Le λ–calcul

Olivier Ridoux – 2021

On ne fait jamais rien seul. Mon intérêt pour le λ-calcul vient de recherches sur la compilation de λProlog engagées dans les années 1980–90 avec Pascal Brisset au sein de l'équipe de Yves Bekkers. Tous deux sont partis bien trop vite (23 mai 2010 et 7 octobre 2020), mais sont toujours présents dans mon paysage du λ-calcul. OR

Introduction

Le λ-calcul est un des plus importants modèles théoriques employés en informatique, jouant un rôle fondamental dans le développement scientifique du domaine, mais aussi présent très concrètement dans des langages de programmation aussi peu fondamentalistes que Python ou JavaScript.

Comme souvent en science, il contient sa part de sérendipité puisqu'il a d'abord échoué à ce pourquoi il a été inventé, et ses inventeurs ont été les premiers surpris de ce qu'il permettait vraiment. Il est aussi un très bel exemple de ce que l'informatique n'est pas uniquement gouvernée par un progrès technologique frénétique puisque sa conception date de 1930 et que sa spécification est restée inchangée, mais intensément utilisée depuis cette date.

Outre sa mise en œuvre dans des langages comme Python ou JavaScript, il est aussi à l'origine de langages de programmation plus disciplinés comme LISP dès 1958 et les langages de la famille ML à partir des années 70. Plus généralement, le λ-calcul est un modèle explicatif commun pour un très grand nombre de phénomènes de programmation, l'appel de fonction/procédure, la récursivité, le typage, les structures de données, etc. D'une certaine façon, tous les langages de programmation doivent quelque chose au λ-calcul.

Le λ-calcul est aussi une théorie très belle, très simple dans son expression initiale, qui a connu de nombreuses ramifications, donc très fructueuse. En particulier, ses variantes typées entretiennent des liens très riches avec la logique, faisant de cette théorie une passerelle entre programmation et démonstration, ouvrant par là la voie à une forme de certification de programme.

Définitions

Le λ-calcul est défini par une notation, celle des λ-termes (ou λ-expressions), et par des règles de réduction qui portent sur celle-ci. La grammaire suivante donne une présentation assez standard de la notation des λ-termes :

$$\Lambda ::= V \quad \text{(les variables)}$$
$$\mid \lambda V.\Lambda \quad \text{(les abstractions)}$$
$$\mid (\Lambda\ \Lambda) \quad \text{(les applications)}$$

Ex. si x, y et z sont des variables, $\lambda x.x$ et $\lambda x.\lambda y.(y\ (z\ x))$ sont des λ-termes. On convient des raccourcis suivants : (1) omettre les parenthèses imbriquées à gauche, ex. $(x\ y\ z)$ pour $((x\ y)\ z)$; (2) fusionner les cascades d'abstractions, ex. $\lambda xy.(x\ y)$ pour $\lambda x.\lambda y.(x\ y)$; (3) noter par un exposant les applications répétées d'un même λ-terme, ex. $(x^3\ y)$ pour $(x\ (x\ (x\ y)))$. On note les méta-variables par des majuscules, E, F, G, etc., et les variables par des minuscules, x, y, z, etc.

L'abstraction est un quantificateur (① au verso), c-à-d. $\lambda x.$E lie x dans E. Ex. dans $\lambda x.(x\ \lambda y.(y\ x))$, x est libre dans la partie soulignée et liée dans la partie surlignée, de même y est lié dans la partie soulignée. On appelle combinateur un λ-terme où toutes les occurrences de variables sont liées. On donne aux combinateurs des noms selon leur fonction, ou selon la tradition, ex. $\omega = \lambda x.(x\ x)$ et $\Omega = (\omega\ \omega)$.

Les règles de réduction (② au verso) sont les suivantes :

$$\lambda x.\,E \rhd \lambda y.\,E_{[y/x]} \quad (\alpha\text{-conversion})$$
$$(\lambda x.\,E\ F) \rhd E_{[F/x]} \quad (\beta\text{-réduction})$$

La α-conversion est le renommage des variables quantifiées. La β-réduction exprime que lorsque une abstraction $\lambda x.$E est appliquée à un λ-terme F (un β-rédex), cela se réduit en E où on a substitué F à la variable x. On appelle λ le système qu'elles forment, et la λ-réduction par passage au contexte comme en ②.

Comme pour toute quantification, des restrictions imposent que ne sont substituées que les occurrences libres de x dans E, et que ni y ni aucune variable libre de F ne doit se trouver liée par la substitution. Il est toujours possible de satisfaire ces restrictions en appliquant des α-conversions à $\lambda x.$E. Ex. $(\lambda xa.(x\ a)\ a)$ ne peut pas se β-réduire car la variable a libre se trouverait liée indument, mais la α-conversion $[b/a]$ produit $(\lambda xb.(x\ b)\ a)$ qui se β-réduit en $\lambda b.(a\ b)$.

Une λ-réduction peut s'arrêter sur un λ-terme λ-normal, comme ci-dessus, et on dit qu'elle converge, ou bien produire indéfiniment des λ-termes non λ-normaux, ex. $\Omega \longrightarrow \Omega \longrightarrow \dots$. Un même λ-terme peut être le point de départ d'une λ-réduction qui converge et d'une autre qui ne converge pas, ex. $(\lambda xy.x\ \ a\ \ \Omega)$.

Confluence et convergence du λ-calcul

Le Théorème de confluence de Church–Rosser montre que le λ-calcul vérifie la propriété de confluence (② au verso) ; $\rhd^*\lhd_\beta$ est donc une relation d'équivalence.

Des paires de λ-termes ne sont pas bien traitées par la β-équivalence. Les membres de ces paires se comportent toujours identiquement quand on les applique aux mêmes λ-termes, mais ils ne sont pas λ-équivalents. Ex. $\lambda x.(a\ x)$ et a. En effet, quel que soit E, $(\lambda x.(a\ x)\ E) \equiv_\beta (a\ E)$, mais $\lambda x.(a\ x)$ et a sont λ-normaux et différents ; ils ne sont donc pas λ-équivalents. Une nouvelle règle de réduction tient compte de ces paires de λ-termes :

$$\lambda x.\,(E\ x) \rhd E \quad (\eta\text{-réduction})$$

Ici, on impose que x ne soit pas libre dans E, et contrairement aux autres règles, si la restriction n'est pas satisfaite aucune α-conversion ne peut la satisfaire. Le λ-calcul avec η-réduction jouit des mêmes propriétés de confluence que sans.

Le théorème de confluence a pour conséquence que lorsqu'elles convergent, toutes les réductions d'un même λ-terme convergent vers le même point. C'est bien, mais peut-on décider si un λ-terme a une réduction qui converge ? La réponse est non, car cela reviendrait à résoudre le problème de l'arrêt. Tant pis, mais existe-t-il des stratégies pour trouver une réduction qui converge si il en existe une ? La réponse est oui ; la stratégie de réduction à gauche a cette propriété. Elle consiste à toujours réduire le β-rédex le plus à gauche. Enfin, si plusieurs réductions d'un même λ-terme convergent, convergent-elles toutes aussi efficacement ? La réponse est encore non : ex. $(\lambda xy.x\ \ a\ \ (\lambda sz.(s^n\ z)\ \lambda x.x\ b))$ peut converger en 1, 2, …, n+1 coups.

Ces observations conduisent à la définition et à la comparaison de stratégies qui dictent dans quel ordre sont faites les réductions. Ex. ces stratégies sont utilisées pour spécifier comment sont passés les paramètres dans les langages de programmation. Ex. soit un λ-terme $(\lambda xyz.E\ X\ Y\ Z)$, on peut :

- réduire d'abord les X, Y et Z. C'est la stratégie par valeur, qui a l'avantage que même si il y a beaucoup d'occurrences de x dans E, on ne réduira X qu'une seule fois, et l'inconvénient que si x n'a pas d'occurrence dans E, on aura réduit X pour rien, y-compris si la réduction de X ne converge pas ; ce n'est pas une réduction à gauche, mais elle est très largement adoptée dans les langages de programmation !
- réduire d'abord $(\lambda xyz.E\ X)$, puis $(\lambda yz.E_{[X/x]}\ Y)$, etc. C'est la stratégie paresseuse, qui a les avantages et inconvénients duaux de la première. On arrive à mitiger les inconvénients en représentant les λ-termes par des graphes qui partagent les occurrences de variables. C'est la stratégie d'un langage comme Haskell. Cette stratégie est liée à la curryfication qui considère une fonction à n paramètres comme une fonction à 1 paramètre qui retourne une fonction à 1 paramètre, etc.

Expressivité

Le λ-calcul a été conçu initialement pour résoudre des difficultés logiques de la théorie des ensembles, et ça n'a pas réussi. Cependant, en « jouant » avec les λ-termes, Church et ses collaborateurs d'alors, Rosser et Kleene, se sont progressivement rendu compte qu'ils arrivaient à coder de plus en plus de domaines de valeurs et de fonctions avec des λ-termes.

Voici quelques exemples du codage en λ-termes de valeurs ou opérateurs usuels. Ces codages sont souvent appelés codage de Church.

- $\mathrm{ID} = \lambda x.x$ représente la fonction identité.
- $\mathrm{T} = \lambda xy.x$ et $\mathrm{F} = \lambda xy.y$ représentent les valeurs de vérité True et False. De cette façon $\mathrm{IF} = \lambda bxy.(b\ x\ y)$ ($\equiv_\eta$ ID) représente une conditionnelle qui retourne x ou y selon que b est T ou F, et $\mathrm{NOT} = \lambda bxy.(b\ y\ x)$ et $\mathrm{AND} = \lambda b_1b_2.(\mathrm{IF}\ b_1\ b_2\ \mathrm{F})$ ($\equiv_\beta \lambda b_1b_2.(b_1\ b_2\ \mathrm{F})$) représentent les connecteurs logiques ¬ et ∧.
- $0 = \lambda sz.z$ ($\equiv_\alpha$ F), $1 = \lambda sz.(s\ z)$, $2 = \lambda sz.(s\ (s\ z))$ et $n = \lambda sz.(s^n\ z)$ représentent les entiers. Le λ-calcul compte avec des bâtons, les s. Ajouter 1 revient à ajouter un bâton ; $\mathrm{INCR} = \lambda nsz.(n\ s\ (s\ z))$ l'ajoute à la fin. $\mathrm{FOR} = \lambda nfi.(n\ f\ i)$ ($\equiv_\eta$ ID) itère f n fois à partir d'une valeur initiale i. Ex. $\mathrm{ADD} = \lambda n_1n_2.(\mathrm{FOR}\ n_1\ \ \mathrm{INCR}\ n_2)$ ($\equiv_\beta \lambda n_1.(n_1\ \mathrm{INCR})$) et $\mathrm{MUL} = \lambda n_1n_2.(\mathrm{FOR}\ n_1\ (\mathrm{ADD}\ n_2)\ 0)$. Et $\mathrm{EQZ} = \lambda n.(n\ (\lambda\ x.\ \mathrm{F})\ \mathrm{T})$ représente le test à 0.
- Retirer 1 à un entier a représenté un challenge pour les premiers explorateurs du λ-calcul. Une des façons d'y arriver est de passer par la représentation des constructeurs de donnée. $\mathrm{CONS} = \lambda xyc.(c\ x\ y)$ représente la paire (x, y), et les combinateurs $\pi_1 = \lambda p.(p\ \mathrm{T})$ et $\pi_2 = \lambda p.(p\ \mathrm{F})$ représentent ses deux projections.

$\mathrm{DECR} = \lambda n.(\pi_1\ (\lambda f.\ (n\ f\ (\mathrm{CONS}\ 0\ 0))\ \ \lambda p.(\mathrm{CONS}\ (\pi_2\ p)\ (\mathrm{INCR}\ (\pi_2\ p)))))$ représente alors la décrémentation. DECR applique n fois à la paire (0, 0) la fonction f qui transforme la paire (x, y) en $(y, y{+}1)$, produisant ainsi $(n{-}1, n)$, puis projette la partie gauche de la paire résultante.

- $\mathrm{Y} = \lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x)))$ est un combinateur de point-fixe. Il a la propriété que $(\mathrm{Y\ F}) \equiv_\beta (\mathrm{F}\ (\mathrm{Y\ F}))$, c-à-d. Y calcule un point fixe de F.

Ex. $\mathrm{FAC} = (\mathrm{Y}\ \ \lambda f.\ \lambda n.(\mathrm{IF}\ (\mathrm{EQZ}\ n)\ 1\ (\mathrm{MUL}\ n\ (f\ (\mathrm{DECR}\ n)))))$ est une définition récursive de la fonction factorielle. Remarquer que Y occupe essentiellement la position du letrec de la programmation fonctionnelle.

Finalement, en 1936 Church publie que tout ce qui peut être calculé peut l'être dans le λ-calcul, presque en même temps que Turing publie que ses machines calculent tout ce qui peut être calculé. Très vite, ils démontrent que les deux formalismes sont équivalents, et postulent que machines de Turing et λ-calcul identifient une notion absolue de ce qui peut être calculé. C'est la Thèse de Church-Turing.

Le théorème de Böhm

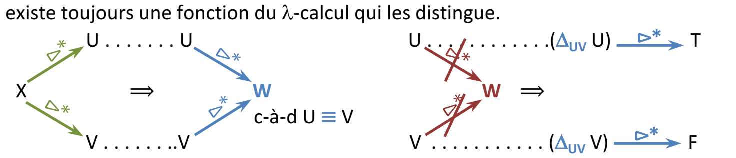
Les combinateurs présentés page 4 (section Expressivité) suggèrent de nombreuses variantes. Ex. $\mathrm{INCR'} = \lambda nsz.\ (s\ (n\ s\ z))$ qui ajoute un bâton au début de la représentation d'un entier plutôt qu'à la fin, ou $\mathrm{ADD'} = \lambda\ n_1n_2.\ (n_2\ \mathrm{INCR}\ n_1)$. On peut alors se demander si on peut étendre la β-équivalence en posant que ces variantes sont équivalentes, un peu comme on l'a fait en ajoutant la η-réduction pour rendre équivalents des λ-termes qui se comportaient partout pareil. Le théorème de Böhm répond par la négative et de façon spectaculaire.

Böhm s'est demandé si étant donnés deux λ-termes U et V, normaux et distincts, il est toujours possible de construire un λ-terme Δ_{UV} qui les discrimine : c-à-d. tel que $(\Delta_{UV}\ \mathrm{U}) \equiv \mathrm{T}$ et $(\Delta_{UV}\ \mathrm{V}) \equiv \mathrm{F}$. On dit que Δ_{UV} sépare U et V. Ex. EQZ sépare 0 de n'importe quel autre entier de Church. Böhm (1968) montre que deux λ-termes β-normaux séparables ne peuvent être η-équivalents, et réciproquement. Il n'y a pas deux λ-termes normaux pas η-équivalents et pas séparables.

Ce théorème montre que, parce qu'ils sont séparables, INCR et INCR', ou ADD et ADD', ex. $\Delta_{\mathrm{INCR\ INCR'}} = \lambda i.(i\ \lambda uv.\mathrm{T}\ \lambda w.\mathrm{F}\ i)$, ne peuvent pas être posés comme équivalents. Le faire rendrait T et F équivalents par passage au contexte du nouvel axiome, et par conséquence n'importe quels λ-termes X et Y équivalents. En effet, on aurait $(\mathrm{T\ X\ Y}) \equiv (\mathrm{F\ X\ Y})$, $(\mathrm{T\ X\ Y}) \rhd \mathrm{X}$ et $(\mathrm{F\ X\ Y}) \rhd \mathrm{Y}$, donc $\mathrm{X} \equiv \mathrm{Y}$.

En fait, les combinateurs INCR et INCR', ou ADD et ADD', produisent des résultats équivalents quand ils sont appliqués à des entiers de Church, mais ils se comportent différemment pour d'autres λ-termes. Au contraire, deux λ-termes η-équivalents se comportent vraiment partout pareil.

On peut voir la propriété de séparabilité comme le dual de la propriété de confluence. La confluence dit que d'un même λ-terme des réductions différentes peuvent toujours confluer, et si convergence, converger vers la même forme normale, alors que la séparabilité dit que pour deux formes normales distinctes il existe toujours une fonction du λ-calcul qui les distingue.



Plus fondamentalement, le Théorème de Böhm démontre une forme de maximalité de la théorie du λ-calcul.

Les variantes typées

Devant l'échec du λ-calcul comme réponse au paradoxe de Russell, Curry a eu l'idée de forcer une stratification de la logique en introduisant des types qui déterminent si un λ-terme est une fonction, une fonction de fonction, etc. Il y eut de nombreuses variantes, et celle qui est décrite ici est plutôt celle de Church.

Le langage des types simples est le suivant :

$$\mathrm{T} ::= \mathrm{E} \quad \text{(les types élémentaires)}$$
$$\mid (\mathrm{T} \longrightarrow \mathrm{T}) \quad \text{(les types fonctionnels)}$$

Ex. si ι et o sont des types élémentaires, ι, $(\iota \longrightarrow (\iota \longrightarrow o))$ et $((\iota \longrightarrow o) \longrightarrow o)$ sont des types. On convient d'omettre les parenthèses externes et les plus à droite, ex. $\iota \longrightarrow \iota \longrightarrow o$ et $(\iota \longrightarrow o) \longrightarrow o$. On note des types génériques par des minuscules grecques éventuellement indicées ou primées, ex. τ', et des types élémentaires particuliers par des minuscules romaines, ex. ι et o.

Le langage des λ-termes simplement typés est le suivant : pour tous types τ et τ',

$$\Lambda_\tau ::= V_\tau \quad \text{(les variables)}$$
$$\mid (\Lambda_{\tau'\longrightarrow\tau}\ \Lambda_{\tau'}) \quad \text{(les applications)}$$
$$\Lambda_{\tau'\longrightarrow\tau} ::= \lambda V_{\tau'}.\ \Lambda_\tau \quad \text{(les abstractions)}$$

Cette formalisation capture assez bien la façon dont fonctionnent les types en programmation. Cependant, elle ne permet pas de caractériser les codages présentés page 4 (section Expressivité). Par exemple, un λ-terme qui représente des entiers peut aussi bien être de type $(\iota\!\longrightarrow\!\iota) \longrightarrow \iota \longrightarrow \iota$ que $((\iota\!\longrightarrow\! o) \longrightarrow \iota \longrightarrow o) \longrightarrow (\iota\!\longrightarrow\! o) \longrightarrow \iota \longrightarrow o$. Pire encore, il n'est pas possible de définir de combinateur de point fixe dans ce cadre ; exit les fonctions récursives.

La réponse pragmatique des langages de programmation est d'enrichir le langage d'un jeu de constantes typées qui représentent les valeurs et les opérations du domaine de calcul. Une réponse plus fondamentale est proposée par le calcul des constructions. Le Cube de Barendregt propose une présentation harmonieuse, c-à-d. dans un cadre unique, de ramifications du système des types simples qui se combinent et culminent avec le calcul des constructions.

Des langages de programmation comme ML ou Scala offrent la possibilité de types polymorphes, ex. $\forall\ \alpha.\ \alpha \longrightarrow \alpha$, et de les inférer. Cela dispense de les écrire dans les programmes. La notion d'isomorphisme de type permet d'abstraire les types dans une direction extensionnelle en observant que certains types sont inter-convertibles par des fonctions simples. Ex. une fonction qui prend en paramètre une paire peut aussi bien être vue comme une fonction qui prend 2 paramètres, et vice-versa. Cela permet de s'affranchir de détails de codage fortuits.

Genèse : une débâcle productive

Le paradoxe de Russell montrait une faille dans la construction de la théorie naïve des ensembles de Frege. Russell lui-même proposa des variantes stratifiées où les ensembles sont ordonnés en classes, et où les éléments d'un ensemble sont toujours d'une classe inférieure à celle de l'ensemble. De cette façon, la question de savoir si un ensemble d'ensembles se contient ne se pose même plus, mais la logique obtenue est trop contrainte pour être utilisable.

Church recherchait une réponse en changeant de langage, et il développa le λ-calcul dans ce but. Aux notations et règles de calcul de la page 2 (section Définition), Church avait rajouté que les λ-termes devaient s'interpréter logiquement, et que la β-équivalence préservait les valeurs de vérité. La β-réduction devenait une forme de déduction.

À cet effet, les symboles ¬, ⟹ et ∃ était prédéfinis dans le système. Ex., un λ-terme (A B) pouvait signifier que B est un A, et alors un λ-terme (¬ A B) pouvait signifier que B n'est pas un A. Rosser et Kleene observent que le λ-calcul permet de former $k = \lambda x.(\neg\ x\ x)$, alors que $(k\ k) \equiv_\beta (\neg\ k\ k)$, ce qui est inconsistant. À nouveau, un manque de stratification permet d'écrire des énoncés autoréférentiels qui conduisent à des paradoxes. Rosser parle d'une débâcle !

Cependant, une relation réussie entre calcul et logique s'établira très progressivement avec la correspondance de Curry-Howard. Celle-ci s'entend dans les λ-calcul typés, et met en vis-à-vis les types et les formules logiques d'une part, et les λ-termes et les preuves d'autre part. Selon cette correspondance, si le λ-terme T est de type τ, c'est qu'il peut être interprété comme une preuve de la formule logique qui correspond à τ. Le λ-terme T est en fait la preuve que le type τ est habité. Ex. la correspondance entre le *modus ponens* et la règle de typage des applications :

$$\frac{A \quad A \Longrightarrow B}{B}\ (\textit{modus ponens}) \qquad \frac{X : \alpha \quad F : \alpha \longrightarrow \beta}{(F\ X) : \beta}$$

Cette correspondance a été reconnue progressivement pour des théories de types de plus en plus sophistiquées : les types simples dans les années 1930, qui correspondent à un calcul des propositions intuitionnistes, jusque dans les années 1960–70 avec des théories de types qui correspondaient à un calcul des prédicats, puis à une publication par William Howard en 1980 qui rendra cette relation très populaire. De cette correspondance sont issus des environnements de preuve qui permettent de certifier des programmes : ex. le système Coq.

Conclusion

Le λ-calcul constitue un domaine d'étude qui va de la logique aux langages de programmation. Il présente de nombreuses variantes : avec ou sans type, et si avec, quels types ; avec ou sans constantes, et si avec, lesquelles ; les stratégies de réduction ; leur mise en œuvre algorithmique ; leur compilation, etc. Il présente même des variantes plus orientées vers la modélisation des systèmes communicants. Il traverse ainsi une bonne partie de la science informatique.

Il est présent dans de nombreux langages de programmation. On peut même penser qu'il reste des traits du λ-calcul qui « attendent » toujours d'être incorporés dans les langages de programmation, en particulier les systèmes de types de haut niveau qui permettent d'exprimer des traits de polymorphisme ou de paramétricité qui manquent ou sont largement améliorables dans les langages de programmation.

La formalisation de la logique moderne au début du XX^e siècle demandait le développement d'une notion formelle de calcul. Cela a conduit à des formalismes comme celui des machines de Turing ou celui des fonctions récursives de Gödel. Le développement du λ-calcul semble plus relever du « pas de côté » (lire Bruno Latour sur les pas de côté) puisque le λ-calcul a d'abord été conçu comme un formalisme logique qui éviterait les écueils de la théorie naïve des ensembles, mais a commencé par échouer dans ce rôle avant d'y arriver bien plus tard.

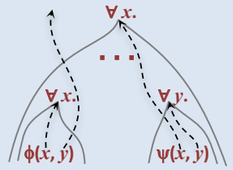
Bibliographie

- « *Highlights on the History of the Lambda-Calculus* », J. Barkley Rosser (*Annals of the History of Computing*, 1984). Contient des réflexions sur la preuve du théorème de Church-Rosser, et une description très vivante des instants où Church, Rosser et Kleene découvraient l'expressivité du λ-calcul.
- « *The Lambda Calculus: Its Syntax and Semantics* », Henk Barendregt (North Holland, 1984). Un classique de la présentation du λ-calcul.
- « Logique, réduction, résolution », René Lalement (Masson, 1990). Une somme sur la logique et le calcul.
- « *Introduction to generalized type systems* », Henk Barendregt (*Journal of Functional Programming*, 1991). Une reconstruction élégante de plusieurs tentatives pour doter le λ-calcul de systèmes de types sophistiqués.
- « *Isomorphisms of Types: from λ-calculus to information retrieval and language design* », Roberto Di Cosmo (Birkhaüser, 1995). Une belle présentation du concept d'isomorphisme de type.
- « La machine en logique », Pierre Wagner (PUF, 1998). Une présentation précise et philosophique de la correspondance de Curry-Howard présenté ici comme la définition formelle de l'intelligence artificielle, celle des machines.

Mathématiques et éléments de contexte : mathématiques et éléments de contexte — lire, apprendre, comprendre.
Pliage : recto visible (autre face), traits **gris** rentrants et traits **rouges** saillants. Découper selon le trait rouge entre les deux ●, puis achever le pliage.

① Les quantificateurs et leur portée

Le concept de **quantificateur** évoque le plus souvent celui des quantificateurs du calcul des prédicats, ∀ et ∃. Au delà de leur sens **logique**, ces quantificateurs ont un sens purement **syntaxique**. Ils **lient** des variables et leur donnent une **portée**. Dans une formule $\forall x.\ \phi$ ou $\exists x.\ \phi$, on dit que le quantificateur **lie** la variable x dans la sous-formule ϕ. Cela signifie que les occurrences libres (c-à-d. pas sous une autre quantification) de x dans ϕ sont indépendantes d'autres occurrences qui seraient liées par un autre quantificateur.



En fait, plutôt que de dire que ∀ lie x dans ϕ, on ferait mieux de dire que x est le nom d'une instance du quantificateur ∀, et que les occurrences de x dans ϕ désignent l'instance du quantificateur qui les lie (flèches pointillées dans le diagramme). De cette façon, une variable désigne simplement la première instance de quantification englobante qui porte le même nom, d'où une notion de **portée**. Et tout autre moyen qu'un nom, ex. un rang, rendrait ce service de désigner une instance englobante particulière.

Les noms des variables liées par un quantificateur sont tellement anecdotiques qu'on peut les changer arbitrairement, mais dans la limite de ne pas confondre les instances de quantificateur. Le renommage est formalisé par une substitution, notée $[y/x]$, qui exprime que les occurrences libres de x peuvent être remplacées par y, mais seulement si ça ne cause pas une liaison accidentelle de y. Ex. $\forall x.\ \exists y.\ x + y = 0$ peut subir la substitution $[z/x]$ mais pas la substitution $[y/x]$ car l'occurrence de x dans $x + y$ se trouverait liée par la quantification de y.

Les quantifications logiques ne sont absolument pas les seules à montrer ce comportement. Les sommations, ex. $\Sigma_{i\in[1,\ n]}\ x_i$, les produits, ex. $\Pi_{j\in[1,\ n]}\ x_j$, les intégrales, ex. $\int f(x).\ dx$, les dérivées, ex. $\partial\ f(y)\ /\ \partial y$, les notations d'ensemble en compréhension, ex. $\{\ 2\times n \mid n \in \mathbb{N}\ \}$, ont toutes le même type de comportement. Ce sont toutes des quantifications qui lient respectivement i, j, x, y et n. Les déclarations des paramètres de fonction dans les langages de programmation se comportent elles-aussi comme des quantifications ; les déclarations tout court, les boucles FOR aussi. Dans tous les cas, le sens d'une variable de programme est déterminé par **qui la quantifie**. Les requêtes **SQL** aussi montrent ces phénomènes, et ça se manifeste de façon critique dans la construction des **requêtes imbriquées**.

Toutes les quantifications introduisent des noms, les variables sont en fait des noms de quantification, et on peut les renommer en leur appliquant une substitution à condition de ne pas utiliser le nom d'une quantification qui englobe les occurrences de la variable à renommer. En appliquant une substitution $[E/x]$ (qui généralise une substitution $[y/x]$) dans un contexte K, soit $K_{[E/x]}$, aucune occurrence de variable libre dans E ne doit être liée accidentellement par une quantification qui englobe une occurrence de x dans K. On ne peut pas changer les noms des occurrences de variables libres de E car ils référencent des quantifications qu'on ne voit pas. Par contre, on peut toujours changer les noms des quantification englobant x dans K.

Cela donne l'idée de scinder une quantification en d'une part son rôle strictement syntaxique, le même pour toutes les quantifications imaginables, et son rôle sémantique, très différent d'une quantification à une autre. Il faut disposer pour cela d'une quantification neutre, qu'on pourrait noter λ, comme un **individu** λ, et de symboles distincts pour chaque sorte de quantification. On écrirait alors $\forall(\lambda x.\ \phi)$, $\exists(\lambda x.\ \phi)$, $\int(\lambda x.\ f(x))$, $\{\}(\lambda n.\ (2\times n \mid n \in \mathbb{N}))$, $\partial(\lambda y.\ f(y))$ et $\Sigma(\lambda i.\ (x_i \mid i \in [1,\ n]))$. C'est d'ailleurs ce qu'on ferait en **Python** si on n'y disposait pas déjà de listes en compréhension, ex. **`[2*x for x in range(10)]`** ; on écrirait **`list(map(lambda x: 2*x, range(10)))`**.

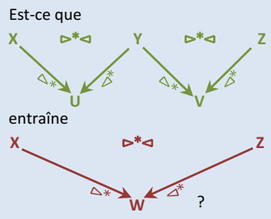
Conclusion : les quantifications sont beaucoup plus variées que les seules quantifications du calcul des prédicats, mais présentent toutes un aspect syntaxique commun, celui de lier des variables à des contextes nommés. Les noms des contextes n'importent pas, du moment qu'on ne crée pas de confusion entre contextes. On peut donc imaginer une quantification neutre, $\lambda x.\ \ldots$, sans sémantique particulière si ce n'est de créer un contexte, avec pour seule opération la possibilité de substituer quelque chose aux variables liées, $[E/x]$, en respectant les contraintes des contextes.

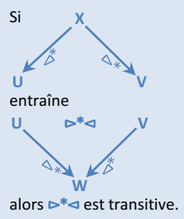
② Réduction, passage au contexte, équivalence, confluence, convergence

Considérons Λ un langage d'arbres finis, soit sous forme graphique, soit sous forme textuelle non-ambiguë, ex. le langage des **expressions arithmétiques**. On appellera **schéma d'arbre** une notation d'arbre comportant des **méta-variables** qui représentent n'importe quel arbre qu'il serait légal de placer là. Les arbres représentés par un schéma sont appelés les **instances** du schéma d'arbre. Ex. $A + B$ est le schéma des expressions arithmétiques qui sont des sommes, et $2 + 3$ est une instance de ce schéma. Dans un schéma les multiples occurrences d'une même méta-variable sont significatives de ce que toutes ces occurrences représentent strictement le même arbre (au contraire des multiples occurrences d'un même non-terminal dans une règle de grammaire algébrique qui n'indiquent pas autant d'arbres identiques, mais autant d'arbres de même catégorie syntaxique). Ex. $A + A$ est le schéma des expressions arithmétiques qui sont des sommes d'une expression et d'elle-même. Par extension, un arbre est un schéma d'arbre qui ne comporte pas de méta-variable. On appellera **contexte** d'une ou plusieurs occurrences dans un arbre cet arbre où les **positions** de ces occurrences ont été relevées. On notera $K_{[A]}$ un contexte qui contient une ou plusieurs occurrences de A où A peut être un schéma d'arbre : ex. $K_{[A+B]}$ serait le contexte des sommes dans une expression. Cette notation est ambiguë, ex. si il a des sommes dans des sommes, et on devra préciser le cas échéant les occurrences qui sont visées. On notera $K_{[B/A]}$ le contexte $K_{[A]}$ où on a remplacé les occurrences de A par autant de B.

On définit des **règles de réduction**, c-à-d. des règles qui expriment le remplacement d'un arbre, le **rédex**, par un autre, le **réduit**. Une règle de réduction sera notée **schéma de rédex ⊳ schéma de réduit** où toutes les méta-variables du schéma de réduit doivent être des méta-variables du schéma de rédex ; c'est donc en fait un schéma de règle. Comme pour les schémas d'arbre, les multiples occurrences d'une même méta-variable dans un schéma de règle représente strictement le même arbre, ex. $A + A \rhd 2 \times A$ qui exprime une sorte de factorisation. On peut assortir une règle de réduction de conditions annexes pour exprimer des contraintes d'application, ex. $A + B \rhd B + A$, si $||A|| > ||B||$ où $||.||$ représente le nombre de nœuds d'un arbre.

Finalement, on définit une **relation de réduction** entre 2 arbres X et Y, notée $X \rhd Y$, par **passage au contexte** (on dit aussi **par congruence**) : $X \rhd Y$ ssi $X = K_{[U]}$ et $Y = K_{[V/U]}$ et $U \rhd V$ est une instance d'une règle de réduction et U représente une occurrence unique dans $K_{[U]}$. Ex. (1) $(5 + 5) + (5 + 5) \rhd (5 + 5) + (2 \times 5) \rhd (2 \times 5) + (2 \times 5) \rhd 2 \times (2 \times 5)$, et (2) $(5 + 5) + (5 + 5) \rhd 2 \times (5 + 5) \rhd 2 \times (2 \times 5)$. On note $\rhd^*$ un enchaînement de réductions, y compris aucune (**fermeture réflexive-transitive**), ex. $(5 + 5) + (5 + 5) \rhd^* 2 \times (2 \times 5)$ et $2 \times 5 \rhd^* 2 \times 5$. Si il y a plusieurs règles de réduction, et en l'absence de précision, $\rhd$ désigne l'une d'entre elles.





On définit ensuite une relation qu'on voudrait être d'équivalence : $X \rhd^*\lhd Y$ ssi il existe U tel que $X \rhd^* U$ et $Y \rhd^* U$. Pour que ce soit une relation d'équivalence, il faudrait que ce soit une relation transitive, et cette définition ne le garantit pas. Ex. on peut avoir $X \rhd^* U$ et $Y \rhd^* U$, et $Y \rhd^* V$ et $Z \rhd^* V$, donc $X \rhd^*\lhd Y$ et $Y \rhd^*\lhd Z$, mais si on n'a pas $U \rhd^*\lhd V$ on risque de ne pas avoir $X \rhd^*\lhd Z$. On appelle **confluence** la propriété que si $X \rhd^* U$ et $X \rhd^* V$ alors $U \rhd^*\lhd V$. La confluence est dite **faible** (on dit parfois locale) si les $\rhd^*$ partant de X sont remplacés par des $\rhd$. Cette propriété dépend du système de règles de réduction adopté, et elle est suffisante pour que $\rhd^*\lhd$ soit une relation d'équivalence, et on la notera alors ≡. On aurait pu construire une relation d'équivalence par fermeture **réflexive-transitive-symétrique** de $\rhd$, mais des règles de réduction comme $0 \times A \rhd 0$ ne se prêtent pas à être lues symétriquement car toutes les méta-variables de ce que serait le réduit ne sont pas des méta-variables de ce que serait le rédex. En fait, la littérature offre beaucoup de variations sur ce thème : (1) définir une relation d'équivalence puis se demander si une relation de co-convergence la représente bien, (2) définir une relation de co-convergence puis se demander si elle est une relation d'équivalence, et de nombreuses variantes entre les deux.

On dit d'un arbre qui ne contient pas de rédex qu'il est **irréductible** ou **normal**, et on se demande alors si toutes les réductions convergent vers des arbres normaux ; c'est la propriété de **normalisation forte**. Cela dépend encore du système de règles de réduction. Ex. la règle de réduction $A + B \rhd B + A$ sans condition annexe permet d'entretenir indéfiniment une réduction qui ne converge pas. On peut alors se demander si pour chaque arbre au moins une réduction converge, c'est la **normalisation faible**. Le fait que des réductions convergent et d'autres non mène à se demander si des stratégies permettent de toujours trouver une convergence si elle existe, et à quel prix.

Il est courant de donner des noms aux règles de réduction et aux systèmes qu'elles forment. On pourra alors en préfixer les mots **réduction**, **équivalence**, **normal**, etc. et en indicer les symboles $\rhd$, $\rhd^*$, ≡, etc. : ex. **ξ-réduction** pour réduction selon la règle ou le système ξ, ou $\equiv_\xi$ pour équivalence selon la règle ou le système ξ.

③ Le paradoxe de Russell

Gottlob Frege publie en 1879 (puis 1884, 1893 et 1903) la première présentation d'une logique avec **quantificateurs**. Il propose au passage une première **théorie des ensembles** qui fondent les quantifications. Mais cet édifice est sapé par Bertrand Russell qui présente un paradoxe qui rend cette logique **inconsistante**.

Le paradoxe repose sur la possibilité d'un **ensemble de tous les ensembles** et de caractériser des ensembles d'ensembles en **compréhension**. L'ensemble de tous les ensembles est forcément un élément de lui-même. Mais tous les ensembles ne sont pas éléments d'eux-mêmes. Ex. l'ensemble des entiers n'est pas un entier. Le prédicat « s'appartenir » partitionne donc l'ensemble de tous les ensembles en deux sous-ensembles. Question : que dire du sous-ensemble des ensembles qui ne s'appartiennent pas ? S'appartient-il, ou non ?

> *Si il s'appartient, il n'est pas un ensemble qui ne s'appartient pas, donc il ne s'appartient pas. Si il ne s'appartient pas, il appartient à l'ensemble des ensembles qui ne s'appartiennent pas, donc à lui-même. D'où une* ***inconsistance****.*

On perçoit facilement que le paradoxe vient de la trop grande facilité avec laquelle on a formé des ensembles d'ensembles en utilisant le **principe de compréhension**. Ce principe dit qu'il suffit qu'un prédicat soit défini sur un ensemble pour que cela caractérise deux sous-ensembles, celui des éléments pour qui le prédicat est vrai, et les autres. D'où de nombreuses tentatives de restreindre cette faculté, en particulier en stratifiant le langage des ensembles, des ensembles d'ensembles, etc.

④ Gottlob, Bertrand, Haskell, et les autres

Gottlob Frege (1848—1925, Allemagne) : depuis son ***Begriffschrift*** (1879), ***Die Grundlagen der Arithmetik*** (1884), jusqu'à ***Die Grundgesetze der Arithmetik*** (1893—1903), Frege consacre sa carrière à une première présentation d'un calcul des prédicats, son **idéographie**, et d'une théorie des ensembles.

Bertrand Russel (1872—1970, Royaume-Unis) : on lui doit directement peu de chose, mais il a été un des grands animateurs des travaux de fondation de la logique moderne, ex. dans son travail avec **Alfred North Whitehead**. Il est aussi **Prix Nobel de Littérature** et un **pacifiste**.

Haskell Curry (1900—1982, États-Unis) : co-inventeur, avec **Moses Schönfinkel**, du modèle de calcul des **combinateurs**. Ce calcul peut être entièrement simulé par le λ-calcul, et vice-versa.

Alonzo Church (1903—1995, États-Unis) : inventeur du λ-calcul pour contrer les inconsistance que Russell, et d'autres, avait exhibées dans la théorie des ensembles de Frege, puis découvreur avec Kleene et Rosser de la puissance calculatoire insoupçonnée de ce calcul.

J. Barkley Rosser (1907—1989, États-Unis) : étudiant puis collaborateur de Church à l'occasion des premiers travaux sur le λ-calcul.

Stephen Cole Kleene (1909—1994, États-Unis) : étudiant de Church à l'occasion des premiers travaux sur le λ-calcul. On lui doit des présentations consolidées de beaucoup des systèmes utilisés aujourd'hui en informatique : ex. en **théorie des langages** ou en théorie de la **calculabilité**.

Corrado Böhm (1923—2017, Italie) : grand explorateur des capacités du λ-calcul en programmation, compilation, etc.

Henk Barendregt (1947, Pays-Bas) : auteur d'une présentation consolidée du λ-calcul qui est une bible du domaine, ainsi que d'une présentation harmonisée des systèmes de λ-calcul typés qui convergent vers le calcul des constructions : le **λ-cube**, ou **Cube de Barendregt**.