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

Table des matières

Description du langage LOOP

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

Instructions

Variables $x_0, x_1, \ldots$ contenant des entiers

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

P : *suite d'instructions*

Utilisation du langage

- Entrées dans $x_1, x_2, \ldots$; 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 $x_i \notin P$

$\rightarrow$ Fonctions de $\mathbb{N}^k$ dans $\mathbb{N}$ et autre *via* codage

Exemples

Addition

Entrées : $x_1 = a$
$x_2 = b$
Sortie : $x_0 = a + b$

Multiplication

Entrées : $x_1 = a$
$x_2 = b$
Sortie : $x_0 = a \times b$

Décrément

Entrée : $x_1 = a$
Sortie : $x_0 = a \dot{-} 1 =: \quad = \mathsf{m}$

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

  $\cdots$

  n. $x_i \leftarrow x_i + 1$
- $\ldots$

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. $\quad x_{k'} \leftarrow x_{k'} \dot{-} 1$
5. LOOP $x_{k'}$:
6. $\quad x_i \leftarrow 1$
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. $\quad x_i \leftarrow 0$
5. $x_t \leftarrow (x_k < x_j)$
6. LOOP x_t :
7. $\quad x_i \leftarrow 0$
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. $x_i \leftarrow 1$
3. LOOP x_b :
4. $\quad P_{(x_i)}$
5. $\quad x_i \leftarrow x_i + 1$
6. $x_b \leftarrow 0$
7. $x_i \leftarrow 0$

Branchement conditionnel

IF $x_i \neq 0$ THEN P ELSE Q

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

Des programmes LOOP (un peu plus) évolués

Nombres de Fibonacci

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

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. $x_t \leftarrow x_t \dot{-} x_k$
10. $x_t \leftarrow 0$

Primalité

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

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 $\mathbb{N}^k \to \mathbb{N}$
- Fonctions $\Sigma^* \to \Sigma^*$: 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

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

Goodstein (1947)

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

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

Notation et croissance

Puissances itérées

Knuth (1976)

$$a \uparrow^0 b = H_2(a,b) = a \times b$$
$$a \uparrow^1 b = H_3(a,b) = a^b = a \uparrow b$$
$$a \uparrow^n b = H_{n+2}(a,b)$$

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	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}65536$	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}65536$	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}65536$
4	2	4	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}65536$	$\cdots$	$\cdots$	$\cdots$

Fonction d'Ackermann

Définition

Ackermann (1928)

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

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

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

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 $A(m - 1, A(m, n - 1))$

Terminaison

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

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

Version itérative ?

- Utilisation d'une pile d'appels $\rightarrow$ 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

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

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 $m_1 < m_2$

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 $\{1, \ldots, n\}$ en sous-ensembles $X_1, \ldots, X_k$
  - Opérations : $\textsc{union}(X_i, X_j) = X_i \cup X_j$; $\textsc{find}(x) = X_i$ 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

LOOP et fonctions totales

Théorème (*très* facile !)

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

Corollaire

Il existe des algorithmes non programmables avec LOOP

Preuve

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

Conséquence

Corollaire

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

Preuve

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 **récursives primitives**

La boucle *while* est nécessaire

- Les fonctions récursives 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