Langages fonctionnels non-stricts: De la paresse et de l'indulgence

Guy Tremblay

Dépt. d'informatique, UQAM

juillet 1997

1

Qu'est-ce qu'un langage non-strict?

Selon B. Goldberg, dans "*Functional Programming Languages*" (Computing Surveys, Jan. 1996):

> *In a class of functional languages called* non-strict *functional languages, of which* `Haskell` *is the most popular, no function evaluates its arguments unless they are needed.*
>
> *Another feature of non-strictness is that it allows the construction of infinite data structures.*

Vraiment?

2

Aperçu

- Qu'est-ce qu'un langage strict? Qu'est-ce qu'un langage non-strict?
- À quoi sert l'évaluation non-stricte *sans* paresse?
- À quoi sert l'évaluation paresseuse?
- Comment, dans un contexte parallèle, les différentes formes d'évaluation se distinguent-elles?

3

1. Langage strict vs. langage non-strict

1.1 Langage fonctionnel strict

Une fonction f est dite stricte lorsque:

$$x = \bot \Rightarrow f\ x = \bot$$

Informellement: le résultat ne peut pas être produit tant que l'argument n'est pas disponible.

Stratégie de mise en oeuvre

$=$ appel par valeur

$\Rightarrow$ toutes les fonctions sont strictes!

4

1.2 Langage fonctionnel non-strict

Fonction non-stricte:

$$\neg\,(x = \bot \Rightarrow f\ x = \bot)$$
$$\equiv$$
$$x = \bot \wedge\ f\ x \neq \bot$$

Langage *non-strict*

= toutes les fonctions peuvent être non-strictes

⇒ une fonction peut produire un résultat même si un argument n'est pas défini

Différentes sortes de langages non-stricts:

- Langages paresseux
- Langages indulgents

5

Langage paresseux

Forme la plus connue de langage non-strict

En fait:
souvent considérée comme la seule forme

Stratégie de mise en oeuvre:

= *appel-par-nécessité*

= gel/dégel des arguments + mémorisation

⇒ Une expression est évaluée *seulement si* sa valeur est *requise* pour produire le résultat de la fonction

6

Langage indulgent

Langage non-strict . . . mais *non-paresseux*

Pas de gel/dégel des arguments:

⇒ un argument est *toujours* évalué

Pas de contraintes *fixes* d'ordonnancement

= un argument *n'a pas* à être évalué avant l'appel

Stratégie parallèle de mise en oeuvre:

- fonction *et* arguments évalués *en parallèle*

Stratégie séquentielle aussi possible:

- cf. K. Traub 1988, 1989

7

Figure sur les diverses classes de langages fonctionnels et leur pouvoir expressif

7-1

2. À quoi sert l'évaluation non-stricte *sans* paresse?

2.1 Évaluation non-stricte et récursivité

Récursivité: courante pour les fonctions et types

```
> -- Valeurs fonctionnelles
> fac 0 = 1
> fac x = x * fac (x-1)

> -- Types
> data ListeI  =  NilN  |  ConsN Int ListeI
```

Pour les valeurs non-fonctionnelles?

```
> desUns  =  ConsN 1 desUns
```


Figure montrant mise en oeuvre de `desUns`

2.2 Structures récursives non-strictes

Graphes cycliques

Structures non-cycliques

Structure récursive non-stricte et non-cyclique:

```
> deux2 = (2, fst deux2)
```

Caractéristique:

= ∃ dépendance entre composante et résultat

Programmation dynamique

Liste des `n` premiers nombres de Fibonacci:

```
> fib_liste n =
>   let
>     l = 1 : 1 : [l!!(i-1) + l!!(i-2) | i <- [2..n]]
>   in
>     l
```

2.3 Programmes *circulaires*

"Using Circular Programs to Eliminate Multiple Traversals of Data" (R. Bird 1984)

Recherche des éléments supérieurs à la moyenne:

```
>  trouverSupMoyenne [] =
>    error "La liste ne doit pas etre vide."
>  trouverSupMoyenne xs =
>     let
>       (laMoyenne, listeSupM)
>                  = trvSupMoyenne laMoyenne xs 0 0 []
>     in
>       listeSupM
>
>  trvSupMoyenne laMoyenne [] total nb listeSupM =
>     (total `div` nb, listeSupM)
>  trvSupMoyenne laMoyenne (x : xs) total nb listeSupM =
>     let
>       listeSupM' = if x > laMoyenne then (x : listeSupM)
>                                     else listeSupM
>     in
>       trvSupMoyenne laMoyenne xs
>                     (total + x) (nb + 1) listeSupM'
```


3. À quoi sert l'évaluation paresseuse?

Paresse ⇒ une expression est évaluée *seulement si* elle est absolument nécessaire à la production du résultat

3.1 Structures de données infinies

```
>  aPartirDe n = n : (aPartirDe (n+1))
>
>  naturels = aPartirDe 0
>  pairs = [2 * n | n <- naturels ]
>
>  estPremier n = ...
>  premiers = [ n | n <- naturels, estPremier n ]
>
>  take 1000 premiers
```

Intérêt: Problèmes de fouille

Auparavant, il faut souligner que l'exécution, dans un langage indulgent, des programmes qui suivent pourrait, si la mise en oeuvre est *équitable**, produire les résultats désirés. Par contre, des calculs non-nécessaires, possiblement sans fins dans le cas de structures infinies, pourraient être générés. En d'autres mots, les deux approches produiraient le même résultat mais l'approche indulgente utiliserait, à cause des structures infinies, une quantité non-bornée de ressources.

†

*Informellement, une mise en oeuvre est équitable ("*fair*") si elle assure le progrès constant de toute expression prête à s'évaluer.

†Bien que cela puisse sembler une approche avec *spéculation*, ceci ne l'est pas plus qu'une approche d'évaluation stricte, qui elle aussi évalue tous les arguments effectifs.

3.2 Modularité

Racine carrée par Newton-Raphson (J. Hughes)

```
>  prochaineApprox x a = (a + (x / a)) / 2
>
>  genererApproximations x0 =
>     let  premiereApprox = (x0 + 1) / 2
>          approxSuivantes
>                 = map (prochaineApprox x0) approximations
>          approximations
>                 = premiereApprox : approxSuivantes
>     in   approximations
>
>  approxAcceptable eps (a0 : a1 : rest) =
>        if abs (a0 - a1) <= eps
>          then a1
>          else approxAcceptable eps (a1 : rest)
>
>  racineCarre x eps =
>        approxAcceptable eps (genererApproximations x)
```

Fouille parmi une suite (potentiellement) *infinie* d'approximations

Modularité = Critère de sélection défini par fonction `approxAcceptable`

3.3 Structures de données à coûts amortis

Travaux récents de C. Okasaki (1995, 1996): structures de données purement fonctionnelles avec coûts amortis intéressants

⇒ permet de borner le coût d'une *séquence* d'opérations

3.4 Mécanisme d'exception

```
> diviser x 0 err = err
> diviser x y = x / y
>
> r = diviser a b (error "Division par 0")
```


3.5 Lorsqu'un langage paresseux est utilisé, la paresse est-elle nécessaire?

- Programmes circulaires de Bird et Allison: indulgent OK *(sauf un programme)*!
- Programmes de Sekar et al. pour *ee-* et *dd-analysis*: indulgent (ou strict) OK!
- Programmes de Hartel et al. ("*Benchmarking Implementations of Lazy Functional Languages*"): paresse parfois utilisée mais de façon limitée

La paresse dans les programmes de Hartel:

- Listes infinies de nombres aléatoires ou de constantes

  ⇒ construction explicite de *Streams*
- Sélecteur à l'extérieur d'une condition:

```
> let
>    x = head l
> in
>    if null l
>       then ...
>       else ... x ...
```

  ⇒ Déplacement du sélecteur
- Analyse syntaxique via *combinator parsing*:

  Nécessite vraiment la paresse

  ⇒ seulement deux (2) programmes pas traduits!

4. Parallélisme dans l'évaluation stricte vs. non-stricte: Exemples

Quantité de parallélisme (du − au + parallèle)

- Paresseux: parallélisme très faible même avec optimisations (5–10 instructions)
- Strict: parallélisme limité pcq. évaluation stricte impose séquentialité (synchronisation de type *barrière*)
- Indulgent: grande quantité de parallélisme!
  - Évaluation en parallèle des arguments et du corps de la fonction
  - Synchronisation implicite producteur/consommateur

Parallélisme de pipeline (prod./consom.)

Exemple:

```
map f [] = []
map f (x : xs) = (f x) : (map f xs)

double x = 2*x;
inc x = x+1;

resultat = map double (map inc [1..100])
```

Strict:

- On construit `[1..100]` puis ...
- On applique `inc` aux éléments de `[1..100]` puis ...
- On applique `double` au résultat

Paresseux pur:

- Alternance (style co-routine) entre exécution de `inc` et `double`

Indulgent:

- Exécution en parallèle (style **pipeline**) de `inc` et `double`
- Synchronisation (implicite) lorsqu'un élément n'est pas disponible

Parallélisme spatial

Exemple:

```
data arbre = Feuille Int | Noeud arbre arbre

feuillesDe a = feuillesAccum a [];

feuillesAccum (Feuille i) feuilles
  = i : feuilles
feuillesAccum (Noeud gauche droit) feuilles
  = feuillesAccum gauche (feuillesAccum droit feuilles)
```

Parallélisme:

- Strict ⇒ Le parcours de l'arbre `droit` doit se terminer *avant* que le parcours de l'arbre `gauche` puisse débuter.
- Indulgent ⇒ Les arbres `gauche` et `droit` peuvent être parcourus en parallèle.

Parallélisme et exécution non-déterministe

= Haskell indulgent + *I/M-structures*

I-structures = *Incremental structures*

≈ variables *logiques* (affectation ≈ unification)

⇒ une seule écriture

⇒ sans transparence référentielle mais déterministe

M-structures = *Mutable structures*

≈ variables impératives avec synchronisation implicite

⇒ Possibilité de plusieurs écritures mais, à chaque fois, le champ doit être vide (vidé par lecture)

⇒ non-déterministe: si plusieurs lecteurs en attente, alors un seul est réactivé

Exemple: Problème de l'histogramme

```
data Arbre = Feuille Int | Noeud Arbre Arbre

histogramme a n =
   let
      histo = Marray (1, n) [ (j, 0) | j <- [1..n] ]
      seq traverser a histo
          resultat = histo
   in
       resultat


traverser (Feuille n) histo =
   let
      histo![n] = histo![n] + 1
   in
      ()
traverser (Noeud gauche droit) histo =
   let
      traverser gauche histo
      traverser droit histo
   in
      ()
```

- Solution parallèle (parcours des sous-arbres)
- Localement non-déterministe mais globalement déterministe

5. Comment, dans un contexte parallèle, peut-on modéliser formellement les différentes formes d'évaluation?

Modèle formel *opérationnel* pour mieux comprendre les différents modes d'évaluation

Particularité: contexte d'évaluation *parallèle*

⇒ aucune contrainte séquentielle d'exécution
... sauf si absolument nécessaire

Notation utilisée: LOTOS

5.1 Évaluation non-paresseuse

Gestion de l'environnement (bloc d'activation)

Trois types de requêtes:

- `demander` un argument;
- `obtenir` un argument *après* l'avoir demandé
- `definir` un argument

```
process GererEnv[definir, demander, obtenir]
                (env : Env) : exit :=
  demander ?i : Nat [estDefini(env, i)];
  obtenir !i !env[i];
  GererEnv[definir, demander, obtenir](env)
  []
  definir ?i : Nat ?v : Valeur [not estDefini(env, i)];
  GererEnv[definir, demander, obtenir](definir(env, i, v))
  []
  exit
endprocess
```

Évaluation stricte

Évaluation d'une expression (haut niveau)

```
process Eval[entree, sortie] : noexit :=
  entree ?e : Expr ?fenv : FEnv;
  hide definir, demander, obtenir in
     GererEnv[definir, demander, obtenir](creer(0))
     |[demander, obtenir]|
     EvalExpr[demander, obtenir, sortie](e, fenv)
  >>
  Eval[entree, sortie]
endproc
```

Évaluation d'un argument effectif

```
process EvaluerArg[demander, obtenir, definirArg]
                (e : Expr, i : Nat, fenv : FEnv) : exit :=
   hide sortie in
    EvalExpr[demander, obtenir, sortie](e, fenv)
    |[sortie]|
    sortie ?v : Valeur;  definirArg !i !v;  exit
endprocess
```

Évaluation stricte

Évaluation d'une expression

```
process EvalExpr[demander, obtenir, sortie]
                (e : Expr, fenv : FEnv) : exit :=
  ...
  [] [sorte(e) == Variable] ->
     demander !variable(e);
     obtenir !variable(e) ?v : Valeur;
     sortie !v; exit
  ...
  [] [sorte(e) == Appel] ->
     let args = arguments(e) in
     let n = longueur(args) in
     hide definirArg, demanderArg, obtenirArg in
       GererEnv[definirArg, demanderArg, obtenirArg]
               (creer(n))
       |[definirArg, demanderArg, obtenirArg]|
       (EvaluerArgs[demander, obtenir, definirArg]
                   (args, fenv, 0)
        >>
        EvalExpr[demanderArg, obtenirArg, sortie]
                (corps(fenv, fonction(e)), fenv)
        )
endprocess
```

Évaluations des arguments effectifs

```
process EvaluerArgs
        [demander, obtenir, definirArgs]
        (args: ArgsEffectifs, fenv: Fenv, k: nat) : exit :=
  [k ~= longueur(args)] ->
     EvaluerArg[demander, obtenir, definirArgs]
               (args[k], k, fenv)
     |||
     EvaluerArgs[demander, obtenir, definirArgs]
                (args, fenv, k+1)
  []
  [k = longueur(args)] ->
     exit
endprocess
```

Évaluation indulgente

Appel de fonction:

```
process EvalExpr[demander, obtenir, sortie]
                (e : Expr, fenv : FEnv) : exit :=
  ...
  [] [sorte(e) == Appel] ->
     let args = arguments(e) in
     let n = longueur(args) in
     hide definirArg, demanderArg, obtenirArg in
       GererEnv[definirArg, demanderArg, obtenirArg]
               (creer(n))
       |[definirArg, demanderArg, obtenirArg]|
       (EvaluerArgs[demander, obtenir, definirArg]
                   (args, fenv, 0)
        |||
        EvalExpr[demanderArg, obtenirArg, sortie]
                (corps(fenv, fonction(e)), fenv)
       )
endprocess
```

⇒ évaluation des arguments et du corps de la fonction *en parallèle* (avec synchronisation)

5.2 Évaluation paresseuse

Appel de fonction

```
  []
  [sorte(e) == Appel] ->
     let args = arguments(e) in
     let n = longueur(args) in
     hide definirArg, demanderArg, obtenirArg in
       GererEnv[demander, obtenir, definirArg,
                demanderArg, obtenirArg]
               (args, creer(n), fenv)
       |[demanderArg, obtenirArg]|
       EvalExpr[demanderArg, obtenirArg, sortie]
               (corps(fenv, fonction(e)), fenv)
```

Gestion de l'environnement

```
process GererEnv[demanderNonlocal, obtenirNonlocal,
                 definir, demander, obtenir]
                (args : ArgsEffectifs, env : Env,
                 fenv : FEnv) : exit :=
  demander ?i : Nat;
  ([estDemande(env, i)] ->
      GererEnv[demanderNonlocal, obtenirNonlocal, definir,
               demander, obtenir]
              (args, n, fenv)
   []
   [! estDemande(env, i)] ->
     EvaluerArg[demanderNonlocal, obtenirNonlocal, definir]
               (args[i], i, fenv)
     |[definir]|
     GererEnv[demanderNonlocal, obtenirNonlocal, definir,
              demander, obtenir]
             (args, demander(env, i), fenv)
   )
  []
  obtenir ?i : Nat !env[i] [estDefini(env, i)];
  GererEnv[demanderNonlocal, obtenirNonlocal, definir,
           demander, obtenir]
          (args, env, fenv)
  []
  definir ?i : Nat ?v : Valeur [not estDefini(env, i)];
  GererEnv[demanderNonlocal, obtenirNonlocal, definir,
           demander, obtenir]
          (args, definir(env, i, v), fenv)
endprocess
```

Quelques remarques:

- L'approche indulgente est non-stricte ... mais très semblable à l'approche stricte
  ⇒ plus simple que la paresse
- L'approche paresseuse est plus complexe à cause du gel/dégel des expressions
  ⇒ utilisation de fermetures
  ⇒ moins de parallélisme: évaluation amorcée au dernier moment
- Modèle opérationnel pcq. différence entre indulgence et paresse se situe à ce niveau (gestion des ressources)

6. Conclusion

- Un langage peut être non-strict sans être paresseux (= indulgent)
- Langage indulgent:
  - Moins expressif que langage paresseux !
  - Mieux approprié pour programmation parallèle ?!
- Travail en cours:
  - Compilateur pour `pH`
  - Problème difficile = Partitionnement
    Indulgence ⇒ Parfois *trop* de parallélisme
    * Réduction via aggrégation en *thread*s
    * Mécanismes d'étranglement, d'amortissement

Partitionnement

= Regrouper les instruction en *threads* pour réduire les coûts de synchronisation

Thread = séquence d'instructions (ordonnées statiquement) qui, lorsqu'amorcée, s'exécute sans interruption ou synchronisation

⇒ synchronisation dynamique seulement *entre* les *threads*

Nécessité de l'ordonnancement dynamique

- Communication interprocesseurs
- Indulgence ⇒ contexte dynamique peut influer sur l'ordre des instructions:

```
f u v = (u*u, v+v)
g z = let (s, t) = f z s in t
h z = let (s, t) = f t z in s
```