TAS
Cours 01 - Lambda-Calcul Pur

Romain Demangeon

TAS - M2 STL

19/09/2024

SORBONNE UNIVERSITÉ

Plan du Cours 01

1. Généralités sur le typage.
2. Le λ-calcul pur.

Vocabulaire : Type

Type de Données

Un type de données (ou type) est un ensemble de données partageant des propriétés et des interactions.

- Relation d'appartenance : "x de type T" $\Leftrightarrow$ "$x \in T$"
- exemple : `int` en `C` contient les nombres positifs ou négatifs représentables sur un nombre fixe de bits (dépendant de l'architecture).
  - opérations arithmétiques.
- exemple : `bool` en `OCaml` contient deux valeurs de vérité `true` et `false`.
  - test (`if`), fonctions booléennes.

Vocabulaire : Primitifs, Composés, Sous-typage

- types primitifs : données de base directement fournies par le langage (entiers, booléens)
- types composés : construits à partir d'un ou plusieurs types.
  - conteneurs `C[T]`: tableaux ou listes,
  - produits $\mathtt{T}\times\mathtt{U}$: juxtaposition de données (`struct` en `C`)
  - unions $\mathtt{T}+\mathtt{U}$: union de deux types (types algébriques en `OCaml`).
  - objets : regroupement de données exposées aux mêmes méthodes.
- une même donnée peut appartenir à plusieurs types.
- si tous les `S` sont des `T` alors `S` est un sous-type de `T`
  - c'est l'inclusion $\mathtt{S}\subseteq\mathtt{T}$,
  - pour les types objets, un `S` dispose (au moins) de toutes les méthodes de `T`.

Vocabulaire : Abstraction

- un type qui ne révèle pas la structure de ses éléments est dit abstrait.
- un type abstrait peut contenir des données dont bleu la représentation doit rester privée.
  - exemple : ensembles.
  - exemple : types de fonctions.
- des langages autorisent la création de types de données abstrait.

SORBONNE UNIVERSITÉ

Typage

- les opérations (primitives, fonctions, méthodes) s'appliquent à un type de données particulier T.
- le programme peut fonctionner ou échouer lorsqu'une fonction destinée à opérer sur T est appelée avec un S.
  - si $S \leq T$, c'est bon.
  - sinon ???
  - exemple : utiliser un entier comme une fonction.
- le contrôle de la légitimité de l'utilisation d'une opération s'appelle la vérification de typage (*typechecking*, ou "typage").
  - notion dépendante du langage.
  - bas-niveau (assembleur) : peu de vérifications
  - haut-niveau : typage plus ou moins strict
- typage dynamique : typage à l'exécution
- typage statique : typage à la compilation / interprétation.
- le typage accroit la sûreté d'exécution des programmes.

Types de bases

- les types de base (primitifs) sont directement manipulés par les programmes d'un langage.
  - entiers, flottans, booléens, chaînes de caractères, ...
- ils permettent :
  - une manipulation correcte de la mémoire.
  - l'adéquation de l'utilisation des primitives.
- la force du typage des primitives dépend du langage
  - en OCaml :

    ```
    print_endline("hello" + 1)
        }
    ```

    Erreur de typage
  - en Javascript:

    ```
    alert("hello" + 1)
    ```

    Ok."hello1"

Types de bases : Enumérations

- Enumérations : types de base particuliers donnés par une grammaire
- construction d'un type union
- naturel dans les langages avec reconnaissance de motifs
  - utilisation en décomposition suivant la grammaire
  - certains types primitifs peuvent être considérés commme énumérations (booléens)

```
type carte = Roi | Dame | Valet | Petite of int

let points c =
  match c with
    Roi -> 4
  | Dame -> 3
  | Valet -> 2
  | Petite 1 -> 11
  | Petite 10 -> 10
```

Statique vs. Dynamique

- vérification du typage de l'utilisation d'une primitive

```
sqrt "hello" (* en OCaml)
Math.sqrt("hello") // en Javascript
```

- statique au moment de l'analyse du code source du programme.
  - ici, le compilateur OCaml considère que la primitive attend un flottant, et est appliquée à ce qu'elle reconnaît comme une chaîne de caractères.
  - arrêt de l'analyse, échec de compilation
- dynamique au moment de l'exécution du programme.
  - ici, Javascript, en calculant l'expression, reconnait que `"hello"` n'est pas un nombre et renvoie `NaN`, sans erreur d'exécution (JS est permissif).

Statique vs. Dynamique

- vérification du typage de l'utilisation d'une fonction

```
let x = "hello" in x 0 (* en OCaml)
x = 'hello'; x(0); // Javascript
```

- ici, le compilateur OCaml infère que x n'est pas une fonction, erreur de typage et arrêt de l'analyse.
- ici, Javascript, en calculant l'expression, reconnait que `"hello"` n'est pas une fonction et produit une erreur d'exécution.

Données structurées

- moyen donnés par le langage pour représenter des donnés complexes
- une organisation des données $\longrightarrow$ un type
  - tableau, enregistrements, listes, *n*-uplets, ...
- chaque type T vient avec des opérateurs
  - de construction : produit des éléments de T à partir de données (y compris de type T) (`cons`, `(.,.)`, ...)
  - de destruction (déconstruction) : décompose un élément de T pour récupérer une partie de son contenu (`.[.]`, `hd`, ...)

Typage des données structurées

```
let succ x = x + 1
let rec map f xs = match xs with
    [] -> []
  | hd :: tl -> ( f hd )::( map f tl )

map succ [1; 2; 3] (* reussit *)
map succ 1 (* echoue *)
map succ [| 1; 2; 3 |] (* echoue *)
```

- système de types efficace : empêche l'utilisation d'une primitive au mauvais type
- ici, l'inférence déduit que le second argument de `map` doit être une liste
  - en fait, elle calcule son type `('a -> 'b) -> 'a list -> 'b list`
  - elle le déduit de l'utilisation des primitives `hd` et `tl`
  - elle empêche l'application de `map` à un entier ou un vecteur.

Définition de types

- les langages permettent souvent la définition de nouveaux types.
- la définition produit directement des moyens de construction et de destruction
- l'inférence synthétise les types des fonctions qui les manipulent
  - ici, `val sum : btree -> int`

```
type btree =
    Leaf of int
  | Node of { key : int ; left : btree ; right : btree }

let mytree =
  Node { key = succ (2);
         left = Node { key = 0; left = Leaf (-1); right = Leaf 1 };
         right = Leaf 3}

let rec sum bt = match bt with
    Leaf n -> n
  | Node { key = k ; left = lt ; right = rt } ->
               k + sum lt + sum rt
```

Types fonctionnels

- un système de types est d'ordre supérieur quand il s'étend aux fonctions et/ou aux objets.
- les types fonctionnels (ou type flèches) `A -> B` vérifient l'usage des fonctions
  - appel `f(a, b)` : vérification de l'adéquation entre les paramètres et les arguments.
  - appel `f(a, b)` : calcul du type du résultat et de son utilisation dans le programme.
- exemples de vérification statique de l'adéquation :
  - ```
    let f = (fun x -> x + 1) in
    let g = (fun x -> (fun y -> x [y])) in
    g f 0
    ```
  - ```
    let f = (fun x -> x + 1) in
    let dv = [ "Inferno" ; "Purgatorio" ; "Paradiso" ]
    map f dv
    ```
- pour les langages objets, on doit vérifier lors d'un appel de méthodes `o.m()` que l'objet `o` dispose bien d'une méthode `m`.

Vocabulaire : Types Abstraits

- définition d'un type masquant les détails de mise en oeuvre aux "clients" les utilisant.
- le type est exposé par :
  - un nom (abstrait, donc) de type éventuellement paramétré par des types,
  - la liste des opérations qui le manipulent.
- exemple classique : une pile définie par
  - un nom de type `St[T]` paramétré par `T`
  - `create : Unit -> St[T]`
  - `push : T * St[T] -> St[T]`
  - `empty : St[T] -> Bool`
  - `top : St[T] -> T`
  - `pop : St[T] -> St[T]`
- la liste des fonctions, avec leur types, est appelée l'interface dy type abstrait.

Vocabulaire : Types Abstraits (II)

- Pour donner une sémantique à un type abstrait on le munit d'une spécification
  - un ensemble de propriétés satisfaites par les opérations.
- ici:
  - `empty(create)` est vrai.
  - `empty(push(e, p))` est faux.
  - `top(create())` produit une erreur.
  - `top(push(e,p))` produit `e`.
  - `pop(create())` produit une erreur.
  - `pop(push(e,p))` produit `p`.
- les détails d'implémentation sont masqués.
- chaque type abstrait est traité indépendamment par le langage (même si deux types abstraits ont la même implémentation)
- certains compilateurs (`Java`, `OCaml`) vérifient qu'une implémentation est correcte vis-à-vis de son interface

Polymorphisme

- le typage apporte de la flexibilité à la programmation.
  - il permet d'écrire un composant génrique de manière indépendante de la nature de ses arguments.
- le polymorphisme indique que le type des arguments et du résultat d'un composant peut varier, tout en restant contraint par le typage (et donc sûr).
- exemple : trouver un élément dans une liste.
- Classification du polymorphisme :
  - paramétrique : un unique code générique pouvant traiter des arguments de types différents (généricité en langage objet)
  - de surcharge : code d'un composant qui diffère en fonction du nombre et du type des arguments.
  - d'inclusion : code réutilisé en fonction des relations entre les types (généralement sous-typage), aussi appelé polymorphisme objet.

Généricité

- certains langages permettent l'introduction d'un paramètre de type dans le code

```
func echange <E>( a : inout E, b : inout E){
  let temp = a ;
  a = b ;
  b = temp
}
```

- le code générique n'examine pas la structure des entités (arguments) du type paramétré.
- il examine des relations de type entre les entités
  - ici, le fait que `a` et `b` ont même type.

Généricité (II)

- la généricité peut apparaître dans un type structuré.
  - c'est (quasiment) toujours le cas : les fonctions totalement générique sont limitées.
- par exemple, au sein d'un conteneur

```
enum arbre<E> {
case Empty
indirect case
  Node ( arbre<E>, E , arbre<E>)
}

func long <E>(a : arbre <E>) -> Int {
  switch(a) {
    case .Empty: return 0
    case .Node (let fg, _, let fd):
      return long (a:fg) + 1 + long(a:fd)
  }
}
```

- l'ordre supérieur permet d'écrire du code paramétré par des fonctions.

```
map : ('a -> 'b) -> 'a list -> 'b list
```

Surcharge

- écrire du code différent pour le même nom de fonction qui dépend du type et du nombre des paramètres

```
func mem (e : Int, bi : Int, bs : Int) -> Bool {
  return (bi <= e) && (e <= bs)
}

func mem (c : Character, s : String) -> Bool {
  return( s.characters.contains(c))
}

func mem <E : Equatable>(e : E, a : arbre <E>) -> Bool {
  switch(a) {
    case .Empty : return false
    case .Node (let fg , let etiq , let fd):
      if (e == etiq) {return true}
      else {return(mem(e : e, a : fg) || mem(e : e, a : fd))}
    }
}
```

- dans les langages objets typés statiquement un algorithme de résolution de surcharge calcule quelle méthode est appelée en utilisant une relation d'ordre sur l'ensemble des méthodes possibles.

Polymorphisme d'inclusion

- sous-typage sémantique(subsomption) : $S \leq T$ quand partout où l'on a besoin d'un T, on peut utiliser un S.
  - en objet : la méthode qu'on veut appeler (sur T) est bien présente (dans S)
- le sous-typage est produit automatiquement par un mécanisme de classes (héritage)
  - ou un mécanisme différent commes les prototypes de Javascript.

```
protocol Animal {
  func exprimer () -> String
}
class Chat : Animal {
  func exprimer () -> String {
    return "Miauler"
  }
}
class Chien : Animal {
  func exprimer () -> String {
    return "Aboyer"
  }
}
class Chihuahua : Chien {
  override func exprimer () -> String {
    return super.exprimer () + " fort"
  }
}
```

Modèles de Calcul

- Début du XXeme siècle : *qu'est ce que le calcul* ?
- Trois théories mathématiques différentes :
  - Machines de Turing (Turing 1936): automate + rubans + lecture/écriture,
  - Fonctions récursives (Kleene 1931): schéma de définition de fonctions,
  - λ-calcul (Church 193?): syntaxe épurée fonctionnelle.
- Thèse de Church

Forme physique (prouvée)

Le λ-calcul, les machines de Turing et les fonctions récursives ont la même expressivité.

Forme psychologique (conjecturée)

Le cerveau humain aussi.

le λ-calcul

- le λ-calcul est un modèle de calcul, Turing-complet.
- en λ-calcul, "tout est fonction".
- le λ-calcul est à l'origine de la programmation fonctionnelle.
- en tant que modèle calcul, il est utilisé pour réfléchir aux notions de programmes, de fonctions, de calculabilité, de complexité, ...
- sa syntaxe simple donne accès à :
  - des preuves mathématiques claires,
  - une implémentation facile.
- son côté syntaxique est un défaut :
  - gestion des variables,
  - α-conversion,
  - substitutions implicites.

Syntaxe

$$M, N ::= x \mid \lambda x.M \mid M\ N$$

- grammaire BNF qui détermine un ensemble (infini) de termes (λ-termes).
  - chaque terme peut être vu comme un programme.
- on dispose d'un ensemble infini de variables (des noms)
- un λ-terme est soit :
- une variable x,
- une abstraction $\lambda x.M$: une fonction anonyme $x \mapsto M$ avec un paramètre x et un corps M
- application $M\ N$: notée comme en OCaml, M terme "fonction" et N terme "argument".
- ainsi $\lambda x.(\lambda y.x)$ correspond à l'expression OCaml :

```
(fun x -> (fun y -> x))
```

α-conversion

- variable liée (par un λ): $\lambda x.((x\ z)\ \lambda y.(x\ y))$
- variable libre (non liée): $\lambda x.((x\ z)\ \lambda y.(x\ y))$
- α-conversion : renommage des variables liées $\lambda x.M = \lambda y.(M[y/x])$ si $y \notin M$.
- On s'autorise cette opération à tout moment.
  - c'est "le même" terme (à α-équivalence près)
  - par exemple $\lambda x.(\lambda y.(x\ z)) \equiv_\alpha \lambda y.(\lambda u.(y\ z))$
  - mais $\lambda x.(\lambda y.(x\ z)) \not\equiv_\alpha \lambda z.(\lambda y.(z\ x))$
- Convention de Barendregt: variable liées différentes deux à deux et différentes des variables libres.
- On s'efforce, dans la suite, à ce que les termes présentés respectent cette convention.

On évite, par exemple, $(\lambda x.x)\ (\lambda x.x\ x)$ ou $(\lambda x.x)\ x$

Sémantique

Substitution

- $M[N/x]$: terme M où l'on a remplacé chaque occurence de x par N
- $(M[N_1/x_1])[N_2/x_2] = (M[N_2/x_2])[(N_1[N_2/x_2])/x_1]$
- opération magique instantanée.
  - implémentation ? λ-calcul avec substitutions explicites.

$$(\beta)\frac{}{(\lambda x.M)\ N \longrightarrow M[N/x]} \qquad \frac{M \longrightarrow M'}{M\ N \longrightarrow M'\ N}$$

$$\frac{N \longrightarrow N'}{M\ N \longrightarrow M\ N'} \qquad \frac{M \longrightarrow M'}{\lambda x.M \longrightarrow \lambda x.M'}$$

Exemples

Premier Exemple

$$
\begin{array}{rl}
 & (\lambda x.x\ x)\ (\lambda y.y) \\
\longrightarrow & (x\ x)[(\lambda y.y)/x] \\
= & (\lambda y.y)\ (\lambda y.y) \\
=_\alpha & (\lambda x.x)\ (\lambda y.y) \\
\longrightarrow & (\lambda y.y)
\end{array}
$$

Non-Déterminisme

$$
\begin{array}{rl}
 & (\lambda x.x\ x)\ ((\lambda y.y)\ (\lambda z.z)) \\
\longrightarrow & ((\lambda y.y)\ (\lambda z.z))\ ((\lambda u.u)\ (\lambda v.v)) \\
\longrightarrow & \ldots
\end{array}
$$

ou

$$
\begin{array}{rl}
 & (\lambda x.x\ x)\ ((\lambda y.y)\ (\lambda z.z)) \\
\longrightarrow & (\lambda x.x\ x)\ (\lambda z.z) \\
\longrightarrow & \ldots
\end{array}
$$

Termes usuels

Notations

- $\lambda xy.M$ pour $\lambda x.(\lambda y.M)$ (regroupement des paramètres)
- $M_1\ M_2\ M_3$ pour $(M_1\ M_2)\ M_3$ (parenthèsage à gauche)
- $\lambda x.M\ N$ pour $\lambda x.(M\ N)$ (maximalité du λ)

- $I = \lambda x.x$ (identité)
- $K = \lambda xy.x$ (sélection gauche ou T)
- $F = \lambda xy.y$ (sélection droite ou $\perp$)
- $S = \lambda xyz.(x\ z)\ (y\ z)$ (application généralisée)
- $\delta = \lambda x.(x\ x)$ (auto-application)
- $\Omega = \delta\ \delta$ (divergence)
- $Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$ (combinateur paradoxal)

Graphes de Réduction

Définition

Un graphe de réduction est un multi-graphe dirigé connexe où les sommets sont des termes. Il y a une arête de M vers N pour chaque manière d'obtenir $M \longrightarrow N$.

$$
\begin{tikzcd}
& (I\ I)\ (I\ I) \arrow[dl] \arrow[dr] & \\
I\ (I\ I) \arrow[dr, shift left] \arrow[dr, shift right] & & (I\ I)\ I \arrow[dl] \\
& (I\ I) \arrow[d] & \\
& I &
\end{tikzcd}
$$

Graphes de Réduction (II)

$$\begin{tikzcd} K\ I\ \Omega \arrow[r] \arrow[loop above] & (\lambda y.I)\ \Omega \arrow[r] \arrow[loop above] & I \end{tikzcd}$$

avec $\color{blue}{Tr} = \lambda x.(x\ x\ x)$

$$Tr\ \ Tr \longrightarrow Tr\ \ Tr\ \ Tr \longrightarrow Tr\ \ Tr\ \ Tr\ \ Tr \longrightarrow \ldots$$

$$\begin{tikzcd} (\lambda x.I)\ (Tr\ \ Tr) \arrow[r] \arrow[d] & (\lambda x.I)\ (Tr\ \ Tr\ \ Tr) \arrow[r] \arrow[dl] & (\lambda x.I)\ (Tr\ \ Tr\ \ Tr\ \ Tr) \arrow[r] \arrow[dll] & \ldots \\ I & & & \end{tikzcd}$$

Couples

Encodage

$(x, y) \stackrel{def}{=} \lambda f.f\ x\ y$

- $(.,.) = \lambda abf.f\ a\ b$ (encouplage)
- $\Pi_2^1 = \lambda c.(c\ K)$ (projection gauche)
- $\Pi_2^2 = \lambda c.(c\ F)$ (projection droite)
- $\leftrightarrow = \lambda cf.f\ (\Pi_2^2\ c)\ (\Pi_2^1\ c)$ (échange)

Entiers de Church

Encodage

$n \stackrel{def}{=} \lambda fe.f\ (f\ \ \dots\ (f\ \ e)\dots)$ (n applications de f à e)

- $0 = F$ (zéro)
- $1 = \lambda fe.(f\ \ e)$ (unité)
- $add = \lambda nmfe.n\ f\ (m\ f\ \ e)$ (addition)
- $mult = \lambda nmfe.n\ (m\ \ f)\ e$ (multiplication)
- $power = \lambda nmfe.(m\ \ n)\ f\ \ e$ (puissance)
- pour *pred* il faut définir $\sigma : (x, y) \mapsto (x + 1, x)$ en utilisant les couples.

Listes

Encodage

$[e_1; e_2; \ldots; e_k] \stackrel{def}{=} \lambda cn.c\ e_1\ (c\ e_2\ (\ldots\ (c\ e_k\ n) \ldots))$

- $[] = 0$ (liste vide)
- $[.] = \lambda ecn.c\ e\ n$ (encapsulation)
- $cons = \lambda elcn.c\ e\ (l\ c\ n)$ (construction)
- $append = \lambda l_1 l_2 cn.(l_1\ c\ (l_2\ c\ n))$ (concaténation)
- $hd = \lambda L.L\ K\ 0$ (tête)
- pour tl (queue) il faut définir la fonction
  $\sigma : a\ ,\ (x, y) \mapsto ((cons\ a\ y), y)$
- $map = \lambda flcn.l\ (\lambda as.(c\ (f\ a)\ s))\ n$
- $filter = \lambda plcn.l\ (\lambda as.(p\ a)\ (c\ a\ s)\ s)\ n$
- $reduce = \lambda fal.l\ f\ a$

arbres

$(e, A_g, A_d) \stackrel{def}{=} \lambda cn.c\ e\ A_g\ A_d$

Limites du λ-calcul

Substitutions

- écrire $M[N/x]$ c'est triché.
- $\lambda_{\sigma\Uparrow}$: calcul implémentant les substitutions de manière explicite
  - on descend dans le terme avec l'argument et on remplace au bon endroit.

α-conversion

- le λ-calcul est trop syntaxique.
  - la notion de variable est décevante.
  - difficulté d'implémentation
- λ avec indices de De Bruijn: $M, N ::= n \mid \lambda.M \mid M\ N$
  - le nombre de λ à traverser pour trouver le λ liant.
  - $K = \lambda\lambda.1$ et $S = \lambda\lambda\lambda.(2\ 0)\ (1\ 0)$
- Réseaux d'interaction.

Non-Déterminisme

Un modèle de calcul est non-déterministe quand deux réductions aboutissant à deux termes différents sont possibles depuis un même terme.

- $\exists t, t_1, t_2.(t \longrightarrow t_1) \wedge (t \longrightarrow t_2) \wedge (t_1 \neq t_2)$

- langages usuels: déterministes,
- machines de Turing: non-déterministes (cf. P & NP),
- réseaux de Petri et CCS: non-déterministes.
- λ-calcul: non-déterministe
  - $(I\ I)\ (I\ I) \longrightarrow I\ (I\ I)$
  - $(I\ I)\ (I\ I) \longrightarrow (I\ I)\ I$
- un "endroit où l'on peut réduire" est appelé redex.
- on a parfois le choix entre plusieurs redex.

Confluence Locale

$\longrightarrow^*$: cloture réflexive transitive de $\longrightarrow$ (i.e. $\longrightarrow^n$ pour un certain $n \geq 0$)

Un modèle de calcul est localement confluent quand deux réductions aboutissant à deux termes différents sont joignables en 0 ou plus réductions.

- $\forall t, t_1, t_2.(t \longrightarrow t_1) \wedge (t \longrightarrow t_2) \Rightarrow \exists u.(t_1 \longrightarrow^* u) \wedge (t_2 \longrightarrow^* u)$

$$
\begin{tikzcd}
 & t \arrow[dl] \arrow[dr] & \\
t_1 \arrow[dr, dotted, "*"] & & t_2 \arrow[dl, dotted, "*"] \\
 & u &
\end{tikzcd}
$$

Confluence

Un modèle de calcul est confluent quand deux séries de réductions aboutissant à deux termes différents sont joignables en 0 ou plus réductions.

- $\forall t, t_1, t_2 . (t \longrightarrow^* t_1) \wedge (t \longrightarrow^* t_2) \Rightarrow \exists u . (t_1 \longrightarrow^* u) \wedge (t_2 \longrightarrow^* u)$

$$\begin{tikzcd}
 & t \arrow[dl, "*"] \arrow[dr, "*"] & \\
t_1 \arrow[dr, dotted, "*"] & & t_2 \arrow[dl, dotted, "*"] \\
 & u &
\end{tikzcd}$$

Différence entre les deux notions

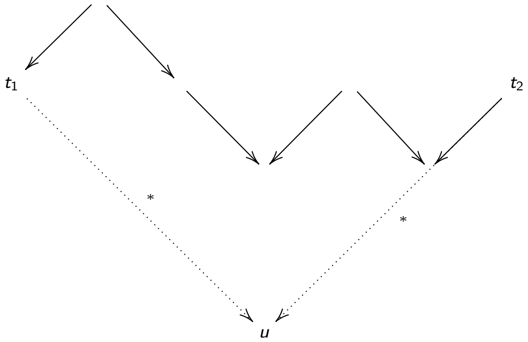
$$\longleftarrow \quad \rightleftarrows \quad \longrightarrow$$

Church-Rosser

- $u \equiv t$ (équivalence) quand il existe une suite $(u_i)_{0 \leq i \leq k}$ avec $u_0 = u$ et $u_k = t$ telle que, $\forall i < k, (u_i \longrightarrow u_{i+1}) \vee (u_{i+1} \longrightarrow u_i)$.
- $u \equiv t$ quand on peut utiliser $\longrightarrow$ dans les deux sens pour les relier.

Un modèle de calcul est Church-Rosser quand deux termes équivalents sont joignables en 0 ou plus réductions.

- $\forall t, t_1, t_2.(t_1 \equiv t_2) \Rightarrow \exists u.(t_1 \longrightarrow^* u) \wedge (t_2 \longrightarrow^* u)$



Cas du λ-calcul

- le λ-calcul est Church-Rosser.
- implication pratique: pas de concurrence.
  - "aucun choix n'est définitif"
- nombre de réductions:
  - $K\ I\ (power\ 10\ 2\ I\ I) \longrightarrow\ \longrightarrow I$,
  - $K\ I\ (power\ 10\ 2\ I\ I) \longrightarrow^{(N)} I$ (avec N tres grand).
- "vu qu'on est confluent, autant réduire au plus rapide".

Normalisation Faible

Une forme normale n d'un terme t est un réduit de t qui ne se réduit pas

- $(t \longrightarrow^* n) \wedge \neg(\exists s, n \longrightarrow s)$

- I est une forme normale de $S\ K\ K$.

Un terme est faiblement normalisant s'il admet une forme normale.

- $S\ K\ K$ est faiblement normalisant.
- $K\ I\ \Omega$ est faiblement normalisant.
- Ω n'est pas faiblement normalisant.

Un modèle de calcul est faiblement normalisant si tous les termes sont faiblement normalisants.

- le λ-calcul n'est pas faiblement normalisant.

Unicité de la forme normale

- soit un terme t qui a deux formes normales n_1 et n_2,
- comme $t \longrightarrow^* n_1$ et $t \longrightarrow^* n_2$, par confluence, il existe n tel que $n_1 \longrightarrow^* n$ et $n_2 \longrightarrow^* n$,
- comme n_1 est une forme normale $n_1 = n$,
- comme n_2 est une forme normale $n_2 = n$,
- finalement $n_1 = n_2$.

Les termes du λ-calcul ont au plus une forme normale.

Normalisation Forte

Un terme est fortement normalisant (terminant) s'il n'existe pas de suite infinie de réductions depuis ce terme.

- $\nexists (t_n)_{n \in \mathbb{N}}.(t_0 = t) \wedge (\forall k \in \mathbb{N}, t_k \longrightarrow t_{k+1})$

- $S\ K\ K$ est fortement normalisant.
- Ω n'est pas fortement normalisant.
- $K\ I\ \Omega$ n'est pas fortement normalisant.

Un modèle de calcul est fortement normalisant (terminant) si tous les termes sont fortement normalisants.

- le λ-calcul n'est pas fortement normalisant (divergent).

Combinateur de point fixe

- λ-calcul divergent:
  - comment exprimer la récursion ?
  - nécessaire pour la Turing-completude.

Un combinateur de point fixe *fix* est une fonction(nelle), qui quand elle est appliquée à une fonction F, donne un point fixe de F.

- $F\ (\mathit{fix}\ F) \equiv (\mathit{fix}\ F)$

- plusieurs combinateurs de points fixe en λ-calcul.
- $Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$
  - vérifier que $(Y\ F) \equiv F\ (Y\ F)$.
- Y permet de programmer des fonctions récursives.

Factorielle

- $isZero = \lambda n.n\ (\lambda y.F)\ K$ (égalité avec 0)
- $F = \lambda fn.(isZero\ n)\ 1\ (mult\ n\ (f\ (pred\ n)))$ (fonctionnelle)
- $fact = Y\ F$ (factorielle, point fixe de la fonctionnelle)

$$
\begin{array}{ll}
 & fact\ 3 \\
= & (Y\ F)\ 3 \\
\equiv & F\ (Y\ F)\ 3 \\
\equiv & (isZero\ 3)\ 1\ (mult\ 3\ ((Y\ F)\ (pred\ 3))) \\
\equiv & (mult\ 3\ ((Y\ F)\ 2)) \\
\equiv & (mult\ 3\ ((isZero\ 2)\ 1\ (mult\ 2\ ((Y\ F)\ (pred\ 2))))) \\
\equiv & (mult\ 3\ (mult\ 2\ ((Y\ F)\ 1)))
\end{array}
$$

λ-calcul pur

$M, N ::= x \mid \lambda x.M \mid M\ N$

$$(\beta)\frac{}{(\lambda x.M)\ N \longrightarrow M[N/x]} \qquad \frac{M \longrightarrow M'}{M\ N \longrightarrow M'\ N}$$

$$\frac{N \longrightarrow N'}{M\ N \longrightarrow M\ N'} \qquad \frac{M \longrightarrow M'}{\lambda x.M \longrightarrow \lambda x.M'}$$

- Syntaxe et sémantique du λ-calcul pur.
- Modèle de programmation fonctionnelle réaliste ?
  - non-déterminisme.
  - réduction sous les λ.

Une stratégie est un ensemble de règle de réduction tel qu'un terme à au plus un réduit.

- une stratégie est une sous-relation déterministe de $\longrightarrow$.

Appel par Valeur de Gauche à Droite (LtR-CbV)

$$
\begin{array}{lcl}
M, N & ::= & x \mid \lambda x.M \mid M\ N \\
V & ::= & x \mid \lambda x.M \mid x\ V
\end{array}
$$

$$
(\beta)\frac{}{(\lambda x.M)\ V \longrightarrow M[V/x]} \qquad \frac{M \longrightarrow M'}{M\ N \longrightarrow M'\ N}
$$

$$
\frac{N \longrightarrow N'}{V\ N \longrightarrow V\ N'}
$$

- Syntaxe des valeurs: termes qu'on ne peut pas réduire.
  - variable, abstraction, ou application bloquée.
- β-réduction uniquement si l'argument est une valeur.
- On réduit d'abord la fonction et apres l'argument.
- On ne réduit pas sous le λ.
- C'est la stratégie de la plupart des langages de programmation.

Appel par Nom (CbN)

$$M, N \quad ::= \quad x \mid \lambda x.M \mid M\ N$$

$$(\beta)\,\frac{}{(\lambda x.M)\ N \longrightarrow M[N/x]} \qquad\qquad \frac{M \longrightarrow M'}{M\ N \longrightarrow M'\ N}$$

- Syntaxe classique.
- β-réduction classique.
- On ne peut pas réduire l'argument.
- On ne réduit pas sous le λ.
- C'est la stratégie de *Haskell* (stratégie paresseuse).

Continuations

- programmer par continuation, c'est manipuler explicitement le futur du résultat
  - il est passé en paramètre.

```
let add x y k = k (x + y)

let n = add 1 2 (fun x -> add x 3 (fun y -> y))
```

- Programmer par continuation permet de manipuler l'environnement.
  - dans certain langage: `call/cc`

Stratégies CPS

$$
\begin{array}{lcl}
[x] & = & x \\
[\lambda x.M] & = & \lambda x.[\![M]\!] \\
[\![V]\!] & = & \lambda k.k\ [V] \\
[\![M\ N]\!] & = & \lambda k.[\![M]\!]\ (\lambda a.[\![N]\!](\lambda b.a\ b\ k))
\end{array}
$$

- - $[\![\delta I]\!] I$
  - $= (\lambda k.[\![\delta]\!]\ (\lambda a.[\![I]\!]\ (\lambda b.a\ b\ k)))\ I$
  - $\longrightarrow [\![\delta]\!]\ (\lambda a.[\![I]\!]\ (\lambda b.a\ b\ I))$
  - $= (\lambda k.k[\delta])\ (\lambda a.[\![I]\!]\ (\lambda b.a\ b\ I))$
  - $\longrightarrow (\lambda a.[\![I]\!]\ (\lambda b.a\ b\ I))[\delta]$
  - $\longrightarrow ([\![I]\!]\ (\lambda b.[\delta]\ b\ I))$
  - $\longrightarrow (\lambda k.k[I])\ (\lambda b.[\delta]\ b\ I))$
  - $\longrightarrow (\lambda b.[\delta]\ b\ I))\ [I]$
  - $\longrightarrow ([\delta]\ [I]\ I))$
  - $\ldots$
- Encodage par continuation: implémente la stratégie CbV-LtR.

Réduction Standard

- tout terme M s'écrit

$$\lambda x_1 x_2 \ldots x_k . M_1 \ M_2 \ \ldots \ M_n$$

avec $k \geq 0$, $n > 0$ et M_1 une variable ou une abstraction
  - (et si M_1 est une abstraction $n > 1$).
- si M_1 est une variable on dit que M est en forme normale de tête.
- sinon $M_1 = \lambda x.N_1$ et $(\lambda x.N_1 \ M_2)$ est le redex de tête.
- la réduction standard est la stratégie qui
  1. réduit le redex de tête s'il existe.
  2. applique la réduction standard aux $M_2 \ldots M_n$ si M est en forme normale de tête.
- la réduction standard termine si M a un forme normale.