

## Disclaimer

Les deux parties de ce document sont indépendantes et peuvent être lues dans n'importe quel ordre. L'objectif initial de la partie 1 était de discuter qu'est-ce que c'est une théorie mathématique en utilisant l'arithmétique comme exemple. Cependant, cette tentative s'est transformée vite dans un essai sur l'histoire de la mathématique de la fin XIX<sup>ème</sup> siècle jusqu'au milieu du XX<sup>ème</sup>. On se concentre principalement sur les problèmes de consistance et notamment sur le deuxième problème de Hilbert. Il est évident que notre approche ne peut jamais être complète, donc on se restreint sur le contexte historique des avancées en formalisation d'arithmétique ainsi que la notion de calculabilité. Intentionnellement, on évite au maximum les définitions formelles et formules et invite le lecteur curieux de faire ses propres recherches en partant du Wikipédia (plutôt anglais) qui présente la plupart des définitions manquantes.

Au contraire, l'objectif de la partie 2 est de donner au lecteur le goût du  $\lambda$ -calcul : on présente une définition plutôt complète, puis on montre, comment construire un langage de programmation élémentaire mais aussi puissant que la machine de Turing. Ce langage est connu comme le paradigme fonctionnel.

Finalement, l'ordre de lecture conseillé est plutôt le suivant : un passage "diagonale" de la partie 2, puis lecture complète de la partie 1 et la relecture de la partie 1 pendant laquelle on devra voir le sens derrière la forêt de détails techniques nécessaires pour introduire un système formel tel que  $\lambda$ -calcul.

## Part 1. Histoire de $\lambda$ -calcul

Qu'est-ce que c'est un  $\lambda$ -calcul ? Les codeurs sont familiers avec une notion de  $\lambda$ -fonction - une fonction anonyme qui est utilisée dans les morceaux de code qui ne méritent pas d'avoir un nom : clé de tri, les petites transformations dans les requêtes à la SQL etc. Cependant, la notion de  $\lambda$ -fonction a été prise d'un système de calcul aussi puissant que la machine de Turing (est inventée dans les mêmes années 30s). Dans cet article on va discuter l'histoire de son invention pour mieux comprendre le concept.

### Plan

1. Crise des fondements
2. Axiomes de Peano

3. 1ere version de Lambda calculs
4. Theoreme de Goedel
5. Machine de Turing et calculabilité
6. Thèse de Church-Turing
7. Impact et applications

### Crise des fondements

Qui n'a jamais vu cette phrase : “*Cette phrase est fausse*” ? Probablement, personne. Tout le monde sait qu'elle n'est ni vrai ni fausse (effectivement, s'il on suppose qu'elle est vrai, alors elle doit être fausse et à l'invers). Conue depuis presque 3000 ans sous un nom d'un paradoxe du menteur, les mathématiciens ont eu une habitude de vivre avec jusqu'à la fin de XIXème siècle. Cépéndant, les “invités inattandus” ont arrivés de temps en temps :

- **Paradoxe du barbier.** Un barbier rase tous qui ne se rasent pas eux-mêmes et seulement ceux-ci; Qui rase ce barbier ?
- **Paradoxe sorite.** Un grain isolé ne constitue pas un tas. L'ajout d'un grain ne fait pas d'un non-tas, un tas. Donc on ne peut pas construire un tas par l'ajout des grains.
- **Paradoxe du crocodile.** Un crocodile méchant vous attrape et propose de deviner votre destin. Si votre devine est incorrecte, il vous mange. La réponse ? – “Tu vas me dévorer” !

Pour une liste exhaustive des paradoxes simples, on peut consulter un livre de Martin Gardner. “Aha! Gotcha. Paradoxes to puzzle and delight”. Maintenant, discutons un autre saboteur de logique : *Paradoxe de l'ensemble de Russel*. Disons qu'un ensemble est *simple* s'il n'appartient pas à lui-même. Par exemple, l'ensemble des tout les gens est simple, car cet ensemble n'est pas une personne. Ainsi, l'ensemble des tout les ensemble n'est pas simple par son définition. *L'ensemble de Russel* est un ensemble qui contient tout les ensembles simples et rien d'autre. Est-ce qu'un ensemble de Russel est simple ? Si c'est le cas, par construction il contient lui-même. Donc il n'est pas simple. Mais s'il n'est pas simple il doit contenir lui-même, ce que signifie qu'il est simple. *Contradiction*.

Les mathématiciens n'étude pas ni crocodiles, ni barbiers. Les questions des menteurs sont plutôt les compétences de code penal ou les philosophes.

Cependant dans une version de Russel ce paradoxe n'utilise que les constructions formelles de mathématique. Cela signifie que telles constructions sont contradictoires elles-mêmes : si nous avons prouvé qu'une formule propositionnelle est à la fois vrai et fausse, le même peut avoir lieu pour n'importe quel théorème. Si on ajoute qu'au début de XXème siècle paradoxe de Russel n'a pas été le seul paradoxe connu, on voit bien qu'est-ce que c'était le *crise de fondements* en mathématique.

Ce crise a été reflété sous le numéro 2 dans une liste des 23 fameux problèmes de Hilbert déterminants le développement du mathématique en XXème siècle.

**2ème problème de Hilbert** *Déterminer la consistance de l'arithmétique.*

Dans le sens plus formel, la consistance signifie le suivant.

1. Il y a des axiomes dont on peut déduire tout les théorèmes de l'arithmétique.
2. Aucun axiomes ne peut pas être déduit des autres.
3. Il n'existe pas d'une proposition X, tel que les axiome implique X ainsi que "non X".

L'étape 1 est plutôt constructif : en pratique, il est suffisant de produire les nombres (entiers, rationnels, réels) avec ses propriétés habituels. Dans l'étape 2 il faut prouver que les axiomes sont indépendants l'un des autres. Équivalentment, si on supprime n'importe quel axiome, l'étape 1 n'est plus vrai. L'étape 3 est le plus compliqué.

## Axiomes de Peano

L'arithmétique est un domaine de mathématique qui étudie les nombres et relations entre eux. Elle est appliquée partout de premières années de l'école jusqu'à les concepts modernes d'astrophysique. Cependant, pour construire les bases de l'arithmétique, il est presque suffisant de bien déterminer les nombres naturels ainsi que les action qu'on peut faire avec. (Les nombres entiers est une extension pour que opération  $x - y$  renvoie toujours un nombre valide, les nombres rationnels apparaissent si on étudie la division. Finalement, les nombres algébrique sont responsable pour résoudre les équations polynomiales et le reste - pour "fermer des trous"). Classiquement, les nombres naturels peuvent être définis de même façon qu'on fait quand les petits enfants apprennent à compter: ce résultat est connus depuis la fin de XIXème siècle comme les axiomes de Peano:

1. 1 est naturel;

2. le nombre suivant d'un nombre naturel est naturel;
3. rien ne suivi de 1;
4. si  $a$  suit  $b$  et  $a$  suit  $c$ , alors  $b = c$ ;
5. axiome de recurrence (i.e., si un prédicat  $A(x)$  est vrai pour  $x = 1$  ainsi que  $A(n)$  implique  $A(n + 1)$ , alors  $A(x)$  est vrai pour tout  $n$  naturel).

Montrons que avec ces axiomes sont suffisant pour construire tout les nombres.

- **Nombres naturels** sont déjà construit. Ok, pour être honête, nous avons construit les objets “bizzares” et les avons appelés les nombres naturels – prenez l’habitude que mathématique fondamental est une façon de parler des choses évidents depuis l’école avec des mots très sophistiqués. Mais la vrai avantage de cet approche est son **correctité** absolu.
- **Zero** est indispensable pour compter. Rien plus simple - ajoutons un objet nouveau spéciale, qui est (i) suivi de 1. Appelons lui “0”. Posons que pour tout  $n$  naturel (ii)  $n+0 = 0+n = n$  et aussi (iii)  $n \cdot 0 = 0 \cdot n = 0$  (servira pour le futur).
- On peut compter et même calculer la somme, mais les mathématiciens veulent plus de symmetrie, donc l’opération réciproque de “+” doit exister. D’accord: par définition, la différence  $a - b$  est un nombre  $c$  tel que  $a = b + c$ . Que vaut  $2 - 5$ ? Oups, il n’existe pas tel  $c$  que  $c + 5 = 2$  - n’oubliez pas qu’on veut la stricte absolue, donc on ne peut utiliser que des nombres déjà calculés. Nous n’avons pas de choix: disons que “ $2 - 5$ ” et un nouveau nombre. Ainsi que “ $1 - 2$ ”, “ $42 - 45$ ” et même “ $239 - 261$ ”. Cela semble beaucoup, mais remarquons que  $2 - 5$  est égal au  $42 - 45$  et aussi au  $0 - 3$ . Pour simplicité, omettons zero et écrivons juste -3. Félicitations ! Vous venez de construire les nombres **négatifs** et donc les nombres **entières** ! Cet opération s’appelle la **closure** et est très typique pour la génération des nouveaux objets.
- Les **nombres rationnels** arrivent par la même logique que : si nous pouvons calculer le produit, alors nous voulons diviser. Les résultats de tout les divisions possibles ( $1/2$ ,  $-2/3$ ,  $2/4$ ,  $37/17$ ,  $5/5$ , etc.) forment les nombres rationnels – vous devez le souvenir bien depuis l’école - d’habitude ils sont appris plus tard que les nombres négatifs.

- Imaginons l'axe de nombres. D'un point de vue, il est très dense – pour n'importe quel nombre rationnel, il existe un autre nombre rationnel qui est “autant proche de lui qu'on veut”. Il n'est pas suffisant ! Malheureusement,  $\sqrt{2}$  n'est pas rationnel (à propos : 7ème problème de Hilbert s'agit de prouver que  $\sqrt{2}^{\sqrt{2}}$  n'est pas rationnel). Donc les **nombres réels** sont définis par une autre closure. Informellement on remplit des “trous” sur l'axe des nombres. Pour les plus curieux qui souhaitent la construction formelle, la page de wikipedia est bien explicative. Personnellement, je préfère la construction par des coupes de Dedekind.

**todo: introduce the main manipulations for the lambda, combinators, SKI**

Heureusement, la vraie preuve d'une consistance des axiomes de Peano est un problème beaucoup plus sophistiqué que l'invention de ses axiomes, et l'histoire n'est donc que commencée.

## 1ère version de Lambda calcul

En 1932 Church a proposé une autre construction qui est connue comme  **$\lambda$ -calcul non-typé**. Malheureusement, son étudiant, Kleene a prouvé que cette construction n'a pas été consistante.

$\lambda$ -calcul a formalisé une application d'une fonction. L'écriture envisage la compréhension d'une fonction comme une “règle”. Et l'écriture classique  $f(x)$  pointe plutôt sur le résultat de ce règle.

Rappelons brièvement, qu'est ce que c'est. (Sinon, wiki et les autres articles ou “Eggs and crocodiles”) Le brique principal est une fonction. Au lieu de  $f(x)$  on écrit  $\lambda x.f$ . Si on parle de la valeur de  $f(x)$  quand  $x = a$ , on écrit  $\lambda x.f a$ . Naturellement, on peut définir une composition... Pour transformer des propositions on a une règle de  $\beta$ -reduction.

Malgré sa simplicité et abstraité, cette construction permet néanmoins rédefinir tout les opérations arithmétiques, la logique Booléen etc. Est-ce que  $\lambda$ -calcul non-typé est un bon candidat pour le rôle de fondement de mathématique ? La réponse est **non**: à cause de Paradoxe de Kleene-Rosser proposé en 1935 par J. B. Rosser et Stephen Kleene qui a été un étudiant de Church.

Ironiquement, ce paradoxe, beaucoup plus sophistiqué dans sa version initiale, n'a pas très loin des paradoxes plus simples décrites au début de cet article. Commençons par une phrase “si cette phrase est vraie, alors  $X$ ”, où  $X$  est un énoncé quelconq.

- Par les propriétés d'implications ("faux  $\Rightarrow$  X" est toujours vrai), cette phrase ne peut pas être fautive.
- S'il est vrai, alors X est vrai.
- Nous venons de prouver que n'importe quel énoncé est vrai, e.g. les États-Unis et la Chine ont une frontière commune (ce que peut probablement expliquer la construction de "The Great Wall").

Il peut être formulé en termes de  $\lambda$ -calculs. Mais la "rootcause" de tout le paradoxe de ce type est la même – l'autoréférence : la phrase entière est contenue dans sa première moitié. Remarquons qu'interdire les autoréférences dans la logique n'est pas la solution parfaite, car la logique devient trop restrictive par rapport au langage naturel.

Considérons une fonction  $r$  définie comme  $r = \lambda x.((xx) \rightarrow y)$ .  $(rr)$   $\beta$ -se réduit en  $(rr) \rightarrow y$ . Si  $(rr)$  est fautive, alors  $(rr) \rightarrow y$  est vrai par le principe d'explosion, mais cela est contradictoire avec la  $\beta$ -réduction. Donc  $(rr)$  est vrai. On en déduit que  $y$  est aussi vrai. Comme  $y$  peut être arbitraire, on a prouvé que n'importe quelle proposition est vraie. Contradiction.

## Théorème de Gödel

Les deux paradoxes discutés ci-dessus, sont basés sur le même concept de l'autoréférence : une proposition ou n'importe quel objet qui référence lui-même (e.g., ensemble des tous les ensembles). Faut-il interdire l'autoréférence dans les constructions mathématiques ? L'idée n'est pas séduisante si on rappelle que avec les paradoxes, nous avons jeté dans la poubelle tous les constructions récursives.

Néanmoins, l'autoréférence a une influence forte sur le fondement de mathématique. Un résultat clé et le plus connu comme le théorème de l'incomplétude a été prouvé par Kurt Gödel en 1930. Une des interprétations prétend que la consistance d'un système d'axiomes ne peut pas être prouvée en n'utilisant que ces axiomes (voici l'autoréférence !). En particulier, pour prouver la consistance d'arithmétique il faut ajouter les axiomes supplémentaires (qui a été vite fait, en 1936). Le seul problème est que maintenant il faut prouver un autre système...

Pour ceux qui veulent plonger dans le sujet de l'autoréférence, nous pouvons conseiller un livre "Gödel, Escher, Bach : Les Brins d'une Guirlande Éternelle" de Douglas Hofstadter.

La crise des fondements a déclenché plusieurs études sur le sujet. Nous avons brièvement présenté deux modèles qui ont été les candidats sur le rôle de base minimale de l'arithmétique. Cependant, le  $\lambda$ -calcul non typé est

contradictoire car il contient des paradoxes. La consistance de l'arithmétique Péano a été prouvée un an après, en utilisant la récurrence transfinie par Gerhard Gentzen. D'après le théorème de Gödel, l'ajout d'une proposition supplémentaire dans le système des axiomes a été nécessaire. C'était une idée qui a été manquante pendant presque 50 ans entre la publication des axiomes de Péano et la preuve de Gentzen.

Pour résumer le sujet de l'arithmétique, disons que dans la version moderne on construit le fondement toujours à partir des axiomes de Péano. Pour la consistance, au lieu de la récurrence transfinie, on rajoute la théorie des ensembles de Zermelo-Fraenkel avec l'axiome de choix. Néanmoins, chez les mathématiciens il n'y a pas de consensus si le deuxième problème de Hilbert est résolu ou non.

## **Machine de Turing et calculabilité**

Cependant, comme il est souvent en science, il faudrait étudier le même domaine de point de vue un peu différent. Cela a été fait sur l'autre continent par un jeune étudiant Alan Turing. Il a cherché une solution pour un problème de la décision posé en 1928 par Hilbert et Ackermann : "trouver un algorithme qui détermine dans un temps fini, s'il un énoncé est vrai ou faux". La formalisation d'un terme algorithme a conduit au concept de machine de Turing connu par tout le monde. Entre autres, le théorème de Gödel a été reformulé en termes d'une machine de Turing. Le résultat a été aussi négatif, connu comme un théorème de Turing-Church: "il existe des énoncés pour lesquels on ne peut pas déterminer" (vérifier l'énoncé et le nom d'un théorème).

## **Thèse de Church-Turing**

Le résultat positif. S'il existe des fonctions, qu'il ne peuvent pas être décidées, on se pose la question, qu'est-ce que ce sont les fonctions simples, i.e. les fonctions que l'on peut effectivement calculer. Intuitivement, c'est dont la valeur peut être calculée avec un crayon si on a suffisamment de papier et du temps. Mais vous comprenez déjà que les mathématiciens n'acceptent pas les solutions intuitives... Le problème de décision est lié avec un problème de calculabilité. Qu'est-ce que signifie qu'une fonction peut être calculée ? Souvent on se réfère sur "des méthodes d'un crayon et de papier". Indépendamment, chaque des deux a proposé que toute fonction calculable en termes de crayon et papier peut être calculée par son méthode (lambda-calcul ou la machine de Turing). Les deux propositions - ne sont pas des théorèmes, peuvent pas être prouvées car on ne peut pas formaliser autrement calculabilité. (On doit remarquer

ici qu'il y avait le troisième mécanisme de déterminer la calculabilité - les fonctions récursives primitives). Relativement vite il a été prouvé que tous les 3 mécanismes sont équivalents. Donc, n'importe lequel peut être utilisé comme une définition de fonction effectivement calculable.

## Résumé. Impacts de $\lambda$ -calcul

Le concept de  $\lambda$ -calcul a joué un rôle tellement important dans l'informatique théorique que l'on peut voir ses échos en pratique : dans la plupart des langages de programmation une notion de  $\lambda$ -fonction représente une fonction "anonyme". Cette notion rend le terme connu par des ingénieurs mais la plupart ne connaît pas les détails cachés derrière. Cela provoque souvent les discussions dans StackOverflow similaires à celui-ci : *"Another obvious case for combinators is obfuscation. A code translated into the SKI calculus is practically unreadable. If you really have to obfuscate an implementation of an algorithm, consider using combinators, here is an example."*

En réalité le concept a eu quatre impacts principaux.

1. *Formalisation d'une notion de calculabilité.* Avant les années 1930s, la définition de calculabilité pouvait être caricaturisée comme "calculable à l'aide du papier, crayon et suffisamment du temps". En plus il y avait une intuition que les fonctions récursives doivent définir la classe des fonctions calculables. L'invention de  $\lambda$ -calcul et machine de Turing a relancé une discussion sur la notion de calculabilité. Comme tous les trois concepts ont été prouvés équivalents, les mathématiciens se sont mis d'accord à les utiliser comme une définition formelle de calculabilité.
2. *Preuves de calculabilité.* Car les trois concepts sont équivalents, n'importe lequel peut être utilisé pour prouver la calculabilité d'un nouveau objet. Donc on peut considérer  $\lambda$ -calcul comme un outil de plus (en réalité utilisé plus souvent pour prouver qu'un objet n'est pas calculable).
3. *Preuves formelles.* La version des  $\lambda$ -calcul typés peut être appliquée dans la théorie des preuves. Ainsi, les certains langages de preuves formelles tels que Coq ou AUTOMATH sont basés sur ce modèle.
4.  $\lambda$ -calcul est un *langage de programmation* primitif (en nombre de constructions). Comme la machine de Turing est le fondement de tous les langages impératifs,  $\lambda$ -calcul est une base pour les langages fonctionnels tels que Haskell ou OCaml.



## Part 2. Les formalités.

### Introduction informelle dans la programmation fonctionnelle

Le concept de langage de programmation le plus élémentaire connu par tout le monde est une machine de Turing. Sa ruban contenant les instructions et les données se traduit facilement dans la programmation impérative – un paradigme implémenté par “OVER9000” des langages populaires. Dans ce paradigme, le processus du calcul est décrit en termes des instructions qui changent l’état de “calculateur”. Les caractéristiques de programmes impératifs sont :

- L’état se change par des instructions de l’affectation ( $v = E$ ).
- Les instructions sont exécutées consécutivement ( $C1; C2; C3$ ).
- Il y a un mécanisme de branchement (`if`, `switch`).
- Il y a un mécanisme de boucles (`while`, `for`).

Exemple (le calcul d’un factoriel impératif):

```
res = 1;
for i = 1..n:
    res = res * i;
```

C’est un programme le plus compliqué dans cet article. Cependant, on peut voir clairement que l’exécution est impérative car il est composé de instructions consécutives qui translatent le calculateur de l’état initial à son état final. Une partie de l’état final (variable `res`) est interprétée comme un résultat du calcul.

En parallèle de l’approche programme comme l’instruction, il existe un paradigme fonctionnel qui présente un programme comme une fonction. Par exemple, le factoriel est une expression qui dépend de l’entrée `n`. L’exécution de ce programme est une suite de réduction de cette expression jusqu’à l’expression triviale qui ne contient que le résultat. De plus,

- Il n’y a pas de notion des états ainsi que des variables.
- Pas de variables – pas de l’opération de l’affectation.
- Pas de cycles, car il n’y a pas de différences entre les itérations.
- L’ordre de calcul n’est pas important car les expressions sont indépendantes.

En revanche, le paradigme fonctionnel nous donne:

- La récursion à la place des boucles.
- Fonctions d'ordre supérieur, i.e., les fonctions qui prennent à l'entrée et renvoient autres fonctions.
- Filtrage par motif.

Bien sûr, quelqu'un peut répliquer que toutes ses détails sont présents dans la plupart des langages modernes. En fait, les langages modernes sont multi-paradigmes – ils prennent les meilleurs des tous. Par contre, langage machine et donc Assembler restent les langages purs impératifs. De plus, rajoutons qu'en programmation fonctionnelle, toutes les fonctions sont *pures*, i.e., ne dépendent que des ces paramètres.

Dans la suite de cet article, nous construisons  $\lambda$ -calcul qui joue le rôle de “machine de Turing” pour la programmation fonctionnelle.

**Remarque.** *La construction complète sont technique et même la définition de  $\lambda$ -calcul dépasse largement la taille de cet article. Nous essayons plutôt de donner une idée comment les primitives de la programmation impérative peuvent être exprimés en termes de  $\lambda$ -calcul. Ainsi, on ne donnera pas l'exemple plus sophistiqué que le calcul d'un factoriel.*

## Deux opérations<sup>1</sup>

Dans  $\lambda$ -calcul nous n'avons que deux moyen pour construire les expressions : *application* et *abstraction*.

**1. Application.** La notion  $\mathbf{f} \ \mathbf{x}$  signifie que  $\mathbf{f}$  est appliqué à  $\mathbf{x}$ . Du point de vue de codeur on peut dire qu'un algorithme  $\mathbf{f}$  est appliqué à l'entrée  $\mathbf{x}$ . Cependant nous construisons un *système formel* où il n'y a pas de différence entre les algorithmes et les données, donc l'auto-application est aussi autorisée :  $\mathbf{f} \ \mathbf{f}$ .

**2. Abstraction.** Soit  $M \equiv M[x]$  est une expression qui (probablement) contient  $x$ . Dans ce cas, la notion  $\lambda x.M$  signifie une fonction  $x \rightarrow M[x]$  qui mappe  $x$  à  $M[x]$ . Ainsi,  $\lambda$ -abstraction est un moyen de créer une fonction anonyme en partant d'une expression  $M$ .

**Example.** *Considerons  $\lambda$ -expression :  $(\lambda x.2x+8)17$ . Le calcul est une série de réductions d'une paire abstraction-application :*

$$(\lambda x.2 \cdot x + 8)17 = (x := 17)2 \cdot 17 + 8 = 42.$$

---

<sup>1</sup>based on Moskvina's presentations

Cette réduction s'appelle  $\beta$ -réduction. Si les expressions sont liées par la  $\beta$ -réduction, on dit qu'elles sont  $\beta$ -équivalentes. Le règle formel de  $\beta$ -équivalence est suivant :

$$(\lambda x.M)N =_{\beta} M[x := N]$$

Dans  $\lambda$ -calcul sans type il n'y a rien à part de l'application, l'abstraction et  $\beta$ -réduction. En plus, pour omettre les parenthèses, nous avons les accords suivants :

- L'application est gauche-associative, i.e.,  $FXYZ := (((FX)Y)Z)$ .
- L'abstraction est droite-associative, i.e.,  $\lambda xyz.M := (\lambda x.(\lambda y.(\lambda z.M)))$ .
- L'abstraction s'applique à tous ce qu'elle arrive à "toucher", i.e.,  $\lambda x.MNK := \lambda x.(MNK)$ .

**Variables libres et liées.** Considerons un terme  $M[x]$ . On dit que variable  $x$  est *libre* dans  $M$ . Par contre, dans l'abstraction  $\lambda x.M[x]$ , variable  $x$  devient liée par un  $\lambda$ .

**Example.** Dans le terme ci-dessous, les variables  $x$  et  $y$  sont liées,  $z$  et  $w$  sont libres.

$$(\lambda y.(\lambda x.xz)y)w$$

## What is $\lambda$ ?<sup>2</sup>

En mathématique la notion d'une fonction est liée avec une application, i.e., règle qui transform paramètre dans un résultat. Au contraire,  $\lambda$ -calcul est une théorie des "fonctions comme formules". La différence est ce que une formule formelle n'est pas obligatoirement se traduit en règle bien précise. Commençons par un exemple. En arithmétique on peut écrire :

Soit  $f$  est une fonction  $x \rightarrow x^2$ . Considerons  $A = f(5)$ .

En langage de lambda, on peut écrire simplement:

$$(\lambda x.x^2)(5).$$

L'expression  $\lambda x.x^2$  signifie une fonction qui mappe  $x$  à  $x^2$ . Puis ce terme est succédé par une *application* de cette fonction au nombre 5. L'un des avantage de cette notation est simplicité dans la construction des fonctions

---

<sup>2</sup>based on Selinger's lecture notes

de l'ordre supérieur : si  $f : X \rightarrow X$  est une fonction, alors la composition  $f \circ f$  peut s'écrire comme  $\lambda x.f(f(x))$ . Ceci n'est pas simple, mais l'opération qui mappe  $f$  à  $f \circ f$  s'écrit mécaniquement de manière suivante :

$$\lambda f.\lambda x.f(f(x)).$$

(On peut le comparer avec  $g : (X \rightarrow X) \rightarrow (X \rightarrow X)$  où  $g(f) = f \circ f$  pour tout  $f : X \rightarrow X$ .) Le vrai avantage est visible si on considère un terme  $f$  suivant:  $\lambda x.x$ . Rien inattendu, c'est une fonction d'identité. Mais que vaut  $f(f)$ ? Par définition, si on pose  $x = f$ ,

$$f(f) = (\lambda x.x)(f) = f.$$

Remarquons que  $f(f)$  n'a jamais du sens dans mathématique classique car la fonction ne peut pas être incluse dans sa propre domaine de définition.

**todo:  $\lambda$  sans type est simplement typé**

## Combinateurs

Le cas spécial de  $\lambda$ -termes sans type sont les termes qui n'ont pas des variables libres. Ils s'appellent *combinateurs*. Voici les exemples des combinateurs classiques :

- **I** =  $\lambda x.x$  – combinateur d'identité
- **K** =  $\lambda xy.x$  – “suppresseur”
- **S** =  $\lambda fgx.fx(gx)$  – “distributeur”

En fait, tout les combinateurs peuvent être exprimés en termes de ces trois – on dit qu'ils forment la base chez les combinateurs. Cependant, cette base n'est pas minimal, car **I** = **SKK**. **Théorème** Tout les combinateurs peuvent être exprimés en termes de **K** et **S**.

Mais **I** est très utile pour simplifier les calculs car sans lui les formules sont trop longues. Pour cette raison, on parle plutôt du système **S, K, I**. Autres exemples des combinateurs avec leurs représentations en base **S, K** et **S, K, I** (todo!) :

- $\omega = \lambda x.xx = \mathbf{SII}$  (verify!)
- $\Omega = \omega\omega = (\lambda x.xx)\lambda x.xx$
- $\mathbf{C} = \lambda fxy.fyx = \mathbf{S}((\mathbf{S}(\mathbf{KS})\mathbf{K})(\mathbf{S}(\mathbf{KS})\mathbf{K})(\mathbf{KK}))$
- $\mathbf{B} = \lambda fgx.f(gx) = \mathbf{S}(\mathbf{KS})\mathbf{K}$

- $W = \lambda xy. xyx = \mathbf{SS}(\mathbf{K}(\mathbf{SKK}))$

Notons qu'on peut penser de logique combinatoire comme du  $\lambda$ -calcul sans symbol  $\lambda$  – les deux systèmes sont équivalents, la différence n'est que dans le brique de base :

- Dans  $\lambda$ -calcul, nous utilisons l'application et l'abstraction des fonctions aux variables.
- Dans logique combinatoire on part des fonctions d'ordre supérieur, i.e., les fonctions qui ne contiennent pas de variables libres.

Les constructions logiques dans le monde combinatoire seront probablement (ou pas) présentés en autres articles. Dans le futur, nous ne considérons que  $\lambda$ -calcul.

## Prise en main<sup>3</sup>

Dans *lambda*-calcul sans type nous n'avons qu'un seul primitif - des fonctions. Donc, si on veut l'utiliser pour la programmation, c'est à nous de réaliser même les objets les plus élémentaires, tels que les nombres ou les constantes booléennes.

$tru := \lambda t. \lambda f. t$  est une fonction qui renvoie son premier argument,  
 $fls := \lambda t. \lambda f. t$  est une fonction qui renvoie son deuxième argument.

Les termes *tru* et *fls* vont jouer le rôle de **vrai** et **faux**. Cependant, pour l'instant ils ne sont que des formules formelles qui manquent du contexte. Comme ce contexte, définissons la fonction *if*

$$if := \lambda b. \lambda x. \lambda y. bxy$$

Ici, *b* est une condition de branchement, *x* et *y* sont des branches **then** et **else**. Donc, pour “montrer” que *tru* et *fls* correspondent aux constantes logiques, nous avons besoin de prouver le suivant:

$$\begin{aligned} if\ tru\ t\ e &= t, \\ if\ fls\ t\ e &= e \end{aligned}$$

Pour les formules formelles “prouver” signifie “partir d'une expression à gauche, appliquer les certaines règles afin d'obtenir une expression à droite”. Donc c'est le moment de parler de ces règles. Pour  $\lambda$ -calcul il y en a deux :

---

<sup>3</sup>Basé sur l'article russe <https://habr.com/ru/post/215991/>

1.  **$\alpha$ -equivalence.** Informellement<sup>4</sup>, on dit que deux termes sont équivalents s'il coïncident à renommage des variables abstraites près. Par exemple,  $\lambda x.f(x) \equiv_{\alpha} \lambda y.f(y)$ . Par une variable abstraite on comprend une variable qui est présente à gauche et à droit du point, e.g.,  $x$  et  $y$  eux-mêmes ne sont pas  $\lambda$ -équivalents car il ne sont pas abstraites.

## 2. $\beta$ -reduction.

Notons que dans  $\lambda$ -calcul, il n'y a rien sauf application, abstraction et  $\beta$ -équivalence.

*Preuve (if fls t e = e).*

$$\begin{aligned}
 \text{if fls } t \ e &= (\lambda b. \lambda x. \lambda y. b \ x \ y) \ \text{fls } t \ e && \text{par définition de if} \\
 &= (\lambda x. \lambda y. \text{fls } x \ y) \ t \ e && \text{par } \beta\text{-reduction de } \lambda b \\
 &= (\lambda y. \text{fls } t \ y) \ e && \text{par } \beta\text{-reduction de } \lambda x \\
 &= \text{fls } t \ e && \text{par } \beta\text{-reduction de } \lambda y \\
 &= (\lambda t. \lambda f. f) \ t \ e && \text{par définition de fls} \\
 &= (\lambda f. f) \ e && \text{par } \beta\text{-reduction de } \lambda t \\
 &= e && \text{par } \beta\text{-reduction de } \lambda f
 \end{aligned}$$

□

Un lecteur curieux peut vérifier par lui-même que  $\text{if } \text{tru } t \ e = e$ . De plus, un vrai passionné peut essayer de trouver les bonnes expressions pour conjonctions (“and”), disjonction (“or”) ainsi que negation (“not”).

**spoiler.**

- $\text{and} = \lambda x. \lambda y. x \ y \ \text{fls}$
- $\text{or} = \lambda x. \lambda y. x \ \text{tru } y$
- $\text{not} = \lambda x. x \ \text{fls } \text{tru}$

## Pourquoi langage de programmation?

Comment peut-on montrer que langages de programmation **X** et **Y** sont équivalents ? Il faut montrer deux proposition : (i) un programme quelconque écrit en **X** peut être réécrit en **Y** et (ii) un programme quelconque écrit en **Y** peut être réécrit en **X**. Helas, prouver ces deux réductions entre **citation?**

---

<sup>4</sup>la définition formelle peut être trouvée dans n'importe quel livre, mais sa complexité dépasse largement les objectifs du blog.

$\lambda$ -calcul et le machine de Turing est assez technique et demande quelques dizaines de pages écrites. Donc nous nous limiterons à la démonstration de deux mécanismes :

- Pour le *branchement*, nous avons montré ci-dessus que le terme **if** joue le rôle de même opérateur dans la programmation.
- Ci-dessous nous montrerons que la *réursion* est aussi possible en  $\lambda$ -calcul. Ce mécanisme va jouer le rôle des boucles qui n'existent pas dans ce système.

Informellement, on comprend très bien que ces deux mécanismes sont suffisants pour écrire n'importe quel programme.<sup>5</sup> De plus, la construction de récursion n'est pas simple du tout.

### Calcul du factoriel. Ingrédients

Supposons que nous sommes beaucoup avancés dans le sujet et réussis à construire les fonctions suivantes (rappelons que par défaut il n'y a pas ni nombres ni opérations arithmétiques à  $\lambda$ -calcul).

- **1**, juste nombre 1, mais il faut le construire à l'aide de application et abstraction;
- **isZero**, si argument de cette fonction est égal à 0, elle renvoie *true*, sinon *false*;
- **mult** renvoie un produit de ces deux arguments.
- **pred** prend à l'entrée un nombre naturel et calcule son prédécesseur (souvenez-vous des axiomes de Peano, si vous avez déjà lu la partie 1). Pourtant, cette fonction est la plus complexe : la construction a été inventée par Kleene pendant l'extraction de son dent de sagesse. Aujourd'hui, l'anesthésie n'est pas pareille...

### Calcul du factoriel. 1ère approche

Tous ces ingrédients nous permettent d'introduire un factoriel assez naturellement :

$$\mathbf{fact} = \lambda x. \mathbf{if} (\mathbf{isZero} \ x) \ 1 \ (\mathbf{fact} \ (\mathbf{pred} \ x))$$

Rien de miracle, si  $x$  est égal à 0, on renvoie 1, sinon – le produit de  $x$  et factoriel de  $x - 1$ . Si on remplace **fact** par sa définition, on obtient une série infinie de réductions. We have a problem...

---

<sup>5</sup>La vraie difficulté est dans la formalisation de cette dernière proposition, ainsi que dans la propre construction pour la récursion arbitraire qui est faite à l'aide des combinateurs.

## Calcul “lazy”

J’espère que vous protestiez contre cela, en argumentant que pour calculer **fact 0**, nous n’avons pas besoin de substitutions infinies car nous savons déjà que le troisième argument de **if** sera ignoré. Tout a fait, mais les règles de jeu “Informatique théorique” nous impose d’utiliser que les opérations bien précis : si on prétend que  $\lambda$ -calcul est un langage de programmation, alors on doit être capable de proposer un algorithme qui l’exécute et donc aucun ambiguïté n’est pas toléré. Dans notre cas on a “oublié” de fixer l’ordre de calcul. Considerons un terme suivant :

$$(\lambda x.x) ((\lambda x.x) (\lambda z. (\lambda x.x)z))$$

Pour simplicité on peut le réécrire :

$$\mathbf{id} (\mathbf{id} (\lambda z. \mathbf{id} z))$$

Ce terme-là contient 3 redexes. Nous n’avons plusieurs choix de l’ordre des réductions :

- **$\beta$ -reduction complète.** Le redex est choisi au hasard à chaque étape. Il est facile de voir que si l’expression initiale est finie, le résultat ne dépend pas de l’ordre de calcul (rappelons qu’il n’y a pas de notion d’état, donc les effets de bord sont impossibles). Voici une des réductions possibles d’une expression ci-dessus :

$$\begin{aligned} & \mathbf{id} (\mathbf{id} (\lambda z. \mathbf{id} z)) \\ &= \mathbf{id} (\mathbf{id} (\lambda z. z)) \\ &= \mathbf{id} (\lambda z. z) \\ &= \lambda z. z \end{aligned}$$

- **L’ordre normal.** À chaque étape on choisit un redex le plus gauche (i.e., le plus externe) :

$$\begin{aligned} & \mathbf{id} (\mathbf{id} (\lambda z. \mathbf{id} z)) \\ &= \mathbf{id} (\lambda z. \mathbf{id} z) \\ &= \lambda z. \mathbf{id} z \\ &= \lambda z. z \end{aligned}$$

- **L’appel par nom.** L’ordre de calcul est identique à l’ordre normal. En plus, on interdit les réductions à l’intérieur de l’abstraction. Dans



notre exemple on s'arrête sur l'étape avant dernier :

$$\begin{aligned} & \underline{\text{id} (\text{id} (\lambda z. \text{id } z))} \\ &= \underline{\text{id} (\lambda z. \text{id } z)} \\ &= \lambda z. \text{id } z \end{aligned}$$

Une version optimisée de cette stratégie est utilisé par Haskell par défaut. C'est le calcul "lazy".

- **L'appel par valeur.** On commence par un redex le plus gauche (externe), dans la partie droite duquel il y a une valeur – un terme clos qui ne peut plus être réduit :

$$\begin{aligned} & \underline{\text{id} (\text{id} (\lambda z. \text{id } z))} \\ &= \underline{\text{id} (\lambda z. \text{id } z)} \\ &= \lambda z. \text{id } z \end{aligned}$$

Cette stratégie est utilisée dans la plupart des langages de programmation : pour exécuter une fonction, on calcule d'abord tous ces arguments.

Remarquons que les tout les stratégies sauf calcul lazy formellement interdit la récursion. Le mécanisme de "lazyness" est fait exactement pour éviter les calculs non-nécessaires. En réalité, cette mécanisme est utilisé dans la plupart des langages, mais pas dans l'exécution de fonction. Dans le code suivant : `if a and b: ...`, si `a` est déjà fausse, il est probable que `b` ne sera jamais calculé, ce que nous oblige de porter plus d'attention sur les effets de bords possible.

## Combinator de point fixe

(attention, beaucoup de lettres cursives...)

**Définition.** Un point fixe de  $\lambda$ -fonction  $f$  est une fonction  $x$  tel que

$$f\ x \equiv_{\beta} x$$

**Théorème.** En  $\lambda$ -calcul (ainsi qu'en logique combinatoire), pour chaque terme  $x$  il existe au moins un terme  $p$  tel que  $xp = p$ . De plus, il existe un combinateur  $Y$  tel que  $Yx = xYx$ .

*Preuve.* Pour prouver ce théorème, construisons un tel combinateur :

$$Y = \lambda f. \underline{(\lambda x. f(x x))}(\lambda x. f(x x))$$

Appliquons la réduction à l'expression soulignée :

$$Y = \lambda f. f((\lambda x. f(x x))(\lambda x. f(x x)))$$

On en déduit que  $Y f = f Y f$ . Donc  $Y f$  est un point fixe de  $f$ .  $Y$  s'appelle un *combinateur du point fixe*. En cette formule-là il a été introduit par Haskell Curry <sup>6</sup> (on rappelle que combinateur est un terme dont tout les variables sont liées par  $\lambda$ -abstraction).  $\square$

## Calcul du factoriel. Y à l'aide !

L'un des rôles du combinateur de point fixe est de se faire priver de la récursion en  $\lambda$ -calcul – celui-ci nous permettra de calculer le factoriel sans un truc avec calcul “lazy”. Considérons une fonction :

$$\mathbf{fact}' = \lambda f. \lambda x. \mathbf{if}(\mathbf{isZero} \ x) \mathbf{1}(\mathbf{mult} \ x(f(\mathbf{pred} \ x)))$$

Cette fonction est très ressemblant à **fact**. La seule différence est ce que **fact'** au lieu d'appliquer lui-même sur **pred x**, applique  $f$  qui est son paramètre. Donc, si on pose  $f := \mathbf{fact}$ , notre fonction calcule le factoriel. Autrement dit, **fact' fact = fact**, i.e., **fact** est un point fixe de **fact'**. Donc

$$\mathbf{fact} = \mathbf{Yfact}'$$

Cette fonction n'est pas récursive et elle calcule le factoriel du  $x^7$

## Résumé

Si vous pensez que  $\lambda$ -calcul est simple et les règles décrites sont évidentes, je vais vous déranger : essayer de calculer ce terme-là. **todo** Cependant ce n'est que  $\lambda$ -représentation de nombre 2 (vous vous probablement souvenez qu'il n'y

---

<sup>6</sup> Un combinateur d'un point fixe n'est pas unique (en réalité, il y a un nombre infini de tels). Par exemple, celui a été proposé par Alan Turing :

$$\Theta = (\lambda x. \lambda y. (y(xxy)))(\lambda x. \lambda y. (y(xxy)))$$

<sup>7</sup> En revanche le calcul devient le-e-e-ente : pour calculer  $5!$  il faut faire 66066  $\beta$ -réductions ! Evidemment, j'ai jamais vérifié à la main, je trouve ce nombre dans mes notes du cours et je ne garantis pas que mon prof a fait les calculs lui-même. . .

a pas de nombres en  $\lambda$ -calcul). Cette réduction a été fait par Kleene, pendant son visite au dentiste (pourtant, les analgésiques modernes sont moins efficace en mathématique). Vu le nombre des efforts que nous avons besoin pour les calculs les plus primitifs,  $\lambda$ -calcul sans type reste une construction purement théorique, qui représente un modèle de calcul aussi puissant que la machine de Turing. En enrichissant ce modèle par des constantes et système de types, on s'approche relativement vite au langages fonctionnels existants comme Haskell. Cependant, on perd le simplicité de la construction.

Maintenant, je vous conseil de relire la partie 1 pour voir le contexte dans lequel ce système a été introduit aux années 30s. Si vous n'avez vraiment rien compris, comment on fait les calculs avec  $\lambda$ , ce site va surement aider d'avoir l'idée ce que s'est passé.