < 42 -> peutEtrePlusGrand;
analyseReponseALaGrandeQuestion(_N) -> peutEtrePlusPetit.

question() ->
  {ok, Term} = io:read("Donne un nombre : "),
  Term.

enchaine(A,B) ->
  B(A()).


main([]) ->
  io:format('~nResultat : ~p ~n',[enchaine(fun prog:question/0, fun prog:
 ↪analyseReponseALaGrandeQuestion/1)]).
```

```
Overwriting prog.erl
```

[170]:
```
!erlc prog.erl ; echo -e "42." | escript prog.erl
```

```
Donne un nombre :
Resultat : cEstLaSolution
```

4.3 Map-Reduce et autres foldl, filter

La programmation fonctionnelle moderne retrouve La structure de données de ses origines (les listes de lisp).

Le retour des parcours sur les listes.

Plusieurs fonctions d'ordre supérieur associent concernent les listes :

- map(F,L)->L : application d'une fonction à tous les éléments d'une liste
- foldl(F,Acc,L)->Acc (ou foldr ou reduce) : réduction d'une liste à un élement par accumulation d'exécutions succéssives sur chaque éléments d'une liste (en commençant au début/à gauche pour foldl, ou par la fin/à droite pour foldr)
- filter(P,L)->L : qui réduit une liste aux éléments respectant un certains prédicat
- all(P,L)->Bool (ou any) : qui évalue une propriété booléenne sur tous les éléments d'une liste et revoie la conjonction (ou la disjonction) des réponses
- etc.

[62]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

map(_F,[]) -> [];
map(F,[E|L]) -> [F(E)|map(F,L)].
```

```
foldl(_F,Acc,[]) -> Acc;
foldl(F,Acc,[E|L]) -> foldl(F,F(E,Acc),L).

filter(_P,[])->[];
filter(P,[E|L])->
  suiteFilter(P,P(E),[E|L]).

suiteFilter(P,true,[E|L]) -> [E|filter(P,L)];
suiteFilter(P,_,[_E|L]) -> filter(P,L).

main([]) ->
  io:format('~nResultat : ~p ~n',[map(fun (X) -> 10*X end, [1,2,3,4])]),
  io:format('~nResultat : ~p ~n',[foldl(fun (X,Acc) -> X+Acc end, 0,␣
 ↪[1,2,3,4])]),
  io:format('~nResultat : ~p ~n',[filter(fun (X) -> (X rem 2) == 0 end,␣
 ↪[1,2,3,4])]).
```

```
Overwriting prog.erl
```

[63]:
```
!escript prog.erl
```

```
Resultat : [10,20,30,40]

Resultat : 10

Resultat : [2,4]
```

5 Autres utilisations

C'est plus exotique, mais il y a d'autres utilisations des listes qui permettent de simuler une évaluation paresseuse (à la Haskell) ou des structures incomplètes (à la ProLog).

5.1 Structure incomplète

Précédement, on a fait une concaténation terminale avec accumulateur "fausse" (elle produisait des listes qui n'étaient plus simples et dont certaines parties étaient mises à l'envers). Pour résoudre les 2 problèmes, il fallait ajouter les éléments en queue de liste et non pas en début de liste comme c'était proposé (et comme c'est le plus simple). Pour ajouter en fin de liste, on peut faire un algo linéaire de parcours de la liste et ajout en fin, mais il faudrait trouver mieux, ajouter en fin de liste en temps constant. C'est possible.
Pour ajouter en fin, dans une liste, c'est pratique si on a déjà un pointeur sur la fin. En Erlang, il n'y a pas de pointeur, mais il y a des fonctions. Si, au lieu d'ajouter dans une liste, on opère avec une fonction qui renvoie une liste avec un paramètre de fonction qui pointe sur la fin, cela donne le même résultat.

On a ainsi, un ajout en fin, en temps constant :

`ajoutPourListeAvecFonctionQueue(E,F)->F([E]).`

il faut “juste” fournir en entrée une liste avec fonction/pointeur de queue.

`ajoutPourListeAvecFonctionQueue(4,fun (Z) -> [1,2,3|Z] end).`

et si on conserve la structure de listeAvecFonctionQueue, on peut enchainer les ajouts en fin

`ajoutPourListeAvecFonctionQueue(E,F)->fun (Z) -> F([E|Z]) end.`

par contre, à la fin, si on a conservé la structure de liste avec fonction/pointeur de queue, on a une fonction au lieu d'une liste, pour retrouver la liste il faut applique le fonction sur qlq chose, par exemple sur la liste vide (ce qui semble le plus naturel si on pense que la liste est finie).

`L([]).`

et on peut donc même obtenir une concaténation terminale avec accumulateur, l'appel initial prend un accumulateur vide, et à la fin si on veut le liste on peut rompre la structure de listeAvecFonctionQueue an appliquant la fonction de queue sans re-créer une nouvelle fonction de queue :

[75]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).


ajoutPourListeAvecFonctionQueue(E,F)->fun (Z) -> F([E|Z]) end.

concatenerPourListeAvecFonctionQueue([],B,F)->F(B);
concatenerPourListeAvecFonctionQueue([E|L],B,F)->
  NF = ajoutPourListeAvecFonctionQueue(E,F),
  concatenerPourListeAvecFonctionQueue(L,B,NF).

main([]) ->
  io:format('~nResultat : ~p␣
  ↪~n',[concatenerPourListeAvecFonctionQueue([1,2],[3,4,5,6],fun (Z) -> Z␣
  ↪end)]).
```

```
Overwriting prog.erl
```

[76]:
```
!escript prog.erl
```

```
Resultat : [1,2,3,4,5,6]
```

ou plus compact :

[77]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).


concatenerQ([],B,F)->F(B);
```

```
concatenerQ([E|L],B,F)->concatenerQ(L,B,fun (Z) -> F([E|Z]) end).

main([]) ->
  io:format('~nResultat : ~p ~n',[concatenerQ([1,2],[3,4,5,6],fun (Z) -> Z␣
 ↪end)]).
```

```
Overwriting prog.erl
```

[78]:
```
!escript prog.erl
```

```
Resultat : [1,2,3,4,5,6]
```

ou, mieux, en masquant le mécanisme interne :

[173]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

concatener(A,B) ->
  concatenerQ(A,B,fun (Z) -> Z end).

concatenerQ([],B,F)->F(B);
concatenerQ([E|L],B,F)->concatenerQ(L,B,fun (Z) -> F([E|Z]) end).

main([]) ->
  io:format('~nResultat : ~p ~n',[concatener([1,2],[3,4,5,6])]).
```

```
Overwriting prog.erl
```

[174]:
```
!escript prog.erl
```

```
Resultat : [1,2,3,4,5,6]
```

5.2 (pseudo) Evaluation paresseuse

Certains langages ont une évaluation paresseuse des parresseuses (à la demande seulement si c'est nécessaire), mais erlang a une évaluation des arguments en premier (avant évaluation du corps des fonctions). Mais avec les fonctions, on peut simuler une évaluation paresseuse (des paramètres). Cela permet, par exemple, d'optimiser certains calculs (ne seront faits que les calculs nécessaires) et même d'utiliser des objets (potentiellement) "infinis".

Exemple, pour créer des Suites infinies

[66]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).
```

```
listeAPartirDe(N) ->
    fun() ->
            [N|listeAPartirDe(N+1)]
    end.

main([]) ->
  L0 = listeAPartirDe(0),
  io:format('~nResultat : ~p ~n',[L0()]).  %%une application, un élement (et la␣
 ↪suite) !
```

```
Overwriting prog.erl
```

[67]:
```
!escript prog.erl
```

```
Resultat : [0|#Fun<erl_eval.45.65746770>]
```

[73]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).


listeAPartirDe(N) ->
    fun() ->
            [N|listeAPartirDe(N+1)]
    end.

main([]) ->
  L0 = listeAPartirDe(0),
  [E0|L1]=L0(),    %% plusieurs application, à chaque fois, un élément et la␣
 ↪suite ...
  [E1|L2]=L1(),
  io:format('~nResultat : ~p ~n',[[E0,E1|L2()]]).
```

```
Overwriting prog.erl
```

[74]:
```
!escript prog.erl
```

```
Resultat : [0,1,2|#Fun<erl_eval.45.65746770>]
```

et ainsi de suite...
(pour avoir les valeurs il faut évaluer explicitement, c'est contraignant d'un côté, mais en même temps, si l'évaluation était automatique, comme la suite est infinie, cela ne serait pas une bonne solution)