Langages : interprétation et compilation

Pablo Rauzy
pr@up8.edu
pablo.rauzy.name/teaching/liec

UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS
UFR MITSIC / L3 informatique

Séance 9
Compilation vers le λ-calcul

Compilation vers le λ-calcul

- On a déjà vu le λ-calcul en cours et en TP.
- Rappels :
  - le λ-calcul est un modèle de calculabilité,
  - Il comporte des variables, des fonctions anonymes (abstractions), et des applications.
  - Syntaxe : `Λ ::= x,y,… | λx.Λ | ΛΛ`.

- On a déjà vu le λ-calcul en cours et en TP.
- Rappels :
  - le λ-calcul est un modèle de calculabilité,
  - Il comporte des variables, des fonctions anonymes (abstractions), et des applications.
  - Syntaxe : `Λ ::= x,y,… | λx.Λ | ΛΛ`.
- Mais comment calcule-t-on effectivement avec le λ-calcul ?

Réductions

- Soit `T` un terme de la forme `(λx.E)A`.
- `T` est une application dont le terme gauche est une abstraction.
- C'est ce qu'on appelle un *β-redex*.
- On peut lui appliquer une *β contraction* :
  - `T = (λx.E)A` devient alors `T' = E[x/A]`.
  - On note $T \rightarrow_\beta T'$.
  - `E[x/A]` est ce qu'on appelle une *substitution* :
    C'est le terme `E` avec les occurrences de la variable `x` remplacées par le terme `A`.

Réductions

- Soit `T` un terme de la forme `(λx.E)A`.
- `T` est une application dont le terme gauche est une abstraction.
- C'est ce qu'on appelle un *β-redex*.
- On peut lui appliquer une *β contraction* :
  - `T = (λx.E)A` devient alors `T' = E[x/A]`.
  - On note $\texttt{T} \rightarrow_\beta \texttt{T'}$.
  - `E[x/A]` est ce qu'on appelle une *substitution* :
    C'est le terme `E` avec les occurrences de la variable `x` remplacées par le terme `A`.
- Si il existe une suite de termes $\texttt{T}_i$ tels que $\texttt{T}_0 \rightarrow_\beta \texttt{T}_1 \rightarrow_\beta \texttt{T}_2 \rightarrow_\beta \cdots$:
  - on dit que $\texttt{T}_i$ se *β réduit* à $\texttt{T}_j$ pour $i \leq j$
  - on le note $\texttt{T}_i \rightarrow_\beta^* \texttt{T}_j$.

Exemples

- Réduisons ensemble les termes suivants :
  1. `(λx.x)y`
  2. `(λx.xy)z`
  3. `(λx.xx)(λy.y)`
  4. `(λx.xx)(λx.xx)`
  5. `(λx.λy.x)(λx.x)((λx.xx)(λx.xx))`
  6. `(λf.λx.fx)x`

Exemples

- Réduisons ensemble les termes suivants :
  1. `(λx.x)y`
  2. `(λx.xy)z`
  3. `(λx.xx)(λy.y)`
  4. `(λx.xx)(λx.xx)`
  5. `(λx.λy.x)(λx.x)((λx.xx)(λx.xx))`
  6. `(λf.λx.fx)x`
- Voyons maintenant quelques notions qui vont nous aider à formaliser tous les phénomènes ce qu'on vient de rencontrer.

Variables libres et variables liées

- C'est la même chose en λ-calcul que ce qu'on a vu en programmation.
- On calcule l'ensemble des variables libres d'un terme récursivement sur sa structure :
  - $FV(\mathsf{x}) = \{\mathsf{x}\}$
  - $FV(\lambda \mathsf{x}.\mathsf{T}) = FV(\mathsf{T}) \backslash \{\mathsf{x}\}$
  - $FV(\mathsf{T}_1 \mathsf{T}_2) = FV(\mathsf{T}_1) \cup FV(\mathsf{T}_2)$

Termes clos et termes ouverts

- Un terme est dit *clos* si il ne contient aucune variable libre.
- Il est ouvert sinon.
- On appelle aussi un terme clos un *combinateur*.

α-équivalence

- Deux termes et sont dit *α équivalents* si on peut obtenir l'un à partir de l'autre en ne faisant que renommer de variables liées.
- Il faut toutefois faire attention à ne pas *capturer* de variables libres.
- Exemples :
  - $\texttt{λx.x} \leftrightarrow_\alpha \texttt{λy.y}$
  - $\texttt{λx.xy} \leftrightarrow_\alpha \texttt{λz.zy}$
  - $\texttt{λx.xy} \not\leftrightarrow_\alpha \texttt{λy.yy}$

β-équivalence

- On dit que U et V sont *β équivalents* si il existe T tels que $U \rightarrow_\beta^* T$ et $V \rightarrow_\beta^* T$.
- On le note $U =_\beta V$.
- Exemples :
  - $(\lambda x.x)y =_\beta y$ (β contraction de l'un à l'autre)
  - $(\lambda x.x)y =_\beta (\lambda z.z)y$ (par α équivalence entre les termes)
  - $(\lambda x.x)y =_\beta (\lambda x.y)z$ (par β contractions en parallèle)
  - $(\lambda f.\lambda x.fx)x =_\beta \lambda y.xy$ (attention à la capture de variables lors de la substitution !)

Forme normale

- Un terme est dit en *forme normale* si il ne contient plus de β-rédex.
- Cette forme normale n'existe pas toujours (cf. exemple 4).
- Quand elle existe, il n'y en a qu'une seule.

Stratégie de normalisation

- Une stratégie de normalisation correspond à une façon de choisir l'ordre dans lequel on réduit les β-rédex d'un terme.
- Cette stratégie a une influence sur le résultat du calcul (cf. exemple 5).
- Il existe trois stratégies principales :
  - *normal order*
  - *call-by-name*
  - *call-by-value*

Normal order

- On commence toujours par réduire le β-rédex le plus externe le plus à gauche.
- Cette stratégie va toujours converger vers la forme normale.

- Aucun (?) langage de programmation n'utilise cette stratégie.

Call-by-name

- On évalue jamais sous une abstraction.
- On évalue pas les arguments tant qu'on en a pas besoin.
- Cette stratégie ne trouve pas toujours la forme normale.

- Utilisée par quelques langages, comme TEX.

Call-by-value

- On évalue jamais sous une abstraction.
- On commence par évaluer les arguments.
- Cette stratégie trouve encore moins souvent la forme normale.

- Utilisée dans plein de langage de programmation.

- Sans prétendre avoir présenter le λ-calcul de manière complètement formelle et rigoureuse, on en sait maintenant assez pour jouer avec.

- Je vous ai dit plusieurs fois que c'était un langage de programmation complet.
- Prouvons-le !

- Alonzo Church est le chercheur qui a inventé le λ-calcul dans les années 1930.
- Il a aussi montré qu'on pouvait encoder dans ce langage toutes sortes de données et structure de données.

- Voyons quelques exemples.

- Que veut-on faire avec un booléen ?

- Que veut-on faire avec un booléen ?
  - L'utiliser pour le branchement conditionnel, c'est à dire exécuter une branche ou une autre.

- Que veut-on faire avec un booléen ?
  - L'utiliser pour le branchement conditionnel, c'est à dire exécuter une branche ou une autre.
- Du coup, on va encoder les booléens c'est des fonctions qui prennent deux arguments mais qui n'exécutent que le premier (pour `true`) ou le second (pour `false`) :
  - `true` est `λt.λf.(t _)`
  - `false` est `λt.λf.(f _)`

- Du coup, implémenter le `if` est trivial :

- Du coup, implémenter le `if` est trivial :
  - `if` est `λc.λt.λf.(c t f)`

- Du coup, implémenter le `if` est trivial :
  - `if` est `λc.λt.λf.(c t f)`
- On peut aussi avoir `and`, `or`, et `not` assez facilement :

- Du coup, implémenter le `if` est trivial :
  - `if` est `λc.λt.λf.(c t f)`
- On peut aussi avoir `and`, `or`, et `not` assez facilement :
  - `and` est `λa.λb.(if a b false)`
  - `or` est `λa.λb.(if a true b)`
  - `not` est `λa.(if a false true)`

- L'idée est de représenter chaque entier par une fonction qui prend deux arguments et applique le premier au second autant de fois que l'entier qu'on veut représenter.
  - 0 est donc `λf.λz.z`.
  - 1 est donc `λf.λz.(f z)`.
  - 2 est donc `λf.λz.(f (f z))`.
  - etc.
- Remarquez que si on passe en argument la fonction $+1$ et 0 (les "vraies" valeurs, pas celles encodées dans le λ-calcul), on obtient la ("vraie") valeur du nombre.

- On veut maintenant définir des opérations sur nos représentations des entiers :
  - `succ` est
  - `+` est
  - `*` est
  - `zero?` est
  - `pred` est
  - `-` est

Opérations sur les entiers

- On veut maintenant définir des opérations sur nos représentations des entiers :
  - `succ` est `λn.λf.λz.(f (n f z))`
  - `+` est
  - `*` est
  - `zero?` est
  - `pred` est
  - `-` est

- On veut maintenant définir des opérations sur nos représentations des entiers :
  - `succ` est `λn.λf.λz.(f (n f z))`
  - `+` est `λn.λm.λf.λz.((m f) (n f z))`
  - `*` est
  - `zero?` est
  - `pred` est
  - `-` est

- On veut maintenant définir des opérations sur nos représentations des entiers :
  - `succ` est `λn.λf.λz.(f (n f z))`
  - `+` est `λn.λm.λf.λz.((m f) (n f z))`
  - `*` est `λn.λm.λf.λz.((m (n f)) z)`
  - `zero?` est
  - `pred` est
  - `-` est

Opérations sur les entiers

- On veut maintenant définir des opérations sur nos représentations des entiers :
  - `succ` est `λn.λf.λz.(f (n f z))`
  - `+` est `λn.λm.λf.λz.((m f) (n f z))`
  - `*` est `λn.λm.λf.λz.((m (n f)) z)`
  - `zero?` est `λn.(n (λ_.false) true)`
  - `pred` est
  - `-` est

Opérations sur les entiers

- On veut maintenant définir des opérations sur nos représentations des entiers :
  - `succ` est `λn.λf.λz.(f (n f z))`
  - `+` est `λn.λm.λf.λz.((m f) (n f z))`
  - `*` est `λn.λm.λf.λz.((m (n f)) z)`
  - `zero?` est `λn.(n (λ_.false) true)`
  - `pred` est `λn.λf.λz.((n (λignore1.λg.(g (ignore1 f))) (λ_.z)) (λx.x))`
  - `-` est

Opérations sur les entiers

- On veut maintenant définir des opérations sur nos représentations des entiers :
  - `succ` est `λn.λf.λz.(f (n f z))`
  - `+` est `λn.λm.λf.λz.((m f) (n f z))`
  - `*` est `λn.λm.λf.λz.((m (n f)) z)`
  - `zero?` est `λn.(n (λ_.false) true)`
  - `pred` est `λn.λf.λz.((n (λignore1.λg.(g (ignore1 f))) (λ_.z)) (λx.x))`
  - `-` est `λn.λm.((m pred) n)`

- Pour les listes, on a besoin de :
  - `cons` est
  - `car` est
  - `cdr` est
  - `nil` est
  - `null?` est
  - `pair?` est

- Pour les listes, on a besoin de :
  - `cons` est `λa.λb.λc.λn.(c a b)`
  - `car` est
  - `cdr` est
  - `nil` est
  - `null?` est
  - `pair?` est

- Pour les listes, on a besoin de :
  - `cons` est `λa.λb.λc.λn.(c a b)`
  - `car` est `λp.(p (λa.λb.a) error)`
  - `cdr` est
  - `nil` est
  - `null?` est
  - `pair?` est

- Pour les listes, on a besoin de :
  - `cons` est `λa.λb.λc.λn.(c a b)`
  - `car` est `λp.(p (λa.λb.a) error)`
  - `cdr` est `λp.(p (λa.λb.b) error)`
  - `nil` est
  - `null?` est
  - `pair?` est

Paires et listes

- Pour les listes, on a besoin de :
  - `cons` est `λa.λb.λc.λn.(c a b)`
  - `car` est `λp.(p (λa.λb.a) error)`
  - `cdr` est `λp.(p (λa.λb.b) error)`
  - `nil` est `λc.λn.(n _)`
  - `null?` est
  - `pair?` est

- Pour les listes, on a besoin de :
  - `cons` est `λa.λb.λc.λn.(c a b)`
  - `car` est `λp.(p (λa.λb.a) error)`
  - `cdr` est `λp.(p (λa.λb.b) error)`
  - `nil` est `λc.λn.(n _)`
  - `null?` est `λp.(p (λa.λb.false) (λn.true))`
  - `pair?` est

- Pour les listes, on a besoin de :
  - `cons` est `λa.λb.λc.λn.(c a b)`
  - `car` est `λp.(p (λa.λb.a) error)`
  - `cdr` est `λp.(p (λa.λb.b) error)`
  - `nil` est `λc.λn.(n _)`
  - `null?` est `λp.(p (λa.λb.false) (λn.true))`
  - `pair?` est `λp.(p (λa.λb.true) (λn.false))`

- La seule chose qui nous reste à mettre en œuvre pour disposer d'un vrai langage de programmation est la récursion.
- Ce n'est pas évident, prenons un exemple.

- La seule chose qui nous reste à mettre en œuvre pour disposer d'un vrai langage de programmation est la récursion.
- Ce n'est pas évident, prenons un exemple.
  - `fact` est `λn.(if (zero? n) 1 (* n (fact (pred n))))`

- La seule chose qui nous reste à mettre en œuvre pour disposer d'un vrai langage de programmation est la récursion.
- Ce n'est pas évident, prenons un exemple.
  - `fact` est `λn.(if (zero? n) 1 (* n (fact (pred n))))`
  - Ça ne fonctionne pas, ici `fact` est une variable libre.

- La seule chose qui nous reste à mettre en œuvre pour disposer d'un vrai langage de programmation est la récursion.
- Ce n'est pas évident, prenons un exemple.
  - `fact` est `λn.(if (zero? n) 1 (* n (fact (pred n))))`
  - Ça ne fonctionne pas, ici `fact` est une variable libre.
  - On peut essayer de passer la fonction à elle même, `fact` serait donc :
    `((λf.λn.(if (zero? n) 1 (* n ((f f) (pred n)))))`
    `(λf.λn.(if (zero? n) 1 (* n ((f f) (pred n))))))`
  - Cette fois-ci ça marche !

- La seule chose qui nous reste à mettre en œuvre pour disposer d'un vrai langage de programmation est la récursion.
- Ce n'est pas évident, prenons un exemple.
  - `fact` est `λn.(if (zero? n) 1 (* n (fact (pred n))))`
  - Ça ne fonctionne pas, ici `fact` est une variable libre.
  - On peut essayer de passer la fonction à elle même, `fact` serait donc :
    `((λf.λn.(if (zero? n) 1 (* n ((f f) (pred n)))))`
    `(λf.λn.(if (zero? n) 1 (* n ((f f) (pred n))))))`
  - Cette fois-ci ça marche !
  - Le combinateur `U` prend une fonction est l'applique à elle même. On peut donc réécrire `fact` :
    `(U (λf.λn.(if (zero? n) 1 (* n ((U f) (pred n))))))`

Combinateur Y

- Le combinateur `U` montre que c'est possible d'utiliser la récursion, mais ce qui serait vraiment bien, ce serait de pouvoir écrire facilement n'importe quelle fonction récursive.
- C'est ce que permet de faire le combinateur `Y`, en trouvant le point fixe d'une fonction.
  - C'est à dire que `(Y F)` $=_\beta$ `f` et que `(F f)` $=_\beta$ `f`.
- Il nous faut donc deux choses :
  1. exprimer une fonction récursive comme le point fixe d'une fonction pas récursive,
  2. écrire le combinateur `Y`.

- Si on sait qu'on reçoit la bonne fonction récursive en argument, ça devient très simple.
- En reprenant notre exemple de la fonction factorielle :
  - `fact` est `λf.λn.(if (zero? n) 1 (* n (f (pred n))))`

- Dérivons le combinateur `Y` :
  - On veut que `(Y F)` $=_\beta$ `f` et que `(F f)` $=_\beta$ `f`.

- Dérivons le combinateur `Y` :
  - On veut que `(Y F)` $=_\beta$ `f` et que `(F f)` $=_\beta$ `f`.
  - Par réécriture on a `(Y F)` $=_\beta$ `(F (Y F))`.

- Dérivons le combinateur `Y` :
  - On veut que `(Y F)` $=_\beta$ `f` et que `(F f)` $=_\beta$ `f`.
  - Par réécriture on a `(Y F)` $=_\beta$ `(F (Y F))`.
  - On peut utiliser une abstraction : `Y` est `λF.(F (Y F))`.

- Dérivons le combinateur `Y` :
  - On veut que `(Y F)` $=_\beta$ `f` et que `(F f)` $=_\beta$ `f`.
  - Par réécriture on a `(Y F)` $=_\beta$ `(F (Y F))`.
  - On peut utiliser une abstraction : `Y` est `λF.(F (Y F))`.
  - Bien sûr on se retrouve avec le même problème (`Y` est libre !), mais aussi le fait que le calcul ne termine pas car `Y` se rappelle immédiatement.

- Dérivons le combinateur `Y` :
  - On veut que `(Y F)` $=_\beta$ `f` et que `(F f)` $=_\beta$ `f`.
  - Par réécriture on a `(Y F)` $=_\beta$ `(F (Y F))`.
  - On peut utiliser une abstraction : `Y` est `λF.(F (Y F))`.
  - Bien sûr on se retrouve avec le même problème (`Y` est libre !), mais aussi le fait que le calcul ne termine pas car `Y` se rappelle immédiatement.
  - On sait que pour tout terme `t`, `t` $=_\beta$ `λx.(t x)`.

- Dérivons le combinateur `Y` :
  - On veut que `(Y F)` $=_\beta$ `f` et que `(F f)` $=_\beta$ `f`.
  - Par réécriture on a `(Y F)` $=_\beta$ `(F (Y F))`.
  - On peut utiliser une abstraction : `Y` est `λF.(F (Y F))`.
  - Bien sûr on se retrouve avec le même problème (`Y` est libre !), mais aussi le fait que le calcul ne termine pas car `Y` se rappelle immédiatement.
  - On sait que pour tout terme `t`, `t` $=_\beta$ `λx.(t x)`.
  - Du coup on commence par régler le second problème :
    `Y` est `λF.(F (λx.((Y F) x)))`.

- Dérivons le combinateur `Y` :
  - On veut que `(Y F)` $=_\beta$ `f` et que `(F f)` $=_\beta$ `f`.
  - Par réécriture on a `(Y F)` $=_\beta$ `(F (Y F))`.
  - On peut utiliser une abstraction : `Y` est `λF.(F (Y F))`.
  - Bien sûr on se retrouve avec le même problème (`Y` est libre !), mais aussi le fait que le calcul ne termine pas car `Y` se rappelle immédiatement.
  - On sait que pour tout terme `t`, `t` $=_\beta$ `λx.(t x)`.
  - Du coup on commence par régler le second problème :
    `Y` est `λF.(F (λx.((Y F) x)))`.
  - On sait déjà régler le premier problème avec le combinateur `U` :
    `Y` est `(U (λy.λF.(F (λx.(((U y) F) x)))))`.

- Dérivons le combinateur `Y` :
  - On veut que `(Y F)` $=_\beta$ `f` et que `(F f)` $=_\beta$ `f`.
  - Par réécriture on a `(Y F)` $=_\beta$ `(F (Y F))`.
  - On peut utiliser une abstraction : `Y` est `λF.(F (Y F))`.
  - Bien sûr on se retrouve avec le même problème (`Y` est libre !), mais aussi le fait que le calcul ne termine pas car `Y` se rappelle immédiatement.
  - On sait que pour tout terme `t`, `t` $=_\beta$ `λx.(t x)`.
  - Du coup on commence par régler le second problème :
    `Y` est `λF.(F (λx.((Y F) x)))`.
  - On sait déjà régler le premier problème avec le combinateur `U` :
    `Y` est `(U (λy.λF.(F (λx.(((U y) F) x)))))`.
  - On peut inliner ça : `Y` est `((λy.λF.(F (λx.(y y F x)))) (λy.λF.(F (λx.(y y F x)))))`

- On choisit un sous ensemble assez complet de Scheme :
  - les entiers, les opérations de bases sur les entiers,
  - les booléens, les opérations de bases sur les booléens,
  - les listes, les opérations de bases sur les listes,
  - le constructeur de clôture `lambda`,
  - et les constructions `let` et `letrec`.
- En utilisant ce qu'on a vu jusque là, on va le compiler vers le λ-calcul.
- Il nous reste quelques détails à voir.

Définitions avec `let`

- `(let ((var val)) body)` est équivalent à `((lambda (var) body) val)`
- On peut facilement dérouler de la même façon les assignations multiples avec `let` (en fait on implémente plutôt `let*`).

Fonctions à plusieurs arguments

- Les fonctions de plusieurs arguments sont *currifiées*.
- C'est à dire qu'on transforme une fonction de n arguments en une fonction d'un seul argument qui renvoient une clôture à $n-1$ argument où le nom du premier argument est lié à sa valeur.
- Exemple (en appliquant que les transformations du `let` et du `lambda`) :
  - `(lambda (a b) (+ a b))`
    devient
    `(lambda (a) (lambda (b) ((+ a) b)))`
  - donc
    ```
    (let ((add (lambda (a b) (+ a b))))
       (add 1 2))
    ```
    devient
    `((lambda (add) ((add 1) 2))(lambda (a) (lambda (b) ((+ a) b))))`

Démo !

- Étudions et exécutons le code ensemble.