Lambda Calcul

II. Arithmétique

Marc Lorenzi - 26 mai 2018

In [1]:
```
from lambda_parser import parse
from lambda_terms import *
```

Nous allons voir dans ce notebook comment on peut modéliser l'arithmétique des entiers naturels **à l'intérieur du λ-calcul**.

On suppose que le lecteur a lu le notebook numéro I, qu'il maîtrise la syntaxe des termes, la notion de β-réduction et celle de forme normale. Tout ce que nous avons déjà vu est dans le module `lambda_terms` .

Très peu de code Python dans ce notebook. Nous allons ici **utiliser** les fonctions que nous avons écrite dans le premier notebook. Un bon mathématicien ne déroge jamais à la règle SRACQP : "se ramener à ce qui précède" :-).

1. Les entiers de Church

Étant donnés $n \in \mathbb{N}$ et deux termes M, N, nous allons noter $M^n N = M(M(\dots(MN)))$, où M apparaît n fois, avec la convention que $M^0 N = N$. Remarquons que $M^1 N = MN$. Remarquons aussi que pour tout $n \in \mathbb{N}$ et tout terme N, $M^{n+1} N = MT$ où $T = M^n N$. On peut facilement montrer que pour tous entiers n et p (récurrence), on a $M^{n+p} N = M^n (M^p N)$.

NB : Attention, $M \dots MN = ((\dots((MM)M)\dots M)N)$ et donc n'a rien à voir avec $M^n N$. Ah, ces opérations non associatives ...

Pour tout $n \in \mathbb{N}$, nous définissons maintenant le n-ième entier de Church, noté C_n :

Définition : $C_n = \lambda f.\, \lambda x.\, f^n x$.

On a donc $C_n M T \to_\beta^* M^n T$ pour tous termes M, T. En quoi les entiers de Church modélisent-ils les entiers naturels ? Eh bien si l'on arrive, par exemple, à définir un terme A tel que $A C_m C_n \to_\beta^* C_{m+n}$, l'ensemble des entiers de Church muni de l'"opération" d'addition sera en quelque sorte isomorphe à notre classique monoïde $(\mathbb{N}, +)$. Dit autrement :

Tout calcul dans $(\mathbb{N}, +)$ sera modélisé par des β-réductions dans l'ensemble des termes.

Avant tout, un petit échauffement.

Révision : pour tous termes A et B, on a $C_n A B \to_\beta A^n B$.

Démonstration : $C_n A B = (\lambda f.\, U) A B$ où $U = \lambda x.\, f^n x$. Ainsi,
$C_n A B \to_\beta U[f := A] B = (\lambda x.\, A^n x) B$. Encore un redex !
$(\lambda x.\, A^n x) B \to_\beta (A^n x)[x := B] = A^n B$, d'où le résultat.

La fonction `church` prend un entier n en paramètre et renvoie le terme $\overline{n}$.

In [2]:
```
def church(n):
    f = new_var('f')
    T = new_var('x')
    for k in range(n):
        T = new_app(f, T)
    T = new_abs('f', new_abs('x', T))
    return T
```

In [3]:
```
print_term(church(10))
```

```
λf.λx.f(f(f(f(f(f(f(f(f(fx)))))))))
```

2. L'addition

Soient f une fonction, x un objet, et $m, n \in \mathbb{N}$. On a la relation bien connue $f^{m+n}(x) = f^m(f^n(x))$. Supposons maintenant que toutes ces lettres sont des termes. Que devient l'égalité ? Eh bien, $C_{m+n} f x \to_\beta^* f^{m+n} x$. Et $C_m f (C_n f x) \to_\beta^* f^m (f^n x)$. Nous allons donc définir notre opérateur d'addition comme étant le terme $\lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, m f (n f x)$. Et advienne que pourra.

In [4]:
```
add_alias('ADD', '#m.#n.#f.#x.mf(nfx)')
```

Testons. Ça fait combien $3 + 2$? La fonction `parse` a été un peu boostée pour accepter les entiers naturels.

In [5]:
```
T = parse('ADD 3 2')
print_term(T)
```

```
ADD (λf.λx.f(f(fx)))(λf.λx.f(fx))
```

Calculons une forme normale de notre terme. La fonction `do` du module `lambda_terms`, appelle `parse`, puis `expand`, et enfin `normal_form`.

In [6]:
```
M = do('ADD 3 2')
```

```
0 (λm.λn.λf.λx.mf(nfx))(λf.λx.f(f(fx)))(λf.λx.f(fx))
1 (λn.λf.λx.(λf.λx.f(f(fx)))f(nfx))(λf.λx.f(fx))
2 λf.λx.(λf.λx.f(f(fx)))f((λf.λx.f(fx))fx)
3 λf.λx.(λx.f(f(fx)))((λf.λx.f(fx))fx)
4 λf.λx.f(f((λf.λx.f(fx))fx))
5 λf.λx.f(f(f((λx.f(fx))x)))
6 λf.λx.f(f(f(f(fx))))
--------------------
6 réductions
```

Comptez les `f` dans la réponse : il y en a 5.

Théorème 1 : $3 + 2 = 5$.

Un cas particulier de notre petit raisonnement ci-dessus est celui du calcul du successeur d'un entier. En prenant le second entier égal à 1, dans `ADD`, nous voilà gratuitement en possession d'une fonction qui calcule $n + 1$ à partir de n. Elle s'appelle la fonction *successeur*.

In [7]:
```
add_alias('SUCC', '#n.#f.#x.f(nfx)')
```

In [8]:
```
T = do('SUCC 4')
```

```
0 (λn.λf.λx.f(nfx))(λf.λx.f(f(f(fx))))
1 λf.λx.f((λf.λx.f(f(f(fx))))fx)
2 λf.λx.f((λx.f(f(f(fx))))x)
3 λf.λx.f(f(f(f(fx))))
--------------------
3 réductions
```

3. La multiplication

Même raisonnement que pour l'addition, ou peu s'en faut. On a $f^{mn}(x) = (f^n)^m(x)$. Passant aux termes et aux entiers de Church, il vient $C_{mn}fx \to_\beta^* f^{mn}x$ et $C_m(C_nf)x \to_\beta^* (f^n)^m(x)$. Ainsi, on meurt d'envie de poser pour l'opérateur de multiplication
$P = \lambda m.\,\lambda n.\,\lambda f.\,\lambda x.\,m(nf)x.$

In [9]:
```
add_alias('PROD', '#m.#n.#f.#x.m(nf)x')
```

In [10]:
```
N = do('PROD 3 4')
```

```
0 (λm.λn.λf.λx.m(nf)x)(λf.λx.f(f(fx)))(λf.λx.f(f(f(fx))))
1 (λn.λf.λx.(λf.λx.f(f(fx)))(nf)x)(λf.λx.f(f(f(fx))))
2 λf.λx.(λf.λx.f(f(fx)))((λf.λx.f(f(f(fx))))f)x
3 λf.λx.(λx.(λf.λx.f(f(f(fx))))f((λf.λx.f(f(f(fx))))f((λf.λx.f(f(f(f
x))))fx)))x
4 λf.λx.(λf.λx.f(f(f(fx))))f((λf.λx.f(f(f(fx))))f((λf.λx.f(f(f(fx)))
)fx))
5 λf.λx.(λx.f(f(f(fx))))((λf.λx.f(f(f(fx))))f((λf.λx.f(f(f(fx))))fx)
)
6 λf.λx.f(f(f(f((λf.λx.f(f(f(fx))))f((λf.λx.f(f(f(fx))))fx)))))
7 λf.λx.f(f(f(f((λx.f(f(f(fx))))((λf.λx.f(f(f(fx))))fx)))))
8 λf.λx.f(f(f(f(f(f(f(f((λf.λx.f(f(f(fx))))fx)))))))
9 λf.λx.f(f(f(f(f(f(f(f((λx.f(f(f(fx))))x)))))))
10 λf.λx.f(f(f(f(f(f(f(f(f(f(fx))))))))))
--------------------
10 réductions
```

Et en plus ça marche.

Théorème 2 : $3 \times 4 = 12$.

Vous en avez peut-être assez de compter les `f` ? Moi aussi. Voici une fonction qui le fera pour nous.

In [11]:
```
def unchurch(T):
    T = expr(expr(T))
    count = 0
    while is_app(T):
        count += 1
        T = right(T)
    return count
```

In [12]:
```
unchurch(N)
```

Out[12]:
```
12
```

Youpi.

4. Puissances

Plus rien ne peut nous arrêter ... $f^{m^n} = (((f^m)^m)\ldots)^m$, n fois. En termes de termes, si j'ose dire, $C_{m^n} f \to_\beta (((f^m)^m)\ldots)^m$ et $C_n C_m f$ aussi. Dit autrement, pour élever à la puissance m^n, on itère n fois la fonction "élévation à la puissance m". Bigre, d'habitude les paramètres de C_n sont une fonction f et un objet x. Oui, mais en λ-calcul, TOUT est fonction, même l'"entier" C_m, qui est justement l'élévation à la puissance m ! Finalement, l'opérateur d'exponentiation est $E = \lambda m.\, \lambda n.\, nm$. Tout simplement.

In [13]:
```
add_alias('POW', '#m.#n.nm')
```

In [14]:
```
T = do('POW 2 3')
```

```
0 (λm.λn.nm)(λf.λx.f(fx))(λf.λx.f(f(fx)))
1 (λn.n(λf.λx.f(fx)))(λf.λx.f(f(fx)))
2 (λf.λx.f(f(fx)))(λf.λx.f(fx))
3 λx.(λf.λx.f(fx))((λf.λx.f(fx))((λf.λx.f(fx))x))
4 λx.λv1.(λf.λx.f(fx))((λf.λx.f(fx))x)((λf.λx.f(fx))((λf.λx.f(fx))x)
v1)
5 λx.λv1.(λv1.(λf.λx.f(fx))x((λf.λx.f(fx))xv1))((λf.λx.f(fx))((λf.λx
.f(fx))x)v1)
6 λx.λv1.(λf.λx.f(fx))x((λf.λx.f(fx))x((λf.λx.f(fx))((λf.λx.f(fx))x)
v1))
7 λx.λv1.(λv1.x(xv1))((λf.λx.f(fx))x((λf.λx.f(fx))((λf.λx.f(fx))x)v1
))
8 λx.λv1.x(x((λf.λx.f(fx))x((λf.λx.f(fx))((λf.λx.f(fx))x)v1)))
9 λx.λv1.x(x((λv1.x(xv1))((λf.λx.f(fx))((λf.λx.f(fx))x)v1)))
10 λx.λv1.x(x(x(x((λf.λx.f(fx))((λf.λx.f(fx))x)v1))))
11 λx.λv1.x(x(x(x((λv1.(λf.λx.f(fx))x((λf.λx.f(fx))xv1))v1))))
12 λx.λv1.x(x(x(x((λf.λx.f(fx))x((λf.λx.f(fx))xv1)))))
13 λx.λv1.x(x(x(x((λv1.x(xv1))((λf.λx.f(fx))xv1)))))
14 λx.λv1.x(x(x(x(x(x((λf.λx.f(fx))xv1))))))
15 λx.λv1.x(x(x(x(x(x((λv1.x(xv1))v1))))))
16 λx.λv1.x(x(x(x(x(x(xv1))))))
--------------------
16 réductions
```

In [15]:
```
unchurch(T)
```

Out[15]: 8

Théorème 3 : $2^3 = 8$.

Aparté : L'une des règles de la substitution, celle qui est compliquée et évite la capture de variable libre, nous saute au visage. Regardez la variable v_1 dans le résultat ci-dessus ...

5. Prédécesseur, soustraction

5.1 Prédécesseur d'un entier

Là nous devons passer du côté obscur de la force. Additionner c'est facile, mais comment soustraire ? Nous allons montrer que l'on peut prendre comme opérateur de calcul du prédécesseur d'un entier de Church :

$$P = \lambda n.\, \lambda f.\, \lambda x.\, n(\lambda g.\, \lambda h.\, h(gf))(\lambda u.\, x)(\lambda u.\, u)$$

Euh oui mais le prédecesseur de 0 n'existe pas. Soyons précis : si $n \geq 1$, le prédécesseur de n est $n-1$. Et on convient que le "prédécesseur" de 0 est 0.

Théorème : pour tout $n \geq 0$, on a $PC_{n+1} \to_\beta^* C_n$.

Démonstration : Sautez la démo si vous voulez. C'est une démo démoniaque. Je vais prouver que $PC_{n+1}fx \to_\beta^* C_n fx$ pour rester lisible. Il "suffira" ensuite de rajouter des $\lambda f.\, \lambda x.$ (en admettant, je ne le prouverai pas ici, qu'on peut le faire).

Posons $F = \lambda g.\, \lambda h.\, h(gf)$, $\Phi = \lambda u.\, x$, et $I = \lambda u.\, u$. On a donc $PC_{n+1}fx \to_\beta^* F^{n+1}\Phi I$.

Montrons par récurrence sur n que pour tout $n \in \mathbb{N}$, pour tous termes T, U, on a $F^{n+1}TU \to_\beta^* U(f^n(Tf))$.

- Pour $n = 0$, on a $FTU \to_\beta^* U(Tf) = U(f^0(Tf))$, ce que l'on voulait.
- Soit $n \geq 1$. Supposons la propriété vraie pour l'entier $n-1$. On a $F^{n+1}TU =_\beta F(F^nT)U \to_\beta^* U(F^nTf)$. Par l'hypothèse de récurrence (avec $U = f$!), on a donc $F^{n+1}TU \to_\beta^* U(f(f^{n-1}(Tf))) = U(f^n(Tf))$, ce que l'on voulait.

Il ne reste plus qu'à prendre $T = \Phi$ et $U = I$:

$$PC_{n+1}fx \to_\beta^* C_{n+1}F\Phi I = F^{n+1}\Phi I \to_\beta^* I(f^n(\Phi f)) \to_\beta f^n(\Phi f) \to_\beta^* f^n x = C_n fx.$$

Bilan : vous n'avez rien compris. Mais je vous assure que c'est bon.

Exercice : Montrez que $PC_0 \to_\beta^* C_0$. Ça c'est jouable.

In [16]:
```
add_alias('PRED', '#n.#f.#x.n(#g.#h.h(gf))(#u.x)(#u.u)')
```

Testons.

In [17]:
```
T = do('PRED 6')
```

```
0 (λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))(λf.λx.f(f(f(f(f(fx))))))
1 λf.λx.(λf.λx.f(f(f(f(f(fx))))))(λg.λh.h(gf))(λu.x)(λu.u)
2 λf.λx.(λx.(λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf))(
(λg.λh.h(gf))((λg.λh.h(gf))x))))))(λu.x)(λu.u)
3 λf.λx.(λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf))((λg.
λh.h(gf))((λg.λh.h(gf))(λu.x))))))(λu.u)
4 λf.λx.(λh.h((λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf)
))((λg.λh.h(gf))(λu.x)))))f))(λu.u)
5 λf.λx.(λu.u)((λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf
))((λg.λh.h(gf))(λu.x)))))f)
6 λf.λx.(λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf))((λg.
λh.h(gf))(λu.x)))))f
7 λf.λx.(λh.h((λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf)
))(λu.x))))f))f
8 λf.λx.f((λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf))(λu
.x))))f)
9 λf.λx.f((λh.h((λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf))(λu.x)))f))
f)
10 λf.λx.f(f((λg.λh.h(gf))((λg.λh.h(gf))((λg.λh.h(gf))(λu.x)))f))
11 λf.λx.f(f((λh.h((λg.λh.h(gf))((λg.λh.h(gf))(λu.x))f))f))
12 λf.λx.f(f(f((λg.λh.h(gf))((λg.λh.h(gf))(λu.x))f)))
13 λf.λx.f(f(f((λh.h((λg.λh.h(gf))(λu.x)f))f)))
14 λf.λx.f(f(f(f((λg.λh.h(gf))(λu.x)f))))
15 λf.λx.f(f(f(f((λh.h((λu.x)f))f))))
16 λf.λx.f(f(f(f(f((λu.x)f)))))
17 λf.λx.f(f(f(f(fx))))
--------------------
17 réductions
```

Ça fonctionne. Mais que de peine pour soustraire 1 ! La soustraction est le talon d'Achille de notre modèle. Je ne veux pas être efficace, mon but est de montrer que c'est **FAISABLE**.

5.2 Soustraction

Soustraire n, c'est itérer n fois le prédécesseur. Pour être précis, si $m \geq n$ le terme ci-dessous se réduit à C_{m-n}. Sinon, il se réduit à C_0. En fait on va trouver cela très pratique bientôt.

In [18]:
```
add_alias('SUB', '#m.#n.(n PRED)m')
```

Préparez vous à un choc brutal.

In [19]:
```
T = do('SUB 5 3')
```

```
0 (λm.λn.n(λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))m)(λf.λx.f(f(f(f(fx))
)))(λf.λx.f(f(fx)))
```

```
))))(λf.λx.f(f(fx)))
1 (λn.n(λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))(λf.λx.f(f(f(f(fx))))))(
λf.λx.f(f(fx)))
2 (λf.λx.f(f(fx)))(λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))(λf.λx.f(f(f(
f(fx)))))
3 (λx.(λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))((λn.λf.λx.n(λg.λh.h(gf))
(λu.x)(λu.u))((λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))x)))(λf.λx.f(f(f(
f(fx)))))
4 (λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))((λn.λf.λx.n(λg.λh.h(gf))(λu.
x)(λu.u))((λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))(λf.λx.f(f(f(f(fx))))
)))
5 λf.λx.(λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))((λn.λf.λx.n(λg.λh.h(gf
))(λu.x)(λu.u))(λf.λx.f(f(f(f(fx))))))(λg.λh.h(gf))(λu.x)(λu.u)
6 λf.λx.(λf.λx.(λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))(λf.λx.f(f(f(f(f
x)))))(λg.λh.h(gf))(λu.x)(λu.u))(λg.λh.h(gf))(λu.x)(λu.u)
7 λf.λx.(λx.(λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))(λf.λx.f(f(f(f(fx))
)))(λg.λh.h(g(λg.λh.h(gf))))(λu.x)(λu.u))(λu.x)(λu.u)
8 λf.λx.(λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))(λf.λx.f(f(f(f(fx)))))(
λg.λh.h(g(λg.λh.h(gf))))(λu.λu.x)(λu.u)(λu.u)
9 λf.λx.(λf.λx.(λf.λx.f(f(f(f(fx)))))(λg.λh.h(gf))(λu.x)(λu.u))(λg.λ
h.h(g(λg.λh.h(gf))))(λu.λu.x)(λu.u)(λu.u)
10 λf.λx.(λx.(λf.λx.f(f(f(f(fx)))))(λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))
))))(λu.x)(λu.u))(λu.λu.x)(λu.u)(λu.u)
11 λf.λx.(λf.λx.f(f(f(f(fx)))))(λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))
(λu.λu.λu.x)(λu.u)(λu.u)(λu.u)
12 λf.λx.(λx.(λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.
h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(
g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))x)
))))(λu.λu.λu.x)(λu.u)(λu.u)(λu.u)
13 λf.λx.(λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(
λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg
.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))(λu.λu
.λu.x)))))(λu.u)(λu.u)(λu.u)
14 λf.λx.(λh.h((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λ
h.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.
h(g(λg.λh.h(g(λg.λh.h(gf))))))(λu.λu.λu.x))))(λg.λh.h(g(λg.λh.h(gf))
))))(λu.u)(λu.u)(λu.u)
15 λf.λx.(λu.u)((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.
λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh
.h(g(λg.λh.h(g(λg.λh.h(gf))))))(λu.λu.λu.x))))(λg.λh.h(g(λg.λh.h(gf)
))))(λu.u)(λu.u)
16 λf.λx.(λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(
λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg
.λh.h(g(λg.λh.h(gf))))))(λu.λu.λu.x))))(λg.λh.h(g(λg.λh.h(gf))))(λu.
u)(λu.u)
17 λf.λx.(λh.h((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λ
h.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))(λu.λu.λ
u.x)))(λg.λh.h(g(λg.λh.h(gf))))))(λg.λh.h(g(λg.λh.h(gf))))(λu.u)(λu.
u)
18 λf.λx.(λg.λh.h(g(λg.λh.h(gf))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))
))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(λg.λ
h.h(gf))))))(λu.λu.λu.x)))(λg.λh.h(g(λg.λh.h(gf))))))(λu.u)(λu.u)
19 λf.λx.(λh.h((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λ
h.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))(λu.λu.λ
```

```
u.x)))(λg.λh.h(g(λg.λh.h(gf))))(λg.λh.h(gf))))(λu.u)(λu.u)

20 λf.λx.(λu.u)((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.
λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))(λu.λu.
λu.x)))(λg.λh.h(g(λg.λh.h(gf))))(λg.λh.h(gf)))(λu.u)
21 λf.λx.(λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(
λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))(λu.λu.λu.x)))
(λg.λh.h(g(λg.λh.h(gf))))(λg.λh.h(gf))(λu.u)
22 λf.λx.(λh.h((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λ
h.h(g(λg.λh.h(gf))))))(λu.λu.λu.x))(λg.λh.h(g(λg.λh.h(gf))))))(λg.λh
.h(g(λg.λh.h(gf))))(λg.λh.h(gf))(λu.u)
23 λf.λx.(λg.λh.h(g(λg.λh.h(gf))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))
))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))(λu.λu.λu.x))(λg.λh.h(g(λg
.λh.h(gf)))))(λg.λh.h(gf))(λu.u)
24 λf.λx.(λh.h((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λ
h.h(g(λg.λh.h(gf))))))(λu.λu.λu.x))(λg.λh.h(g(λg.λh.h(gf))))(λg.λh.h
(gf))))(λg.λh.h(gf))(λu.u)
25 λf.λx.(λg.λh.h(gf))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.
h(g(λg.λh.h(g(λg.λh.h(gf))))))(λu.λu.λu.x))(λg.λh.h(g(λg.λh.h(gf))))
(λg.λh.h(gf)))(λu.u)
26 λf.λx.(λh.h((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λ
h.h(g(λg.λh.h(gf))))))(λu.λu.λu.x))(λg.λh.h(g(λg.λh.h(gf))))(λg.λh.h
(gf))f))(λu.u)
27 λf.λx.(λu.u)((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.
λh.h(g(λg.λh.h(gf))))))(λu.λu.λu.x))(λg.λh.h(g(λg.λh.h(gf))))(λg.λh.
h(gf))f)
28 λf.λx.(λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))((λg.λh.h(g(λg.λh.h(g(
λg.λh.h(gf))))))(λu.λu.λu.x))(λg.λh.h(g(λg.λh.h(gf))))(λg.λh.h(gf))f
29 λf.λx.(λh.h((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))(λu.λu.λu.x)(λg.
λh.h(g(λg.λh.h(gf))))))(λg.λh.h(g(λg.λh.h(gf))))(λg.λh.h(gf))f
30 λf.λx.(λg.λh.h(g(λg.λh.h(gf))))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))
))))(λu.λu.λu.x)(λg.λh.h(g(λg.λh.h(gf)))))(λg.λh.h(gf))f
31 λf.λx.(λh.h((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))(λu.λu.λu.x)(λg.
λh.h(g(λg.λh.h(gf))))(λg.λh.h(gf))))(λg.λh.h(gf))f
32 λf.λx.(λg.λh.h(gf))((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))(λu.λu.λ
u.x)(λg.λh.h(g(λg.λh.h(gf))))(λg.λh.h(gf)))f
33 λf.λx.(λh.h((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))(λu.λu.λu.x)(λg.
λh.h(g(λg.λh.h(gf))))(λg.λh.h(gf))f))f
34 λf.λx.f((λg.λh.h(g(λg.λh.h(g(λg.λh.h(gf))))))(λu.λu.λu.x)(λg.λh.h
(g(λg.λh.h(gf))))(λg.λh.h(gf))f)
35 λf.λx.f((λh.h((λu.λu.λu.x)(λg.λh.h(g(λg.λh.h(gf))))))(λg.λh.h(g(λ
g.λh.h(gf))))(λg.λh.h(gf))f)
36 λf.λx.f((λg.λh.h(g(λg.λh.h(gf))))((λu.λu.λu.x)(λg.λh.h(g(λg.λh.h(
gf)))))(λg.λh.h(gf))f)
37 λf.λx.f((λh.h((λu.λu.λu.x)(λg.λh.h(g(λg.λh.h(gf))))(λg.λh.h(gf)))
)(λg.λh.h(gf))f)
38 λf.λx.f((λg.λh.h(gf))((λu.λu.λu.x)(λg.λh.h(g(λg.λh.h(gf))))(λg.λh
.h(gf)))f)
39 λf.λx.f((λh.h((λu.λu.λu.x)(λg.λh.h(g(λg.λh.h(gf))))(λg.λh.h(gf))f
))f)
40 λf.λx.f(f((λu.λu.λu.x)(λg.λh.h(g(λg.λh.h(gf))))(λg.λh.h(gf))f))
41 λf.λx.f(f((λu.λu.x)(λg.λh.h(gf))f))
42 λf.λx.f(f((λu.x)f))
43 λf.λx.f(fx)
```

```
--------------------

43 réductions
```

C'est correct, mais catastrophiquement inefficace. Là n'est pas le problème, en tout cas pas ici.

La soustraction des entiers est CALCULABLE.

L'efficacité des calculs n'est pas au menu du jour.

Nous discutons de CALCULABILITÉ et pas de COMPLEXITÉ.

Théorème : $5 - 3 = 2.$

La preuve se fait en 43 réductions. Je vous invite à vérifier les étapes à la main :-).

6. Bilan

Les fonctions d'addition, soustraction, multiplication, puissance sont donc calculables dans le λ-calcul. Il resterait à considérer la division. Nous le ferons dans le prochain notebook. Après tout, une division n'est qu'une suite de soustractions, une simple **boucle** et hop, c'est fait ! Oui, mais comment faire des boucles ? Patience...

7. Et ensuite ?

Dans le prochain notebook, nous définirons à l'intérieur du λ-calcul les booléens `TRUE` et `FALSE` . Puis nous apprendrons à faire des tests en écrivant un terme `IF` tel que `IF B M N` se réduit à `M` si `B` est vrai, et à `N` si `B` est faux.

Nous apprendrons également à modéliser des **couples**, des **triplets**, des **listes**.

Enfin, nous verrons qu'il est possible de modéliser la notion de récursivité grâce à des termes spéciaux que l'on appelle des **combinateurs de point fixe**. À titre d'exemple, nous écrirons un terme `F` tel que FC_n soit la somme des entiers de 0 à n. Et aussi un terme `FIB` tel que `FIB` C_n ait pour forme normale C_{F_n} où F_n est le $n$ième nombre de Fibonacci.

Nous terminerons par la résolution du problème des **tours de Hanoi**.

In [ ]: