Introduction

L'objectif initial de la première partie était de discuter autour de la définition d'une théorie mathématique correcte en prenant l'arithmétique comme exemple. Cette tentative s'est vite transformée en un essai sur l'histoire des mathématiques de la fin du XIX$^{\text{ème}}$ jusqu'au milieu du XX$^{\text{ème}}$ siècle, en se concentrant principalement sur les problèmes de consistance et notamment sur le deuxième problème de Hilbert.

Il est évident que notre approche ne peut être considérée comme complète, on s'est restreint au contexte historique des progrès concernant la formalisation de l'arithmétique ainsi que la notion de calculabilité. Intentionellement, on a évité au maximum les définitions formelles. On invite plutôt le lecteur curieux à faire ses propres recherches en partant de Wikipédia (plutôt la version anglaise) qui contient la plupart des définitions manquantes.

A l'opposé, l'objectif de la deuxième partie était de donner au lecteur le goût du λ-calcul : on présente une définition plutôt complète, puis on montre comment construire un language de programmation élémentaire mais aussi puissant que la machine de Turing. Ce langage est connu comme le paradigme fonctionnel.

De fait, les deux parties de ce document sont indépendantes. Cependant, il est conseillé de commencer par un passage "diagonale" sur la partie 2, puis une lecture complète de la partie 1 et enfin la relecture de la partie 2 pendant lequelle on pourra faire attention au sens global derrière la forêt de détails techniques nécessaire pour introduire un système formel tel que le λ-calcul.

Partie 1. Histoire du λ-calcul

Qu'est-ce que le λ-calcul ? Les développeurs sont en général à l'aise avec la notion de λ-fonction : une fonction anonyme qui est utilisée dans des morceaux de code qui ne méritent pas d'avoir un nom : clé de tri, petite transformation dans des requêtes similaires à du SQL... Cependant, la notion de λ-fonction a été reprise d'un système de calcul aussi puissant que la machine de Turing (et inventée dans les mêmes années 30). Dans cet article, on présente l'histoire de l'invention du λ-calcul afin de mieux appréhender ce concept difficile.

Plan

1. Crise des fondements
2. Axiomes de Peano
3. 1ère version du λ-calcul
4. Theorème de Gödel
5. Machine de Turing et calculabilité
6. Thèse de Church-Turing
7. Impact et applications

Crise des fondements

Qui a déjà vu cette phrase : *"Cette phrase est fausse"* ? Probablement personne. Tout le monde sait qu'elle n'est ni vraie ni fausse (si on suppose qu'elle est vraie, alors elle doit être fausse et inversément). Connue depuis presque 3000 ans sous le nom du paradoxe du menteur, les mathématiciens ont eu l'habitude de vivre avec jusqu'à la fin de XIX$^{\text{ème}}$ siècle. D'autres formulations similaires ont vu le jour depuis :

- **Paradoxe du barbier.** Dans un village, un barbier rase tous les habitants du village 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 de grains.
- **Paradoxe du crocodile.** Un méchant crocodile vous attrape et vous propose de deviner votre destin. Si votre réponse est incorrecte, il vous mange. La réponse ? – "Tu vas me dévorer" !

Pour une liste exhaustive des paradoxes simples, on peut consulter le livre de Martin Gardner : "Aha! Gotcha. Paradoxes to puzzle and delight".

Maintenant, discutons d'un autre saboteur de logique : le *Paradoxe de l'ensemble de Russel*. Disons qu'un ensemble est *simple* s'il n'appartient pas à lui-même. Par exemple, l'ensemble de tous les gens est simple, car cet ensemble n'est pas une personne. Ainsi, l'ensemble de tous les ensembles n'est pas simple par définition. *L'ensemble de Russel* est un ensemble qui contient tous 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'étudient ni les crocodiles, ni les barbiers. Les questions de menteur font plutôt partie des compétences des philosophes ou du code pénal. Cependant, dans la version de Russel, ce paradoxe n'utilise que des constructions formelles des mathématiques. Cela signifie que de telles constructions sont contradictoires elles-mêmes : si nous avons prouvé qu'une formule propositionnelle est à la fois vraie et fausse, n'importe quel théorème peut également devenir vrai et faux à la fois. Si on ajoute à cela qu'au début de XX$^{\text{ème}}$ siècle, le paradoxe de Russel n'a pas été le seul paradoxe connu, on a un aperçu de ce qu'a été la *crise de fondements* en mathématiques.

Cette crise a été formalisée dans le second élément de la liste des 23 problèmes de Hilbert déterminants le développement des mathématiques au XX$^{\text{ème}}$ siècle.

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

Dans le sens le plus formel, la consistance peut être définie par les trois propositions suivantes :

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

L'étape 1 est plutôt constructive : en pratique, il est suffisant de produire les nombres (entiers, rationnels, réels) avec leurs propriétés habituelles. Dans l'étape 2, il faut prouver que les axiomes sont indépendants les uns des autres. Ce qui est équivalent à dire que si on supprime n'importe quel axiome, l'étape 1 n'est plus vrai. L'étape 3 est la plus compliquée.

Axiomes de Peano

L'arithmétique est le domaine des mathématiques qui étudie les nombres et leurs relations. Elle se retrouve partout, des premières années d'école primaire jusqu'aux concepts modernes d'astrophysique. Cependant, pour construire les bases de l'arithmétique, il est presque suffisant de bien déterminer les nombres naturels ainsi que leurs interactions (les nombres entiers sont une extension des nombres naturels pour que l'opération $x - y$ renvoit toujours un nombre valide ; les nombres rationnels aparaissent si on étudie la division ; les nombres algébriques sont utilisés pour résoudre les équations polynomiales et le reste, c'est pour "boucher les trous").

Classiquement, les nombres naturels peuvent être définis de la même façon qu'on l'explique aux enfants lorsqu'ils apprenent à compter. Ce résultat est connu depuis la fin du XIX$^{\text{ème}}$ siècle comme les axiomes de Peano :

1. 1 est naturel;
2. le nombre suivant d'un nombre naturel est naturel;
3. rien n'est suivi de 1;
4. si a suit b et a suit c, alorc $b = c$;
5. axiome de récurrence (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 maintenant que ces axiomes sont suffisants pour construire l'ensemble des nombres naturels.

- Les **nombres naturels** sont déjà construits. Pour être honête, nous avons construit des objets "bizzares" et les avons appelés des nombres naturels – prenez l'habitude qu'en mathématique fondamentale, il est courant de parler de choses évidentes avec des mots très sophistiqués. Le vrai avantage de cette approche est son aspect absolument **correct**.
- **Zéro** est indispensable pour compter. Rien de plus simple - ajoutons un nouvel objet spécial, qui est (i) suivi de 1. Appelons le "0". Posons que pour tout n naturel (ii) $n+0 = 0+n = n$ et aussi (iii) $n\cdot 0 = 0\cdot n = 0$ (cela servira dans le futur).
- On peut ainsi compter et même calculer la somme, mais les mathématiciens veulent plus de symétrie, ils aimeraient l'opération réciproque de "+". 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 de c tel que $c + 5$

= 2 – n'oubliez pas qu'on souhaite une exactitude absolue, donc on ne peut utiliser que des nombres déjà calculés. Nous n'avons pas le choix: disons que "2 - 5" est un nouveau nombre. Ainsi que "1 - 2", "42 - 45" et même "239 - 261". Cela semble beaucoup, mais remarquons que "2 - 5" est égal à "42 - 45" et aussi à "0 - 3". Par simplicité, omettons zéro et écrivons juste -3. Félicitations ! Vous venez de construire les nombres **négatifs** et donc les nombres **entiers** ! Cet opération s'appelle une **clôture** et est usuelle pour générer de nouveaux objets.

- Les **nombres rationnels** arrivent en utilisant la même logique : si nous pouvons calculer le produit, alors nous voulons également diviser. Les résultats de toutes les divisions possibles (1/2, -2/3, 2/4, 37/17, 5/5, etc.) forment les nombres rationnels.

- Imaginons tous les nombres rationnels sur un axe. D'un certain point de vue, cet axe est très dense – pour n'importe quel nombre rationnel, il existe un autre nombre rationnel qui est "aussi proche de lui que l'on veut". Mais ce n'est pas suffisant ! Malheureusement, $\sqrt{2}$ n'est pas rationnel (à propos : dans le $7^{\text{ème}}$ problème de Hilbert, il s'agit de prouver que $\sqrt{2}^{\sqrt{2}}$ n'est pas rationnel). Donc les **nombres réels** sont définit par une autre clôture. Informellement, on remplit les "trous" sur l'axe des nombres.

  Pour les plus curieux qui souhaitent aborder la construction formelle, la page de wikipedia sur la construction des nombres est bien faite et très explicite. Personnellement, je prefère la construction par coupure de Dedekind.

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

Heureusement pour nous, la preuve de la consistance des axiomes de Peano est un problème beaucoup plus sophistiqué que l'invention de ses axiomes, et l'histoire ne fait donc que commencer...

1ère version du λ-calcul

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

Le λ-calcul a formalisé l'application d'une fonction. Il envisage la compréhension d'une fonction comme une "règle". L'écriture classique $f(x)$ pointe plutôt sur le résultat de cette règle.

Rappelons brièvement ce que c'est. La brique principale est la 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 $\lambda x.fa$. Naturellement, on peut définir une composition de fonctions. Pour transformer des propositions, on a une règle de β-reduction.

Malgré sa simplicité et son caractère abstrait, cette construction permet de redéfinir toutes les opérations arithmétiques, la logique booléenne... Est-ce que le λ-calcul non-typé est un bon candidat pour le rôle de fondement des mathématiques ? La réponse est **non**: à cause du paradoxe de Kleene-Rosser proposé en 1935 par J. B. Rosser et Stephen Kleene (l'étudiant de Church).

Ironiquement, ce paradoxe, beaucoup plus sophistiqué dans sa version initiale, n'est pas très éloigné des paradoxes plus simples décrits au début de cet article. Commençons par la phrase suivante "si cette phrase est vraie, alors X", où X est un énoncé quelconque.

- Par la propriété d'implication ("faux $\rightarrow$ X" est toujours vrai), cette phrase ne peux pas être fausse.
- Si elle est vraie, alors X est vrai.
- Nous venons de prouver 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").

Cette phrase peut être formulée en termes de λ-calcul. Mais la cause principale de tous les paradoxes de ce type est la même – l'autoréférence : la phrase entière est contenue dans sa première motié. Remarquons qu'interdire les autoréférences dans la logique n'est pas la solution parfaite, car la logique devient trop restreinte par rapport au langage naturel.

Considérons une fonction r définie comme $r = \lambda x.((xx) \rightarrow y)$. (rr) β-se réduit en $(rr) \rightarrow y$. Si (rr) est faux, alors $(rr) \rightarrow y$ 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 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 réference lui-même (par exemple, l'ensemble de tous les ensembles). Faut-il interdire l'autoréference dans les constructions mathématiques ? L'idée n'est pas séduisante si on rappelle qu'avec les paradoxes, nous avons jeté à la poubelle toutes les constructions récursives.

Néanmoins, l'autoréférence a une influence forte sur le fondement des mathématiques. Un résultat clé 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'autoréference !). En particulier, pour prouver la consistance de l'arithmétique, il faut ajouter des axiomes supplémentaires (ce qui a été rapidement fait, en 1936). Le seul problème est que, maintenant, il faut prouver un autre système...

Pour ceux qui veulent creuser le sujet de l'autoréférence, nous vous conseillons le livre suivant : "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 candidats au rôle de base minimale de l'arithmétique. Cependant, le λ-calcul non typé est contradictoire car il contient des paradoxes. La consistance de l'arithmétique Peano 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. Ce fut l'élément manquant pendant presque 50 ans, entre la publication des axiomes de Peano et la preuve de Gentzen.

Pour résumer le sujet de l'arithmétique, disons que, dans la version moderne, on utilise toujours les axiomes de Peano comme méthode de construction. Pour la consistance, on rajoute la théorie des ensembles de Zermelo-Fraenkel avec l'axiome du choix au lieu de la récurrence transfinie.

Néanmoins, encore aujourd'hui, il n'y a pas de véritable consensus chez les mathématiciens pour savoir si le deuxième problème de Hilbert est résolu ou non.

Machine de Turing et calculabilité

Comme souvent en science, il est utile d'étudier le même domaine d'un point de vue un peu différent. Cela a été fait en Angleterre par un jeune étudiant, Alan Turing. Il a cherché une solution au problème de la décision posé en 1928 par Hilbert et Ackermann : "trouver un algorithme qui détermine, dans un temps fini, si un énoncé est vrai ou faux". La formalisation d'un tel algorithme a conduit au concept de machine de Turing connu par tout le monde. En outre, le théorème de Gödel a été reformulé en utilisant le concept de machine de Turing.

Le résultat a été aussi négatif, connu sous le nom du théorème de Turing-Church: "il existe des énoncés pour lesquels on ne peux pas déterminer". **todo: vérifier l'enoncé et le nom du théorème**

Thèse de Church-Turing

S'il existe des fonctions qui ne peuvent pas être décidées, alors on peut se poser la question de ce que sont les fonctions simples, i.e. les fonctions que l'on peut effectivement calculer. Intuitivement, ce sont celles dont la valeur peut être calculée avec un crayon si on a suffisament de papier et de temps. Mais vous savez bien que les mathématiciens n'aiment pas les solutions intuitives... Le problème de décision est lié à un problème de calculabilité. Que signifie qu'une fonction peut être calculée ?

Souvent, on se refère à "des méthodes de crayon et de papier". Indépendement, chaque des deux (Church et Turing) a proposé que toute fonction calculable en termes de crayon et de papier peut être calculée par sa méthode (λ-calcul ou machine de Turing). Les deux propositions ne sont pas des théorèmes et ne peuvent pas être prouvées car on ne peut pas formaliser autrement la calculabilité. On doit remarquer ici qu'il y avait un troisième méchanisme pour déterminer la calculabilité : les fonctions récursives primitives. Relativement vite, il a été prouvé que les 3 méchanismes sont équivalents. Donc, n'importe lequel peut être utilisé comme une définition de fonction effectivement calculable.

Impact et applications

Le concept de λ-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, on retrouve la notion de λ-fonction qui représente une fonction "anonyme". Cette notion rend le terme connu par tous les développeurs mais la plupart ne connaissent pas les détails qui se cachent derrière. Cela provoque souvent des discussions dans StackOverflow similaires à celle-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 caricaturée comme "calculable à l'aide de papier, de crayon et de suffisament de temps". En plus, il y avait l'intuition que les fonctions récursives doivent définir la classe des fonctions calculables. L'invention du λ-calcul et de la machine de Turing a relancé la discussion sur la notion de calculabilité. Comme les trois concepts ont été prouvés équivalents, les mathématiciens se

sont mis d'accord pour les utiliser comme une définition formelle de calculabilité.

2. *Preuves de calculabilité.* Puisque les trois concept sont équivalents, n'importe lequel peut être utilisé pour prouver la calculabilité d'un nouvel objet. On peut donc considérer le λ-calcul comme un outil de plus (en réalité plus souvent utilisé pour prouver qu'un objet est n'est pas calculable).

3. *Preuves formelles.* La version du λ-calcul typé peut être appliquée dans la théorie des preuves. Ainsi, certains langages de preuves formelles tels que Coq ou AUTOMATH sont basés sur ce modèle.

4. Le λ-calcul est un *langage de programmation* primitif (en nombre de constructions). Comme la machine de Turing est le fondement de tous les langages impératifs, le λ-calcul est une base pour les langages fonctionnels tels que Haskell ou OCaml.

Partie 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.