Langages et paradigmes

Olivier Lecarme

Master d'Informatique, première année

2007–2008

Septième partie VII

Modèles de programmation

Introduction

- Généralités
  - Le modèle impératif ou constructif domine largement les langages de programmation, spécialement dans le monde industriel et commercial.
  - Les modèles fonctionnel et logique occupent cependant des niches solides et ont des implémentations de qualité industrielle.

généralités

- Tout ce qui concerne les noms, portées et visibilité, ainsi que le typage, les expressions, la récursivité, est commun à la plupart des modèles.
- Les frontières entre les différentes catégories sont souvent floues :
  - énoncés impératifs dans les langages qui ne le sont pas
  - programmation fonctionnelle dans les langages impératifs
  - existence de langages multi-paradigmes
  - langages hybrides tels que Icon

généralités

- Nous avons déjà vu des concepts importants pour les langages fonctionnels :
  - polymorphisme
  - portée dynamique
  - récursivité comme mécanisme d'itération
  - utilisation intensive de listes
  - beaucoup d'allocation dynamique et nécessité d'un récupérateur de mémoire
- Beaucoup moins de présentation de concepts pour la programmation en logique, qui a vingt ans de moins que la programmation fonctionnelle.

Historique

- Les modèles impératif et fonctionnel proviennent des modèles de calcul de mathématiciens dans les années 30 :
  - Alan Turing propose un modèle de calcul opérationnel, la machine de Turing, sorte d'automate à pile disposant d'une mémoire linéaire de capacité illimitée.
  - Alonzo Church propose un modèle de calcul fonctionnel, le lambda-calcul, qui s'appuie sur des expressions paramétrées et la substitution des paramètres.

historique

- deux autres modèles encore :
  - A. A. Markov propose un modèle de calcul algorithmique, les algorithmes de Markov, qui s'appuient sur la substitution de caractères dans des chaînes à l'aide de règles extrêmement simples et strictes.
  - Stephen Kleene et Emil Post proposent des modèles beaucoup plus abstraits qui ne conduisent pas à des implémentations.
  - Il a été démontré que tous ces modèles sont équivalents.

historique

- Le but est de formaliser le concept de fonction calculable. Cela permet de distinguer formellement deux mécanismes de démonstration :
  - démonstration constructive, qui montre comment obtenir l'objet désiré (un programme est une démonstration constructive de l'existence de ce qu'il calcule)
  - démonstration non-constructive, qui montre simplement que l'objet doit exister

historique

- Les langages de programmation actuels proviennent de ces modèles :
  - machine de Turing pour la plupart des langages impératifs
  - lambda-calcul pour les langages fonctionnels
  - algorithmes de Markov pour Snobol et un peu Icon
- Le modèle logique est lié à l'idée de démonstration constructive, mais de manière plus abstraite :
  - on écrit un ensemble d'axiomes
  - l'ordinateur découvre une démonstration constructive d'un ensemble particulier de valeurs qui satisfont les axiomes pour un ensemble de données

Exemple

- On veut calculer le PGCD des valeurs `a` et `b`.
- En programmation impérative :

  Vérifier si `a = b`. Si oui, imprimer l'un des deux et terminer. Sinon, remplacer le plus grand par leur différence et recommencer.

- En programmation fonctionnelle :

  Le PGCD de `a` et `b` est défini comme étant `a` si `a = b`, ou sinon comme étant le PGCD de `c` et `d`, où `c` est le minimum de `a` et `b` et `d` est leur différence.

exemple

- En programmation logique :

  La proposition `PGCD(a,b,p)` est vraie si `a`, `b` et `p` sont tous les trois égaux, ou s'il existe les nombres `c` et `d` tels que `c` est le minimum de `a` et `b` (c'est-à-dire que `min(a,b,c)` est vrai) et `d` est leur différence (c'est-à-dire que `diff(a,b,d)` est vrai), et que `PGCD(c,d,p)` est vrai.

- On voit que pour les deux derniers modèles, il faut normalement ajouter une méthode d'évaluation du programme, alors que le modèle impératif est auto-évaluable.

Programmation fonctionnelle

- Généralités
  - La programmation fonctionnelle au sens propre définit les résultats d'un programme comme une fonction mathématique des données, sans idée d'état interne ni d'effet de bord.
  - Les langages Miranda, Haskell et Sisal sont purement fonctionnels, ainsi que la proposition de Backus.
  - Les langages fonctionnels pratiques (Lisp, Scheme et même ML) contiennent tous des aspects impératifs.

généralités

Pour que la programmation soit praticable, les langages fonctionnels comprennent de plus :

- Fonctions de première classe et fonctions d'ordre supérieur :
  - si les fonctions sont des objets de première classe, on doit pouvoir en créer de nouvelles à l'exécution, comme on fait pour tout autre objet
  - une fonction d'ordre supérieur prend une fonction comme argument ou produit une fonction comme résultat

généralités

- Polymorphisme très général :
  - il permet d'utiliser une fonction sur des types d'arguments très généraux
  - les langages fonctionnels courants sont à typage dynamique
- Types et opérateurs pour les listes :
  - les listes ont une définition récursive par essence
  - leur manipulation récursive (tête, queue) est naturelle

généralités

- Récursivité :
  - en l'absence d'effets de bord, c'est le seul moyen de répéter une action
  - cela devient donc le mécanisme fondamental d'évaluation d'un langage fonctionnel
- Résultats de fonctions de type quelconque :
  - indispensable pour ne pas limiter le type de calculs
  - tout autre mécanisme nécessiterait un effet de bord

généralités

- Constructeurs pour objets structurés : même remarque que pour les résultats de fonctions.
- Récupération de mémoire :
  - sans effet de bord, il faut à tout moment créer de nouveaux objets
  - la mémoire n'étant pas infinie, il faut donc récupérer ceux qui ne servent plus
- Comme ces aspects sont quasiment nécessaires à la programmation fonctionnelle, les apôtres de ce modèle les considèrent comme indispensables à tout langage.

généralités

- Comme ces aspects ne sont pas indispensables à la programmation impérative, les apôtres de ce modèle les considèrent (au moins pour certains) comme inutilement coûteux et de peu d'intérêt :
  - Fortran jusqu'à la version de 1977 n'a pas de récursivité
  - Pascal ne permet pas qu'une fonction ait un résultat structuré
  - dans la plupart des langages impératifs, les constructeurs d'objets structurés manquent ou sont incomplets
  - la récupération de mémoire n'est pas appliquée aux objets en pile, et n'est pas universellement utilisée

généralités

- les fonctions d'ordre supérieur sont une spécificité des langages fonctionnels
- Certaines propriétés de Lisp et Scheme ne sont pas nécessaires aux langages fonctionnels, ni même caractéristiques :
  - homogénéité des programmes et des données (homoïconicité)
  - auto-définition (l'évaluateur de Lisp se décrit simplement en Lisp)
  - interaction avec l'utilisateur par une boucle de lecture, évaluation et affichage

Difficultés de la programmation fonctionnelle

- L'idée de programmation strictement fonctionnelle est séduisante :
  - l'absence d'effets de bord facilite la lecture des programmes
  - toute expression est évaluée indépendamment de son contexte
  - les raisonnements sur les programmes sont facilités

difficultés

De nombreux paradigmes de programmation s'appuient de manière centrale sur l'idée d'affectation ou d'effet de bord :

- Utilisation d'entrées-sorties :
  - il faut des mécanismes élaborés pour modéliser la simple idée de lecture séquentielle
  - c'est encore pire pour l'affichage graphique ou pour les fichiers à accès direct

difficultés

- Initialisation de structures complexes :
  - il est facile dans un langage fonctionnel de construire une liste à partir d'une autre
  - c'est beaucoup plus difficile pour un tableau, spécialement à plusieurs dimensions, et surtout si l'ordre d'accès n'est pas purement séquentiel
- Construction de statistiques :
  - comptage d'occurences dans une large collection de données
  - établissement d'un dictionnaire, d'un lexique, d'une table associative
  - la méthode naturelle est de mettre à jour des entrées dans une structure de statistique

difficultés

- Modification sur place :
  - si les données sont très encombrantes, il faut éviter de les dupliquer
  - tri sur place plutôt que tri par copie
  - calculs scientifiques sur de grandes matrices
- Presque tous ces cas se ramènent au problème de modification triviale :
  - la programmation fonctionnelle pure oblige à construire une copie de l'objet structuré pour modifier un seul de ses composants
  - si le langage fonctionnel fournit des outils de modification sur place, il introduit donc l'effet de bord et l'affectation

difficultés

- Propositions des apôtres de la programmation fonctionnelle :
  - un bon compilateur doit découvrir les récursivités terminales et les remplacer par des itérations
  - un bon compilateur doit découvrir les modifications triviales et éviter la copie des structures
  - la programmation fonctionnelle est moins répandue parce que les étudiants commencent par la programmation impérative

difficultés

- Mais :
  - toutes les architectures de machines actuelles suivent le modèle de Von Neumann
  - les tentatives d'architectures spécialisées pour la programmation fonctionnelle ont toutes été des échecs
  - et pourtant de grandes universités américaines ou françaises font aborder la programmation par Scheme ou ML

Programmation logique

- Généralités
  - Il s'agit de démontrer des théorèmes à partir d'axiomes.
  - Le programmeur fournit le théorème, qui est le but à atteindre, et l'implémentation essaie de trouver une collection d'axiomes et d'étapes de déduction qui conduisent à ce théorème.
  - Les axiomes sont écrits comme des clauses de Horn :

$$T \leftarrow C_1, C_2, \ldots, C_n$$

T est la tête et les termes C_i contituent le corps.

généralités

- H est vrai si tous les termes C_i sont vrais.
- La combinaison de plusieurs clauses permet d'en déduire de nouvelles par résolution :
  - A et B impliquent C
  - C implique D
  - alors A et B impliquent D
- Pendant la résolution, les variables libres peuvent prendre des valeurs par le processus d'unification :
  - humide(x) ⟵ fleuri(X)
  - humide(Nice)
  - donc fleuri(Nice)
- Le seul langage logique réellement utilisé est Prolog, défini par Colmerauer et Roussel à Marseille en 1973.

Prolog

- Le programmeur construit une base de données de clauses supposées vraies.
- Une clause est faite de termes, qui sont des constantes, des variables ou des structures :
  - Une constante est un atome (similaire à Lisp ou Scheme) ou un nombre.
  - Une variable est un identificateur commençant par une majuscule, elle peut prendre une valeur arbitraire au cours du processus d'unification.
  - Une structure est un prédicat logique ou une structure de donnée, elle prend la forme d'un foncteur et d'une liste d'arguments.

Prolog

- Les clauses peuvent être des faits, des règles ou des buts, et se terminent par un point :
  - un fait est une clause de Horn sans partie droite :

```
humide(nice).
```

  - une règle a une partie droite :

```
neigeux(X) :- humide(X), froid(X).
```

  - un but n'a pas de partie gauche :

```
?- humide(Ville).
```

Prolog

- Un but n'apparaît pas dans un ensemble de clauses, c'est une question posée au moteur de Prolog, qui répond par un ensemble de valeurs de variables qui rend vraie la clause donnée comme but.

Résolution et unification

- Principe de résolution (Robinson, 1965) :

  Si C_1 et C_2 sont deux clauses telles que la tête de C_1 correspond à l'un des termes de C_2, alors on peut remplacer le terme de C_2 par le corps de C_1.

résolution et unification

- Exemple :

```
habite(toto, nice).
habite(jojo, cannes).
habite(lili, grasse).
habite(tata, cannes).
voisin(X, Y) :- habite(X, V), habite(Y, V).
```

Si `X` prend la valeur `tata` et `V` la valeur `cannes`, on peut remplacer le premier terme de la dernière clause par le corps (vide) de la quatrième, ce qui donne la nouvelle clause :

```
voisin(tata, Y) :- habite(Y, cannes).
```

donc `Y` est voisin de `tata` s'il habite `cannes`.

résolution et unification

- Le mécanisme qui associe une valeur à une variable est l'unification :
  - une constante s'unifie avec elle-même
  - deux structures s'unifient si elles ont le même foncteur et le même nombre d'arguments, et que les arguments s'unifient en correspondance
  - une variable s'unifie avec n'importe quoi

Extensions

- Pour rapprocher Prolog d'un langage de programmation, on apporte des extensions lexicales ou syntaxiques :
- Le but `=(A,B)` peut s'écrire `A = B`.
- Les expressions arithmétiques s'écrivent comme dans les langages ordinaires, avec en plus le foncteur `is` qui unifie son premier argument avec la valeur arithmétique de son deuxième argument.

extensions

- Les notations pour les listes sont inspirées de Lisp :
  - la liste vide est `[]`
  - le constructeur de liste est `.(a, [])`
  - la liste `.(a, .(b, .(c, [])))` s'abrège sous la forme `[a, b, c]`
  - on peut expliciter la limite entre tête et queue de la liste avec l'opérateur `|`, ce qui permet d'écrire les deux clauses suivantes :

```
member(X, [X | T]).
member(X, [H | T]) :- member(X, T).
```

Ordre d'exécution

- Le mécanisme d'unification sert à trouver une suite d'étapes de résolution qui construisent le but à partir de la base de faits, ou à démontrer que cette suite n'existe pas.
- Deux stratégies sont possibles :
  - partir des clauses existantes et chercher à en dériver le but, par chaînage avant
  - partir du but et procéder à l'envers vers les clauses existantes, par chaînage arrière.

ordre d'exécution

- C'est la deuxième stratégie qu'utilise Prolog : l'arbre des recherches possibles est exploré en profondeur et de gauche à droite.
- Les clauses étant ordonnées et l'algorithme de recherche connu, le résultat est déterministe.
- En revanche, l'ordre des clauses est très important, en particulier pour éviter une descente infinie.
- Pour donner au programmeur un certain contrôle sur le déroulement de la recherche, on introduit une coupure, notée **!**, qui fige toutes les unifications faites jusqu'au point où on la rencontre.

Difficultés de la programmation logique

- Le « Programme d'ordinateurs de cinquième génération » du Ministère de l'industrie japonais (mi-70) devait utiliser Prolog comme langage machine.
- En fait, l'idée même a été abandonnée.
- On peut considérer les clauses de Horn comme un bon formalisme de spécification du problème à résoudre.
- Prolog est un mécanisme d'implémentation de cette spécification, avec les difficultés mentionnées.

difficultés

- Il y a quelques limites théoriques importantes :
  - certains aspects de la logique ne sont pas couverts
  - les clauses sont déclaratives, mais l'ordre d'exploration du moteur de Prolog est déterminant
  - le programmeur doit souvent en tenir compte pour que le programme soit efficace, ou simplement se termine
  - il n'y a pas de moyen propre et clair d'exprimer la négation