Langage de Programmation 2 (LP2)

RICM3

Cours 10

Pascal Lafourcade

Polytech

2009 - 2010

Points fixes

Décidabilité

Complexité

Exemple

Récursive terminale

Threads

Object

Conclusion

Définition par point fixe

Rêvons

let x such that $x = 1. +. 0.5$ *. x in $5.$ *. x

Définition

Étant donné une fonction f, un *point fixe* de f est une valeur x telle que $x = f(x)$.

Exemple :

$$f_2 = \lambda x.\,(1. +. \frac{x}{2.})$$

Lorsque le point fixe de f existe, alors il est unique,

Ainsi let x such that $x = f\ x$ constitue une définition acceptable

Comment calculer le point fixe ?

Sous certaines conditions, il suffit de chercher la limite d'une suite de la forme :

$$a_0;\ a_1 = f\ a_0;\ \ldots\ a_{n+1} = f\ a_n;\ \ldots$$

Exemple :

Pour $f = f_2$ et $a_0 = 1$, on obtient la suite

$$1\,;\ 1,5\,;\ 1,75\,;\ \ldots\ 2 - \frac{1}{2^n}\ \ldots$$

Point fixe en programmation fonctionnelle

Les conditions ne sont pas réunies pour que, en pratique, on puisse définir des valeurs numériques par point fixe.

Mais cela fonctionne pour des valeurs fonctionnelles

Exemple

La fonction factorielle est un point fixe de la fonctionnelle suivante

let f_fact = fun $g \rightarrow$ fun $n \rightarrow$ if $n = 0$ then 1 else n * g $(n-1)$

En prenant pour g_0 une fonction quelconque, la limite de

$$g_0\,;\ g_1 = \text{f_fact}\ g_0\,;\ \ldots\ g_{n+1} = \text{f_fact}\ g_n\,;\ \ldots$$

est exactement la fonction factorielle

Notation d'une définition par point fixe

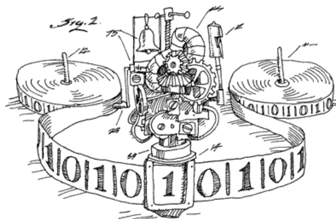
Au lieu de

let fact such that fact = f_fact fact

On écrit

let rec fact = fun n → f_fact fact n

Machine de Turing (1912-1954)



1. **ruban infini** : chaque case contient un symbole parmi un alphabet fini avec un symbole spécial « blanc ».
2. **tête de lecture/écriture** : se déplace vers la gauche ou vers la droite du ruban.
3. **registre d'état** : mémorise l'état courant de la MT. (le nombre d'états est toujours fini, et contient un état initial)
4. **table d'actions** : indique quel symbole écrire, comment déplacer la tête de lecture.

Si aucune action n'existe pour une combinaison donnée d'un symbole lu et d'un état courant, la machine s'arrête.

Décidabilité, Indécidabilité ...

Soit RE les langages acceptés par une MT.

A	w_0	w_1	w_2	...	w_j	...
M_0	O	N	N	...	O	...
M_1	N	O	O	...	O	...
M_2	N	O	N	...	O	...
⋮	⋮	⋮	⋮	...	⋮	...
M_i	N	O	N	...	O	...
⋮	⋮	⋮	⋮	...	⋮	...

$A[M_i, w_j]$

- O : si la MT M accepte le mot m_i
- N : sinon (boucle ou termine pas)

Un Langage indécidable.

Soit RE les langages acceptés par une MT.

$$L = \{w | w = w_i \wedge A[M_i, w_i] = N\}$$

Si $L \in RE$, accepté par une MT, mais pas M_k de la liste car

- Si M_k accepte w_k, $A[M_k, w_k] = O$, donc $w_k \notin L$, contradiction.
- Sinon $A[M_k, w_k] = N$ donc $w_k \in L$, contradiction.

Donc L est indécidable.

Fonction récursives primitives

Définition

- Fonction nulle
- Succ
- Projections

Composition de fonction recursives primitives.

Exemples : Pred, addition, multiplication

Fonction d'Ackerman

Ack(m,n) =

- $n + 1$ si $m = 0$
- $Ack(m-1, 1)$si $n = 0$
- $Ack(m-1, Ack(m, n-1))$ sinon

Quelques exemples :

- $Ack(1, n) = n + 2$
- $Ack(2, n) = 2n + 3$
- $Ack(3, n) = 8.2^n - 3$
- $Ack(4, n) = 2.2^{2^{2^{\cdots^{2}}}}$
- $Ack(4, 4) > 2^{65536} > 10^{80}$

La fonction d'Ackerman n'est pas récursive primitive.

"Nul n'est jamais assez fort pour ce calcul"

Fonction récursives primitives

Indécidabilité de la récursion primitive

Exemple : Post Correspondence Problem (PCP)

Definition ((PCP))

Soit Σ un alphabet fini. **Input :** Suite de paires $\langle u_i, v_i \rangle_{1 \leq i \leq n}$
$u_i, v_i \in \Sigma^*, \; n \in \mathbb{N}$
Question : Existence de $k, i_1, \ldots, i_k \in \mathbb{N}$ tel que
$u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$?

Example

u_1	u_2	u_3	u_4	v_1	v_2	v_3	v_4
aba	bbb	aab	bb	a	aaa	$abab$	$babba$

Solution : $\mathbf{1431}$

$u_1 \cdot u_4 \cdot u_3 \cdot u_1 = aba \cdot bb \cdot aab \cdot aba = a \cdot babba \cdot abab \cdot a = v_1 \cdot v_4 \cdot v_3 \cdot v_1$

Mais pas de solution pour $\langle \mathbf{u_1}, \mathbf{v_1} \rangle, \langle \mathbf{u_2}, \mathbf{v_2} \rangle, \langle \mathbf{u_3}, \mathbf{v_3} \rangle$

PCP est indécidable

Complexité

Nombre d'opérations nécessaires pour faire un cacul.

Complexité en mémoire, en espace, en temps ...
Complexité en moyenne, dans le pire des cas (worst-case analysis).

Exemples

- Successeur (n) : Complexité : 1
- Factorielle (n) = if n=0 then 0 else fact(n-1)*n :
  - n multiplications
  - n soustraction
  - n+1 appel récursif

  Complexité : 3n+1
- Complexité de SAT pour n variables : 2^n.

Complexité

O(.)

Une fonction $g(n)$ est dite $O(f(n))$ s'il existe c et n_0 telles que

$$\forall n > n_0, g(n) \leq cf(n)$$

La fonction $c_1 n^2 + c_2 n$ est $O(n^2)$ car

$$c_1 n^2 + c_2 n \leq (c_1 + c_2) n^2$$

Factorielle(n) a une complexité en $O(n)$, car

$$3n + 1 \leq 100n$$

Qu'est-ce qu'un algorithme efficace ?

$$O(n^2), O(n^{100}), O(2^n), O(4), O(log(n)), O(3n)\,?$$

Algorithmes

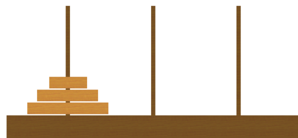
- Polynomiaux (P)
- Non-Polynomiaux (NP)

Conjecture

$$P \neq NP$$

http://qwiki.stanford.edu/wiki/Complexity_Zoo

Tour de Hanoï (Edouard Lucas 1883)



Règles

- Ne jamais poser un disque sur un disque plus petit.
- Ne déplacer qu'un disque à la fois.

BUT : Déplacer les 64 disques d'une colonne sur une autre.

Algorithme

D M A

D M A

D M A

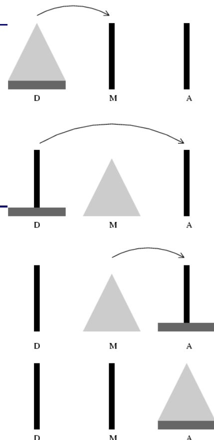
D M A

Déplacer n disques de A vers C en passant par B :

- Déplacer (n-1) disques de A vers B en passant par C ;
- Déplacer 1 disque de A vers C ;
- Déplacer (n-1) disques de B vers C en passant par A.

Code Ocaml

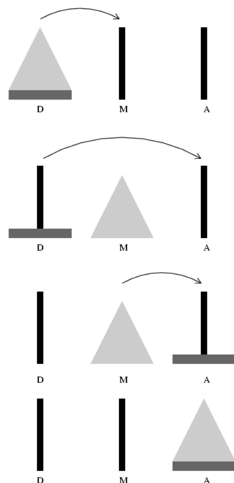
```
let mouvement de vers =
print_string ("Deplace un disque de  "^de^" vers "^ vers);
print_newline();;
```



Exemple :

```
hanoi "A" "B" "C" 3 ;;

(* Deplace un disque de  A vers C
Deplace un disque de  A vers B
Deplace un disque de  C vers B
Deplace un disque de  A vers C
Deplace un disque de  B vers A
Deplace un disque de  B vers C
Deplace un disque de  A vers C *)
```



Calcul du nombre de mouvement

```
let rec compte_hanoi_naif   = function
| 0 -> 0
| n -> compt_hanoi_naif (n-1)+
       1+
       compte_hanoi_naif  (n-1);;

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

- compte-hanoi 3 = 7
- compte-hanoi 10 = 1023
- compte-hanoi 16 = 65535

Nombre de mouvements : Conjecture

Pour n disques, il faut $2^n - 1$ mouvements

Preuve par récurrence

Encore plus vite

```
let compte_hanoi_rapide n = power 2.0 n - .1.0;;
```

Comparaison

- compte-hanoi-naif : $2^n - 1$ additions ($O(2^n)$)
- compte-hanoi : n multiplications ($O(n)$)
- compte-hanoi-rapide : 2 exp et soustraction ($O(2)$)

Si 100 millions d'additions par seconde $n = 64$, nous avons plus de 5845 années, un temps linéaire ou bien constant.

Exemple : factorielle

```
let rec fact  n = if n = 0 then 1 else n* (fact (n-1));;

let rec factaux x res =
  match x with
  |0 -> res
  |n -> factaux (x-1) (n * res)
let fact n = factaux n 1 ;;
```

Quelle est la différence entre ces deux fonctions ?

Récursivité terminale

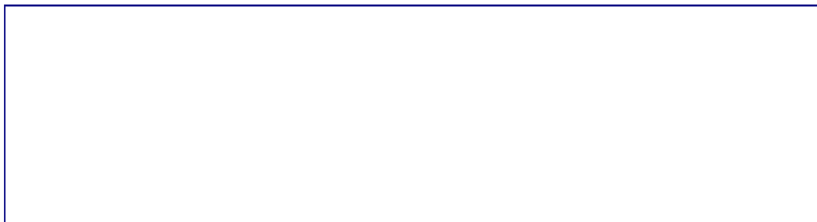
Définition

Une fonction f est récursive terminale (ou final recursive) si tout appel récursif est de la forme $f(...);$;

La valeur retourner est directement la valeur issue de l'appel récursif, sans qu il n'y ait aucun autre opération sur cette valeur.
Lors de la compilation une récursion terminale devient une itération.
Ainsi économie de la pile d'exécution, donc espace constant

Exemple : Minimum d'une liste

```
let rec minlist  l  = match l with
| [] -> failwith  "liste vide"
| x :: r - > min (x , minlist r) ;;
```



Deux fonctions ...

```
let rec f x = 0 + f x;;
```

```
val f : 'a -> int = <fun>
```

```
# f 0;;
```

Stack overflow during evaluation (looping recursion ?).

```
let rec f x =  f ( x + 1);;
```

```
val f : int -> 'a = <fun>
```

Boucle Infinie

Concurrence

Introduction aux concepts de la programmation parallèle, les modèles à mémoire partagée et à mémoire répartie.

Partage d'une zone mémoire

- Mémoire partagée. (P_i et une M)
- Mémoire répartie. (P_i et M_i)

Concurrence

Indépendance causale entre plusieurs actions, comme l'exécution de plusieurs instructions en « même temps ».

Dans les threads, le code et la zone mémoire des données sont partagés.

Exemple

Un programme et deux processus

```
------------------------------------
|     Programme principal          |
|                                  |
|       let x = ref 1;;            |
------------------------------------
|  processus P     |  processus Q   |
|                  |                |
|  x := !x + 1;;   |  x := !x * 2;; |
------------------------------------
```

À la fin de l'exécution de P et Q, que vaut la référence x

Fonctionnel vs impératif

- Impératif : chaque processus peut modifier la mémoire (partagée)
- Fonctionnel : chaque processus ne peut pas modifier la mémoire (distincte)

Thread

Permet de lancer de nouveaux processus correspondants à l'appel d'une fonction avec son argument.

Mutex et Condition

Outils de synchronisation que sont les verrous de zone d'exclusion mutuelle et les attentes sur condition.

Event

Implante un mode de communication de valeurs du langage par événement.

5 modules en Ocaml

- `Thread` : création et exécution de processus légers (type Thread.t) ;
- `Mutex` : création, pose et libération de verrous (type Mutex.t) ;
- `Condition` : création de conditions (signaux), attente et réveil sur condition (type Condition.t) ;
- `Event` : création de canaux de communication (type 'a Event.channel), des valeurs y transitant (type 'a Event.event), et des fonctions de communication.
- `ThreadUnix` : redéfinition des fonctions d'entrées-sorties du module Unix pour qu'elles ne soient pas bloquantes.

Commande

ocamlc -thread -custom threads.cma fichiers.ml -cclib -lthreads
ocamlmktop -thread -custom -o threadtop thread.cma -cclib -lthreads

Syntaxe

Processus léger

- Thread.create ; ;
  - : ('a -> 'b) -> 'a -> Thread.t = <fun>
- Thread.join t1 ; ; attend la fin de t1.
- Thread.kill t1 ; ; arrête l'execution de t1.
- Thread.exit ; ; met fin à sa propre activité.
- Thread.delay ; ; susspend pdt x secondes.
  - : float -> unit = <fun>

Mutex

```
# Mutex.create ;;
- : unit -> Mutex.t = <fun>
# Mutex.lock ;;
- : Mutex.t -> unit = <fun>
# Mutex.unlock ;;
- : Mutex.t -> unit = <fun>
# Mutex.try_lock;;
- : Mutex.t -> bool = <fun>
```

Si le verrou est déjà pris, la fonction retourne false. Dans l'autre cas, la fonction prend le verrou et retourne true.

Exemples

- Dîner des philosophes
- Producteur - Consommateur
- ...

Vocabulaire

- **Classe** : spécification d'jun ensemble d'objets
- **Objet** : élément ou instance de classe
- **Héritage** : relation d'extension/spécialisation entre classe
- **Attribut** ou champs : données nommées d'une classe ou d'une instance
- **Méthode** : fonction appartenant à une classe ou une instance
- **Appel** (de méthode) : activation (du code) d'une méthode.

Syntaxe

Class id id_1, ..., id_n =
  object
...
    val id = expr
...
    val mutable id = expr
...
    method id id_1, ..., id_n = expr
...
end

Exemple *

```
class counter = object
  val mutable n = 0
  method incr = n <- n+1
  method reset = n <- 0
  method get = n end;;
```

En pratique

```
let c = new  counter;;

c#incr; c#incr; c#get;;

(new counter)#get;;

(new counter) = (new  counter);;
```

Classes et paramétres *

```
class cpt c0 d = object
  val mutable c = c0
  method incr = c <- c+d
  method reset = c <- c0
  method get = c
 end;;
 class cpt :  int -> int ->  object
    val mutable c : int
    method get : int
    method incr : unit
    method reset : unit
  end
```

En pratique

```
let c = new cpt 0 1;;
val c : cpt = <obj>
```

Héritage

$$\textbf{inherit}\ \text{id}\ expr_1 \ldots expr_n$$

- id : nom classe mère
- exp_i : paramètres de création

- La classe fille hérite des variables et méthode de la classe mère.
- méthode de la classe mère sont accessibles dans la définition
- Méthodes de la classe mère et fille sont utilisables.

Initialisation

inithializer exp

```
class cpt1 c0 s = object
  initializer Printf.printf "Nouveau compteur \n "
  ...
```

Exemple*

```
class cpt1 c0 s = object
 inherit cpt c0 s
 method to_string = Printf.sprintf
     "< init=%d; step = %d; value =%d >" c0 s c
end;;
```

Pratique

```
let c= new cpt1 5 2 ;;
c#incr; c#get;;
c#to_string;;
```

self*

Une méthode ne peut être invoquée hors d'une instance/objet
Mais une classe peut définir un objet qui utilise ses propres méthodes.

object (id)

exemple

```
class cpt1 =
   object (moi)
     inherit cpt 0 1
     method double = moi#incr;moi#incr
end;;
```

L'usage est de prendre self à la place de moi. :-)

Redéfinition (super)

inherit ... **as** id

Utiliser les méthodes de la classe mère et même les redéfinir.

```
class cpt2 =
   object (moi)
     inherit cpt 0 1 as super
     method double = super#reset;super#incr;super#incr
end;;
```

Changement de double en triple ;-)

Héritage multiple

```
class counter c0 = object
  val mutable n = 0
  method get = n
  method incr = n <- n+1
end;;

class aff  = object
 method print_valv = Printf.sprintf
      "la valeur de %s vaut :%d"  s v
end;;

class cpt c0  = object (self)
 inherit aff
 inherit counter 0
 method  double = self#incr;self#incr;
    self#print_val "counter " self#get
end;;
```

Aujourd'hui

- Complexité
- Primitivité récursive
- Threads
- Objet

Note LP2

Note finale = 70% Examen + 10 % CC + 20% TP

1 RV + 1 exo fait.

CC

- Exercices rendus chaque semaine.
- Corrigé par le professeur en début de TD.
- Seulement 5 copies corrigés par semaine choisies au hasard par un programme.
- Correction de 2 exercices par étudiant.
- Note = Max des 2 notes.
- Si absence justifiée pas de correction (prévenir à l'avance).

Possibilité de rendre d'autres travaux pour correction détaillée.

Note LP2

Note finale = 70% Examen + 10 % CC + 20% TP

TP : 4 TPs + 1 TP-Projet

- TP = 60% TP-Projet + 40% TPs
- TPs et TP-Projet par binôme.
- TP-Projet à rendre pour le 22 Avril 2010 minuit par email.
- 4 TPs à rendre une semaine après le TP par email (minuit).
- Note TPs = Max des 2 TPs corrigés
- Si TP pas rendu alors 0.
- Si TOUS les TPs rendus +1 note TP.
- Si absence justifiée pas de correction (prévenir à l'avance).

Possibilité de demander une correction ciblée de parties de TP.

Programme du Cours

1. Bases de OCaml (Objective Categorical Abstract Machine Language) ...
2. Récurrence structurelle
3. Evaluation et compléments en Ocaml.
4. Modules, Foncteurs, Input Output
5. Exception, flots
6. Inférence de type (Typage)
7. Polymorphisme, référence, mutable.
8. Lambda Calcul
9. Lambda Calcul
10. Mémoire et point fixe

Programme TD

- listes
- arbres
- tas
- tris
- files
- flots
- foncteurs
- modules
- graphes
- automates
- typage
- ....

Programme TP

1. Introduction à OCAML
2. Code de Huffmann
3. Modules, foncteurs, graphes
4. Analyse lexicale par flots
5. Espace, graphisme : PROJET Doom.

Objectif du cours

Savoir faire

- Changer de paradigme.
- Programmer dans un langage FONCTIONNEL.
- Conception d'algorithmes efficaces et corrects.
- Initiation aux calculs de complexité.

Outils

- Programmation fonctionnelle en Objective Caml.
  (Caml = Categorical Abstract Machine Language)
- Typage
- Raisonnement par récurrence structurelle.
- Preuve de programmes.

Compétences acquises

- programmation fonctionnelle
- travail en binôme
- respect des deadlines
- organisation personnelle (gestion du temps)

CONNAÎTRE plusieurs langages de programmation pour CHOISIR le mieux approprié dans chaque situation

Prochaine fois

Examen 3h00
Salle XXX
La semaine du 17 mai 2010

Conclusion

Merci de votre attention

"J'entends, J'oublie

Je vois, Je me rappelle

Je fais, Je comprends

Confucius

Bibliogrpahie

- Pierre Wolper “Introduction à la Calculabilité”
- Pierre Weis et Xavier Leroy “Le Langage Caml”