Slide 1

Le passage par valeur et le passage par référence - La récursivité

Slide 2

0. Résumé des épisodes précédents

Slide 3

Le noyau impératif, puis la notion de fonction

Θ(t,e,m,G) = (v,m')

Σ(p,e,m,G) = (normal, m') ou (return,v,m')

Slide 4

Pour les fonctions

Θ(f(t_1, ..., t_n),...)

On va chercher le corps p de la fonction dans G

On l'exécute Σ(p,e',m',G)

On récupère le résultat

Slide 5

I. Le passage par valeur et le passage par référence

Slide 6

```
class Troisverres {
 static int a;
 static int b;

 public static void main (String [] args) {
  a = 4;
  b = 7;
  int c = a; a = b; b = c;
  System.out.println(a);
  System.out.println(b);}}
```

Slide 7

```
static int a;
static int b;

static void swap (int x, int y) {
 int z = x; x = y; y = z;}

public static void main (String [] args) {
 a = 4;
 b = 7;
 swap(a,b);
 System.out.println(a);
 System.out.println(b);}
```

Slide 8

Ce que l’on fait

ce que l’on veut faire

Slide 9

Une syntaxe explicite (Pascal)

```
procedure swap(x : integer, y : integer)
...
```

```
procedure swap(var x : integer, var y : integer)
...
```

Slide 10

Caml sans le !

Θ`(x,[x = r],[r = 4]) = r` (et non 4)

```
let swap x y =
 let z = ref 0 in (z := !x; x := !y; y := !z)
```

```
a:= 4;
b := 7;
swap a b;
print_int !a;
print_newline();
print_int !b;
print_newline()
```

En C comme en Caml

Θ`(x,[x = r],[r = 4]) = 4` (et non `r`)

Mais nouvelle construction `&` (inverse du `!` de Caml)

Θ`(&x,e,m) = r`

`*` homologue du `!` de Caml

`*t = u` homologue du `t := u` de Caml

Slide 11

`x = &a` construit `e = [a=r,x=r'], m = [r=4,r'=r]`

En C comme en Caml

```
void swap(int* x, int* y) {
 int z;
 z = *x; *x = *y; *y = z;}
int main () {
 a = 4;
 b = 7;
 swap(&a,&b);
 printf("%d\n",a);
 printf("%d\n",b);
 return 0;}
```

Slide 12

Et en Java ?

Les types enveloppés

(à suivre)

II. Les fonctions Θ et Σ sont-elle bien définie ?

Slide 15

Θ(t + u,e,m,G) = (v + w,m'')

où (v,m') = Θ(t,e,m,G)

et (w,m'') = Θ(u,e,m',G),

t et u sont des expressions plus petites que t + u

Définition par récurrence

Slide 16

Avec les fonctions

Σ(f(t_1,...,t_n),e,m,G)

utilise

Σ(p,e'',m'',G)

où p est le corps de f

Bien formée ?

Slide 17

Si

– appel des fonctions f, g, ... dans le programme principal,

– pas d'appel de fonctions dans le corps de f, g, ...

alors définition de Θ et Σ bien formée

Slide 18

Si

– fonctions f_1, f_2, f_3, ..., f_n,

– uniquement appel de f_1, ..., f_{i-1} dans le corps de f_i

alors définition de Θ et Σ bien formée

Fortran

Isolations successives de morceaux de programme

Slide 19

Le cas général

L'environnement global G est global

Toutes les fonctions peuvent être appelées dans le corps de chaque fonction

```
static int f (final int x) {return f(x);}
```

Θ`(f(x),e,m,G)` utilise Θ`(f(x),e,m,G)`

Définition circulaire de Θ

Slide 20

III. Appeler une fonction dans le corps de cette même fonction

Slide 21

Très utile

```
static int fact(final int x) {
 if (x == 0) return 1;
 return x * fact(x - 1);}
```

Θ`(fact(x),[x = 4],[],G)` est défini en utilisant

Θ`(fact(x),[x = 3],[],G)` qui est défini en utilisant

Θ`(fact(x),[x = 2],[],G)` qui est défini en utilisant

Θ`(fact(x),[x = 1],[],G)` qui est défini en utilisant

Θ`(fact(x),[x = 0],[],G)` qui vaut `(1,[])`

Slide 22

Les définitions mutuellement récursives

```
static boolean pair (final int n) {
 if (n == 0) return true;
 return impair(n - 1);}

static boolean impair (final int n) {
 if (n == 0) return false;
 return pair(n - 1);}
```

Slide 23

Les déf. récursives ne sont pas des déf. par récurrence

```
static int f (final int n) {
 if (n <= 1) return 1;
 if (n % 2 == 0) return (1 + f(n / 2));
 return 2 * f(n + 1);}
```

Le calcul la valeur de `f` en 11 demande celui de sa valeur en 12 qui demande celui de sa valeur en 6 qui demande celui de sa valeur en 3 qui demande celui de sa valeur en 4 qui demande celui de sa valeur en 2 qui demande celui de sa valeur en 1

Slide 24

Plus intéressant : la fonction d'Ackermann

$A_0(y) = 2\,y$

$A_1(y) = 2^y$

$A_1(0) = 1$

$A_1(y+1) = 2\,A_1(y) = A_0(A_1(y))$

$A_2(y) = 2^{2^{2^{2^{2^{1}}}}}$ (y fois)

$A_2(0) = 1$

$A_2(y+1) = 2^{A_2(y)} = A_1(A_2(y))$

Slide 25

$A_{x+1}(y) = A_x(A_x(A_x(\ldots A_x(1) \ldots)))$ (y fois)

$A_{x+1}(0) = 1$

$A_{x+1}(y+1) = A_x(A_{x+1}(y))$

Slide 26

La table de la fonction d'Ackermann

	0	1	2	3	4	5
A_0	0	2	4	6	8	10
A_1	1	2	4	8	16	32
A_2	1	2	4	16	65536	2^{65636}
A_3	1	2	4	65536	$2^{2^{2^{\cdots}}}$ (*)	?

(*) 65636 fois

Slide 27

Définition récursive simple

```
static int ack(final int x, final int y) {
 if (x == 0) return 2 * y;
 if (y == 0) return 1;
 return ack(x - 1,ack(x,y - 1));}
```

mais $A_n(n)$ ne peut pas être définie en imbriquant des définitions par récurrence

(Ackermann,1928)

Slide 28

Les définitions récursives ne sont pas des définitions circulaires

Sinon

```
static int fact(final int x) {
 return fact(x);}
```

serait correcte

Slide 29

IV. Définitions récursives et programmes infinis

Slide 30

Éviter la récursivité dans la définition de `fact`

```
static int fact(final int x) {
 if (x == 0) return 1;
 return x * fact(x - 1);}
```

Éviter la récursivité dans la définition de `fact`

Slide 31

```
static int fact(final int x) {
 if (x == 0) return 1;
 return x * fact1(x - 1);}
```

Éviter la récursivité dans la définition de `fact`

Slide 32

```
static int fact1(final int x) {
 if (x == 0) return 1;
 return x * fact1(x - 1);}
```

```
static int fact(final int x) {
 if (x == 0) return 1;
 return x * fact1(x - 1);}
```

Éviter la récursivité dans la définition de `fact`

```
static int fact2(final int x) {
 if (x == 0) return 1;
 return x * fact2(x - 1);}
```

Slide 33

```
static int fact1(final int x) {
 if (x == 0) return 1;
 return x * fact2(x - 1);}
```

```
static int fact(final int x) {
 if (x == 0) return 1;
 return x * fact1(x - 1);}
```

Un programme non récursif

... mais infini

Slide 34

Récursivité : notation finie pour une instruction infinie

Comme la boucle `while`

Potentialité de non terminaison

Approximations

Slide 35

On remplace `fact10` par `giveup`

Tente de calculer la `fact(n)` avec 10 appels récursifs imbriqués au maximum

Abandonne sinon

Les approximations de la factorielle

Slide 36

0	1	2	3	4	5
1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
1	1	$\perp$	$\perp$	$\perp$	$\perp$
1	1	2	$\perp$	$\perp$	$\perp$
1	1	2	6	$\perp$	$\perp$
1	1	2	6	24	$\perp$

Slide 37

```
static int f (final int n) {
 if (n <= 1) return 1;
 if (n % 2 == 0) return (1 + f(n / 2));
 return 2 * f(n + 1);}
```

0	1	2	3	4	5
1	1	$\perp$	$\perp$	$\perp$	$\perp$
1	1	2	$\perp$	$\perp$	$\perp$
1	1	2	$\perp$	3	$\perp$
1	1	2	6	3	$\perp$

Slide 38

Les fonctions Θ_k et Σ_k

Éviter de répliquer les fonctions un nombre infini de fois

$\Theta_k(t,e,m,G)$: idem Θ si moins de k appels de fonctions imbriqués (et pas définie sinon)

Même définition que Θ sauf cas appel de fonction

$\Theta_k(f(t_1,...,t_n),e,m,G)$

utilise $\Sigma_{k-1}(p,e',m',G)$ si $k > 0$

et $\Theta_0/\Sigma_0(f(t_1,...,t_n),e,m,G)$ n'est pas définie

récurrence sur k

Les fonctions Θ et Σ

Slide 39

Θ(t,e,m,G) = lim$_k$ Θ_k(t,e,m,G)

Σ(t,e,m,G) = lim$_k$ Σ_k(t,e,m,G)

Fonctions récursives et définitions au point fixe

Slide 40

Une alternative pour définir les fonctions Θ et Σ

Voir la définition de la factorielle comme une équation

f = x $\mapsto$ si (x == 0) alors 1 sinon x * f(x-1)

f = Φ(f)

Équation au point fixe

Slide 41

Toutes les fonctions ont un point fixe

Dans l'espace des fonctions partielles

Exemples :

`f = x ↦ si (x == 0) alors 1 sinon x * f(x-1)`

`f = x ↦ 1 + f(x)`

`f = x ↦ f(x)`

La fonction la moins définie

Même Θ / Σ (car point fixe = limite)

Slide 42

V. Programmer sans affectation

Slide 43

La factorielle et la factorielle

La boucle et la récursivité : deux moyens (redondants) de construire des programmes infinis

```
static int fact(final int x) {
 int i; int r;
 r = 1;
 for (i = 1; i <= x; i = i + 1) {r = r * i;}
 return r;}
```

```
static int fact(final int x) {
 if (x == 0) return 1;
 return x * fact(x - 1);}
```

Slide 44

Dans la définition récursive

```
static int fact(final int x) {
 if (x == 0) return 1;
 return x * fact(x - 1);}
```

Pas d’affectation (=)

On supprime l’affectation

Toutes les variables sont finales

Séquence et boucle inutiles

Slide 45

Il reste ...

La déclaration de variables finales

Le test

Les définitions récursives de fonctions

= Le noyau fonctionnel du langage

Slide 46

Java = noyau impératif = noyau fonctionnel

Une expression `t`

Une fonction partielle qui à l'entier `n` on associe la valeur `v` telle que $(\mathtt{v},\mathtt{m'}) = \Theta(\mathtt{t},[\mathtt{x} = \mathtt{n}],[],\mathtt{G})$

Exemple : à `2 * x + 3` on associe $\mathtt{x} \mapsto \mathtt{2 * x + 3}$

Idem pour les instructions

À un langage on associe un ensemble de fonctions : les fonctions programmables dans ce langage

Slide 47

Java = noyau impératif = noyau fonctionnel

L'ensemble des fonctions programmables

- en Java,
- dans le noyau impératif,
- dans le noyau fonctionnel

est le même (ensemble des fonctions calculables)

[Faux si les définitions de fonctions ne sont pas récursives]

Slide 48

Execices 2.13, 3.3.

Slide 49

La prochaine fois : les enregistrements