Programmation parallèle et langages fonctionnels: De la paresse à l'indulgence

Guy Tremblay
Dépt. d'informatique
UQAM

15 mars 1996

0. Aperçu de la présentation

1. Évolution des architectures parallèles
2. Modèles de programmation parallèle
3. Langages fonctionnels
4. Langages fonctionnels non-stricts vs. paresseux
5. Parallélisme: Langages fonctionnels *purs* vs. *impurs*
6. Conclusion

1. Évolution des architectures parallèles

1.1 Intuition

Approches pour obtenir une grande puissance de calcul:

1. Super-ordinateur
2. Machine parallèle

Super-ordinateur:

- Petit nombre de processeurs très performants
- Circuits hautement optimisés:
  ⇒ Processeur très coûteux!
- Généralement pas d'usage général, e.g., machines vectorielles.

Machine parallèle:

- Machine avec un *grand* nombre de processeurs
  ⇒ Grande puissance de calcul
- Utilise des processeurs *simples*
  ⇒ Processeurs peu coûteux
  ⇒ Profite de l'augmentation générale des performances des processeurs

1.2 Aperçu de quelques architectures parallèles

(A) Machine MIMD avec bus et mémoire partagée
(B) Machine MIMD avec réseau et mémoire distribuée
(C) Machine multi-contextes

(A) Machine MIMD avec bus et mémoire partagée

Caractéristiques/problèmes:

+ Problème de la cohérence des caches facilement résolu (*snoopy bus*)
− Ne permet pas le parallélisme massif: ≈ 30 processeurs

(B) Machine MIMD avec réseau et mémoire distribuée

Caractéristiques/problèmes:

+ Permet + grand parallélisme (transactions multiples sur le réseau)
0 Problème de la cohérence des caches plus complexe mais résoluble
– Temps de latence mémoire imprévisible
⇒ que faire en cas de faute de cache?

(C) Machine multi-contexte (*multi-threaded*)

Caractéristiques:

- Machine MIMD avec réseau et mémoire distribuée
- Utilise des processeurs multi-contextes:
  - Contexte (*thread*) = IP + FP + registres
  - Conserve plusieurs *contextes indépendants*
    ⇒ plusieurs instructions prêtes à s'exécuter
  - Changements de contexte rapides et peu coûteux
    ⇒ changement en cas de faute de cache

Avantages:

- Meilleure tolérance de latence mémoire
- Meilleur support du parallélisme de granularité fine

2. Modèles de programmation parallèle

2.1 Caractéristiques d'un langage *parallèle* de haut niveau

Aspects à considérer pour la programmation d'une machine parallèle:

- Répartition/placement des données
- Division du travail entre les processeurs
- Gestion des communications entre les processeurs
- Synchronisation

Caractéristiques recherchées:

- Devrait isoler le programmeur de l'architecture
- Devrait fournir un modèle de mémoire simple
- Devrait être facile de raisonner à propos du programme

2.2 Quelques approches de programmation parallèle

Langages séquentiels traditionnels:

- Utilisation d'un compilateur *parallélisant*

⇒ FORTRAN (*dusty decks*) & C

Parallélisme de contrôle:

- Instructions de contrôle explicites: `fork/join`, `task/rendez-vous`

⇒ Processus séquentiels communicants

Parallélisme de données:

- Manipulation parallèle de structures (homogènes)
  - `FORALL I = 1, 100 DO  A[I] = B[I] + C[I]`
  - `A[1:100] = B[1:100] + C[1:100]`

⇒ Modèle SPMD

Langages fonctionnels:

- *Naturellement* parallèle = parallélisme *implicite*
  ⇒ parallélisme = option par défaut
- Modèle logique = mémoire partagée
- Langage pur ⇒ déterministe
- Transparence référentielle ⇒ raisonnement "équationnel"

3. Langages fonctionnels

3.1 Intuition

Langage *purement* fonctionnel = Basé uniquement sur les notions de valeur et fonction

Parallélisme implicite:

- Ordre *partiel* d'exécution basé sur les dépendances de données

Exemple: Calcul des racines $\frac{-b \pm \sqrt{b^2-4ac}}{2a}$

```
racines a b c = (r1, r2)
                where disc = sqrt( b*b - 4 * a * c )
                      r1   = (-b + discr) / (2 * a)
                      r2   = (-b - discr) / (2 * a)
```


Transparence référentielle:

- Absence d'effets de bord
- Substitutivité = Une sous-expression peut être remplacée par n'importe quelle autre ayant la même valeur
- *Definiteness* = Toutes les occurences d'un identificateur dénotent la même valeur

Exemple:

```
r = x + x
      where x = f a

r = (f a) + (f a)
```

Implications:

- Raisonnement équationnel
- Propriété de confluence:
  ⇒ L'ordre d'évaluation ne change pas le résultat
  ⇒ N'importe quel ordre parallèle est valide

3.2 Caractéristiques des langages modernes

Fonctions d'ordre supérieur:

= Fonctions passées en paramètre, retournées comme résultat, rangées dans des structures de données, etc.
⇒ *citoyens de première classe*

- Permet de combiner des fonctions pour en produire de nouvelles

Exemple: Application partielle et composition de fonctions:

```
plus x y = x + y
inc1 x = plus x 1
inc2 = plus 2
inc3 = inc2 . inc1
```

Exemple: Patrons de récursion et filtrage:

```
fold op unite [ ] =
   unite
fold op unite (x : xs) =
   op x (fold op unite xs)

somme = fold (+) 0
produit = fold (*) 1
min = fold (<) +inf
```


Types algébriques et polymorphisme

= Types *génériques* récursifs (sans pointeurs) avec fonctions *polymorphes*

Exemple: Arbre binaire et calcul du nombre de feuilles

```
data Arbre T =
   Feuille T | Noeud (Arbre T) (Arbre T)

nbFeuilles (Feuille x) =
   1
nbFeuilles (Noeud a1 a2) =
   (nbFeuilles a1) + (nbFeuilles a2)
```

Inférence de types

= Vérifications statique des types
... mais les déclarations de type sont optionnelles

3.3 Les différentes classes de langages fonctionnels

Trois (3) grandes classes:

- Strict = appel-par-valeur
- Paresseux = appel-par-nécessité
- Indulgent = non-strict ... mais non-paresseux

Exemple:

```
let
   f x y = if x == 0 then 0 else y
   boucle x = x + boucle (x+1)
in
   f 0 (boucle 0)
```

Résultats de l'évaluation de "`f 0 (boucle 0)`":

- Strict $\Rightarrow \perp$
- Paresseux $\Rightarrow 0$
- Indulgent $\Rightarrow 0$ ... mais génère un calcul infini

Exemple:

```
listeDeUns = 1 : listeDeUns
```

Résultat pour `listeDeUns`:

- Strict $\Rightarrow \perp$
- Paresseux $\Rightarrow$ Liste circulaire/cyclique
- Indulgent $\Rightarrow$ Liste circulaire/cyclique

4. Langages fonctionnels non-stricts vs. paresseux

4.1 Langage strict vs. non-strict

Définition formelle de la stricticité

Soit f une fonction à un (1) argument.

On dit que f est strict si et seulement si:

$$f \perp = \perp$$

Informellement: f est stricte lorsqu'aucun résultat ne peut être produit tant que la valeur de l'argument n'est pas connue.

Langage fonctionnel strict

Langage strict $\Rightarrow$ Mise en oeuvre assure que *toutes* les fonctions sont toujours strictes en tous leurs arguments (sauf `if ... then ... else ...`)

$\Rightarrow$ Appel par valeur

Langage fonctionnel non-strict

Langage non-strict $\Rightarrow$ Une fonction peut retourner un résultat (partiel ou complet) *avant* que les arguments soient connus/reçus

4.2 Intuitions de mise en oeuvre

Paresse

Utilisation de l'*appel par nécessité* (*call-by-need*):

- L'évaluation des arguments actuels est *gelée*
  $\Rightarrow$ les expressions sont encapsulées dans des *glaçons*
- Dans la fonction, une référence à un argument formel *dégèle* le glaçon associé
  $\Rightarrow$ l'expression associée est évaluée (et *mémoïsée*)

Donc: le corps de la fonction est évalué *avant* les arguments

Implication:

Seules les expressions qui sont absolument requises pour la production du résultat sont évaluées

Indulgence

Mise en oeuvre **séquentielle**: aucun ordre fixe n'est imposé pour l'évaluation des arguments vs. le corps de la fonction.

Contrainte à respecter:

> Si un argument devient requis durant l'évaluation de la fonction, alors on doit s'assurer que sa valeur est disponible avant de poursuivre ($\Rightarrow$ changement de contexte)

Mise en oeuvre **parallèle**: les arguments et le corps de la fonction sont évalués *en parallèle*.

Contrainte à respecter:

> Respect des dépendances de données $\Rightarrow$ l'évaluation du corps de la fonction doit être suspendue lorsqu'un argument n'est pas encore disponible.

Implication:

> Certaines expressions non-requises peuvent être évaluées

4.3 Pouvoir expressif de la paresse et l'indulgence

Structures de données non-stricts

Exemple: Constructeurs non-stricts

```
let
   uns = 1 : uns
in
   uns
```

```
let
   a = (2, fst a)
in
   a
```


Exemple: Graphes (cycliques ou non)

```
data Graphe T = Sommet T [(Graphe T)]

sommets = [
   Sommet "S0" [sommets!!1, sommets!!2],
   Sommet "S1" [sommets!!3],
   Sommet "S2" [sommets!!0, sommets!!3],
   Sommet "S3" []
]
leGraphe = head sommets
```


Programmation dynamique

Exemple: Tableau des n premiers nombres de Fibonacci

```
fib_array n =
   let
      A = Array (0, n) (
            (0 := 1) :
            (1 := 1) :
            [(i := A(i-1)+A(i-2)) | i <- [2..n]]
          )
   in
      A
```

Programmes circulaires

Exemple: Recherche du maximum d'une liste et des nombres inférieurs d'au plus 10% du maximum

```
trouverMaxs xs =
   let
      (leMax, presqueMaxs) =
         trouverMaxs' leMax xs minInt []
   in
      (leMax, presqueMaxs)


trouverMaxs' leMax [] maxCourant presqueMaxs =
   (maxCourant, presqueMaxs)
trouverMaxs' leMax (x : xs) maxCourant presqueMaxs =
   let
      nouveauMax = max(maxCourant, x)
      presqueMaxs' = if (x >= 0.9*leMax)
                        then (x : presqueMaxs)
                        else presqueMaxs
   in
      trouverMaxs' leMax xs nouveauMax presqueMaxs'
```


4.4 Pouvoir expressif de la paresse

Structures de données *potentiellement infinies*

Exemple: Les nombres naturels

```
aPartirDe n = n : (aPartirDe (n+1))

naturels = aPartirDe 0

pairs = [ 2 * n | n <- naturels ]
```

Exemple: Les k premiers nombres premiers

```
estPremier n = ... -- Détermine si n est premier ou non

nbPremiers = [ n | n <- naturels, estPremier n ]

take k nbPremiers
```


Exemple: Calcul de la racine carrée d'un nombre via séquence d'approximations successives

```
prochaineApprox x a =
   (a + (x / a)) / 2

genApproxs x =
   let
      premier = (x + 1) / 2
      suivants = map (prochaineApprox x) approxs
      approxs = premier : suivants
   in
      approxs

within eps (a0 : a1 : rest) =
   if abs(a0 - a1) <= eps
      then a1
      else within eps (a1 : rest)

sqrt x eps =
   within eps (genApproxs x)
```


Structures de données à coûts amortis

Exemple: File FIFO

```
data File a = File Int [a] Int [a]
   (* Invariant(File lnAv av lnArr arr) =
         lnAv = |av| ∧ lnArr = |arr| ∧ lnAv ≥ lnArr *)

fileVide =
   File 0 [] 0 []

estVide (File lnAv av lnArr arr) =
   (lnAv == 0)

ajouter x (File lnAv av lnArr arr) =
   mkFile lnAv av (lnArr+1) (x : arr)

retirer (File (lnAv+1) (x : av) lnArr arr) =
   (x, mkFile lnAv av lnArr arr)

mkFile lnAv av lnArr arr
   | lnArr <= lnAv =
      File lnAv av lnArr arr
   | lnArr == lnAv + 1 =
      File (lnAv + lnArr) (av ++ reverse arr) 0 []
```

5. Parallélisme: Langages fonctionnels *purs* vs. *impurs*

5.1 Parallélisme et langage fonctionnel *pur*

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

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]`
- On applique `inc` à chacun des éléments de `[1..100]`
- On applique `double` à chacun des éléments du résultat produit par `map inc [1..100]`

Paresseux pur:

- Alternance (style co-routine) entre exécution de `inc` et `double` (tampon de taille 1 entre `inc` et `double`)

Indulgent:

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

Parallélisme spatial

Exemple:

```
data arbre = Feuille Integer | Noeud arbre arbre

feuillesDe a = feuillesAccum a [];

feuillesAccum (Feuille i) feuilles =
   i : feuilles
feuillesAccum (Noeud g d) feuilles =
   feuillesAccum g (feuillesAccum d 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 gauches et droits peuvent être parcourus en parallèle.

5.2 Programmation fonctionnelle *impure* avec pH

pH = *parallel Haskell*
= Haskell indulgent + *I/M-structures* du langage Id

Langage Id (MIT) = Langage à 3 niveaux

- Niveau purement fonctionnel
  - évaluation indulgente
  - tableaux et boucles
- Niveau avec *I-structures*
  ≈ variables *logiques* (affectation ≈ unification)
  ⇒ sans transparence référentielle mais déterministe
- Niveau avec *M-structures*
  ≈ variables *impératives* avec mécanisme implicite de synchronisation
  ⇒ non-déterministe

I-structures: Structures de données incrémentales

I-structures = *Incremental structures*

- Permettent la création de structures de données *partiellement définies* (i.e., avec des champs vides)
- L'accès à un champ vide entraine la suspension/ mise en attente du consommateur
  ⇒ résultat déterministe

Exemple: Mise en oeuvre des listes cycliques

Définition cyclique:

```
let
   uns = 1 : uns
in
   uns
```

Mise en oeuvre avec *I-structures*:

```
let
   uns = Iarray(1, 2) []
   uns![1] = 1
   uns![2] = uns
in
   uns
```


Exemple: Mise en oeuvre parallèle des tableaux

```
mkArray n initF =
   Array (0, n) [(i := initF i) | i <- [0..n]]
```

```
-- Mise en oeuvre avec I-structures + boucle
mkArray n initF =
   let
      A = Iarray(0, n) [];
      for i <- [0..n] do
            A![i] = initF i
         finally ()
   in
      A
```


M-structures: Effets de bord et non-déterminisme

M-structures = *Mutable structures*

- Données *mutables* ≈ variables impératives
- Synchronisation intégrée

- Un champ mutable peut être lu et/ou écrit *plusieurs* fois mais:
  - Lecture (`mfetch`)
    - Pré-condition: Le champ ne doit pas être vide, sinon mise en attente du lecteur
    - Post-condition: Le champ est vide
  - Écriture (`mstore`)
    - Pré-condition: Le champ doit être vide
    - Post-condition: Si lecteurs en attente, l'un d'eux est réactivé

  ⇒ Lecture/écriture atomique
  ⇒ Synchronisation implicite

Exemple: Problème de l'histogramme

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

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

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

Caractéristiques:

- Solution parallèle
- Localement non-déterministe (ordre des incréments) mais globalement déterministe (totaux cumulatifs)

⇒ Règle générale, l'obtention d'un résultat déterministe dépend du programme, non du langage

6. Conclusion

- Si les langages fonctionnels deviennent un jour populaires pour les machines parallèles (?) ce sera par l'utilisation de langages indulgents

- Des éléments *impurs* devront probablement être incorporés ou adaptés
  - Gestion des ressources
  - Traitement des E/S
  - Annotations pour répartition des données

Travaux en cours et futurs

- Exploration des limites des langages indulgents vs. paresseux

- Mise en oeuvre de pH sur architecture EARTH (*Efficient Architecture for Running THreads*) du prof. Gao à McGill

  Problème majeur
  ≠ Extraction du parallélisme
  = Partitionnement/regroupement des instructions en *threads* plus longs