Structures Discrètes 4

λ-calcul et μ-récursion

Jan-Georg Smaus

UT3/Dép. d'Info.

Année 2022/2023

Ce Cours

Dans les six séances données par Jan-Georg Smaus, nous allons étudier deux thèmes :

- Les fonctions μ-récursives, qui sont une classe de fonctions sur les entiers naturels définies par les ingrédients constante zéro, identité, succésseur, composition, récursion simple, minimalisation. Il s'avère que cette classe donne un formalisme de calcul qui est aussi puissant que les machines de Turing : chaque fonction μ-récursive peut être calculée par une machine de Turing, et chaque machine de Turing peut être simulée par une fonction μ-récursive.

Ce Cours

Dans les six séances données par Jan-Georg Smaus, nous allons étudier deux thèmes :

- Les fonctions μ-récursives, qui sont une classe de fonctions sur les entiers naturels définies par les ingrédients constante zéro, identité, succésseur, composition, récursion simple, minimalisation. Il s'avère que cette classe donne un formalisme de calcul qui est aussi puissant que les machines de Turing : chaque fonction μ-récursive peut être calculée par une machine de Turing, et chaque machine de Turing peut être simulée par une fonction μ-récursive.
- Leλ-calcul, qui est un formalisme d'expressions avec un certain mécanisme de réécriture, que vous avez déjà rencontré en étudiant OCaml. Il s'avère que dans le λ-calcul, on peut coder les entiers naturels et effectivement les fonctions μ-récursives, et donc le λ-calcul est un formalisme de calcul aussi puissant que les machines de Turing ou bien les fonctions μ-récursives.

Plan

Les fonctions numériques

Le point de vue ici est plus "mathématique" que "informatique" : nous ne regardons pas un formalisme de calcul ou réécriture comme les machines de Turing ou le λ-calcul (à voir plus tard), mais nous nous concentrons sur ce qui est calculé : des fonctions des nombres vers les nombres. Par exemple,

$$f(m,n) := m \cdot n^2 + 3 \cdot m^{2 \cdot m + 17}$$

peut être calculé pour toutes valeurs de m et n car c'est une compostion de fonctions toutes calculables, car toutes définies récursivement basées surs des fonctions plus simples ...
Nous considérons dans ce chapître des fonctions $\mathbb{N}^k \mapsto \mathbb{N}$.
Ce chapître suit très étroitement le livre de Lewis et Papadimitriou [1].

Plan

Les fonctions basiques

Nous commençons par quelques fonctions $\mathbb{N}^k \mapsto \mathbb{N}$ $(k \geq 0)$ très simples que nous appelons basiques :

Definition 1

1. Pour tout $k \geq 0$, la fonction zéro k-aire et la fonction $\text{zero}_k(n_1, \ldots, n_k) = 0$ pour tous $n_1, \ldots, n_k \in \mathbb{N}$.
2. Pour tout $k \geq j > 0$, la $j$ème fonction d'identité k-aire est la fonction $\text{id}_{k,j,}(n_1, \ldots, n_k) = n_j$ pour tous $n_1, \ldots, n_k \in \mathbb{N}$.
3. La fonction de succésseur est définie comme $\text{succ}(n) = n + 1$ pour tout $n \in \mathbb{N}$.

Composition et récursion

Ensuite nous introduisons deux manières simples de combiner des fonctions pour en obtenir des fonctions plus complexes :

Definition 2

1. Soit $k, \ell \geq 0$, soit g une fonction k-aire, et soient $h_1, \ldots, h_k$ des fonctions ℓ-aires. La composition de g avec $h_1, \ldots, h_k$ est la fonction ℓ-aire définie par

$$f(n_1, \ldots, n_\ell) = g(h_1(n_1, \ldots, n_\ell), \ldots, h_k(n_1, \ldots, n_\ell)).$$

2. Soit $k \geq 0$, soit g une fonction k-aire et h une fonction $(k+2)$-aire. La fonction définie récursivement par g et h est la fonction $(k+1)$-aire f

$$\begin{aligned} f(n_1, \ldots, n_k, 0) &= g(n_1, \ldots, n_k) \\ f(n_1, \ldots, n_k, m+1) &= h(n_1, \ldots, n_k, m, f(n_1, \ldots, n_k, m)) \end{aligned}$$

La récursion primitives

Definition 3

Les fonctions récursives primitives sont les fonctions basiques et toutes les fonctions que l'on peut obtenir en appliquant la composition et la récursion successivement.

Exemples : plus2 et succid3

Une instance du point 1 de la définition 2 : Soit $k = \ell = 1$, soit $g = \text{succ}$ (une fonction unaire), et soit $h_1 = \text{succ}$ aussi (une fonction unaire). La composition de g avec h_1 est donc la fonction unaire

$$f(n_1) = \text{succ}(\text{succ}(n_1)).$$

Appelons f plus2.

Exemples : plus2 et succid3

Une instance du point 1 de la définition 2 : Soit $k = \ell = 1$, soit $g = \text{succ}$ (une fonction unaire), et soit $h_1 = \text{succ}$ aussi (une fonction unaire). La composition de g avec h_1 est donc la fonction unaire

$$f(n_1) = \text{succ}(\text{succ}(n_1)).$$

Appelons f plus2.

Exercice 1

Définissez la fonction plus3 *(*plus3$(n) = n + 3$*) de la même manière.*

Exemples : plus2 et succid3

Une instance du point 1 de la définition 2 : Soit $k = \ell = 1$, soit $g = \text{succ}$ (une fonction unaire), et soit $h_1 = \text{succ}$ aussi (une fonction unaire). La composition de g avec h_1 est donc la fonction unaire

$$f(n_1) = \text{succ}(\text{succ}(n_1)).$$

Appelons f plus2.

Exercice 1

Définissez la fonction plus3 *(*$\text{plus3}(n) = n + 3$*) de la même manière.*

Exercice 2

Définissez la fonction ternaire $\text{succid3}(n_1, n_2, n_3) = \text{succ}(n_3)$.

Mais à quoi succid3 pourrait servir ? . . .

Exemple : plus

On veut définir la fonction binaire plus par récursion. Le point 2 de la définition 2 devient : Soit $k = 1$, soit g donc unaire et h ternaire. La fonction plus, définie récursivement par g et h est la fonction binaire

$$\begin{aligned} \text{plus}(n_1, 0) &= g(n_1) \\ \text{plus}(n_1, m+1) &= h(n_1, m, \text{plus}(n_1, m)) \end{aligned}$$

Exercice 3

Que doivent être g et h ?

Notation pour définir des fonctions

Pour alléger la notation et plus de flexibilité, nous serons désormais moins rigoureux dans la présentation de fonctions primitives récursives :

- L'argument récursif ne s'appelle pas forcément m et n'est pas forcément le dernier.
- Les fonctions g et h ne sont pas forcément explicitées :

$$f(\text{“des variables, et 0 dans une position”}) = \text{“expression de ces variables”}$$
$$f(\text{“des variables, et } \ldots + 1 \text{ dans une position”}) = \text{“expression de ces variables, de '}\ldots\text{' et de l'appel récursif”}$$

Exemples : mult, exp, fonction constante, sgn

$$\begin{aligned} \text{mult}(m, 0) &= \text{zero}(m) \\ \text{mult}(m, n+1) &= \text{plus}(m, \text{mult}(m, n)) \end{aligned}$$

Exemples : mult, exp, fonction constante, sgn

$$\begin{aligned} \text{mult}(m, 0) &= \text{zero}(m) \\ \text{mult}(m, n+1) &= \text{plus}(m, \text{mult}(m, n)) \end{aligned}$$

Exercice 4

Donnez des définitions pour

- exp ;
- *une fonction constante, par exemple* $f(n_1, \ldots, n_k) = 4$;
- *la fonction* sgn *qui rend* 0 *pour* 0 *et* 1 *pour tout autre argument.*

On monte d'un cran ...

Pour une meilleure lisibilité nous écrirons $m + n$ au lieu de plus(m, n), $m \cdot n$ au lieu de mult(m, n), $m \uparrow n$ au lieu de exp(m, n).
Toutes fonctions numériques comme

$$m \cdot (n + m^2) + 178^m$$

sont donc primitives récursives car elles peuvent être obtenues en composant les fonctions ci-dessus.

Exemple : pred

Comme nous sommes limités aux entiers naturels, nous ne pouvons pas espérer définir la soustraction et la division.
Mais nous définissons déjà le prédecesseur :

$$\begin{aligned} \text{pred}(0) &= 0 \\ \text{pred}(n+1) &= n \end{aligned}$$

ainsi que la soustraction non-négative que l'on dénote $\sim$:

$$\begin{aligned} m \sim 0 &= m \\ m \sim (n+1) &= \text{pred}(m \sim n) \end{aligned}$$

Prédicat

Un prédicat primitif récursif est une fonction primitive récursive dont les valeurs ne peuvent être que 0 (“faux”) ou 1 (“vrai”).

Exemples : quelques prédicats

$$\begin{aligned} \text{iszero}(0) &= 1 \\ \text{iszero}(m+1) &= 0 \end{aligned}$$

- $\text{isone}(0) = 0$, $\text{isone}(n+1) = \text{iszero}(n)$
- $\text{geq}(m,n) = \text{iszero}(n \sim m)$
- $\text{lt}(m,n) = 1 \sim \text{geq}(m,n)$
- ϕ or $\psi = 1 \sim \text{iszero}(\phi + \psi)$

Exercice 5

Définissez "ϕ and ψ".

- $\text{equals}(m,n) = \text{geq}(m,n)$ and $\text{geq}(n,m)$

Exemple : définition par cas

Soient f, g deux fonctions primitives récursives et p un prédicat primitf récursif, tous k-aires. La définition de f par cas :

$$f(n_1, \ldots, n_k) = \begin{cases} f(n_1, \ldots, n_k) & \text{si } p(n_1, \ldots, n_k) \\ g(n_1, \ldots, n_k) & \text{autrement} \end{cases}$$

et primitive récursive car elle peut être réécrite comme

$$f(n_1, \ldots, n_k) = p(n_1, \ldots, n_k) \cdot g(n_1, \ldots, n_k) + (1 \sim p(n_1, \ldots, n_k)) \cdot h(n_1, \ldots, n_k).$$

Exemples : Division avec reste

$$\begin{aligned} \mathrm{rem}(0,n) &= 0 \\ \mathrm{rem}(m+1,n) &= \begin{cases} 0 & \text{si } \mathrm{equals}(\mathrm{rem}(m,n),\mathrm{pred}(n)); \\ \mathrm{rem}(m,n)+1 & \text{autrement.} \end{cases} \end{aligned}$$

et

$$\begin{aligned} \mathrm{div}(0,n) &= 0 \\ \mathrm{div}(m+1,n) &= \begin{cases} \mathrm{div}(m,n)+1 & \text{si } \mathrm{equals}(\mathrm{rem}(m,n),\mathrm{pred}(n)); \\ \mathrm{div}(m,n) & \text{autrement.} \end{cases} \end{aligned}$$

Exemple : extraire des chiffres

Supposons que nous voulons extraire le 3ème chiffre le plus faible du nombre 4274688, c'est-à-dire 6. Pour extraire les 3 derniers chiffres, on prend le reste de la division de 4274688 par $10^3 = 1000$, c'est-à-dire 688. Pour en obtenir le premier et donc le résultat, on divise par $10^2 = 100$.

Exemple : extraire des chiffres

Supposons que nous voulons extraire le 3ème chiffre le plus faible du nombre 4274688, c'est-à-dire 6. Pour extraire les 3 derniers chiffres, on prend le reste de la division de 4274688 par $10^3 = 1000$, c'est-à-dire 688. Pour en obtenir le premier et donc le résultat, on divise par $10^2 = 100$. En général, le m-ième chiffre le plus faible de n dans la représentation à base p est défini comme

$$\text{digit}(m, n, p) = \text{div}(\text{rem}(n, p \uparrow m), p \uparrow (m \sim 1))$$

et donc primitif récursif.

Plan

Diagonalisation

Nous avons vu beaucoup d'exemples, de plus on plus complexes.
Mais les fonctions primitives récursives ne sont pas toutes les fonctions "calculables", même si on se limite à des fonctions unaires : toute fonction primitive récursive peut être écrite basée sur les fonctions de base. Donc elle peut être représentée comme une chaîne finie de caractères d'un alphabet fini (même si c'est très fastidieux à mettre en œuvre). Donc

Fait 1

L'ensemble des fonctions primitives récursives est énumérable.

Soit donc $f_0, f_1, f_2, \ldots$ la liste des fonctions primitives récursives. Nous définissons

$$g(n) = f_n(n) + 1$$

Clairement, g est calculable, mais $g \neq f_m$ pour tout m, c'est-à-dire, g n'est pas sur la liste des fonctions primitives récursives.

Calculable et reconnaissable

Toute manière de définir des fonctions calculables ne peut se limiter à des opérations simples comme la composition ou la récursion. De telles opérations produisent des fonctions qui peuvent facilement être identifiées comme telles, et donc énumérées. Une diagonalisation est ensuite possible. Nous définissons maintenant une opération plus subtile qui correspond à une itération non-bornée (une boucle *while*). Elle définie une classe de fonctions pour lesquelle l'appartenance à la classe ne peut pas toujours être décidée.

Minimalisation

Definition 4

Soit g une fonction $(k+1)$-aire, pour un $k \geq 0$. La minimalisation de g est la fonction k-aire f définie ainsi :

$$f(n_1, \ldots, n_k) = \begin{cases} \text{le } m \text{ minimal tel que } g(n_1, \ldots, n_k, m) = 1, \\ \quad \text{si un tel } m \text{ existe} \\ 0 \text{ autrement.} \end{cases}$$

On dénote la minimalisation de g (pas la fonction, mais bien le résultat de son application à $n_1, \ldots, n_k$) par $\mu\, m[g(n_1, \ldots, n_k, m) = 1]$.

La minimalisation d'une fonction g est toujours bien définie, mais il n'y a pas de manière évidente de la calculer ; la méthode

$m := 0$
while $g(n_1, \ldots, n_k, m) \neq 1$ do $m := m + 1$
return m

n'est pas un algorithme car elle n'est pas garantie de terminer

La μ-récursion

Definition 5

On appelle une fonction g minimalisable si pour tout $n_1, \ldots, n_k \in \mathbb{N}$, il existe un $m \in \mathbb{N}$ tel que $g(n_1, \ldots, n_k, m) = 1$, c'est-à-dire, si la méthode ci-dessus termine.
On appelle une fonction μ-récursive si elle peut être obtenue à partir des fonctions basiques en appliquant la composition, la récursion, et la minimalisation des fonctions minimalisables.

Comme on ne peut pas toujours reconnaître une fonction minimalisable, on ne peut pas toujours reconnaître une fonction μ-récursive.

Exemple : log

Nous définissons une version un peu spéciale (pour éviter des problèmes mathématiques) de log : $\log(m, n)$ est la puissance minimale de $m+2$ qui est au moins $n+1$ (c'est-à-dire, $\log(m, n) = \lceil \log_{m+2}(n+1) \rceil$).

$$\log(m, n) = \mu\, p[\mathrm{geq}((m+2) \uparrow p, n+1)].$$

Ceci est bel et bien une définition μ-récursive !
Il s'agit juste d'un simple exemple. La fonction log est en réalité primitive récursive.

Plan

La μ-récursivité et les machines de Turing

Theorem 6

Une fonction $f : \mathbb{N}^k \mapsto \mathbb{N}$ est μ-récursive si et seulement si ell est récursive (calculable par une machine de Turing)

Preuve : TODO, je compte une séance entière pour la présenter.

Rappel : machines de Turing

TODO

Plan

Plan

Le λ-calcul et la programmation fonctionnelle

Nous commençons par une introduction intuitive, faisant référence à des choses que vous connaissez. Plus tard, nous serons bien plus abstraits !
Le λ-calcul est la base théorique de la programmation fonctionnelle, que vous avec étudiée en ILU1 et ILU3 : le langage de programmation OCaml.
En fait, la lettre λ correspond à un mot clé souvent utilisé en OCaml, surtout au début : le mot fun. Quelques exemples pris d'ILU1 :

OCaml	λ-calcul
`fun a -> fun b -> not (a && b)`	$\lambda a.\lambda b.\neg(a \wedge b)$
`fun f -> fun g -> fun x -> f (g x)`	$\lambda f.\lambda g.\lambda x.\, f\,(g\,x)$
`fun x -> x + 10`	$\lambda x.\, x + 10$
`fun x -> x < 10`	$\lambda x.\,(x < 10)$
`fun x l -> x :: l`	$\lambda x\, l.\, x :: l$

On voit des expressions qui sont des fonctions sans nom.

Applications

En OCaml comme en λ-calcul, les applications de fonction à un argument s'écrivent en mettant la fonction suivie de l'argument.

OCaml	λ-calcul
`(fun x -> x + 10)  5`	$(\lambda x.\, x + 10)\ 5$

`(fun f -> fun g -> fun x -> f (g x))`
`  (fun x -> x < 10)`
`  (fun x -> x + 10) 5`
	$(\lambda f.\lambda g.\lambda x.\, f(g\, x))$
	$(\lambda x.\, (x < 10))$
	$(\lambda x.\, x + 10)\ 5$

Applications

En OCaml comme en λ-calcul, les applications de fonction à un argument s'écrivent en mettant la fonction suivie de l'argument.
En OCaml comme en λ-calcul, les applications mènent à une évaluation qui consiste en remplaçant un paramètre par un argument.

OCaml	λ-calcul
`(fun x -> x + 10) 5` $\longrightarrow$ `5 + 10`	$(\lambda x.\, x + 10)\ 5$ $\longrightarrow 5 + 10$

`(fun f -> fun g -> fun x -> f (g x))`
`(fun x -> x < 10)`
`(fun x -> x + 10) 5`
$\longrightarrow$ `(fun x -> x < 10)`
`((fun x -> x + 10) 5)`
	$(\lambda f.\lambda g.\lambda x.\, f(g\, x))$
	$(\lambda x.\, (x < 10))$
	$(\lambda x.\, x + 10)\ 5$
	$\longrightarrow (\lambda x.\, (x < 10))$
	$((\lambda x.\, x + 10)\ 5)$

Applications

En OCaml comme en λ-calcul, les applications de fonction à un argument s'écrivent en mettant la fonction suivie de l'argument.
En OCaml comme en λ-calcul, les applications mènent à une évaluation qui consiste en remplaçant un paramètre par un argument.

OCaml	λ-calcul
`(fun x -> x + 10) 5` $\longrightarrow$ `5 + 10`	$(\lambda x.\, x + 10)\ 5$ $\longrightarrow 5 + 10$

`(fun f -> fun g -> fun x -> f (g x))`
`(fun x -> x < 10)`
`(fun x -> x + 10) 5`
$\longrightarrow$ `(fun x -> x < 10)`
`((fun x -> x + 10) 5)`
$\longrightarrow$ `(fun x -> x < 10) (5 + 10)`
	$(\lambda f.\lambda g.\lambda x.\, f(g\, x))$
	$(\lambda x.\, (x < 10))$
	$(\lambda x.\, x + 10)\ 5$
	$\longrightarrow (\lambda x.\, (x < 10))$
	$((\lambda x.\, x + 10)\ 5)$
	$\longrightarrow (\lambda x.\, (x < 10))\ (5 + 10)$

Applications

En OCaml comme en λ-calcul, les applications de fonction à un argument s'écrivent en mettant la fonction suivie de l'argument.
En OCaml comme en λ-calcul, les applications mènent à une évaluation qui consiste en remplaçant un paramètre par un argument.

OCaml	λ-calcul
`(fun x -> x + 10)  5` $\longrightarrow$ `5 + 10`	$(\lambda x.\, x + 10)\ 5$ $\longrightarrow 5 + 10$

`(fun f -> fun g -> fun x -> f (g x))`
`  (fun x -> x < 10)`
`  (fun x -> x + 10) 5`
$\longrightarrow$ `(fun x -> x < 10)`
`  ((fun x -> x + 10) 5)`
$\longrightarrow$ `(fun x -> x < 10) (5 + 10)`
$\longrightarrow$ `(5+10) < 10`
	$(\lambda f.\lambda g.\lambda x.\, f(g\, x))$
	$(\lambda x.\,(x < 10))$
	$(\lambda x.\, x + 10)\ 5$
	$\longrightarrow (\lambda x.\,(x < 10))$
	$((\lambda x.\, x + 10)\ 5)$
	$\longrightarrow (\lambda x.\,(x < 10))\ (5 + 10)$
	$\longrightarrow (5 + 10) < 10$

Les “built-ins” ?

À ce point, en regardant $(5 + 10) < 10$, vous vous dites sans doute que “$+$” et “$<$” sont des symboles “built-in” avec la sémantique habituelle, et que $(5 + 10) < 10$ sera évalué à $15 < 10$ et donc finalement à *false*.

Les “built-ins” ?

À ce point, en regardant $(5 + 10) < 10$, vous vous dites sans doute que “$+$” et “$<$” sont des symboles “built-in” avec la sémantique habituelle, et que $(5 + 10) < 10$ sera évalué à $15 < 10$ et donc finalement à *false*.

Ceci est effectivement le cas en OCaml. Mais pour le λ-calcul, après cette section introductoire, il faudra faire table rase avec des telles idées ! Il n’y aura pas de constantes “built-in” et pas d’évaluation “habituelle” !

Le `let`

Le `let`, très fréquent en OCaml, correspond à une combinaison d'abstraction (λ) et application :
`let y = 10 in y + y` correspond à $(\lambda y.y + y)\ 10$.

Le λ-calcul et la logique

Quittons maintenant la programmation et considérons la logique.
En logique des prédicats, on a les connecteurs $\forall$ et $\exists$. On se souvient que les quantificateurs lient les variables :
Voilà une formule

$$(Q(x) \vee \exists x.\, \forall y.\, P(f(x), z) \wedge Q(y)) \vee \forall x.\, R(x, z, g(x))$$

avec ses occurrences liées, libres, et liantes.

Le λ-calcul et la logique

Quittons maintenant la programmation et considérons la logique.
En logique des prédicats, on a les connecteurs $\forall$ et $\exists$. On se souvient que les quantificateurs lient les variables :
Voilà une formule

$$(Q(x) \vee \exists x.\, \forall y.\, P(f(x), z) \wedge Q(y)) \vee \forall x.\, R(x, z, g(x))$$

avec ses occurrences liées, libres, et liantes.
Dans le λ-calcul, le λ lie les variables.

- Le λ-calcul est dans un certain sens un formalisme bien plus abstrait que la logique des prédicats (par exemple).
- Le λ est un lieur générique.
- Le λ-calcul peut coder la logique des prédicats (par exemple).

Codage de la logique en λ-calcul

Rappel de la déduction naturelle et sa règle $(E\forall)$:

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[s/x]}\ (E\forall)$$

Le λ-calcul nous permet d'avoir un codage de la logique des prédicats qui marche comme ça : au lieu d'avoir comme lieurs des quantificateurs $\forall$, $\exists$, nous aurons le seul lieur λ et des constantes All, Exists.
Par exemple, $\forall x.\exists y.P(x,y)$ est codé comme
$\text{All}\ (\lambda x.\ \text{Exists}\ (\lambda y.\ (P(x,y))))$.
La règle $(E\forall)$ s'écrit dans ce codage comme

$$\frac{\Gamma \vdash \text{All}\ \Phi}{\Gamma \vdash \Phi\ s}\ (E\forall)$$

Exercice 6

Instanciez la règle pour $\Phi = \text{All}\ (\lambda x.\ \text{Exists}\ (\lambda y.\ (P(x,y))))$ *et* $s = 7$.

Les assistants de preuve

- Ce codage est un mécanisme tellement puissant qu'on l'utilise dans l'implantation des logiciels que l'on appelle des assistants de preuve.
- Il existe des logiques où on ne pense même pas du λ-calcul comme un "codage" d'une autre syntaxe mais comme la syntaxe propre de la logique : les logiques de l'ordre supérieur (HOL).

Plan

Types

- On n'a pas insisté sur ce point ci-dessus, mais Ocaml est un langage de programmation typé ; il existe aussi des langages de programmtion fonctionnels non-typés : Lisp.
- Le λ-calcul existe aussi en typé ou non-typé ; ici on regarde le λ-calcul non-typé.

Types

- On n'a pas insisté sur ce point ci-dessus, mais Ocaml est un langage de programmation typé ; il existe aussi des langages de programmtion fonctionnels non-typés : Lisp.
- Le λ-calcul existe aussi en typé ou non-typé ; ici on regarde le λ-calcul non-typé.
- Rappel : il faut faire table rase avec l'idée que nous nous basons sur des constantes arithmétiques qui auraient la sémantique habituelle. En fait, il n'y aura pas de constantes du tout !

Définition des λ-termes

Definition 7

On suppose un ensemble de variables $x, y \ldots$.
L'ensemble des λ-termes est définit par induction comme le plus pétit ensemble tel que :

- Chaque variable x est un λ-term.
- Si s et t sont des λ-termes, alors l'application $s\,t$ est un λ-term.
- Si s est un λ-terme, alors l'abstraction $\lambda x.\, s$ est un λ-terme.

Dans la suite, nous dirons simplement terme pour un λ-terme.

Example 8

$\lambda x.x$ et $\lambda x.(f\,x)\,y$ sont des termes.

Plus de terminologie

Definition 9

- Dans $\lambda x.s$, l'occurrence de x (en bleu) est liante, et
- toutes les occurrences de x dans s [a] sont des occurrences liées.
- Toute autre occurrence d'une variable est libre.
- Le λ est un (et le seul) lieur de variables.
- Un terme sans occurrence libre de variables est clos ou bien un combinateur.
- Un terme avec au moins une occurrence libre de variable est ouvert.

a. s est une méta-variable.

Example 10

$\lambda x.x$ est un terme clos et $\lambda x.(f\ x)\ y$ est un terme ouvert.

Conventions d'écriture et abréviations de combinateurs

Definition 11

- $s\,t\,u$ correspond à $(s\,t)\,u$: l'application associe à gauche.
- $\lambda x\,y.\,s$ correspond à $\lambda x.\,\lambda y.\,s$.
- $\lambda x.\,s\,t$ correspond à $\lambda x.\,(s\,t)$.

Voilà une liste d'abréviations de combinateurs que nous utiliserons beaucoup :

Definition 12

$$
\begin{array}{rcllrcl}
I & := & \lambda x.\,x & & \omega & := & \lambda x.\,x\,x \\
K & := & \lambda x\,y.\,x & & \Omega & := & \omega\,\omega \\
S & := & \lambda f\,g\,x.\,f\,x\,(g\,x) & & B & := & \lambda f\,g\,x.\,f\,(g\,x) \\
C & := & \lambda f\,x\,y.\,f\,y\,x & & & &
\end{array}
$$

On écrit aussi $s \circ t$ pour $B\,s\,t$.

Nombre d'arguments

Techniquement, les fonction du λ-calcul ne prennent qu'un argument, mais quand une fonction prend un argument après l'autre, il est convenable de dire qu'elle prend plusiers argument. Par exemple, S peut être vue comme une fonction qui prend 3 arguments.

Substitution (rappel)

Pour rappel, voilà un extrait de la définition d'une substitution pour la logique des prédicats :

- **Variable :** $x[s/x] = s$ et $y[s/x] = y$ pour $y \neq x$
- **Conjonction :** $(A \wedge B)[s/x] = (A[s/x] \wedge B[s/x])$
- **Quantificateur universel :**
  1. $(\forall x.A)[s/x] = (\forall x.A)$
  2. $(\forall y.A)[s/x] = (\forall y.(A[s/x]))$ si $y \notin fv(s)$
  3. $(\forall y.A)[s/x] = (\forall y'.(A[y'/y][s/x]))$ si $y \in fv(s)$, où y' est une variable "fraîche" (c.-à-d. $y' \notin fv(s), y' \notin fv(A)$)

La définition pour le λ-calcul est en analogie ...

Substitution pour le λ-calcul

Definition 13

- **Variable :** $x[u/x] = u$ et $y[u/x] = y$ pour $y \neq x$
- **Application :** $(s\,t)[u/x] = (s[u/x])(t[u/x])$
- **Abstraction :**
  1. $(\lambda x.\, s)[u/x] = (\lambda x.\, s)$;
  2. $(\lambda y.\, s)[u/x] = \lambda y.\, (s[u/x])$ si y n'est pas libre dans u ;
  3. $(\lambda y.\, s)[u/x] = \lambda y'.\, (s[y'/y][u/x])$ si y est libre dans u, où y' est une variable "fraîche" (c.-à-d. y' n'est libre ni en s ni en u).

Exercice 7

Calculez les applications de substitution suivantes : $(f\,x\,y)[y/x]$, $(f\,x\,y\,x)[z/x]$, $((\lambda x.\, x)\,x)[z/x]$, $(\lambda x\,y.\, f\,x\,y)[g/f]$, $(\lambda x\,y.\, f\,x\,y)[g\,z/f]$, $(\lambda x\,y.\, f\,x\,y)[g\,x/f]$.

“Fonctions” et “applications”

La substitution est une “fonction” qu’on peut “appliquer” à un terme, mais cela se joue au niveau méta. Ce ne sont pas des fonctions ou applications dans le sens syntaxique du λ-calcul.

Plan

α-conversion

Sémantiquement, les noms des variables liées ne font aucune différence. On appelle α-conversion le renommage d'une variable liée dans un terme, et on dit que deux termes sont α-équivalents ($\equiv_\alpha$) s'ils sont identiques à renommage des variables liées près.

Example 14

$\lambda x.\, x \equiv_\alpha \lambda y.\, y$. Attention : $\lambda x.\lambda x.\, x \equiv_\alpha \lambda y.\lambda x.\, x$, mais $\lambda x.\lambda x.\, x \not\equiv_\alpha \lambda y.\lambda x.\, y$

β-réduction

Definition 15

Un β-rédex est un terme de la forme $(\lambda x.\, s)\, t$.
La β-réduction est définie par le système de règles suivant :

$$\frac{}{(\lambda x.\, s)\, t \longrightarrow_\beta s[t/x]} \qquad \frac{s \longrightarrow_\beta s'}{s\, t \longrightarrow_\beta s'\, t} \qquad \frac{t \longrightarrow_\beta t'}{s\, t \longrightarrow_\beta s\, t'} \qquad \frac{s \longrightarrow_\beta s'}{\lambda x.\, s \longrightarrow_\beta \lambda x.\, s'}$$

La β-réduction est la notion de computation du λ-calcul.

Terminaison de la β-réduction

Il y a des termes pour lesquels la β-réduction termine toujours :

$$S\,K\,K = \underline{(\lambda f\,g\,x.\,f\,x\,(g\,x))\,K}\,K \longrightarrow_\beta (\lambda g\,x.\,K\,x\,(g\,x))\,K = (\lambda g\,x.\,\underline{(\lambda x\,y.\,x)\,x}\,(g\,x))\,K \longrightarrow_\beta (\lambda g\,x.\,\underline{(\lambda y.\,x)\,(g\,x)})\,K \longrightarrow_\beta \underline{(\lambda g\,x.\,x)\,K} \longrightarrow_\beta \lambda x.\,x = I$$

Il y a des termes pour lesquels la β-réduction ne termine jamais :

$$\omega\,\omega = \underline{(\lambda x.\,x\,x)\,(\lambda x.\,x\,x)} \longrightarrow_\beta (\lambda x.\,x\,x)\,(\lambda x.\,x\,x)\ldots$$

Il y a des termes pour lesquels la β-réduction peut ou peut ne pas terminer :

$$K\,I\,(\omega\,\omega) = \underline{(\lambda x\,y.\,x)\,I}\,(\omega\,\omega) \longrightarrow_\beta \underline{(\lambda y.\,I)\,(\omega\,\omega)} \longrightarrow_\beta I$$
$$K\,I\,(\underline{\omega\,\omega}) \longrightarrow_\beta K\,I\,(\underline{\omega\,\omega})\ldots$$

À chaque fois on souligne le β-rédex.

Formes normales

Definition 16

Un terme est normal s'il ne contient aucun β-rédex.
On dit que s évalue à t et on écrit $s \Downarrow t$ si $s \longrightarrow_\beta^* t$ et t est normal. On dit aussi que t est la forme normale de s.

Formes normales

Definition 16

Un terme est normal s'il ne contient aucun β-rédex.
On dit que s évalue à t et on écrit $s \Downarrow t$ si $s \longrightarrow_\beta{}^* t$ et t est normal. On dit aussi que t est la forme normale de s.

Example 17

Tous les combinateurs, I, K, S, ω, B et C sont normaux, mais Ω $(= \omega\omega)$ n'est pas normal.

Formes normales

Definition 16

Un terme est normal s'il ne contient aucun β-rédex.
On dit que s évalue à t et on écrit $s \Downarrow t$ si $s \longrightarrow_\beta{}^* t$ et t est normal. On dit aussi que t est la forme normale de s.

Example 17

Tous les combinateurs, I, K, S, ω, B et C sont normaux, mais Ω $(= \omega\omega)$ n'est pas normal.

Definition 18

Un terme est (faiblement) normalisant s'il a une forme normale.
Un terme est fortement normalisant si aucune chaîne infinie de β-réductions n'existe pour ce terme.

Example 19

$K\, I\, (\omega\, \omega)$ est normalisant et $S\, K\, K$ est fortement normalisant.

Stratégies d' évaluation

Une stratégie d'évaluation (en anglais : leftmost-outermost) consiste en :

- plus-à-gauche : dans une application $s\,t$, si on peut réduire s et t indépendemment, commencer systématiquement par s ;
- plus-à-l'extérieur : dans une application $(\lambda x.\, s)\,t$, commencer par $(\lambda x.\, s)\,t \longrightarrow_\beta s[t/x]$ (au lieu de chercher des rédexes dans s ou t).

On dénote par $\Rightarrow_\beta$ des étapes de $\longrightarrow_\beta$ qui respectent cette stratégie.

Stratégies d' évaluation

Une stratégie d'évaluation (en anglais : leftmost-outermost) consiste en :

- plus-à-gauche : dans une application $s\,t$, si on peut réduire s et t indépendemment, commencer systématiquement par s ;
- plus-à-l'extérieur : dans une application $(\lambda x.\, s)\,t$, commencer par $(\lambda x.\, s)\,t \longrightarrow_\beta s[t/x]$ (au lieu de chercher des rédexes dans s ou t).

On dénote par $\Rightarrow_\beta$ des étapes de $\longrightarrow_\beta$ qui respectent cette stratégie.

Theorem 20 (Curry 1958)

$s \Downarrow t$ si et seulement si $s \Rightarrow_\beta t$.

Explication intuitive :

- Pour $s\,t$, peut-être t est un argument qui finira par être "jeté". Donc mieux vaut ne pas risquer la non-terminaison sur t.
- Pour avoir une forme normale du rédexe $(\lambda x.\, s)\,t$, il faudra de toute façon faire une β-réduction (aucune réduction interne à s ou t fera disparaître ce rédexe). Donc on fait cette réduction de préférence.

Équivalence

Definition 21

On définit par $(\alpha\beta$-)équivalence $(\equiv)$ entre termes la relation d'équivalence minimale comprenant $\equiv_\alpha$ et $\longrightarrow_\beta$.

Theorem 22 (Church-Rosser 1936)

Si $s \equiv t$ alors il existe un terme u tel que $s \longrightarrow_\beta{}^* u$ et $t \longrightarrow_\beta{}^* u$.

Corollary 23

1. *Un terme a au plus une forme normale.*
2. *Si $s \equiv t$ et t est normal, alors $s \longrightarrow_\beta{}^* t$ et $s \Downarrow t$.*
3. *Si s et t sont normaux, alors $s \equiv t$ ssi $s \equiv_\alpha t$.*

Plan

Point fixe

Definition 24

Un point fixe d'une fonction f est un argument t tel que $f\,t \equiv t$.
Un combinateur de point fixe est un combinateur R tel que

$$R\,s \equiv s\,(R\,s)$$

pour tout terme s.

Dans le λ-calcul, toute fonction a un point fixe. Ce fait est important et surprenant.
On ne sait pas encore si un combinateur de point fixe R existe, mais si c'était le cas, par définition $R\,f$ serait un point fixe de f.

Les combinateurs T et Y

Theorem 25

Soient

$$\begin{aligned} T &:= (\lambda x\, f.\, f\,(x\, x\, f))\,(\lambda x\, f.\, f\,(x\, x\, f)) \\ Y &:= \lambda f.\,(\lambda x\, f.\, f\,(x\, x))\,(\lambda x\, f.\, f\,(x\, x)) \end{aligned}$$

T et Y sont des combinateurs de point fixe.

Preuve.

Soit s un terme quelconque.

$$\begin{aligned} &T\, s = (\lambda x\, f.\, f\,(x\, x\, f))\,(\lambda x\, f.\, f\,(x\, x\, f))\, s \longrightarrow_\beta \\ &(\lambda f.\, f\,((\lambda x\, f.\, f\,(x\, x\, f))\,(\lambda x\, f.\, f\,(x\, x\, f))\, f))\, s \longrightarrow_\beta \\ &s\,((\lambda x\, f.\, f\,(x\, x\, f))\,(\lambda x\, f.\, f\,(x\, x\, f))\, s) = s\,(T\, s) \end{aligned}$$

Exercice 8

Montrez que Y est un combinateur de point fixe.

Combinateurs de point fixe et formes normales

Exercice 9

- *Soit R un combinateur de point fixe. Montrez que pour tout terme t, le point fixe $R\,t$ n'est pas normal.*
- *Donnez une explication informelle du fait suivant : si $\lambda x.\,s$ est normal, alors pour toute variable y, le terme $s[y/x]$ (β-réduction de $(\lambda x.\,s)\,y$) est normal.*
- *Concluez qu'un combinateur de point fixe ne peut être normal.*

Prenons du récul ...

- Dans les maths habituelles, est-ce que la fonction $f(x) = \frac{x^2 - \sin x}{\log x \cdot \cos x}$ a un point fixe ? Pas évident !
- Comment alors dans le λ-calcul, toute fonction aurait un point fixe qui peut être donné par un combinateur (relativement) simple ?

Prenons du récul ...

- Dans les maths habituelles, est-ce que la fonction $f(x) = \frac{x^2 - \sin x}{\log x \cdot \cos x}$ a un point fixe ? Pas évident !
- Comment alors dans le λ-calcul, toute fonction aurait un point fixe qui peut être donné par un combinateur (relativement) simple ?
- Nous sommes pour l'instant sur un niveau d'abstraction élevé ! Si on considère que les nombres (entiers naturels) seraient quelque chose de plus concret et utile, nous ne savons rien d'utile pour l'instant.
- Les points fixes ne sont pas normaux. On pourrait espérer que leur application mène à des formes normaux dans certains cas utiles.
- Quand on applique les combinateurs de point fixe à des fonctions qui "n'ont pas vraiment de point fixe", cela devrait se manifester par le fait qu'aucune forme normale ne sera jamais atteignable.

Prenons du récul ...

- Dans les maths habituelles, est-ce que la fonction $f(x) = \frac{x^2 - \sin x}{\log x \cdot \cos x}$ a un point fixe ? Pas évident !
- Comment alors dans le λ-calcul, toute fonction aurait un point fixe qui peut être donné par un combinateur (relativement) simple ?
- Nous sommes pour l'instant sur un niveau d'abstraction élevé ! Si on considère que les nombres (entiers naturels) seraient quelque chose de plus concret et utile, nous ne savons rien d'utile pour l'instant.
- Les points fixes ne sont pas normaux. On pourrait espérer que leur application mène à des formes normaux dans certains cas utiles.
- Quand on applique les combinateurs de point fixe à des fonctions qui "n'ont pas vraiment de point fixe", cela devrait se manifester par le fait qu'aucune forme normale ne sera jamais atteignable.
- Un peu plus concrètement, les combinateurs de point fixe servent à définir des fonctions récursives, que nous connaissons des langages de programmation (fonctionnels).

Définition récursive

Une définition récursive pourrait ressembler à cela :

$$f := s \tag{1}$$

où les arguments de f sont exprimés dans la forme de s ; par exemple, si $s = \lambda x\, y.\, s'$, alors f serait une fonction à deux arguments. En plus, s contiendrait des occurrences de f.

Définition récursive

Une définition récursive pourrait ressembler à cela :

$$f := s \tag{1}$$

où les arguments de f sont exprimés dans la forme de s ; par exemple, si $s = \lambda x\, y.\, s'$, alors f serait une fonction à deux arguments. En plus, s contiendrait des occurrences de f.
Par contre, (1) n'a aucun statut dans le formalisme du λ-calcul, il ne définit rien, il dit juste qu'on désirerait un terme t (comme définition de f) tel que

$$t \equiv s[t/f]$$

ou de manière équivalente

$$t \equiv (\lambda f.\, s)\, t$$

Donc on cherche un point fixe de la fonction $\lambda f.\, s$.
N'importe quel combinateur de point fixe peut le fournir. Si R est un combinateur de point fixe, alors $R\,(\lambda f.\, s)$ donne la définition de f que l'on cherche selon (1).

Plan

Scott or Church, which is better for proving that the λ-calculus can compute μ-recursive functions ?
Or possibly Barendregt as he is the first one I found who really speaks about μ-recursion, which is what I need.

Plan

Le λ-calcul peut calculer toutes les fonctions μ-récursives.

η

H. R. Lewis et C. H. Papadimitriou :
Elements of the Theory of Computation.
Prentice-Hall, 2nd edition édn, 1998.