Disclaimer

Le deux partie de ce document sont indépendantes et peuvent être lues dans n'importe quel ordre. L'objectif initial de partie 1 a était de discuter qu'est-ce que c'est un corrècte théorie mathématique en utilisant l'arithmétique comme exemple. Cependant, ce tentative s'est transformé vite dans un essai sur l'histoire de mathématique de la fin XIXème siècle jusqu'au milieu du XXème. On se concentre principalement sur les problèmes de consistence et notamment sur la deuxième problème de Hilberte. Il est evident que notre approche ne peut jamais être complète, donc on se restreinte sur le contèxte historique des avancement en formalisation d'arithmétique ainsi que la notion de calculabilité. Intentionellement, on évite au maximum les définitions formelles et formules et invite le lécteur curieux de faire ses propres recherches en partant du Wikipédia (plutôt anglais) qui présente la plupart des définitions manquantes.

Au contraire, l'objectif de partie 2 est de donner au lecteur le goût de λ-calcul : on présente une définition plutôt complète, puis on montre, comment construire une language de programmation elementaire mais aussi puisant que la machine de Turing. Ce langage et connu comme la paradigme fonctionnelle.

Finalement, l'ordre de lécture conseillée est plutôt le suivant : un passage "diagonale" de partie 2, puis lecture complète de partie 1 et la relecture de la partie 1 pendant lequelle on devra voir le sense derrière le forêt de detailles techniques nécessaires pour introduire un système formel tel que λ-calcul.

Part 1. Histoire de λ-calcul

Qu'est-ce que c'est un λ-calcul ? Les codeurs sont familiers avec une notion de λ-fonction - une fonction anonyme qui est utilisée dans les morceaux du code qui ne méritent pas d'avoir un méthode nommé : clé de trie, les petits transformation dans les requêtes à la sql etc. Cépendant, la notion de λ-fonction a été pris d'une système de calcul aussi puisant que la machine de Turing (est inventé 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 fondaments
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 fondaments

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). Connue 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 déstin. 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.

Cépendant 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 fondaments* en mathématique.

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

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

Dans le sens plus formel, la consistence 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 propertés 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 domain de mathématique qui étude les nombres et rélations entre eux. Elle est appliquée partout de premiers années de l'école jusqu'à les conceptes modernes d'astrophysique. Cépendant, 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 $x-y$ renvoit toujours un nombre valide, les nombres rationnels aparaissent si on étudie la division. Finalement, les nombres algébrique sont résponsable pour résoudre les équatinos polynomials 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 dépuis la find 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 suive b et a suive c, alorc $b = c$;

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 depuil 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 strictité 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, omittons 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 définit par une autre closure. Informellement on rempli des "trous" sur l'axe des nombres. Pour les plus curieux qui souhaitent la construiction formelle, la page de wikipedia est bien explicatif. Personnellement, je prefère la construiction par des coupes de Dedekind.

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

Heureusement, la vraie preuve d'une consistence 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 **λ-calcul non-typé**. Malheureusement, son étudiant, Kleene a prouvé que cette construction n'a pas été consistente.

λ-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.

Rappelon 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 $\lambda x.fa$. Naturelement, ion peut définir une composition... Pour transformer des propositions on a une règle de β-reduction.

Malgré sa simplicité et abstraité, cette construction permet néanmoins rédefinir tout les opérations arithmétiques, la logique Booléen etc. Est-ce que λ-calcul non-typé est un bon candidat pour le rôle de fondament 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 le propriétés d'implications ("faux =¿ X" est toujours vrai), cette phrase ne peux pas être faux.
- S'il est vrai, alors X est vrai.
- Nous venons de prouvé que n'importe quel enoncé 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 λ-calculs. Mais la "rootcause" de tout les paradoxe de ce type est la même – l'autoréférence : la phrase entière est contenu dans sa première motié. Remarquons qu'intérdire les autoréférences dans la logique n'est pas la solution parfaite, car la logique devient trop restrente par rapport au langage naturelle.

Considerons une fonction r définie comme $r = \lambda x.((xx) \to y)$. (rr) β-se réduit en $(rr) \to y$. Si (rr) est faux, alors $(rr) \to y$ est vrai par le principe d'explosion, mais cela est contradictoire avec la β-réduction. Donc (rr) est vrai. On en déduit que y est aussi vrai. Comme y peut être arbitraire, on a prouvé que n'importe quel proposition est vrai. 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 réference lui-même (e.g., ensemble des tout les ensembles). Faut-il intérdir l'autoréference dans les constructions mathématiques ? L'idée n'est pas séduisant si on rappel que avec les paradoxes, nous avons jeté dans la poubelle tout les construcitons récursives.

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

contradictoire car il contient des paradoxes. La consistence de l'arithmétique Péano a été prouvé un an après, en utilisant la récurrence transfinie par Gerhard Gentzen. D'après le théorème de Gödel, l'ajout d'un proposition supplémentaire dans le système des axiomes a été nécessaire. C'était une idée qui a été manquant 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 consistence, au lieu de la récurrence transfinie, on rajout la théorie des ensemble 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

Cépéndant, 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 jeun étudiant Alan Turing. Il a cherché une solution pour une 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 outre, le théorème de Gödel a été reformuler en thermes d'une machine de Turing. Le résultat a été aussi négative, connu comme une théorème de Turing-Church: "il existe les énoncés pour lesquels on ne peux pas déterminer" (vérifier l'enoncé et le nom d'un théorème).

Thèse de Church-Turing

Le résultat positif. S'il existe les fonctions, qu'il peuvent pas être décidées, on se pose la question, qu'est-ce que ce sont les fonction simple, 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 suffisament 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 une problème de calculabilité. Qu'est-ce que signifie qu'une fonction peut être calculée ? Souvent on se refère sur "des méthodes d'un crayon et de papier". Indépendement, chaque des deux a proposé que toute fonction calculable en thèrmes de crayon et papier peut être calculé par son méthode (lambda-calcul ou la machine de Turing). Les deux propositions - ne sont pas les théorèmes, peuvent pas être prouvées car on ne peut pas formaliser autrement calculabilité. (On doit remarquer

ici qu'il y avait le troisième mechanisme de détérminer la calculabilité - les fonction récursive primitive). Relativement vite il a été prouvé que tout les 3 méchanismes sont équivalent. Donc, n'importe lequel peut être utilisé comme une définition de function effectivement calculable.

Résumé. Impacts de λ-calcul

Le concept de λ-calcul a joué une rôle tellement importante dans l'informatique théorique que l'on peut voir ses échos en pratique : dans la plupart des langages de programmation une notion de λ-fonction représente une fonction "anonyme". Cette notion rends la terme connue par des ingénieurs mais la plupart ne connaît pas les détails cachés derrière. Cela provoque souvent les discussions dans StackOverflow similaire 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 suffisament du temps". En plus il y avait une intuition que les fonctions récursives doivent définir la classe des fonctions calculables. L'invention de λ-calcul et machine de Turing a relancé une discussion sur la notion de calculabilité. Comme tout les trois concept ont été prouvés équivalents, les mathématiciens se sont mis d'accord a les utiliser comme une définition formelle de calculabilité.

2. *Preuves de calculabilité.* Car les trois concept sont équivalents, n'importe lequel peut être utilisé pour prouver la calculabilité d'un nouvaux objet. Donc on peut considerer λ-calcul comme un outil de plus (en réalité outilisé plus souvant pour prouver qu'un objet est n'est pas calculable).

3. *Preuves formelles.* La version des λ-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. λ-calul est un *langage de programmation* primitif (en nombre de constructions). Comme la machine de Turing est le fondament de tout les langages impératifs, λ-calcul est une base pour les langages fonctionnels tels que Haskell ou OCaml.

Part 2. Les formalités.

Introduction informelle dans la programmation fonctionnelle

Le concepte de langage de programmation le plus elémentaire connu par tout le monde est un machine de Turing. Sa ruban contenant les instructions et les datas se traduit facilement dans la programmation impérative – un paradigme implémenté par "OVER9000" des langages populaires. Dans ce paradigme, le proces 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 instruction de l'affectation (`v = E`).
- Les instruction sont executé consécutivement (`C1; C2; C3`).
- Il y a un méchanisme de branchement (`if`, `switch`).
- Il y a un méchanisme 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'execution est imperative car il est composé de instructions consecutives qui translatent le calculateur de l'état initial à son état final. Une partie de l'état final (variable `res`) est interprété comme un résultat du calcul.

En parallèle de l'approche programme comme l'instruction, il existe une paradigme fonctionnel qui présente un programme comme une fonction. Par example, le factoriel est une expression qui depende de l'entrée `n`. L'execution 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 indépendant.

En revanche, le paradigme fonctionnel nou 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 prennes les meilleurs des tous. Par contre, langage machine et donc Assembler restent les langages pures impératifs. De plus, rajoutons qu'en programmation fonctionnelle, toutes les fonctions sont *pures*, i.e., ne dependent que des ces paramètres.

Dans la suite de cet article, nous construisont λ-calcul qui joue le rôle de "machine de Turing" pour la programmation fonctionnelle.

Remarque. *La construction complète sont téchnique et même la définition de λ-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 λ-calcul. Ainsi, on ne donnera pas l'example plus sophistiqué que le calcul d'un factoriel.*

Deux operations[1]

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

1. Application. La notion `f x` signifie que `f` est appliqué à `x`. Du point de vue de codeur on peut dire qu'un algorithme `f` est appliqué à l'entrée `x`. Cependant nous construisons un *système formel* ou il n'y a pas de différence entre les algorithmes et les données, donc l'auto-application est aussi autorisée : `f 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 $x \to M[x]$ qui mappe x à $M[x]$. Ainsi, λ-abstraction est un moyen de créer une fonction anonyme en partant d'une expression M.

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

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

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

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

[1] based on Moskvin's presentations

Dans λ-calcul sans type il n'y a rien à part de l'application, l'abstraction et β-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 λ.

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 λ?[2]

En mathématique la notion d'une fonction est liée avec une application, i.e., règle qui transform paramètre dans un résultat. Au contraire, λ-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 :

$$\text{Soit } f \text{ est une fonction } x \to x^2. \text{ Considerons } A = f(5).$$

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 de l'ordre supérieur : si $f : X \to X$ 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 méchaniquement de manière suivante :

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

[2]based on Selinger's lecture notes

(On peut le comparer avec $g : (X \to X) \to (X \to X)$ où $g(f) = f \circ f$ pour tout $f : X \to X$.) 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 inclue dans sa propre domaine de définition.

todo: λ sans type est simplement typé

Combinateurs

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

- $\mathbf{I} = \lambda x.x$ – combinateur d'identité
- $\mathbf{K} = \lambda xy.x$ – "suppresseur"
- $\mathbf{S} = \lambda fgx.fx(gx)$ – "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 $\mathbf{I} = \mathbf{SKK}$. **Théorème** Tout les combinateurs peuvent être exprimés en termes de $\mathbf{K}$ et $\mathbf{S}$.

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

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

Notons qu'on peut penser de logique combinatoire comme du λ-calcul sans symbol λ – les deux systèmes sont équivalent, la difference n'est que dans le brique de base :

- Dans λ-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 contient pas de variables libres.

Les constructions logiques dans le monde combinatoire seront probablement (ou pas) presentés en autres articles. Dans le futur, nous ne considerons que λ-calcul.

Prise en main[3]

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

$tru \quad := \lambda t.\lambda f.t \quad$ est une fonction qui renvoie son premier argument,
$fls \quad := \lambda t.\lambda f.t \quad$ 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 context. Comme ce contexte, définissons la fonction if

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

Ici, b est une condition de branchement, x et branche **then** et y corresponds à **else**. Donc, pour "montrer" que tru et fls corréspondent au 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 certains règles afin d'obtenir une expression à droit". Donc c'est le moment de parles de ces règles. Pour λ-calcul il y en a deux :

1. α**-equivalence**. Informellement[4], on dit que deux termes sont équivalent 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 λ-equivalents car il ne sont pas abstraites.

[3]Basé sur l'article russe `https://habr.com/ru/post/215991/`

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

2. β**-reduction**.

Notons que dans λ-calcul, il n'y a rien sauf application, abstraction et β-equivalence.

Preuve $(if\ fls\ t\ e = e)$.

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

□

Un lecteur courieux peut vérifier par lui-même que $if\ tru\ t\ e = e$. De plus, un vrai passioné peut essayer de trouver les bonnes expressions pour conjunctions ("and"), disjonction ("or") ainsi que negation ("not").

spoiler.

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

Pourquoi langage de programmation?

TODO. Recursion à l'aide de combinators

Comment peut-on montrer que langages de programmation **X** et **Y** sont équivalent ? Il faut montrer deux proposition : (i) un programme quelconque écrit en **X** peut être réecrit en **Y** et (ii) un programme quelconque écrit en **Y** peut être réecrit en **X**. Helas, prouver ces deux réductions entre **citation?** λ-calcul et le machine de Turing est assez technique et demande quelques disaines de pages écrites. Donc nous nous limiterons à la démonstration de deux méchanismes :

- Pour le *branchement*, nous avons montre ci-dessus que terme **if** joue le rôle de même operateur dans la programmation.

- Ci-dessous nous montrerons que la *récursion* est aussi possible en λ-calcul. Ce méchanisme va jouer le rôle des boucles qui n'existe pas dans ce système.

Informellement, on comprends très bien que ces deux méchanismes sont suffisantes pour écrire n'importe quel programme.[5] De plus, la construction de recursion n'est pas simple du tout.

Ingrédients. Supposons que nous sommes beaucoup avancés dans le sujet et réussis à construire les fonctions suivantes (rappellons que par défaut il n'y a pas ni nombres ni opérations arithmétiques à λ-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 au 0, elle renvoie tru, sinon – fls;
- **mult** renvoie un produit de ces deux arguments.
- **pred** prend à l'entrée un nombre naturel et calcul son prédesesseur (souvez-vous des axiomes de Peano, si vous avez déjà lu la partie 1). Pourtant, cette fonction est le plus complexe : le construiction à été inventé par Kleene pendend l'extraction de son dent de sagesse. Aujourd'hui, l'anesthésie n'est pas pareil...

1ère approche. Tout ces ingrédients nous permets d'introduir un factoriel assez naturellement :

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

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 son définition, on obtient une série infinie des réductions. We have a problem...

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 λ-calcul est un langage de programmation, alors on doit être capable de proposer un algorithme qui l'execute et donc aucun ambiguïté n'est pas toléré. Dans notre cas on a "oublié" de fixer l'ordre de calcul. Considerons un terme suivant :

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

[5] La vrai dificulté est dans la formalisation de cette dernière proposition, ainsi que dans le propre construction pour la récurtion arbtitraire qui est fait à l'aide des combinateurs.

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 :

- **β-reduction complète.** Le redex est choisi au hazard à chaque étape. Il est facile de voir que si l'expression initiale est finie, le résultat ne dépends 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 :

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

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

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

- **L'appel par nom.** L'ordre de calcul est identique à l'ordre normal. En plus, on interdit les reductions à l'intérieur de l'abstraction. Dans notre example on s'arrête sur l'étape avant dernier :

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

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

- **L'appel par valeur.** On commence par un redex les plus gauche (externe), dans la partie droite duquel il y a une valeur – un terme clos

qui ne peut plus être réduit :

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

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

Remarquons que les tout les strategies sauf calcul lazy formellement interdit la récursion. Le méchanisme de "lazyness" est fait exactement pour éviter les calculs non-nécessaires. En réalité, cette mechanisme est utilisé dans la plupart des langages, mais pas dans l'execution 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.

Y-combinator – Est-ce qu'on a vraiment besoin de cette partie ?

Résumé

Si vous pensez que λ-calcul est simple et les règles décrites sont évidents, je vais vous déranger : essayer de calculer ce terme-là. **todo** Cependant ce n'est que λ-répresentation de nombre 2 (vous vous probablement souvenez qu'il n'y a pas de nombres en λ-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, λ-calcul sans type reste une construction purement théorique, qui répresente 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 λ, ce site va surement aider d'avoir l'idée ce que s'est passé.