Implantation et optimisation des langages fonctionnels

GLO-66811

Automne 2008

Danny Dubé
Université Laval

Introduction

Objet d'études: les langages fonctionnels

- Langages utilisés dans le cours: Scheme, ML et Haskell
- Recherche particulièrement intense autour de ces trois langages
- Il existe plusieurs autres langages fonctionnels
- Centre d'intérêt du cours: techniques d'implantation (de compilation)

Particularités des langages fonctionnels

- Fournissent une abstraction très puissante et très bien connue en mathématiques: la fonction
- Fonctions comme objets de première classe
  - Création de fonctions à la volée
    Ex: `(set! f (lambda (x) (g (+ x 1))))`
  - Passage en argument, retour comme résultat
    Ex: `(f c (lambda (x) (g (+ x 1))))`
  - Stockage dans et extraction de les structures de données
    Ex:
    ```
    (define lst (list 1 (lambda (x) (g (+ x 1))) 3))
    (set! f (list-ref lst 1))
    ```

Particularités des langages fonctionnels

- Absence ou désapprobation des effets de bord
  - Pas nécessairement fournis mais toujours désapprouvés
  - En Scheme: complètement fournis
  - En ML: fournis sous une forme très restrictive
  - En Haskell: complètement absents

Particularités des langages fonctionnels

- Peu de structures de contrôle
  - Ordre d'évaluation et récursivité
  - Exemple de boucle '`for`':

```
sum = 0;
for (i=1 ; i<=10 ; i++)
  sum += f(i);
```

    transformée en fonction récursive:

```
fun g i s =
  if i <= 10
    then g (i+1) (s+(f i))
    else s;
val sum = g 1 0;
```

  - Importance de l'implantation de la récursion

Particularités des langages fonctionnels

- Structures de données avancées
  - Fonctions, nombres, listes, caractères, booléens, chaînes de caractères, symboles
  - Scheme et ML offrent les tableaux
  - ML offre des objets '`ref`'
  - ML et Haskell offrent les types algébriques. Exemple:

```
data Tree a = Node (Tree a) a (Tree a) | Empty
```

  - ML et Haskell offrent aussi le filtrage (*pattern-matching*). Exemple:

```
treeToList (Node t1 x t2) =
  (treeToList t1) ++ [x] ++ (treeToList t2)
treeToList Empty          = []
```

Particularités des langages fonctionnels

- Typage dynamique versus statique et polymorphisme
  - Détection des erreurs de types à la compilation versus à l'exécution
  - Ex: `(+ 2 "3")` en Scheme
  - Ex: `2 + "3"` en ML ou Haskell
- Continuations réifiables
  - Définition de continuation

    La *continuation* d'une expression au cours de son évaluation consiste en le calcul qu'il reste à faire une fois connu le résultat. Conceptuellement, elle peut être vue comme une fonction à un argument.
  - Exemples:

```
... x * y + z ...
... if <test> then f 1 else g 2 ...
```

  - Réification de la continuation en Scheme ('`call/cc`')

Particularités des langages fonctionnels

- Gestion automatique de la mémoire
- Généralement sécuritaires
- Évaluations stricte et paresseuse
  - Exemple nécessitant l'évaluation paresseuse:

```
take 3 lst
  where lst = 1 : map (+ 1) lst
```

Aspects sécuritaires des langages fonctionnels

- Libère le programmeur des détails fastidieux
  - Programmes moins longs
  - Programmes plus clairs
- Peut fournir un bon niveau de vérification à la compilation
- Surveille, à la compilation ou à l'exécution, toutes les sources d'erreur
- En cas d'erreur, celle-ci est déclarée en termes propres au langage
- Exemple:
  - En Haskell:

```
Prelude> [1,2,3,4]!!8
Program error: Prelude.!!: index too large
```

  - En C:

```
for (i=1, p=lst ; i<=8 ; i++, p=p->next);
Segmentation fault
```

Aperçu des 3 langages fonctionnels

- La syntaxe de Scheme:
  - en notation préfixe
  - est pleinement parenthésée
  - n'a pas de notion d'opérateur
- La syntaxe de ML:
  - en notation infixe
  - possède plusieurs opérateurs
  - délimite les formes syntaxiques par des mots-clés et des caractères graphiques
- La syntaxe de Haskell:
  - relativement similaire à celle de ML
  - encore plus axée sur l'utilisation des caractères graphiques
  - elle permet l'utilisation de l'indentation pour indiquer les niveaux d'imbrication syntaxique

Aperçu des 3 langages fonctionnels

- Scheme:

```
> (define (comp f g)
    (lambda (x)
      (f (g x))))
> (define h (comp (lambda (x) (+ x 1))
                  (lambda (y) (* 3 y))))
> (map h '(1 2 3))
(4 7 10)
```

- ML:

```
- val h = (fn x => x + 1) o (fn y => 3 * y);
val h = fn : int -> int
- map h [1,2,3];
val it = [4,7,10] : int list
```

- Haskell:

```
> map h [1,2,3] where h = (+ 1) . (* 3)
[4,7,10]
```

Aperçu des 3 langages fonctionnels

- En Scheme:

```
(define (sequence n)
  (let loop ((n n) (acc '()))
    (if (= n 0)
      acc
      (loop (- n 1) (cons n acc)))))
(sequence 100)
```

- En ML:

```
fun sequence n = loop n []
and loop 0 acc = acc
|   loop n acc = loop (n - 1) (n :: acc);
sequence 100;
```

- En Haskell:

```
[1..100]

sequence n = loop n []
  where loop 0 acc = acc
        loop n acc = loop (n - 1) (n : acc)
sequence 100
```

Aperçu des 3 langages fonctionnels

- En Scheme:

```
(define (quicksort lst)
  (if (null? lst)
    '()
    (let ((pivot (car lst))
          (rest (cdr lst)))
      (append
       (quicksort (filter (lambda (x) (< x pivot)) rest))
       (cons pivot
             (quicksort (filter (lambda (x) (>= x pivot)) rest)))))))
```

- En ML:

```
fun quicksort []      = []
|   quicksort (y::ys)
      = quicksort (filter (fn x => x < y) ys)
        @ [y] @
        quicksort (filter (fn x => x >= y) ys);
```

- En Haskell:

```
quicksort []     = []
quicksort (x:xs) = quicksort (filter (< x) xs)
                   ++ [x] ++
                   quicksort (filter (>= x) xs)
```

Aperçu des 3 langages fonctionnels

Liens vers les langages Scheme, ML et Haskell:

Page web à propos de Scheme:

`http://www.swiss.ai.mit.edu/projects/scheme/`

`http://www.schemers.org/`

(nous ne considérons que R^4RS et R^5RS dans ce cours)

Page web à propos de ML:

`http://www.cs.cmu.edu/afs/cs/project/fox/mosaic/sml.html`

`http://www.faqs.org/faqs/meta-lang-faq/`

Page web à propos de Haskell:

`http://www.haskell.org/`

Introduction aux compilateurs et interprètes

Dans sa plus simple expression, un compilateur comporte les phases suivantes:

1. Analyse lexicale
    - D'une suite de caractères à une suite de jetons
2. Analyse syntaxique
    - D'une suite de jetons à un arbre de syntaxe abstraite (AST)
3. Analyse statique et optimisation
    - Vérification de certaines propriétés
    - Transformation visant à *améliorer* le programme
4. Génération du code
    - Écriture d'un programme dans le langage cible

La phase 3 peut se contenter de n'effectuer que les opérations strictement requises par le langage source (e.g. vérification des types)

Voir le cours IFT-15752, Compilation et interprétation

Introduction aux compilateurs et interprètes

Modes de compilation des langages avancés

- Natif
  - (+) Meilleur code
  - (−) Plus complexe
  - (−) Non portable
- Vers une machine virtuelle
  - (−) Code lent
  - (+) Code compact
  - (+) Simple
  - (+) Portable
- Vers un langage intermédiaire comme C
  - (+) Bon code
  - (+) Presqu'aussi simple que la machine virtuelle
  - (+) Portable

Introduction aux compilateurs et interprètes

Compilation des langages conventionnels

- Forme SSA (*static single assignment*)
- Analyse de définition-usage
- Allocation de registres
- Sélection d'instructions
- Séquencement d'instructions
- Analyse d'aliasage
- Priorité à l'aspect intra-procédural
- Propagation de copies, de constantes

Introduction aux compilateurs et interprètes

Fonctionnement d'un interprète:

- Interprétation de l'AST (grâce à une fonction récursive d'évaluation)
- Compilation interne vers une machine virtuelle
- Compilation vers des séquences de pointeurs
- Compilation dynamique

Définitions (rappel)

- Corps (d'une fonction) (d'une forme de liaison). Ex:
  `(lambda (x) <corps>)`, `(let ((x 5) (y (- x 3))) <corps>)`
- Forme syntaxique. Ex: appel, appel, affectation
  `(gcd x 5)`, `(expt x 5)`, `(set! x 5)`
- Sucre syntaxique. Ex:
  `(let ((x <init>)) <corps>)` $\equiv$ `((lambda (x) <corps>) <init>)`
- Argument: *ce qui est passé*
- Paramètre: *nom du récipient*
- Runtime: module fixe écrit dans le langage cible et fournissant divers services
- Fixnums: entiers à étendue limitée (par la machine)
- Bignums: entiers à étendue illimitée (ou à limite repoussée très loin)

Considérations diverses

- Spécifications formelles versus informelles
  - Les premières sont généralement mathématiques
  - Les premières sont généralement textuelles
- Sémantique
  - Concrète: description du *véritable* comportement du programme
  - Abstraite: description simplifiée ou moins détaillée du comportement du programme
- Principe de développement d'un compilateur:
  1. Faire fonctionner; 2. optimiser
- Distinction entre analyse statique et optimisation:
  Examen versus transformation
- Incalculabilité de plusieurs problèmes d'analyse