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.

2$^{\text{ème}}$ 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 → 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 $r = \lambda x.((xx) \to y)$. (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 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

Impératif contre fonctionnel

Une fois dit “ordinateur”, je pense de la machine de Turing et à l’envers : si vous dites “machine de Turing” j’imagine tout de suit un PC. Pas de surprise, c’est un concepte de langage de programmation ou de l’ordinateur le plus elémentaire – tellement élementaire que la notion “ordinateur” et “langage de programmation” n’ont pas la différence. La machine de Turing est un ruban contenant les instructions et les datas et un automaton que le lit, efface, réecrit et déplace. Tout ensemble elle se traduit facilement dans un programme impératif – un paradigme implémenté par “over 9000” des langages populaires. Dans ce paradigme, le processus du calcul est décrit en termes des instructions qui changent l’état de “calculateur”. Les caractéristiques des programmes impératifs sont :

- L’état se change par des instruction 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 boucle (`while`, `for`). [1]

Exemple (le calcul d’un factoriel impératif):

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

On voit clairement que ce programme est imperatif car il est composé de instructions consecutives qui translatent l’executeur de l’état initial à son état final. Une partie de l’état final (variable `res`) est interprétée comme un résultat du calcul.

Ce programme est très simple, mais pourriez-vous de proposer un machine de Turing qui calcul un factoriel ? Si vous l’avez jamais fait à la main – amusez-vous, c’est un bon éxo de gymnastique mentale. [2]

En parallèle de l’approche programme comme l’instruction, il existe une paradigme fonctionnel qui présente un programme comme une fonction. Par

[1] Il est suffisant d’avoir une instruction du saut inconditionnel (`goto`). N’importe quelle boucle est équivalent à une combinaison de `if` et `goto`. Mais, comme sa utilisation est considéré comme une grosse bêtise de developpeur afin de ne pas enflammer la guerre sainte, restons sur un mécanisme de boucle.

[2] Si vous preferez de lire ou regarder comment les autres réalisent ce type de perversions mathématiques on ne vous jugera pas.

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, nous définirons un autre langage primitif – λ-calcul qui induit la programmation fonctionnelle de même façon que la machine de Turing induit la programmation impérative. Nous fixons comme objéctif de réécrire une fonction qui calcul un factoriel d'un nombre entier. Pour cela nous passerons par tout les étapes nécessaire :

- grammaire du langage (application et abstraction) ;
- règle d'execution (α-equivalence et β-réduction) ;
- codage des nombres (nombres de Church) et des booléens ;
- récurtion à la place de boucles.

Remarque.

1. *La construction complète est téchnique et longue, donc nous allons sauter les certains détails sans pitié. Notre objectif est de donner une idée comment les primitives de la programmation impérative peuvent être exprimés en termes de λ-calcul. Dans tout les cas n'utilisez pas cette article comme la seule source si un examen sur λ-calcul vous suive. Au moins vous êtes avertis.*

2. *Si vous avez un examen dans une semaine, les sources suivantes sont pas mals :* ***todo****.*

3. *Si vous avez un examen dans demain et vous ne comprenez rien, lisez au moins ça : Alligator Eggs!*

Qu'est ce que λ ?

Définition. *λ-terme ou λ-expression est une expression qui satisfait la grammaire suivante :*

1. $\Lambda \to V$

2. $\Lambda \to \Lambda\ \Lambda$

3. $\Lambda \to \lambda V.\Lambda$

où V est un ensemble des chaîne sur l'alphabet fixe $\Sigma \setminus \{“\lambda”, “.”, “\ ”\}$.

La première règle définie des idéntifiants – variables et fonctions. La deuxième est *application* d'une terme à l'autre. La troixième définie une *abstraction.* Si tous ce qu'il est écrit ci-dessous était claire, vous pouvez passer directement au β-reduction. Sinon, rajoutons du sens au cet abracadabra.

1. Idéntifiants. Initialement, nous considérons comme identifiant n'importe quel chaîne qui ne contient trois caractères spéciaux : “λ”, “.” et “ ” (espace). Ainsi, x, f, 42, *hello* et $x + 5$ sont les idéntifiants.

2. Application. La notion $f\ x$ signifie qu'un terme f est appliqué à la terme x. Du point de vue de codeur on peut dire qu'un algorithme f est appliqué à l'entrée x. Mais, comme nous construisons un *système formel*, on est autorisé beaucoup plus, par example un auto-application : $f\ f$.

3. Abstraction. Soit $\mathbf{M}$ est un λ-terme qui contient x à l'intérieur (on écrit $\mathbf{M} \equiv \mathbf{M}[x]$). Dans ce cas, la notion $\lambda x.M$ signifie une fonction $x \to \mathbf{M}[x]$ qui mappe x à $\mathbf{M}[x]$.

Remarque. *Important ! Le $\mathbf{M}$ ou $\mathbf{M}[x]$ est un pseudonyme pour un λ-terme. Dans la suite, nous remplacerons souvent les certains termes par leurs “pseudonymes” pour simplifier les expressions. Tout ces pseudonymes sont écrit en graisse.*

Une autre moyen de voir l'abstraction est une construction d'une fonction anonyme : imaginons une fonction $f(x) := \mathbf{M}[x]$. Dans la notation du λ-calcul, $f(x)$ corrésponds à $\lambda x.\mathbf{M}[x]$. L'avantage de telle écriture est ce qu'on voit clairement que la fonction depend de x mais il n'y a pas d'ambiguïté avec fonction et ça valeur en x. Finalement, si on veut valeur $f(x = a)$, on écrit

$$\lambda x.\mathbf{M}[x]\ a$$

Cela justifie l'application : $\lambda x.\mathbf{M}[x]$ s'applique à a. Dans la suite nous omettrons souvent $[x]$ et écrirons simplement $\lambda x.\mathbf{M}\ a$. Ainsi, λ-abstraction est un moyen de créer une fonction anonyme en partant d'une expression M.

Les trois règles ci-dessus sont les **seules** opérations autorisées pour construire les expressions λ-calcul. On repète parce qu'il est important que notre univers de λ-calcul ne sait rien sauf construire des phrases avec des ces trois règles : il n'y a pas ni nombres ni opérations arithmétique – rien. Par exemple, l'idéntifiant $x + 5$ n'a pas de sence "calculer la somme", c'est juste une chaîne des caractères, un mot, un objet atomique de la théorie. Ainsi, on ne peut pas "calculer" le "résultat" d'abstraction $f\ x$ – ce n'est qu'une construction formelle.

β-reduction

Définition. *La β-équivalence est définie de manière suivante :*

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

S'il est claire, bienvenu au α-équivalence : variables libres et liées. Sinon, voici une traduction de la langue Klingon.

Soit dans la formule ci-dessus, $\mathbf{M} = f\ x\ z\ x$ [3]. Dans ce cas-là, on peut "calculer" l'application en remplaçant tout les occurences de l'x dans $\mathbf{M}$ par a et enlevant λ :

$$\lambda x.\mathbf{M}\ a = (\lambda x.f\ x\ z\ x)\ a \equiv_\beta f\ a\ z\ a$$

Qu'est-ce que signifie "calculer" pour un λ-terme ? On dit que termes $(\lambda x.f\ x\ z\ x)\ a$ et $f\ a\ z\ a$ sont *β-équivalents* (pour cela on utilise un symbole "$\equiv_\beta$"). L'opération qui enlève λ un remplaçant un terme par son β-équivalent, s'appele un *β-réduction*. Cela veut dire que "calculer" signifie "appliquer les β-réductions pour rendre un λ-terme initial le plus simple possible.

[3] Ne pensez pas de f comme de la fonction qui prends deux paramètre x et z. On rappelle que c'est une formule formelle et f, x ainsi que z sont trois λ-termes qui ont les mêmes "droits".

Remarquons aussi que nous avons besoin de paranthèses car nous ne savons pas jusqu'à quel terme s'applique abstraction. Pour minimiser le nombre de paranthèses en suite, nous fixons les accords suivants sur les priorités entre les opérations :

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

$$\lambda x.\mathbf{M}\ \mathbf{N}\ \mathbf{K} := \lambda x.(\mathbf{M}\ \mathbf{N}\ \mathbf{K})$$

α-équivalence : variables libres et liées

Considerons un terme $\mathbf{M}[x]$ qui contient un idéntifiant (i.e., variable) x. On dit que, dans le terme $\lambda x.M[x]$, la variable x est *liée* par un λ-abstraction. Si une variable n'est pas liée, on dit qu'elle est *libre*. La définition concrète est un peut détaillé (essayez de le construire par vous-même) mais la notion est assez simple et intuitive – vérifier quand même un example ci-dessous.

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

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

Considerons deux termes $\mathbf{fx} := \lambda x.f\ x$ et $\mathbf{fy} := \lambda y.f\ y$. Si on applique chaque de deux termes à un terme quelconque a, on obtient le même résultat :

$$\mathbf{fx}\ a = (\lambda x.f\ x)\ a \equiv_\beta f\ a$$
$$\mathbf{fy}\ a = (\lambda y.f\ y)\ a \equiv_\beta f\ a$$

Cela veut dire que les deux termes qui différent seulment par les variables liées actionnent de même façon. Ce termes-là s'appellent *α-équivalents* :

$$\lambda x.\mathbf{M}[x] \equiv_\alpha \lambda x.\mathbf{M}[y \leftarrow x]$$

On pose que les terme α-équivalents sont égaux. C'est un deuxième et dernier rule des calcules.

Un peu du sens

Dans cette section nous essayons faire des amis entre la définition formelle de λ-calcul et des lambda-fonctions[4] qu'on peut trouver dans la plupart de

[4]pour ne pas confondre, on utilise symbol λ uniquement pour λ-calcul de Church

langages de programmation. Ce qu'il faut retenir est ce que ces deux notions sont très différentes : les intuitions prises de l'une notion servent le plus grand obstacle pour compréhension de la deuxième.

En programmation, souvent on a besoin de passer une fonction comme un paramètre : clé pour un tri ou maximum, une operation à appliquer au tout les éléments d'une collection, etc. Dans ce cas, si la fonction est simple et/ou n'est pas réutilisée, on n'a pas d'envie à elle donner un nom et on passe par un lambda. En autres mots, les deux phrases sont synonimiques et la version avec une lambda-fontion est bien plus courte :

- Soit f est une fonction $x \to x^2$. Considerons $A = f(5)$.
- $A := (\lambda x.x^2)(5)$.

En théorie de λ-calcul, notre intérêt est différent. On "oublie" qu'une fonction f est une règle qui mappe x à $f(x)$. Au lieu de cela, on considère une fonction uniquement comme une formules formelle – i.e., une phrase construite en respectant la certaine grammaire (celle que nous venons de décrire ci-dessous). Une formule formelle ne doit pas forcement être calculable, par exemple

- $1 + 2 + 3$ est une correcte formule formelle. On peut la calculer et obtenir 6.
- $a * b * c$ est une correcte formule formelle. Si dans grammaire de cette formule a, b, c corrésponds au $1, 2, 3$ et $*$ corrésponds à une somme, on peut la calculer et obtenir aussi 6.
- $1+2+3+4+5+\ldots$ est aussi une correcte formule formelle. Évidement, elle ne peut pas être calculée dans le sens commun. Cépendant si on change règles du "jeu", on peut obtenir $1/12\ldots$

Tout les λ-termes sont aussi les formules formelles : si pendant le calcul on tombe sur une forme $x\ y\ z$, ou x, y et z sont les identifiants simples[5], il faut pas essayer de les donner du sens : ils ne sont ni variables, ni fonctions, ils n'ont pas d'arité non plus – ils sont juste les briques à partir desquelles on construit l'expression.

Que peut se passer, si on essaie d'interpreter nos briques? Soint $\mathbf{f} := \lambda x.x$. Clairement, c'est une fonction d'identité car elle mappe x à x. Dans λ-calcul, on peut l'appliquer à n'importe quel terme y compris $\mathbf{f}$:

$$\mathbf{f}\ \mathbf{f} = \lambda x.x\ \mathbf{f} = \mathbf{f}$$

[5]on se refère sur le premier point de la définition de λ-terme, les identifiants sont les chaînes de caractères qui ne contiennent ni "λ", ni ".", ni espace

Autrement dit, nous avons prouvé que $f(f) = f$. Mais cela n'a aucun sens car en mathématiques la fonction ne peut pas être inclue dans sa propre domaine de définition ! Le morale est ce que *l'application de λ-termes reste une operation formelle, il est dangéreux l'intérpreter comme l'application d'une fonction classique à son argument*, malgré la seduction de tel approche. La différence est ce que une formule formelle n'est pas obligatoirement se traduit en règle bien précise.

Prise en main : booléens et branchement[6]

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. Començons par les dernières. Les termes **tru** et **fls** ci-dessous jouent le rôle de "vrai" et "faux" conformément.

$$\begin{array}{lll} \textbf{tru} & := \lambda t.\lambda f.t & \text{est une fonction qui renvoie son premier argument,} \\ \textbf{fls} & := \lambda t.\lambda f.t & \text{est une fonction qui renvoie son deuxième argument.} \end{array}$$

Pour l'instant ces termes ne sont que des formules formelles qui manquent du context. Notre contexte sera le terme de branchement **if** :

$$\textbf{if} := \lambda b.\lambda x.\lambda y.b\ x\ y$$

Ici, b est une condition de branchement, x est une "branche then" et y corresponds à "else". Donc, pour justifier que **tru** et **fls** corréspondent au constantes logiques, nous avons besoin de demontrer deux égalités :

$$\begin{gathered} \textbf{if tru}\ t\ e = t, \\ \textbf{if fls}\ t\ e = e \end{gathered}$$

Faisons donc notre premier calcul en λ, sans avoir oublié que calcul est une serie d'applications de règles décrites ci-dessus (α-équivalence et β-réduction) jusqu'à l'obtention d'un terme le plus simple possible sur lequel on n'arrive pas à appliquer aucune de deux règles.

*Preuve (***if fls** *$t\ e = e$).* Dans la serie de réductions ci-dessus nous soulignons

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

une partie expression à laquelle on applique le règle de calcul.

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

□

Un lecteur curieux 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**

Avec ces opérations supplémentaires, on peut prouver des formules plus longues (ne le faites pas à la maison – le calcul est bien plus long).

$$\textbf{if } ((\textbf{not fls})\ \textbf{or}\ (\textbf{fls and tru}))\ t\ e = t$$

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 :

- **I** $= \lambda x.x$ – combinateur d'identité. Une fois appliqué à un terme quelconque, il renvoie le même terme.
- **K** $= \lambda xy.x$ – "suppresseur". Une fois appliqué à deux termes, il ne renvoie que le premier argument.
- **S** $= \lambda fgx.f\ x\ (g\ x)$ – "distributeur" – il distribue son troisième argument au son premier et deuxième.

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 $\mathbf{I} = \mathbf{SKK}$[7].

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 $\mathbf{S}, \mathbf{K}$ ou $\mathbf{S}, \mathbf{K}, \mathbf{I}$:

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

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) presentées en autres articles. Dans le futur, nous ne considerons que λ-calcul.

Nombres de Church

Dans l'article précedant nous avons parlé des axiomes de Péano qui permet de construire nombres naturels. Maintenant on veut les "traduire" en langage de λ-calcul. Si vous ne les souvenez pas, alors on parte d'une idée qu'un nombre naturel est soit zéro[8], soit un nombre naturel plus un.

[7] dans les formulles combinatoire qui ne contient pas de "λ", on peut omettre l'espace de l'application entre les combinateurs $\mathbf{S}$, $\mathbf{K}$ et $\mathbf{I}$ – cela ne crée pas d'ambiguïté

[8] je suis d'accord, zéro n'est pas naturel mais on en a besoin pour compter quand même

Définition.

$$\begin{aligned}
\bar{\mathbf{0}} &:= \lambda s.\lambda z.z \\
\bar{\mathbf{1}} &:= \lambda s.\lambda z.s\ z \\
\bar{\mathbf{2}} &:= \lambda s.\lambda z.s\ (s\ z) \\
\bar{\mathbf{3}} &:= \lambda s.\lambda z.s\ (s\ (s\ z)) \\
&\dots
\end{aligned}$$

Si on se met sur le chemin glissant de l'interpretation, on peut penser d'un nombre de Church comme d'une fonction de deux arguments dont le premier est une fonction et deuxième est une valeur initiale. Le nombre n applique la fonction n fois à la valeur initiale et retourne le résultat. Dans ce sens résultat "d'application" d'un nombre de Church à (+1) et 0 est son "valeur numerique". Par exemple, $\mathbf{3}\ (+1)\ 0 \equiv 3$.[9]

Successeur

Définissons un terme **succ** qui "calcule" le successeur d'un nombre de Church arbitraire. Combien des arguments[10] devrait avoir ce terme ? Son premier argument doit être un nombre à augmenter. Mais un nombre en λ-calcul est aussi une fonction de deux arguments. Donc **succ** doit prendre trois arguments – un "nombre" n à augmenter, une fonction à appliquer $n+1$ fois et sa valeur initiale z :

$$\mathbf{succ} = \lambda n.\lambda s.\lambda z.\ s\ (n\ s\ z)$$

Exercice. Montrer que $\mathbf{succ}\ \overline{\mathbf{n}} = \overline{\mathbf{n+1}}$.

Solution. Utilisons la définition de **succ** et appliquons la β-réduction de λn :

$$\mathbf{succ}\ \overline{\mathbf{n}} = (\lambda n.\lambda s.\lambda z.\ s\ (n\ s\ z))\overline{\mathbf{n}} = \lambda s.\lambda z.\ s\ (\overline{\mathbf{n}}\ s\ z)$$

Puis, par la définition de $n^{\text{ème}}$ nombre de Church on a

$$\dots = \lambda s.\lambda z.\ s\ ([\lambda s'.\lambda z'.\underbrace{s'\ (s'\ \dots(s'}_{n \text{ fois}}\ z')\dots)]\ s\ z)$$

[9] On prefère d'utiliser le symbole d'équivalence ($\equiv$) au lieu d'égalité pour souligner qu'ici on parle de l'interpretation qui ne fait pas une partie de λ-calcul

[10] Par argument on comprénd des variables liées par abstraction : une fois le terme est appliqué à un autre terme, les variables liées du premier sont remplacé par le "contenu" du deuxième.

Remarqu'ons que pour évité l'ambuiguïté, on a remplacé s et z par s' et z' dans la définition de $\overline{\mathbf{n}}$ car ces variables sont liées par abstraction – α-équivalence nous autorise à le faire. Puis, si on applique la β-réduction aux $\lambda s'$ et $\lambda z'$, on obtient

$$\ldots = \lambda s.\lambda z.\ s\ (\underbrace{s\ (s\ \ldots (s}_{n \text{ fois}}\ z)\ldots) = \lambda s.\lambda z.\ (\underbrace{s\ (s\ \ldots (s}_{n+1 \text{ fois}}\ z)\ldots) = \overline{\mathbf{n+1}}$$

$\square$

Addition, multiplication et puissance

L'addition de deux nombre ressemble l'addition de 1. La seule "petite" différence est ce qu'il faut appliquer "(+1)" m fois, si le deuxième argument est $m^{\text{ème}}$ nombre de Church :

$$\mathbf{plus} = \lambda n.\lambda m.\lambda s.\lambda z.\ n\ s\ (m\ s\ z)$$

Effectivement, on peut l'interpreter comme suivant :

$$(\mathbf{plus}\ \bar{3}\ \bar{3})\ (+1)\ 0 \equiv 6$$

La multiplication de duex nombre ressemble l'addition. La seule différence est ce qu'au lieu de l'addition de 1, il faut additionner deuxième nombre :

$$\mathbf{mult} = \lambda n.\lambda m.\lambda s.\lambda z.\ n\ (m\ s)\ z$$

Exercice. Essayez de déviner que fait la fonction suivante :

$$\mathbf{power} = \lambda n.\lambda m.\lambda s.\lambda z.\ n\ m\ s\ z$$

Paire

Les termes ci-dessous "implémentent" une structure de paire : **pair** prends deux nombres et les emballe dans une paire. Si on applique **fst** ou **snd** à une paire on reçoit le premier ou le deuxième élement réspectivement, car ces deux fonctions remplacent t de **pair** par **tru** ou **fls** réspectivement.

- **pair** $:= \lambda a.\lambda b.\lambda t.\ t\ a\ b$
- **fst** $:= \lambda p.\ p$ **tru**
- **snd** $:= \lambda p.\ p$ **fls**

Soustraction

D'abord, rappelons nous qu'en λ-calcul nous n'avons pas des nombres négatifs. Donc "$n - m$" est défini si et seulement si $n \geq m$. Si c'est le cas, alors on a envie de faire de même façon que l'addition – soustraire m de n est équivalent à soustraire m fois 1 de n :

$$\mathbf{minus} := \lambda n.\lambda m.\ m\ \mathbf{pred}\ n$$

Le vrai problème est une fonction **pred** – il est compliqué d'aller à l'arrière sur l'axe de nombres si tout nos fonctions ne nous permettent qu'aller en avant ! Le focus pour calculer $n-1$ est de partir à nouveau de zéro, mais au lieu de faire n pas, il faut faire $n-1$. Il n'est pas claire, comment peut-on s'arrêter sur $n-1$. Pour cela il nous faut une paire. En ayant une paire $< i-1, i-2 >$, on va construire une paire $< i, i-1 >$. Donc, après n étapes, on aura $< n, n-1 >$.

$$\mathbf{pred} := \lambda n.\lambda s.\lambda z.\ \mathbf{snd}\ (n\ (\lambda p.\ \mathbf{pair}\ (s\ (\mathbf{fst}\ p))\ (\mathbf{fst}\ p))\ (\mathbf{pair}\ z\ z))$$

Si, en regardant sur cette formule vous ne comprenez pas, comment il s'applique, ne vous inquétez pas : moi non plus. La soustraction a été inventé par Kleene pendant l'extraction de son dent de sagesse. Aujourd'hui, l'anesthésie n'est pas pareil...

Est-il un langage de programmation ?

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 ércites. 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.[11] De plus, la construction de recursion n'est pas simple du tout.

Calcul du factoriel. 1ère approche

Listons d'abord les ingrédients, que nous aurons besoin pour calculer un factoriel :

- **1**, juste nombre 1;
- **mult** pour calculer un produit;
- **pred** – rien à dire une fonction magique;
- **isZero** – on a besoin de tester si l'argument est égale à **0** pour déterminer la terminaisont de l'induction. Dans ce cas, **isZero** renvoie **tru**, sinon – **fls**. Nous n'avons pas construit cette fonction, c'est un **exercice**, qui est assez simple si on connait le résultat : **spoiler?**

$$\mathbf{isZero} := \lambda n.n\ (\lambda c.\ \mathbf{fls})\ \mathbf{tru}$$

Tout ces ingrédients nous permets d'introduir un factoriel assez naturellement :

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

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 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

[11] 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.

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 :

- **β-réduction 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 réductions 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.

Combinator de point fixe

Définition. Un point fixe *de λ-fonction f est une fonction x tel que*

$$f\ x \equiv_\beta x$$

Théorème. *En λ-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 $\mathbf{Y}$ tel que $\mathbf{Y}x = x\mathbf{Y}x$.*

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 reduction 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 formle-là il a été introduit par Haskell Curry [12] (on reppel que combinateur est un terme dont tout les variables sont liées par λ-abstraction). □

[12] Un combinateur d'un point fixe n'est pas unique (en réalité, il y a un nombre infini de tels). Par example, celui a été proposé par Alan Turing :

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

Calcul du factoriel. Y à l'aide !

L'un de rôle du combinateur de point fixe est de se faire priver de la récursion en λ-calcul – celui-si nous permettra de calculer le factoriel sans un truc avec calcul "lazy". Considerons 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 $\mathbf{fact}'$ 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 factoirel. Autrement dit, $\mathbf{fact}'\ \mathbf{fact} = \mathbf{fact}$, i.e., **fact** est un point fixe de $\mathbf{fact}'$. Donc

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

Cette fonction n'est pas récursive et elle calcul le factoriel du x[13]

Résumé

Voici une brève introduction en λ-calcul. Déjà à partir des primitifs que nous avons décrit, on peut essayer de "programmer" un émulateur de machine de Turing. On peut constater que les construitions semblent d'être longues et lourdes, mais, si on revient sur le début, la machine de Turing qui calcule un factoriel n'est pas plus simple.

Malheureusement, le système de λ-calcul sans type que l'on vient de construire subit des paradoxes comme un paradoxe de Kleene-Rosser (voir partie 1). Pour les corriger, Church a introduit λ-calcul simplement typé qui est plus compliqué qu'un système que nous avons discuté, car avec l'arrivé des types, les termes ne sont plus compatibles avec tout le monde. Les bonnes sources pour cela sont les vrais livres mathématiques out les notes des cours corréspondants – il me semble presque impossible d'adapter le contenu pour un article (même aussi long).

Pourqoi a-t-on besoin de savoir tout cela ? Plutôt pour "l'ouverture d'ésprit". Pour la même raison qu'on manipule avec la machine de Turing. Pour quelques heures de gymnastique de cerveau finalement. Et de plus, si on arrive à comprendre les manipulations d'arithmétique de Church - on s'approche à comprendre les autres manipulations fonctionnelles et les principes de langages comme Haskell. Quand même, les langage fonctionnel sont typés, donc je vous conseille vivement de jeter au moins un oeil sur la théorie des type aussi.

[13] En revanche le calcul devient le-e-e-ente : pour calculer 5! il faut fait 66066 β-reductions ! Evidemment, j'ai jamais vérifier á la main, je trouver ce nombre dans mes notes du cours et je ne garantie pas que mon prof a fait les calculs lui-même...

Finalement, on a remarqué que λ-calcul est lente. Il est vrai. Mais dans les langages fonctionnels sont bien optimisés grâce au calcul “lazy” et filtrage par motif et ne sont pas plus lentes que les langages impératifs (il faut jamais oublié que les vraies optimisations se passent dans la couche supplémentaire entre la chaise et l'écran…). Merci au tous curieux qui a réussi à se tenir intéressé jusqu'au bout !