Mise en oeuvre parallèle
des langages fonctionnels paresseux:

Une approche basée sur la
propagation abstraite des demandes

Guy Tremblay
Dépt. d'informatique
UQAM

21 novembre 1995

1

0. Aperçu de la présentation

1. Introduction: Langages fonctionnels et programmation parallèle
2. Notions de base
   - 2.1 Langage paresseux et analyse de stricticité
   - 2.2 Interprétation abstraite
3. Propagation abstraite des demandes
   - 3.1 Intuition: Calcul de l'inverse
   - 3.2 Langage source
   - 3.3 Propagation *exacte* des demandes
   - 3.4 Propagation *abstraite* des demandes
4. Aperçu du compilateur
5. Résultats expérimentaux
6. Conclusion

2

1. Introduction

Langage purement fonctionnel = Basé sur les notions de valeur et fonction, sans notion de variable/effet de bord (λ-$calcul$)

Paradigme intéressant pour la programmation des machines parallèles:

- Naturellement parallèle (ordre *partiel* implicite)
- Déterministe (confluence)

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

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

3

Trois (3) classes de langages fonctionnels:

- Strict (appel-par-valeur);
- Paresseux (appel-par-nécessité);
- Indulgent (*Lenient* = non strict mais non paresseux).

Exemple:

```
f x y = If x == 0 Then 0 Else y;
boucle x = x + boucle (x+1);
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

4

Exemple:

```
listeDeUns = 1 : listeDeUns
```

Résultat pour `listeDeUns`:

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

4-1

Caractéristiques de l'évaluation paresseuse:

- Evalue uniquement ce qui est absolument nécessaire.
- Permet la manipulation de structures de données *potentiellement infinies* (produit seulement la partie nécessaire pour le calcul du résultat final).

Exemple:

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

naturels = aPartirDe 0;
pairs = [ 2 * n | n <- naturels ];

estPremier n = ...(* Détermine si n est premier ou non *)
nbPremiers = [ n | n <- naturels; estPremier n ];
nbPremiers!4
```

5

Motivation initiale de ma recherche:

Programmation scientifique (tableaux et boucles) avec langage fonctionnel indulgent (Id) sur machine parallèle à flux de données (TTDA)

$\Rightarrow$ *trop* de parallélisme

$\Rightarrow$ utilisation de mécanismes d'étranglement du parallélisme (*throttling mechanism*).

Programmation avec structures de données *dynamiques* dans langage *paresseux* sur ces mêmes architectures:

Suffisamment de parallélisme? ... Ou trop peu?

6

2. Notions de base

2.1 Langage paresseux et analyse de stricticité

Désavantage majeur des langages paresseux: Mise en oeuvre coûteuse dûe à la création et manipulation de *suspensions* (ou graphes).

$\Rightarrow$ **Beaucoup de travail d'être paresseux!**

Exemple:

```
f x y = If x == 0 Then 0 Else y;
f e1 e2
```

Mise en oeuvre naïve avec suspensions:

```
f x y = If (force x) == 0 Then 0 Else (force y);
f (delay e1) (delay e2)
```

7

Analyse de stricticité = Détermine, à la compilation, les expressions qui sont *sûres* d'êtres évaluées.

Permet l'élimination de certaines suspensions: Si l'expression est sûre d'être évaluée dans la fonction, alors on peut l'évaluer *avant* l'appel.

Exemple:

Sans analyse de stricticité:

```
f x y = If (force x) == 0 Then 0 Else (force y);
f (delay e1) (delay e2)
```

Avec analyse de stricticité:

```
f x y = If x == 0 Then 0 Else (force y);
f e1 (delay e2)
```


2.2 Interprétation abstraite

Interprétation abstraite = Stratégie d'analyse de programmes basée sur une *abstraction* de la sémantique du langage.

Intuition: Les calculs définis par le programme sont effectués sur un domaine *simplifié* (abstrait).

Impact de l'utilisation d'un domaine abstrait:

- Avantage: Les calculs sont *plus simples* et peuvent être réalisés (au moment de la compilation) en un temps fini (terminaison assurée).
- Désavantage: Entraîne certaines *approximations*.

Exemple: La règle des signes

Quel est le signe ($+, -$ ou 0) des valeurs représentées par les expressions suivantes?

$0 * 999$

$-123 * -134$

$1234567890123 * 1234567890123$

$-123 + -432432342$

$1234567890123 + 1234567890123$

$1234567890123 + -1234567880123$

Interprétation abstraite de la règle des signes:

Domaine abstrait:

$\mathcal{S} = \{+, -, 0, ?\}$

Fonction d'abstraction:

$\overline{\ } : \mathbf{Z} \to \mathcal{S}$

$\overline{x} = 0$ si $x = 0$
$\overline{x} = +$ si $x > 0$
$\overline{x} = -$ si $x < 0$

Abstraction des opérations:

$\overline{*}$	0	$+$	$-$	$?$
0	0	0	0	0
$+$	0	$+$	$-$	$?$
$-$	0	$-$	$+$	$?$
$?$	0	$?$	$?$	$?$

$\overline{+}$	0	$+$	$-$	$?$
0	0	$+$	$-$	$?$
$+$	$+$	$+$	$?$	$?$
$-$	$-$	$?$	$-$	$?$
$?$	$?$	$?$	$?$	$?$

Exemple:

$$\begin{array}{lcl} \overline{123 * -3434} & = & \overline{123} \,\overline{*}\, \overline{-3434} \\ & = & + \,\overline{*}\, - \\ & = & - \end{array}$$

$$\begin{array}{lcl} \overline{123 + 12343454} & = & \overline{123} \,\overline{+}\, \overline{12343454} \\ & = & + \,\overline{+}\, + \\ & = & + \end{array}$$

$$\begin{array}{lcl} \overline{123 + -5423} & = & \overline{123} \,\overline{+}\, \overline{-5423} \\ & = & + \,\overline{+}\, - \\ & = & ? \end{array}$$

Analyse de stricticité et interprétation abstraite

Approche courante pour l'analyse de stricticité:

- Définition: f est stricte ssi
$$\forall x \cdot x = \bot \Rightarrow f\ x = \bot$$
- Domaine abstrait $= \{0,1\}$ tel que
$$\begin{array}{lcll} \overline{x} & = & 0 & \text{si } x = \bot \\ \overline{x} & = & 1 & \text{si } x \neq \bot \end{array}$$
- Propriété de l'interprétation abstraite de la sémantique standard:
$$\overline{f}\ 0 = 0 \ \Rightarrow\ f\ \bot = \bot \ \ (f \text{ est stricte})$$
- Sûre mais approximative = pas de danger de faire des optimisations illégales:
$$f \text{ est stricte } \not\Rightarrow\ \overline{f}\ 0 = 0$$
- Avantage: Applicable aux fonctions d'ordre supérieur.
- Désavantage majeur: Résultats peu précis dans le cas des structures de données complexes, e.g., listes, arbres, etc.

3. Propagation abstraite des demandes

= Forme d'analyse de stricticité.

- Point de vue différent (vers l'arrière) de la stricticité d'une fonction: f est stricte ssi
$$\forall x \cdot f\ x \neq \bot \Rightarrow x \neq \bot$$
plutôt que
$$\forall x \cdot x = \bot \Rightarrow f\ x = \bot$$
- Interprétation abstraite ... d'une sémantique *non-standard*: Détermine les données requises pour produire le résultat demandé.
- Validité de la sémantique non-standard (*pas de la science-fiction!*): calcule une approximation de l'inverse d'une fonction.
- Approche naturelle pour une mise en oeuvre utilisant des graphes de flux de données.

3.1 Intuition: Calcul de l'image inverse

Formellement: f est *stricte* ssi
$$\forall x \cdot x = \bot \Rightarrow f\ x = \bot$$

Définition alternative (vers l'arrière): f est stricte ssi
$$\forall x \cdot f\ x \neq \bot \Rightarrow x \neq \bot$$

Lorsque f est définie sur un domaine plat, e.g., $\mathbf{Z}_\bot$:
$$\forall x \cdot f\ x \in \mathbf{Z} \Rightarrow x \in \mathbf{Z}$$

Soit $f^{-1}(\mathbf{Z}) = \{x \in \mathbf{Z}_\bot \mid f\ x \in \mathbf{Z}\}$.

On aura alors: f est stricte ssi
$$f^{-1}(\mathbf{Z}) \subseteq \mathbf{Z}$$

Autre propriété: f est sûre de ne pas se terminer ssi
$$f^{-1}(\mathbf{Z}) \subseteq \{\}$$

Définissons le prédicat suivant:

$$est\text{-}inverse\text{-}exact(f, R, A) \triangleq \\ f\ x \in R \ \Rightarrow\ x \in A$$

Le prédicat suivant est équivalent au précédent:

$$est\text{-}inverse\text{-}exact(f, R, A) \triangleq \\ \forall r \in R \cdot \\ f\ x = r \ \Rightarrow\ \exists a \in A \cdot a = x \ \wedge\ f\ a = r$$

Généralisons alors comme suit:

$$est\text{-}inverse(f, R, A) \triangleq \\ \forall r \in R \cdot \\ f\ x \sqsupseteq r \ \Rightarrow\ \exists a \in A \cdot a \sqsubseteq x \ \wedge\ f\ a \sqsupseteq r$$

Les inférences suivantes seront alors valides:

$$\frac{A = \{x \in \mathbf{Z}_\perp \mid f\ x \in \mathbf{Z}\},\ est\text{-}inverse(f, \mathbf{Z}, A),\ A \subseteq \mathbf{Z}}{f \text{ est stricte}}$$

$$\frac{est\text{-}inverse(f, \mathbf{Z}, \{\})}{f \text{ va boucler à l'infini}}$$

$$\frac{est\text{-}inverse(f, \mathbf{Z}, \{\perp\})}{f \text{ n'a pas besoin de son argument}}$$

Objectif de la propagation abstraite des demandes = calculer une approximation de $\widehat{f}(R)$ où

$\widehat{f}(R)$ = plus petit ensemble inverse de f pour R

$$est\text{-}inverse(f, R, \widehat{f}(R)) \\ \wedge \\ (\forall A \cdot est\text{-}inverse(f, R, A) \ \Rightarrow\ \widehat{f}(R) \subseteq A)$$

Propriété intéressante pour obtenir une abstraction sûre:

$$est\text{-}inverse(f, R, A) \wedge A \subseteq A' \ \Rightarrow\ est\text{-}inverse(f, R, A')$$

3.2 Langage source et sémantique standard

Langage source = Langage purement fonctionnel de premier ordre (sans fonctions d'ordre supérieure, i.e., fonctionnelles) avec types algébriques.

3.3 Propagation exacte des demandes:

- Deux types de demandes:
  - Demande *positive*: La valeur est définitivement requise/nécessaire au calcul du résultat.
  - Demande *négative*: La valeur est sûre de ne pas être requise pour le calcul du résultat.
- Définie via une sémantique dénotationnelle *non-standard*:
  - Fonctionne de l'arrière vers l'avant.
  - Étant donné une demande pour le résultat, détermine les demandes qui auront été faites pour les divers arguments une fois le résultat demandé produit.
- Propagation *exacte* ⇒ Non calculable à la compilation (dépend de la sémantique standard pour les conditions).

Exemple:

```
f x = If x == 0 Then 1 Else x * (f (x-1));
g x y z = If x == 0 Then 0 Else y;
```

$\widehat{f} = \lambda d, (x) \cdot$
if $d = @$
then $[@]$
else if $d = \$ \wedge x \geq 0$
then $[\$]$
else $\bot$

$\widehat{g} = \lambda d, (x, y, z) \cdot$
if $d = @$
then $[@, @, @]$
else if $d = \$$
then if $x = 0$ **then** $[\$, @, @]$ **else** $[\$, \$, @]$
else $\bot$

Note:

- "$" dénote une demande positive complète pour une valeur scalaire (entière).
- "@" dénote une demande négative.

Prédicat *est-inverse*: Établit la relation entre f et $\overline{f}$ (fonction de propagation de demandes de f):

$$
\begin{array}{l}
\mathit{est\text{-}inverse}(f, \overline{f}) \triangleq \\
\quad \forall d \in \mathit{Dmd}, v_1, \ldots, v_{n_i} \in \mathit{Valeur} \cdot \\
\qquad [d_j \mid 1 \leq j \leq k] = \overline{f}\ d\ (v_1, \ldots, v_{n_i}) \\
\qquad \Rightarrow \\
\qquad \forall v \in \mathit{Valeur} \cdot \mathit{satisfait}(v, d) \cdot \\
\qquad\quad f\ (v_1, \ldots, v_{n_i}) \sqsupseteq v \\
\qquad\quad \Rightarrow \\
\qquad\quad \exists u_j \in \mathit{Valeur} \cdot \mathit{satisfait}(u_j, d_j) \cdot \\
\qquad\qquad f\ (u_1, \ldots, u_{n_i}) \sqsupseteq v \\
\qquad\qquad \wedge \\
\qquad\qquad u_j \sqsubseteq v_j \ \ (j = 1, \ldots, k)
\end{array}
$$

Théorème (Validité de la propagation exacte des demandes)

Pour toute fonction f:
$\mathit{est\text{-}inverse}(f, \overline{f})$

3.4 Propagation abstraite des demandes:

- Abstraction $\Rightarrow$ résultats moins précis à cause du traitement des conditions (tient compte de tous les chemins d'exécution possibles).

  Domaine abstrait: Demande abstraite = *ensemble* de demandes exactes;

  Introduit incertitude: $\hat{}\$ = \{@, \$\}$

- Validité et sûreté:
  - Abstraction sûre = Le résultat abstrait contient le résultat exact;
  - Résultat valide = Le résultat abstrait est aussi un inverse (propriété d'un sur-ensemble).

Exemple:

```
f x = If x == 0 Then 1 Else x * (f (x-1));
g x y z = If x == 0 Then 0 Else y;
```

$\widetilde{f} = \lambda d \cdot$
cases d **of**
$\phi \rightarrow [\phi]$
$@ \rightarrow [@]$
$\$ \rightarrow [\$]$
$\hat{}\$ \rightarrow [\hat{}\$]$
end

$\widetilde{g} = \lambda d \cdot$
cases d **of**
$\phi \rightarrow [\phi, \phi, \phi]$
$@ \rightarrow [@, @, @]$
$\$ \rightarrow [\$, \hat{}\$, @]$
$\hat{}\$ \rightarrow [\hat{}\$, \hat{}\$, @]$
end

Note: $\hat{}\$$ = demande complète mais incertaine pour une valeur scalaire.

Caractéristique du domaine abstrait de demandes:

- Permet la représentation de demandes *cycliques* = demandes pour valeurs potentiellement infinies

Exemple (hyper-strictcité):

```
Def somme l =
    { Case l Of
        Nil = 0
    |   Cons x xs = x + somme xs };
```

somme $:
l ⇒ $\{T \rightarrow (\mathbf{Rec}\ x.:\{\mathbf{Nil}, \mathbf{Cons}\ \$\ x\})\}$

Exemple (stricticité de tête):

```
Def contient0 l =
    { Case l Of
        Nil = False
    |   Cons x xs = If x == 0 Then True Else contient0 xs };
```

contient0 $:
l ⇒ $\{T \rightarrow (\mathbf{Rec}\ x.:\{\mathbf{Nil}, \mathbf{Cons}\ \$\ \hat{}\ x\})\}$

Quelques propriétés de la propagation abstraite:

Théorème (Détection correcte des arguments stricts)

$\widetilde{f}\ \$ = [\delta_j \mid 1 \leq j \leq n_i] \wedge \delta_k = \$$
⇒
f est stricte en son k^{ieme} argument

Théorème (Détection correcte des arguments non requis)

$\widetilde{f}\ \$ = [\delta_j \mid 1 \leq j \leq n_i] \wedge \delta_k = ©$
⇒
f n'a pas besoin de son k^{ieme} argument

Théorème (Détection correcte de la non-terminaison)

$\widetilde{f}\ \$ = [\phi \mid 1 \leq j \leq n_i]$
⇒
f est sûre de boucler à l'infini

4. Aperçu d'un compilateur basé sur la propagation abstraite de demandes

Compilateur développé pour une variante paresseuse d'un sous-ensemble du langage Id (sous-ensemble purement fonctionnel de premier ordre).

Mise en oeuvre basée sur l'utilisation de suspensions:

- Le programme est représenté par un graphe de flux de données.
- Les expressions apparaissant dans un contexte non-strict sont encapulées dans des suspensions.
- Les suspensions sont annotées avec des demandes par la propagation de demandes à travers le graphe (processus d'*évaluation partielle* ⇒ création de versions multiples + évaluateurs dynamiques).
- Les suspensions non nécessaires sont éliminées.

5. Résultats expérimentaux

Expérimentation = Compilation et exécution de programmes manipulant différents types de structures de données *dynamiques*, e.g., listes linéaires, arbres, listes de listes, etc.

Modèle d'exécution utilisé pour les expériences = machine idéale:

- nombre *infini* de processeurs;
- *aucun délai* de communication entre les processeurs

Paramètres mesurés:

- Nombre total d'instructions exécutées (TOT);
- Longueur du chemin critique (LCC);
- Parallélisme moyen (PM).

Résultats:

- Réduction de TOT et LCC de 20 à 40%.
- Aucun impact majeur sur PM (parallélisme moyen faible, i.e., 5 à 10).

  Pourtant, beaucoup de parallélisme est disponible, tel que montré par l'utilisation de l'évaluation indulgente (i.e., le faible parallélisme est bien dû à la paresse et non aux programmes choisis).
- Demande initiale plus forte ⇒ Programme plus performant ⇒ Importance d'une *analyse de terminaison*, i.e., analyse de stricticité n'est pas suffisante.
- Peu d'espoir pour les programmes nécessitant réellement l'évaluation paresseuse (i.e., manipulant des structures de données potentiellement infinies).

28

6. Conclusion:

- L'utilisation des langages fonctionnels pour la programmation parallèle ne se fera très certainement *pas* par l'utilisation de langages paresseux.
  - Pour quels problèmes l'évaluation paresseuse est-elle vraiment utile/nécessaire?
  - Pourquoi la plupart des programmes *benchmarks* utilisés pour les compilateurs paresseux n'utilisent-ils pas des programmes nécessitant l'évaluation paresseuse?
- La propagation abstraite de demandes semble intéressante pour l'analyse des structures de données dynamiques.

  Pourrait être utile pour d'autres problèmes que l'analyse de stricticité:
  - Partitionnement des programmes pour les architectures parallèles multicontextes *(multi-threaded architectures)* (pour langage fonctionnel indulgent).
  - Identification du parallélisme pour les programmes impératifs manipulant des structures de données dynamiques.

29