

7. Fonctions récursives primitives

Bruno Grenet

Université Grenoble Alpes – IM²AG
L3 Informatique
UE Modèles de calcul – Machines de Turing



<https://membres-ljk.imag.fr/Bruno.Grenet/MCAL-MT.html>

Introduction

Les fonctions récursives primitives

- ▶ Une tentative de formalisation des algorithmes
- ▶ Début du $xx^{\text{ème}}$, avant les machines de Turing, λ -calcul, fonctions μ -récursives
- ▶ Formalisation insuffisante : certaines fonctions intuitivement calculables ne sont pas récursives primitives

Intérêt moderne

- ▶ Correspond à un langage avec des boucles *for* mais sans *while*
 - ▶ on ne peut pas tout programmer avec seulement des boucles *for*
- ▶ Une fonction récursive primitive ne peut pas être Turing-complète

Plan de ce cours

1. Description du langage LOOP \rightarrow fonctions récursives primitives
2. Description de la fonction d'Ackermann-Péter
3. La fonction d'Ackermann-Péter n'est pas récursive primitive

Table des matières

1. Le langage LOOP

2. La fonction d'Ackermann-Péter

3. Les limites du langage LOOP

Table des matières

1. Le langage LOOP

2. La fonction d'Ackermann-Péter

3. Les limites du langage LOOP

Description du langage LOOP

A. M. Meyer & D. M. Ritchie (1967)

Instructions

Variables x_0, x_1, \dots contenant des entiers

- ▶ $x_j \leftarrow 0$
- ▶ $x_i \leftarrow x_j$
- ▶ $x_j \leftarrow x_j + 1$
- ▶ LOOP x_i :
 P

P : suite d'instructions

Utilisation du langage

- ▶ Entrées dans x_1, x_2, \dots ; sortie dans x_0
- ▶ Variables initialisées à 0 (sauf les entrées)
- ▶ LOOP : répète x_i fois les instructions P
 - ▶ valeur x_i fixée avant l'entrée dans la boucle
 - ▶ on peut supposer que $x_i \notin P$

→ Fonctions de \mathbb{N}^k dans \mathbb{N}

et autre *via* codage

Exemples

Addition

Entrées: $x_1 = a$

$x_2 = b$

Sortie: $x_0 = a + b$

1. $x_0 \leftarrow x_1$

2. Loop x_2 :

3. $x_0 \leftarrow x_0 + 1$

Multiplication

Entrées: $x_1 = a$

$x_2 = b$

Sortie: $x_0 = a \times b$

1. Loop x_1 :

2. Loop x_2 :

3. $x_0 \leftarrow x_0 + 1$

Décrément

Entrée: $x_1 = a$

Sortie: $x_0 = a \div 1$
 $= \max(0, a - 1)$

1. Loop x_1 :

2. $x_0 \leftarrow x_2$

3. $x_2 \leftarrow x_2 + 1$

4. $x_2 \leftarrow 0$

	x_0	x_1	x_2
	0	3	0
1	0	—	1
2	1	—	2
3	2	—	3
	2	—	0

Instructions avancées

LOOP est minimaliste mais on peut programmer les instructions standard

Arithmétique

- ▶ $x_i \leftarrow x_i + x_j$
 1. LOOP x_j :
 2. $x_i \leftarrow x_i + 1$

- ▶ $x_i \leftarrow x_i + n$ où $n \in \mathbb{N}$
 1. $x_i \leftarrow x_i + 1$
 2. $x_i \leftarrow x_i + 1$
 - ...
 - n . $x_i \leftarrow x_i + 1$

- ▶ ...

Test d'inégalité

$$x_i \leftarrow \begin{cases} 1 & \text{si } x_j < x_k \\ 0 & \text{sinon} \end{cases}$$

1. $x_i \leftarrow 0$
2. $x_{k'} \leftarrow x_k$
3. LOOP x_j :
4. $x_{k'} \leftarrow x_{k'} \div 1$
5. LOOP $x_{k'}$:
6. $x_i \leftarrow 1$
7. $x_{k'} \leftarrow 0$

Test d'égalité

$$x_i \leftarrow \begin{cases} 1 & \text{si } x_j = x_k \\ 0 & \text{sinon} \end{cases}$$

1. $x_i \leftarrow 1$
2. $x_t \leftarrow (x_j < x_k)$
3. LOOP x_t :
4. $x_i \leftarrow 0$
5. $x_t \leftarrow (x_k < x_j)$
6. LOOP x_t :
7. $x_i \leftarrow 0$
8. $x_t \leftarrow 0$

Structures de contrôle

Boucle *for*

FOR $x_i = 1$ to n : $P_{(x_i)}$

1. $x_b \leftarrow n$
2. $x_j \leftarrow 1$
3. LOOP x_b :
4. $P_{(x_i)}$
5. $x_i \leftarrow x_i + 1$
6. $x_b \leftarrow 0$
7. $x_j \leftarrow 0$

Branchement conditionnel

IF $x_i \neq 0$ THEN P ELSE Q

1. $x_p \leftarrow 0$
2. $x_q \leftarrow 1$
3. LOOP x_j :
4. $x_p \leftarrow 1$
5. $x_q \leftarrow 0$
6. LOOP x_p :
7. P
8. LOOP x_q :
9. Q
10. $x_p \leftarrow 0$
11. $x_q \leftarrow 0$

$$\left. \begin{array}{l} 1. x_p \leftarrow 0 \\ 2. x_q \leftarrow 1 \end{array} \right\} x_p \leftarrow \begin{cases} 1 & \text{si } x_i \neq 0 \\ 0 & \text{sinon} \end{cases}$$
$$\left. \begin{array}{l} 4. x_p \leftarrow 1 \\ 5. x_q \leftarrow 0 \end{array} \right\} x_q \leftarrow \begin{cases} 0 & \text{si } x_i \neq 0 \\ 1 & \text{sinon} \end{cases}$$

Des programmes LOOP (un peu plus) évolués

Nombres de Fibonacci

$$F_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-2} + F_{n-1} & \text{si } n \geq 2 \end{cases}$$

1. $x_2 \leftarrow 0$ F_0

2. $x_3 \leftarrow 1$ F_1

3. LOOP x_1 :

4. $x_4 \leftarrow x_2 + x_3$ F_{i+1}

5. $x_2 \leftarrow x_3$ F_i

6. $x_3 \leftarrow x_4$ F_{i+1}

7. $x_0 \leftarrow x_2$

8-10. $x_2, x_3, x_4 \leftarrow 0$

Division euclidienne

$$x_i \leftarrow x_j \bmod x_k$$

1. $x_b \leftarrow 1$

2. $x_t \leftarrow x_j$

3. LOOP x_j :

4. IF $x_b \neq 0$:

5. IF $x_t < x_k$:

6. $x_i \leftarrow x_t$

7. $x_b \leftarrow 0$

8. ELSE:

9. $x_t \leftarrow x_t - x_k$

10. $x_t \leftarrow 0$

Primalité

$$x_i \leftarrow \begin{cases} 1 & \text{si } x_j \text{ est premier} \\ 0 & \text{sinon} \end{cases}$$

1. $x_i \leftarrow 1$

2. FOR $x_k = 2$ to $x_j - 1$:

3. IF $x_j \bmod x_k = 0$:

4. $x_i \leftarrow 0$

Bilan

Langage LOOP

- ▶ Langage minimaliste
- ▶ Constructions possibles pour des instructions de *haut niveau*
- ▶ Branchements conditionnels et boucle *for*

Exemples

- ▶ On arrive à programmer beaucoup de fonctions de $\mathbb{N}^k \rightarrow \mathbb{N}$
- ▶ Fonctions $\Sigma^* \rightarrow \Sigma^*$: nécessite un codage
 - ▶ aucune difficulté théorique
 - ▶ aucun intérêt pratique...

bijection $\mathbb{N} \leftrightarrow \Sigma^*$

- ▶ Le langage LOOP permet-il de tout calculer ?
- ▶ Peut-on faire des boucles *while* ?

Table des matières

1. Le langage LOOP

2. La fonction d'Ackermann-Péter

3. Les limites du langage LOOP

Les hyperopérations

Remarque de départ

$a + b$: b itérations de l'opération « +1 » à partir de $a = a + 0$

$a \times b$: b itérations de l'opération « + a » à partir de $0 = a \times 0$

a^b : b itérations de l'opération « $\times a$ » à partir de $1 = a^0$

Et si on continuait ?

Hyperopérations

- ▶ $H_0(a, b) = 1 + b$
- ▶ $H_1(a, b) = a + b = 1 + (a + (b - 1))$
- ▶ $H_2(a, b) = a \times b = a + (a \times (b - 1))$
- ▶ $H_3(a, b) = a^b = a \times (a^{b-1})$
- ▶ ...

Goodstein (1947)

$$H_n(a, b) = \begin{cases} b + 1 & \text{si } n = 0 \\ a & \text{si } n = 1 \text{ et } b = 0 \\ 0 & \text{si } n = 2 \text{ et } b = 0 \\ 1 & \text{si } n \geq 3 \text{ et } b = 0 \\ H_{n-1}(a, H_n(a, b - 1)) & \text{sinon} \end{cases}$$

Notation et croissance

Puissances itérées

$$a \uparrow^0 b = H_2(a, b) = a \times b$$

$$a \uparrow^1 b = H_3(a, b) = a^b = a \uparrow b$$

$$a \uparrow^n b = H_{n+2}(a, b)$$

Knuth (1976)

Croissance: $2 \uparrow^n b$

$n \setminus b$	1	2	3	4	5	6
1	2	4	8	16	32	64
2	2	4	16	65536	2^{65536}	$2^{2^{65536}}$
3	2	4	65536	$2^{2^{\dots^2}} \}^{65536}$	$2^{2^{\dots^2}} \}^{2^{2^{\dots^2}} \}^{65536}$	$2^{2^{\dots^2}} \}^{2^{2^{\dots^2}} \}^{2^{2^{\dots^2}} \}^{65536}$
4	2	4	$2^{2^{\dots^2}} \}^{65536}$	$2 \uparrow^4 4$

Fonction d'Ackermann

Définition

$$\varphi(m, n, p) = \begin{cases} m + n & \text{si } p = 0 \\ 0 & \text{si } n = 0 \text{ et } p = 1 \\ 1 & \text{si } n = 0 \text{ et } p = 2 \\ m & \text{si } n = 0 \text{ et } p > 2 \\ \varphi(m, \varphi(m, n - 1, p), p - 1) & \text{sinon} \end{cases}$$

Ackermann (1928)

Remarque

- ▶ $\varphi(m, n, 3) = H_4(m, n + 1) = m \uparrow^2 (n + 1)$
- ▶ $\varphi(m, n, p) \simeq H_{p+1}(m, n + 1) = m \uparrow^{p-1} (n + 1)$

Fonction d'Ackermann-Péter(-Robinson)

Définition

Péter (1935), Robinson (1948)

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ A(m - 1, 1) & \text{si } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{sinon} \end{cases}$$

Remarque

► $A(m, n) = H_m(2, n + 3) \bar{\uparrow} 3$ pour $m > 0$
 $= 2 \uparrow^{m-2} (n + 3) \bar{\uparrow} 3$

Dans la suite, on utilise A mais ça reviendrait au même avec les autres fonctions

Calcul de la fonction $A(m, n)$

Algorithme récursif

$A(m, n)$:

1. Si $m = 0$: renvoyer $n + 1$
2. Si $n = 0$: renvoyer $A(m - 1, 1)$
3. Renvoyer $A(m - 1, A(m, n - 1))$

Version itérative ?

- ▶ Utilisation d'une pile d'appels → boulot du compilateur
- ▶ Boucle *while*: « tant que la pile d'appels est non vide: ... »
- ▶ Possible et pénible

À m fixé

- ▶ Pour tout m , il existe un programme LOOP pour calculer $A(m, \cdot)$
 - ▶ Utilise $\simeq m$ LOOPS imbriquées
- ▶ Ça n'implique pas qu'il existe un programme LOOP pour calculer $A(\cdot, \cdot)$
 - ▶ On ne peut pas avoir un nombre variables de LOOPS

Terminaison

Appels récursifs sur des couples (m', n') où

- ▶ $m' < m$, ou
- ▶ $m' = m$ et $n' < n$

Propriétés de la fonction d'Ackermann-Péter

Lemme

- ▶ $A(1, n) = n + 2$
- ▶ $A(2, n) = 2n + 3$
- ▶ $n + 1 \leq A(m, n) < A(m, n + 1) \leq A(m + 1, n)$ pour tout m, n

exercice

- $A(1, n) = A(0, A(1, n-1)) = A(1, n-1) + 1 = A(1, n-2) + 2 = \dots = A(1, 0) + n = A(0, 1) + n = 2 + n$
- $A(2, n) = A(1, A(2, n-1)) = A(2, n-1) + 2 = \dots = A(2, 0) + 2n = A(1, 1) + 2n = 3 + 2n$
- $A(m, n) \geq n + 1$: double récurrence :
 - $A(0, n) = n + 1$
 - $A(m, 0) \geq 0 + 1$ car $A(m, n) \geq 1$
 - $A(m, n) = A(m-1, A(m, n-1)) \geq A(m, n-1) + 1 \geq \underbrace{(n-1)}_{\text{réc. sur } n} + 1 + 1 = n + 1$
N *réc. sur m* *N* *réc. sur n*
- $A(m, n+1) = A(m-1, A(m, n)) \geq A(m, n) + 1 > A(m, n)$

Propriétés de la fonction d'Ackermann-Péter

Lemme

- ▶ $A(1, n) = n + 2$
- ▶ $A(2, n) = 2n + 3$
- ▶ $n + 1 \leq A(m, n) < A(m, n + 1) \leq A(m + 1, n)$ pour tout m, n

Théorème

- ▶ $A(m, A(m, n)) \leq A(m + 1, n)$ pour tout m, n
- ▶ $A(m_1, A(m_2, n - 1)) \leq A(m_2, n)$ pour $m_1 < m_2$

$$\cdot A(m+1, n) = A(m, A(m+1, n-1)) \geq A(m, A(m, n))$$

$$\cdot A(m_1, A(m_2, n-1)) \leq A(m_2-1, A(m_2, n-1)) = A(m_2, n)$$

Bilan

Fonctions de type Ackermann

- ▶ Toutes des variantes de la fonction φ d'Ackermann
- ▶ Objectifs : qu'elle croissent aussi vite que possible
- ▶ Algorithmes :
 - ▶ en théorie : aucune difficulté à les calculer
 - ▶ en pratique : absolument impossible de calculer au delà de toutes petites valeurs

Pourquoi les avoir présentées ?

- ▶ Impossible de les programmer dans le langage LOOP
- ▶ Il *faut* une boucle *while* (ou des appels récursifs, ce qui est équivalent)

À part ça, elles servent à quoi ?

- ▶ Structure de données *Union-Find*:
 - ▶ Partition de $\{1, \dots, n\}$ en sous-ensembles X_1, \dots, X_k
 - ▶ Opérations : $\text{UNION}(X_i, X_j) = X_i \cup X_j$; $\text{FIND}(x) = X_i$ si $x \in X_i$
- ▶ Complexité de chaque opération : $O(n\alpha(n))$ où $\alpha(n) = \min\{k : A(k, k) > n\}$

Table des matières

1. Le langage LOOP

2. La fonction d'Ackermann-Péter

3. Les limites du langage LOOP

LOOP et fonctions totales

Théorème (très facile !)

Une fonction $f : \mathbb{N}^k \rightarrow \mathbb{N}$ calculée par un programme LOOP est *totale*

Corollaire

Il existe des algorithmes non programmables avec LOOP

Preuve

thm: Un programme LOOP ne peut pas boucler \rightarrow il est défini pour toute entrée

Cor: - On a vu en TD une preuve par diagonalisation: si toutes les fonctions calculées sont totales, alors il existe des fonctions intuitivement calculables qu'on ne peut pas programmer. (exercice 9)

- Dans une formalisation correcte des algorithmes, le problème de l'arrêt doit être indécidable.

Croissance des fonctions LOOP

Théorème

Soit P un programme LOOP. Alors il existe un entier j tel que

$$S_P < A(j, S)$$

où

- ▶ S est la somme des variables en entrée
- ▶ S_P est la somme des variables en sortie
- ▶ $A(\cdot, \cdot)$ est la fonction d'Ackermann-Péter

Remarques

- ▶ j ne dépend que de P , pas de S
 - ▶ j est \simeq le nombre de boucles imbriquées
- ▶ Ce qui nous intéresse est la croissance des $x_i \rightarrow$ la somme est un *résumé*

Preuve du théorème

Cas de base

$$(1) S_p = S - x_i \leq S < A(0, S) = S+1 \quad j=0$$

$$(2) S_p = S+1 < A(1, S) = S+2 \quad j=1$$

$$(3) S_p = S + x_j - x_i \leq 2S < A(2, S) = 2S+3 \quad j=2$$

? Programme Loop

```

x_i ← 0 (1)
x_i ← x_i + 1 (2)
x_i ← x_j (3)
P; Q (4)
loop x_i : P (5)
    
```

Cas (4) $S \rightsquigarrow S_p \rightsquigarrow S_{p;Q}$

Par hyp. $\exists j_p \text{ tq } S_p < A(j_p, S)$

$\exists j_Q \text{ tq } S_{p;Q} < A(j_Q, S_p)$

$$j = 1 + \max(j_p, j_Q) = 1 + j_{\pi}$$

$$S_{p;Q} < A(j_Q, S_p) \leq A(j_Q, A(j_p, S)) \leq A(j_{\pi}, A(j_{\pi}, S)) \leq A(j, S)$$

Cas (5) Hyp: $\exists j_p \text{ tq } S_p < A(j_p, S)$

- (5) \Leftrightarrow
- $x_i \leftarrow x_i - 1$
 - Loop $x_i : P$
 - P
 - $x_i \leftarrow x_i + 1$

$$\begin{aligned}
 S_1 &= S-1 \\
 S_2 &< A(j, S-1) \\
 S_3 &< A(j_p, S_2) \\
 S_4 &= S_3+1
 \end{aligned}$$

Par récurrence^{sur x_i} : $S_{\text{final}} < A(j_p+1, S) \quad j=j_p+1$

- $x_i = 0$: ✓

- $x_i > 0$: $S_2 \leq A(j, S-1) - 1$

$$S_3 < A(j_p, A(j, S-1) - 1) \leq A(j_p, A(j, S-1)) - 1$$

$$S_4 = S_3 + 1 < A(j_p, A(j, S-1)) = A(j, S)$$

Conséquence

Corollaire

Il n'existe pas de programme LOOP pour calculer $A(m, n)$

Preuve

- Si un programme existe pour $A(m, n)$, un programme existe pour $A(m, m)$
- Soit P un programme LOOP qui calcule $A(m, m)$.
 - + Alors il existe j_p tq $S_p < A(j_p, S)$
 - + Sur l'entrée j_p , $S_p = A(j_p, j_p) < A(j_p, S)$
contradiction

Conclusion

Programmes LOOP

- ▶ Programmes itératifs, sans boucle *while*
- ▶ Permet de faire tout ce qu'on sait faire avec des *for* et *if ... then ... else*
- ▶ Programmes qui terminent toujours

Les fonctions calculées par les programmes LOOP sont dites **récurives primitives**

La boucle *while* est nécessaire

- ▶ Les fonctions récurives primitives ne *capturent* pas la notion d'algorithme
 - ▶ besoin des fonctions *partielles*
 - ▶ exemple de la fonction d'Ackermann-Péter : croissance trop rapide
- ▶ Il suffit de rajouter une boucle *while* pour obtenir un modèle Turing-complet
 - ▶ Langage WHILE
 - ▶ Langage GOTO