3.2.2 Le langage pur (non typé)

Le lambda-calcul a été inventé par Alonzo Church [Church, 1941], pour représenter la notion de fonction mathématique, d'une façon qui évite les problèmes de l'approche basée sur la théorie des ensembles.[1]

3.2.2.1 Syntaxe

Le **vocabulaire** est composé d'un ensemble dénombrable de symboles de variables, de parenthèses, du point, et du symbole λ.

La **syntaxe** est définie par induction :

- Si x est une variable, alors x est un λ-*terme*.
- Si x est une variable, et t un terme, alors $\lambda x.t$ est un λ-*terme* (λ-abstraction)
- Si t_1 et t_2 sont des termes, alors $(t_1)t_2$ est un λ-*terme* (application).

Remarques

- L'application d'un terme φ à un terme ψ est notée $(\varphi)\psi$, à l'inverse de la notation traditionnelle $f(x)$.
- Il n'y a pas de distinction entre variables fonctionnelles et variables arguments. On peut donc former des termes de la forme $\lambda f.(f)f$ (application à soi-même).
- Comme les quantificateurs en logique du premier ordre, l'abstracteur λ est un lieur de variables. On peut définir par conséquent les notions de variables libres et liées : [2]
  (a) La **portée** d'un abstracteur λx dans le terme $\lambda x.\varphi$ est le terme φ.
  (b) Une occurrence d'une variable x dans le terme φ est dite **libre** si elle n'est pas dans la portée d'un abstracteur λx apparaissant dans φ.
  (c) Si $\lambda x.\varphi$ est un sous-terme de ψ et que x est libre dans φ, alors cette occurrence de x est dite **liée** par l'abstracteur λx.
  Exemple : dans le terme $((\lambda x.\lambda y.(x)y)z)x$, la variable x a deux occurrences, l'une libre et l'autre liée.
- La grammaire définissant les λ-termes est non ambigüe, il n'y a qu'une façon de décomposer un terme. On peut définir la notion de sous-terme. Les notations (point et parenthèses) pouvant s'avérer assez lourdes, on adopte les conventions :

$$\begin{aligned} \lambda x_1.\lambda x_2 \ldots \lambda x_n.t &= \lambda x_1 x_2 \ldots x_n.t \\ (\ldots((t)t_1)t_2\ldots)t_m &= t\,t_1 t_2 \ldots t_m \end{aligned}$$

  Exemple : $\lambda xy.xy$ se lit $\lambda x.\lambda y.(x)y$.
- L'application fonctionnelle est toujours monadique (s'applique à un argument). Toutes les fonctions sont "curryfiées" : une fonction à deux arguments est vue comme une fonction à un argument qui s'applique à un argument.

3.2.2.2 Conversions

Équivalence On définit la relation $\equiv$ entre termes. Intuitivement $t_1 \equiv t_2$ indique que t_1 et t_2 dénotent la même fonction.

Plus rigoureusement, on définit $\equiv$ comme
• une relation d'équivalence entre termes

[1] En particulier, le fait que la notation $3x^2+7$ est ambiguë entre la définition de la fonction qui à x associe $3x^2+7$ et l'application de la fonction à un x particulier.

[2] Version inductive (VL = variables libres) :
- $VL(x) = \{x\}$ (où x est une variable)
- $VL(\ (t_1)t_2\ ) = VL(t_1) \cup VL(t_2)$ (où t_1 et t_2 sont des termes)
- $VL(\lambda x.t) = VL(t) \setminus \{x\}$

- une congruence pour la λ-abstraction et l'application, i.e. :
  - $M \equiv N \Rightarrow \lambda x.M \equiv \lambda x.N$ pour toute variable x.
  - $M \equiv N \Rightarrow (M)P \equiv (N)P$ & $(P)M \equiv (P)N$ pour tout terme P.

α-conversion Il est naturel, comme on le fait pour les formules quantifiées en logique du premier ordre, de considérer des termes qui ne diffèrent que par le nom des variables liées comme équivalents. C'est l'α-équivalence :

$$\lambda x.\varphi \equiv \lambda z._{[x:=z]}\varphi$$

La notation $_{[x:=z]}\varphi$ signifie que toutes les occurrences liées de x dans φ sont renommées z. Il ne s'agit pas d'un λ-terme du langage, mais d'une abbréviation pour le processus de renommage des variables, que l'on peut définir par induction :

- $_{[x:=z]}x \rightsquigarrow z$
- $_{[x:=z]}y \rightsquigarrow y$ si $y \neq x$
- $_{[x:=z]}(M)N \rightsquigarrow (_{[x:=z]}M)_{[x:=z]}N$
- $_{[x:=z]}\lambda x.M \rightsquigarrow \lambda z._{[x:=z]}M$
- $_{[x:=z]}\lambda y.M \rightsquigarrow \lambda y._{[x:=z]}M$ si $x \neq y$

Convention sur les variables Soit M un terme, x une variable ; les occurrences de x dans M sont soit libres soit toutes liées. On montre aisément que tout terme construit sans respecter la convention sur les variables est équivalent, à une α-conversion près, à un terme la respectant.

Substitution de termes On introduit une "procédure" de substitution d'un terme à une variable, notée de façon analogue au renommage des variables, et définie inductivement comme suit (on prend les précautions d'usage concernant les variables libres ou liées : aucune variable libre dans le terme que l'on substitue ne doit devenir lié dans le terme résultant) :

- $_{[x:=t]}x \rightsquigarrow t$
- $_{[x:=t]}y \rightsquigarrow y$ si $y \neq x$
- $_{[x:=t]}(M)N \rightsquigarrow (_{[x:=t]}M)_{[x:=t]}N$
- $_{[x:=t]}\lambda y.M \rightsquigarrow \lambda y._{[x:=t]}M$ si y n'est pas libre dans t.

Ce sont les seuls cas de substitution admissibles.

Remarque Les substitutions (de terme à une variable) se comportent comme les affectations d'un langage impératif.

Exemples
- $_{[x:=\lambda x.x]}\lambda y.(x)y \rightsquigarrow \lambda y.(\lambda x.x)y$
- $_{[x:=\lambda xy.xy]}\lambda y.(xz)xy \rightsquigarrow \lambda y.((\lambda x.\lambda y.(x)y)z)(\lambda x.\lambda y.(x)y)y$
  (repasser par la forme entièrement parenthésée)

Beta-conversion (ou β-équivalence) :

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

(suivant la convention sur les variables, la substitution est toujours valide.)

La β-conversion est une règle de **calcul** (on parle de β-réduction lorsque l'on "réduit" $(\lambda x.M)N$ en $_{[x:=N]}M$) ; c'est même la **seule** règle primitive de calcul du λ-calcul (modulo l'α-conversion), et celle que nous étudierons le plus longuement.

Exemples
- $(\lambda x.x)y \equiv y$ ("identité")
- $(\lambda xy.x)zt \equiv z$ ("projection")
- $(\lambda x.xx)y \equiv yy$ ("application à soi-même")
  d'où $(\lambda x.xx)\lambda x.xx \equiv (\lambda x.xx)\lambda x.xx$: la β-réduction ne réduit pas nécessairement la taille du terme.

Remarques
- On appelle **(β)-redex** un terme de la forme $(\lambda x.M)N$, et **contractum** (ou forme contractée) le terme ${}_{[x:=N]}M$.
- Un terme est dit **en forme normale** s'il ne contient pas de redex comme sous-terme. Par exemple $\lambda z.(z)x$, et pas $(\lambda z.z)x$.
- Théorème : il existe des redex qui n'ont pas de forme normale. Par exemple $(\lambda x.xx)\lambda x.xx$.
- Théorème (existence d'un point fixe) : $\forall M \exists N$ t.q. $MN \equiv N$.

  Démonstration Soit $W = \lambda x.(M)xx$ et $N = WW$, $x \notin VL(M)$. Alors
  $N = (\lambda x.(M)xx)W = (M)WW = MN$) □

3.2.2.3 Combinateurs

Le λ-calcul, tel qu'il a été présenté, est extrêmement abstrait, et, à ce titre, peu maniable ; on va s'intéresser ici à des « constructeurs » dérivés.

Définition Un **combinateur** est un λ-terme clos, i.e. sans variable libre.

Remarque : un combinateur se comporte de façon uniforme (indépendante du contexte), et ainsi peut être assimilié à une *constante* du langage. À noter qu'un combinateur est défini (comme tout λ-terme) à une α-conversion près.

Quelques combinateurs et leurs propriétés

Identité $\mathsf{I} =_{\text{def}} \lambda x.x$
Pour tout terme t, on a $(\mathsf{I})t \equiv t$.

Entiers Il existe plusieurs façons de représenter les entiers par des λ-termes ; la plus naturelle est celle de Church, où les entiers sont conçus comme des itérateurs :
$\mathsf{0} =_{\text{def}} \lambda f.\lambda x.x$
$\mathsf{1} =_{\text{def}} \lambda f.\lambda x.(f)x$
$\mathsf{n} =_{\text{def}} \lambda f.\lambda x.(f)(f)\ldots(f)x$, avec f n fois.
On peut alors représenter la fonction successeur, l'addition et la multiplication :
$\mathsf{Succ} =_{\text{def}} \lambda n.\lambda f.\lambda x.(f)((n)f)x$
$+ \equiv \lambda m.\lambda n.\lambda f.\lambda x.((m)f)((n)f)x$
$* \equiv \lambda m.\lambda n.\lambda f.(m)(n)f$

Booléens $\mathsf{T} =_{\text{def}} \lambda x.\lambda y.x$
$\mathsf{F} =_{\text{def}} \lambda x.\lambda y.y$
Ces définitions, conventionnelles, s'expliquent par la forme très simple que reçoit alors la définition par cas : $\texttt{if}\ P\ \texttt{then}\ Q\ \texttt{else}\ R =_{\text{def}} PQR$.
En effet, si P se réduit en T (ie $P \equiv \mathsf{T}$), alors $\texttt{if}\ P\ \texttt{then}\ Q\ \texttt{else}\ R \equiv \mathsf{T}QR \equiv Q$. De même, si $P \equiv \mathsf{F}$, alors on obtient R.
On peut alors définir relativement aisément les combinateurs booléens, en passant par la définition précédente : $\neg P = \texttt{if}\ P\ \texttt{then}\ \mathsf{F}\ \texttt{else}\ \mathsf{T} = P\mathsf{FT}$ (ou $((P)\mathsf{F})\mathsf{T}$). D'où par λ-abstraction : $\mathsf{NOT} =_{\text{def}} \lambda x.((x)\mathsf{F})\mathsf{T}$.
Alors, on peut vérifier que si $P \equiv \mathsf{T}$, alors $\mathsf{NOT}P \equiv ((P)\mathsf{F})\mathsf{T} \equiv ((\mathsf{T})\mathsf{F})\mathsf{T} \equiv \mathsf{F}$.
En exercice (p. 18), on peut définir les opérateurs $\wedge$, $\vee$, et vérifier les propriétés usuelles.