Algorithmique - Programmation 1

Cours 8

Université Henri Poincaré

CESS Epinal

Automne 2008

Plan

Rappel : calcul de la fonction puissance

- Fonction récursive n°0 :

```
# let rec p0 =
    function x ->
       function
            0 -> 1
          |  n -> x * (p0 x (n-1));;

val p0 : int -> int -> int = <fun>
```

Rappel : calcul de la fonction puissance (suite)

- Fonction récursive n°1 :

```
# let rec p1 =
    function x ->
       function
           0 -> 1
       |   n -> if (n mod 2) = 0
                then (p1 x (n/2)) * (p1 x (n/2))
                else (p1 x (n/2)) * (p1 x (n/2))*x;;

val p1 : int -> int -> int = <fun>
```

Rappel : calcul de la fonction puissance (suite)

- Fonction récursive n°2 :

```
# let rec p2 =
    function x ->
       function
           0 -> 1
       |   n -> if (n mod 2) = 0
                then (p2 (x*x) (n/2))
                else (p2 (x*x) (n/2))*x;;

val p2 : int -> int -> int = <fun>
```

Rappel : calcul de la fonction puissance (suite)

- Questions :
  - ces fonctions sont-elles les mêmes ?
  - ces fonctions sont-elles équivalentes ?
  - y'a-t-il une fonction préférable aux autres ? Pourquoi ?
- NB: dans ce qui suit, nous utiliserons le terme programme pour désigner une fonction.

Plan

Rappel : calcul de la fonction puissance

Propriétés de programmes

- Egalité de programmes
- Equivalence de programmes
- Efficacité d'un programme (complexité en temps)
- Complexité en `Caml`
- Améliorer un programme
- Complexité en taille

Egalité de programmes

Définition (Egalité de programmes)

Deux programmes sont égaux si leurs contenus sont égaux au renommage des noms locaux près.

```
# let rec pA =
  function x ->
    function 0 -> 1
   |   n -> if (n mod 2) = 0
               then (p1 x (n/2)) * (p1 x (n/2))
               else (p1 x (n/2)) * (p1 x (n/2))*x;;
# let rec pB =
  function v ->
    function 0 -> 1
   |   k -> if (k mod 2) = 0
               then (p1 v (k/2)) * (p1 v (k/2))
               else (p1 v (k/2)) * (p1 v (k/2))*v;;
```

Equivalence de programmes

Définition (Equivalence de programmes)

Deux programmes sont dits équivalents s'ils calculent les mêmes résultats pour les mêmes données :

$$\forall x,\ (p\ \ x) = (q\ \ x)$$

Définition (Preuve d'un programme)

La preuve d'un programme est la démonstration qu'une fonction `Caml` *est équivalente à la fonction mathématique qu'elle doit représenter.*

- Exemple de fonctions équivalentes :
  `p0`, `p1` et `p2` vues précédemment

Efficacité d'un programme

- L'**efficacité d'un programme** est évaluée par rapport à un contexte d'utilisation
  - tâches de sauvegarde lancées la nuit
  - assistant au pilotage
  - *etc*
- En règle générale, un programme est dit efficace s'il se distingue par la **quantité de mémoire et de temps** qu'il nécessite
- Attention : le **temps d'exécution** dépend de l'algorithme utilisé, mais aussi de la machine et du nombre de tâches en cours
- Question : comment **comparer** l'efficacité de deux programmes ?

Efficacité d'un programme (suite)

Définition (Complexité en temps)

La complexité (en temps) d'un programme est la relation liant la taille des données et le nombre d'opérations élémentaires nécessaires à un calcul.

- La complexité exprime la loi de comportement du **temps de calcul en fonction des données**
- Cette loi permet de :
  - **comparer** les comportements asymptotiques
  - **classifier** les programmes selon le comportement
  - déterminer la **faisabilité** des calculs

Efficacité d'un programme (suite)

- Comment s'exprime la complexité d'un programme ?
  - fonction de complexité *exacte* généralement très difficile à obtenir

    $\longrightarrow$ on définit une **borne supérieure** (complexité dans le pire des cas)
- Utilisation de la notion $\mathcal{O}(n)$:
  - le tri d'un tableau de longueur n par recherche du mininum coûte dans le pire des cas $n \times n$ opérations

    $\longrightarrow$ on dit que ce tri est en $\mathcal{O}(n^2)$

Efficacité d'un programme (suite)

Notation	Type de complexité
$\mathcal{O}(1)$	complexité constante (indépendante de la taille de la donnée)
$\mathcal{O}(log(n))$	complexité logarithmique
$\mathcal{O}(n)$	complexité linéaire
$\mathcal{O}(nlog(n))$	complexité quasi-linéaire
$\mathcal{O}(n^2)$	complexité quadratique
$\mathcal{O}(n^3)$	complexité cubique
$\mathcal{O}(n^p)$	complexité polynomiale
$\mathcal{O}(2^n)$	complexité exponentielle
$\mathcal{O}(n!)$	complexité factorielle

Complexité en Caml

- **Opération élémentaire** : une réécriture
- Pour comparer deux programmes, on ne **compte souvent que la réécriture d'une opération** (*e.g.* application d'une fonction)
- Lorsqu'une fonction est définie à partir d'autres fonctions, sa complexité se détermine :
  - en calculant la **complexité de chaque fonction**
  - en **composant** ces complexités "élémentaires"

Améliorer un programme

- La complexité d'un programme est une propriété intrinsèque de la méthode de calcul utilisée
- Pour améliorer un programme, il faut changer de méthode de calcul
- Techniques souvent efficaces :
  - la **dichotomie** : couper la donnée en deux parties égales (on passe de $\mathcal{O}(n)$ à $\mathcal{O}(log(n))$)
  - la **tabulation** : conserver les résultats des calculs intermédiaires

Améliorer un programme

- Quelques exemples d'**optimisations** de programmes :
  - données triées ou non
  - nombre de parcours de listes dans un tri :

    - comparaisons deux à deux

    - recherche du minimum
  - schéma de Hörner pour l'évaluation de polynômes :

    $P(x) = a_0 + a_1.x + a_2.x^2 + ... + a_n.x^n$

    $P(x) = a_0 + x.(a_1 + x.(a_2 + ...(a_{n-1} + x.(a_n))...)$
  - tabulation des termes de la suite de Fibonacci

Améliorer un programme : suite de Fibonacci

- Calcul de la suite $F_n = F_{n-1} + F_{n-2}$
- Solution naïve :

```
let rec fib =
  function
     0 -> 1
  |  1 -> 1
  |  n -> (fib (n-1)) + (fib (n-2));;
```

- Remarque : `fib` recalcule constamment la valeur pour les `n` antérieurs

Améliorer un programme : suite de Fibonacci

- Suite de Fibonacci revisitée :

```
let fib2 =
  function n ->
    let rec memo_fib =
       function (f1, f2) ->
          function
              0 -> f1
            | n -> memo_fib (f1+f2, f1) (n-1)
    in memo_fib (1,0) n;;
```

- Comparez l'exécution de

```
# fib 8;;
```

et

```
# fib2 8;;
```

Complexité en taille

- Un autre critère important est la quantité de mémoire nécessaire à un algorithme
  - données manipulées (*e.g.* annuaire)
  - mémoire disponible (machine serveur *vs* machine client)
  - représentation des données (*e.g.* polynômes creux vs polynômes pleins)

Conclusion

- Caractéristiques d'un programme : **preuve de correction** (production du résultat attendu), **complexité** (en taille et en temps)
- Principales complexités en temps :
  - $\mathcal{O}(n)$
  - $\mathcal{O}(n^p)$
  - $\mathcal{O}(c^n)$
- Privilégier les algorithmes minimisant les parcours et/ou conservant les résultats intermédiaires