

Lambda Nantes

Compréhension, Interprétation et implémentation du λ -calcul

Workshop 1: 10 Octobre 2024 @ Palo IT Nantes

Arnaud Bailly (Cardano Foundation) et **Xavier Van de Woestyne** (Tarides)

Une nouvelle formule !

En complément des présentations (habituelles), on voudrait proposer un nouveau format, plus interactif, potentiellement plus accessible et probablement plus transportable en visio-conférence !

Une nouvelle formule !

En complément des présentations (habituelles), on voudrait proposer un nouveau format, plus interactif, potentiellement plus accessible et probablement plus transportable en visio-conférence !

- ▶ Des **workshops interactifs**
- ▶ **Indépendants** des langages et technologies (tout est le bienvenu)
- ▶ **Pérennes** dans le temps (en permettant notamment plusieurs sessions sur un même sujet)
- ▶ Construction du plateforme web pour le **suivi** des workshops (dans un futur *proche*)

Pleins d'idées de sujets

- ▶ **Compréhension, interprétation et implémentation du λ -calcul**
- ▶ Implémentation d'applications web, **dans un style fonctionnel** (pour remplacer, par exemple, les affreuses applications propriétaires que l'on utilise, comme Meetup, et **parce que c'est très rigolo de réinventer la roue !**)
- ▶ **Vos sujets** (en tant que suggestion, en tant qu'organisateur, ce *que vous voulez*), nous sommes très ouverts à la contribution et à l'apprentissage

Compréhension, Interprétation et implémentation du λ -calcul

On dit souvent qu'une compréhension fine du λ -calcul, la base de la programmation fonctionnelle est nécessaire pour la comprendre.

Compréhension, Interprétation et implémentation du λ -calcul

On dit souvent qu'une compréhension fine du λ -calcul, la base de la programmation fonctionnelle est nécessaire pour la comprendre.

Je prétend que c'est faux (de la même manière qu'une compréhension fine de la théorie des catégories) *n'est absolument un prérequis (ou une nécessité) pour faire du Haskell efficacement.*

Alors pourquoi ce premier Workshop ?

- ▶ **Ça permet de s'initier à la construction d'un langage**
- ▶ C'est tout de même intéressant
- ▶ Le λ -calcul permet de **comprendre** certains points théoriques liés à la compilation et l'interprétation de langages de programmation
- ▶ Permet de se familiariser avec certains encodages dans des langages plus riches
- ▶ Ça se découpe bien en plusieurs parties

C'est quoi le λ -calcul ?

- ▶ Un système formel inventé dans les années 30, par **Alonzo Church** qui décrit/fonde les concepts de **fonctions et d'applications de fonctions** (qui est, en fait, un langage de programmation théorique)
- ▶ Premier formalisme permettant de décrire et caractériser **les fonctions récurives**, permettant de décrire la notion de **calculabilité** (que l'on appellera, plus tard, la *Turing-completude*)

C'est quoi le λ -calcul ?

- ▶ Un système formel inventé dans les années 30, par **Alonzo Church** qui décrit/fonde les concepts de **fonctions et d'applications de fonctions** (qui est, en fait, un langage de programmation théorique)
- ▶ Premier formalisme permettant de décrire et caractériser **les fonctions récursives**, permettant de décrire la notion de **calculabilité** (que l'on appellera, plus tard, la *Turing-completeness*)

Comme pour le modèle de Herbrand-Gödel et les machines de turing, il est un des fondements de la calculabilité, donc de la programmation et offre le même niveau d'expressivité.

Aujourd'hui

- ▶ Base conceptuelle des langages de programmations fonctionnels modernes (laissant à la machine de Turing le rôle de base conceptuelle des langages impératifs)
- ▶ Permet de servir de base pour des modèles de programmations (certains motifs de conceptions par exemple)
- ▶ Sert le travail de l'ingénieur quotidiennement, notamment pour la construction d'outils (ie: *refactor Engine* qui est généralement une conséquence de la β -reduction ou encore l'indexation d'occurrences via des formes de normalisation)
- ▶ Permet d'implémenter des langages en étant étendu, de servir de cible de compilation et aussi d'outil de comparaison entre des langages (via des procédés d'analyse sémantique)

Aujourd'hui

- ▶ Base conceptuelle des langages de programmations fonctionnels modernes (laissant à la machine de Turing le rôle de base conceptuelle des langages impératifs)
- ▶ Permet de servir de base pour des modèles de programmations (certains motifs de conceptions par exemple)
- ▶ Sert le travail de l'ingénieur quotidiennement, notamment pour la construction d'outils (ie: *refactor Engine* qui est généralement une conséquence de la β -reduction ou encore l'indexation d'occurences via des formes de normalisation)
- ▶ Permet d'implémenter des langages en étant étendu, de servir de cible de compilation et aussi d'outil de comparaison entre des langages (via des procédés d'analyse sémantique)
- ▶ Permet d'organiser des Workshops (dans lesquels on pourra ajouter des fonctionnalités, un système de type etc) !

Une grammaire **incroyablement** simple !

Les constituants du λ -calcul sont appelés des λ -termes.

Variables : $\langle \text{nom} \rangle$

x est une variable de nom x .

Abstraction : $\lambda \langle \text{paramètres} \rangle . \langle \text{corps} \rangle$

$\lambda x. x$ est la fonction **identité**.

Application : $\langle \text{fonction} \rangle \langle \text{lambda-terme} \rangle$

$(\lambda x. x)a$, on applique l'identité à a .

En d'autres termes:

Term (M, N, 0, ...):

$M ::= x$ (variable)

$M ::= \lambda x.M$ (abstraction)

$M ::= M N$ (application)

*Comment résoudre des fonctions à plusieurs argument ? La **curryfication** !*

Avec quelques conventions de parenthésages

$$(M) \quad \equiv \quad M$$

$$\lambda x. (M \ N) \quad \equiv \quad \lambda x. M \ N$$

$$\lambda x. (\lambda y. M) \quad \equiv \quad \lambda x. \lambda y. M$$

$$(M \ N) \ 0 \quad \equiv \quad M \ N \ 0$$

Avec ces trois éléments, on peut déjà construire beaucoup de choses ! Le fait d'utiliser uniquement des fonctions et des applications pour encoder des choses s'appelle l'utilisation de **church-encodings**. (La base de l'OOP en vrai)

Des booléens

- ▶ `TRUE` := $\lambda x.\lambda y.x$
- ▶ `FALSE` := $\lambda x.\lambda y.y$

Avec ces trois éléments, on peut déjà construire beaucoup de choses ! Le fait d'utiliser uniquement des fonctions et des applications pour encoder des choses s'appelle l'utilisation de **church-encodings**. (La base de l'OOP en vrai)

Des booléens

- ▶ $\text{TRUE} := \lambda x. \lambda y. x$
- ▶ $\text{FALSE} := \lambda x. \lambda y. y$
- ▶ $\text{AND} := \lambda p. \lambda q. p \ q \ p$
- ▶ $\text{OR} := \lambda p. \lambda q. p \ p \ q$
- ▶ $\text{NOT} := \lambda p. p \ \text{FALSE} \ \text{TRUE}$

Avec ces trois éléments, on peut déjà construire beaucoup de choses ! Le fait d'utiliser uniquement des fonctions et des applications pour encoder des choses s'appelle l'utilisation de **church-encodings**. (La base de l'OOP en vrai)

Des booléens

- ▶ $\text{TRUE} := \lambda x. \lambda y. x$
- ▶ $\text{FALSE} := \lambda x. \lambda y. y$
- ▶ $\text{AND} := \lambda p. \lambda q. p \ q \ p$
- ▶ $\text{OR} := \lambda p. \lambda q. p \ p \ q$
- ▶ $\text{NOT} := \lambda p. p \ \text{FALSE} \ \text{TRUE}$
- ▶ $\text{IFTD} := \lambda p. \lambda a. \lambda b. p \ a \ b$

De l'arithmétique

- ▶ $0 \quad := \lambda f x. x$
- ▶ $1 \quad := \lambda f x. f \ x$
- ▶ $2 \quad := \lambda f x. f \ (f \ x)$
- ▶ $3 \quad := \lambda f x. f \ (f \ (f \ x))$
- ▶ etc.

De l'arithmétique

- ▶ $0 \quad := \lambda f x. x$
- ▶ $1 \quad := \lambda f x. f \ x$
- ▶ $2 \quad := \lambda f x. f \ (f \ x)$
- ▶ $3 \quad := \lambda f x. f \ (f \ (f \ x))$
- ▶ etc.
- ▶ $SUCC \ := \ \lambda n. \lambda f. \lambda x. f \ (n \ f \ x)$
- ▶ $PLUS \ := \ \lambda m. \lambda n. m \ SUCC \ n$
- ▶ etc.

Première étape: décrire un AST

Un **AST** (ou, Arbre de Syntaxe Abstrait) est une représentation manipulable de notre syntaxe.

Avec un AST, on peut, par exemple, considérer que les deux expressions sont équivalentes :

▶ `let f x = x + 1`

▶ `let f x = (+) x 1`

Il est usuel d'utiliser des **sommes** pour décrire les différentes branches (souvent récursives) de notre arbre !

Concepts de variables **libres**

Une peu naïvement : Ensembles des variables qui ne sont pas des arguments (dans un scope donné). Par exemple:

$\lambda x.x$ # Pas de variables libres

$\lambda x.x+y$ # y est libre

Quand une variable n'est pas libre, elle est **liée**

Réductions

Permettant de décrire des équivalence de Lambda Termes.

- ▶ α -conversion: la **résolution des noms** ($\lambda x.x = \lambda y.y$) (qui entraîne des **Substitutions**)
- ▶ β -conversion: l'**application de fonction** ($(\lambda x.x \times + 2) \ 7 = 7 + 2 = 9$)
- ▶ η -reduction: $(\lambda x.f \ x) = f$ si x n'est pas **libre**

La base de stratégie d'évaluation et d'optimisation ! (En effet, on réduira généralement durant l'évaluation d'un programme)

On a suffisamment de “bases” pour démarrer !