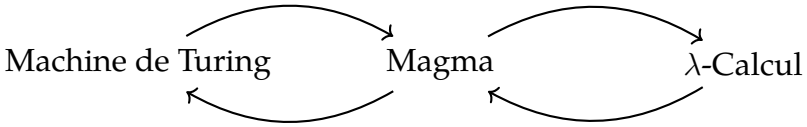
En 1936, les mathématiciens Alonzo Church, Alan Turing et Kurt Gödel ont proposé des formalismes pour le concept de "fonction calculable", c'est à dire les fonctions mathématiques pour lesquelles on peut écrire un algorithme les calculant. Il se trouve que ces trois modèles: le lambda calcul, les machines de Turing et les fonctions μ-récursives fournissent respectivement le même ensemble de fonctions calculables. On se propose donc de démontrer de manière constructiviste l'équivalence entre le lambda calcul et les machines de Turing.

Théorème de Church-Turing Une fonction est calculable par une machine de Turing si et seulement si elle est calculable par un lambda-terme.
On démontrera ceci en montrant que les deux sont équivalents au langage de programmation Magma (*défini plus tard*)



I Définitions

1 Machines de Turing

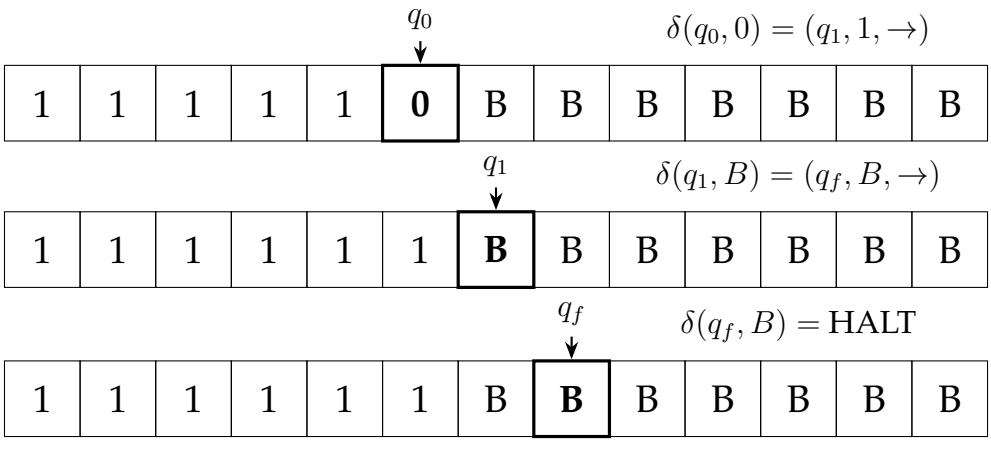
Définition 1 Une machine de Turing est un sextuplet $(Q, \Sigma, \Gamma, q_0, \delta, F)$ où

- Q est un ensemble fini d'états
- Σ est un ensemble fini de symboles d'entrée
- Γ est un ensemble fini de *symboles de bande* contenant B un symbole spécial et tel que $\Sigma \subset \Gamma \setminus \{B\}$
- $q_0 \in Q$ est l'état initial
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \rightarrow\}$ une fonction de transition
- $F \subset Q$ est l'ensemble des états finaux.

Étant donné une *configuration* $(q, i, b) \in Q \times \mathbb{N} \times \Gamma^{\mathbb{N}}$ et T une machine de Turing, on peut "appliquer" T à (q, i, b) en faisant:

- Rien si $q \in F$: $T \cdot (q, i, b) = (q, i, b)$
- Sinon, en notant $(q', \gamma, d) = \delta(q, b_i)$ alors on pose $b'_i = \gamma$ ($b'_j = b_j$ pour $i \neq j$) et $i' = i + 1$ si $d = \rightarrow$ voire $i' = i - 1$ si $d = \leftarrow$ et $i > 0$. Alors $T \cdot (q, i, b) = (q', i', b')$

Exemple 2



On peut alors *exécuter* une machine de Turing T sur une entrée $w \in \Sigma^*$ en calculant la "limite" pour $n \to \infty$ de $T^n \cdot (q_0, 0, wB^\infty)$. Dans les cas où le procédé stationne, on note $T(w) \in \Sigma^*$ le mot formé par les éléments de Σ de la bande, lus de gauche à droite.

Définition 3 Une fonction partielle $f : \Sigma^* \to \Sigma^*$ sera dite calculable par une machine de Turing s'il en existe une telle que lorsque $f(w)$ est défini, T termine sur w et $f(w) = T(w)$.

2 Lambda Calcul

Définition 4 Si V est un ensemble dénombrable de variables (on prendra souvent $V = \Sigma^*$ avec Σ un alphabet fini), l'ensemble Λ des *lambda termes* sur V est défini inductivement par:

- $V \subset \Lambda$ les variables
- $\lambda x.N \in \Lambda$ où $x \in V$ et $N \in \Lambda$. Ceci représente la fonction anonyme $x \mapsto N$ (où x peut apparaître dans N)
- $f\ g \in \Lambda$ où $f, g \in \Lambda$. Ceci représente l'appel de la fonction f avec g comme argument.

Exemple 5 $\lambda a.\lambda b.a$ et $(\lambda x.xx)(\lambda x.xx)$ sont des lambda termes.

Définition 6 La β-réduction est le procédé de transition suivant sur les lambda termes: $(\lambda x.M)\, N \to M[x := N]$. Une β-réduction peut être effectuée à n'importe quel endroit dans le terme. Si $u, v \in \Lambda$, on notera $u \to_\beta v$ pour signifier que u se réduit en v en une réduction. On notera $\to_\beta^*$ la fermeture transitive réflexive de $\to_\beta$.

Définition 7 Deux termes sont dits *α-équivalents* si ils sont identiques, à renommage près des variables liées. Ainsi $\lambda x.z\ x$ et $\lambda y.z\ y$ sont alpha-équivalents, mais pas avec $\lambda x.a\, x$. On considérera systématiquement les lambda termes à renommage près des variables liées.

II Langage Magma

Afin de simplifier notre étude, nous allons nous servir d'un langage de programmation impératif très simple, mais équivalent aux machines de Turing.

Exemple 8 Les programmes suivants exemplifient la plupart des fonctionnalités

```
i = 1
repeat 100
    if i % 15 == 0
        print [true, false]
    elif i % 3 == 0
        print [true]
    elif i % 5 == 0
        print [false]
    else
        print i
    i = i + 1
```

```
i = 0
f = 1
while f < 10
    i = i + 1
    f = 1
    for j in 1..i
        f = f * j
print i
```

Lemme 9 Le langage Magma peut simuler toute machine de Turing.
Démonstration: Soit $T = (Q, \Sigma, \Gamma, q_0, \delta, F)$ une machine de Turing. A renommage près, on suppose $\Gamma = [\![1, |\Gamma|]\!]$, $Q = [\![1, |Q|]\!]$ ainsi que $q_0 = 0$ et $F = [\![1, |F|]\!]$. On construit alors le programme Magma suivant les idées suivantes:

- ✿ Les `array` ne sont pas bornés, on en utilise alors un pour représenter la bande de mémoire (avec les cases non existantes signifiant forcément B)
- ✿ Deux variables q et i pour q et i, initialisés à 0.
- ✿ Une boucle while ($q \neq f_0$ && ... && $q \neq f_{|F|}$), contenant `if/elif` ($q = j$ && b[i] $= k$), une branche pour $1 \leqslant j \leqslant |Q|, 1 \leqslant k \leqslant |\Gamma|$.
- ✿ L'exécution de la machine sur une entrée w se fait en initialisant b à w au début.

Lemme 10 Tout programme magma peut être réduit à une machine de Turing.
Démonstration: Considérons un programme magma. On commence par le réduire à une forme intermédiaire équivalente, dans laquelle on se permet d'avoir des `goto` conditionnels:

- ✿ Si une expression contenant plus qu'une variable est dans un `print`, `while`, etc..., on bouge l'expression vers une variable séparée
- ✿ On découpe les expressions contenant strictement plus qu'un opérateur en plusieurs variables et expressions
- ✿ On remplace toute boucle et tout conditionnel par des `goto` conditionnels
- ✿ On remplace `print` par des ajouts à un tableau

Il ne reste alors plus que des simples instructions de `goto`, copiage de variable et simples opérations. Il est possible (mais technique) de construire la machine de Turing associée, en utilisant par exemple les idées suivantes:

- ✿ On encode chaque entier en unaire, les booléens avec des symboles $\top$ et $\bot$, et un tableau a ses éléments séparés par des |. Puisqu'on peut simuler plusieurs bandes en simultané, on "stocke" chaque variable sur une bande différente.
- ✿ Les états sont de la forme (q, l) où l est la ligne du code originel en cours d'exécution, et q un sous-état qui permet les instructions intermédiaires.
- ✿ Une instruction `goto` revient simplement à modifier la partie l de l'état en fonction de la case associée à la variable.
- ✿ On peut copier une variable dans une autre en allant de gauche à droite jusqu'à atteindre un B.
- ✿ Les opérations élémentaires arithmétiques sont faisable au cas par cas.

III Réduction au Lambda Calcul

Étant donné un code source en Magma, il nous faudrait un lambda-terme calculant la même fonction. Il y a deux problèmes: le lambda calcul est fonctionnel tandis que le magma est impératif: il faudra donc traduire la logique impérative (mutabilité, boucles) en logique fonctionnelle (immutabilité, récursion). De plus, le magma a des booléens, listes et entiers, tandis que le lambda calcul n'a aucun type de données (en dehors de fonction, bien sûr).

1 Librairie standard

Construisons une petite librairie standard de fonctions et de données grâce à des structures de lambda termes.

Définition 11 Un *booléen* est un lambda terme de la forme $\texttt{true} := \lambda a.\lambda b.a$ ou $\texttt{false} := \lambda a.\lambda b.b$

Cette définition encode directement l'opérateur ternaire if then else, et permet assez directement de construire les opérateurs booléens:
- $\rightarrow$ $\texttt{and} := \lambda a.\lambda b.a\, b\, a$
- $\rightarrow$ $\texttt{or} := \lambda a.\lambda b.a\, a\, b$
- $\rightarrow$ $\texttt{not} := \lambda a.a\, \texttt{false}\, \texttt{true}$

On évitera par contre d'utiliser les booléens pour définir des entiers, étand donné qu'il existe une manière plus naturelle d'y parvenir:

Définition 12 Un *numéral de Church* est un lambda terme de la forme $\lambda f.\lambda a.f(\cdots n \text{ fois} \cdots f(a)\cdots) = \lambda f.\lambda a.f^n(a)$. On associera alors $n \in \mathbb{N}$ au numéral avec f^n. Ceci encode un entier n comme une façon de composer n fois une fonction.

On en déduit alors les opérateurs arithmétiques suivants:
- $\rightarrow$ $\texttt{succ} := \lambda n.\lambda f.\lambda a.n\, f\, (f\, a)$, ce qui informellement "ajoute un f"
- $\rightarrow$ $\texttt{add} := \lambda n.\lambda m.\lambda f.\lambda a.n\, f (m\, f\, a)$, ce qui généralise l'idée précédente.
- $\rightarrow$ $\texttt{mult} := \lambda n.\lambda m.\lambda f.\lambda a.n\, (m\, f)\, a$
- $\rightarrow$ $\texttt{pred} := \lambda n.\lambda f.\lambda a.n\, (\lambda g.\lambda h.h\, (g\, f))\, (\lambda u.u)$
- $\rightarrow$ $\texttt{sub} := \lambda n.\lambda m.m\, \texttt{pred}\, n$

  Ainsi que quelques opérations de comparaison, parmi d'autres:
- $\rightarrow$ $\texttt{iszero} := \lambda n.n\, (\lambda u.\texttt{false})\, \texttt{true}$
- $\rightarrow$ $\leqslant := \lambda n.\lambda m.\texttt{iszero}\, (\texttt{sub}\, n\, m)$

Définition 13 On parlera de *paire* pour désigner un terme de la forme $\langle a, b\rangle := \lambda p.p\, a\, b$ (on ne passera que des booléens comme argument). Ceci permet de construire des listes, par exemple $[a, b, c] := \langle a, \langle b, c\rangle\rangle$ et plus généralement $[a_0, ..., a_n] := \langle a_0, [a_1, ..., a_n]\rangle$. Pour des tableaux dans lesquels on veut pouvoir faire des parcours, on enveloppera les listes par des paires $\langle \text{bool}, \text{lst}\rangle$ où bool vaut true si lst est non vide, et false sinon. Ainsi $[\![a_0, ..., a_n]\!] = \langle \text{true}, \langle a_0, [\![a_1, .., a_n]\!]\rangle\rangle$.
Les fonctions principales associés sont $\texttt{fst} := \lambda p.p\, \texttt{true}$ et $\texttt{snd} := \lambda p.p\, \texttt{false}$.

Afin de pouvoir parcourir des listes et tableaux, il nous faut de la récursivité, or toutes les fonctions sont anonymes (on ne peut pas leur donner un nom puis s'y référer dans le corps de la fonction). On utilisera alors l'astuce suivante:

Définition 14 On appelle combinateur $\texttt{Y}$ le terme $\texttt{Y} := \lambda f.(\lambda x.f(x\, x))(\lambda x.f(x\, x))$. On remarquera que $\texttt{Y}\, f \rightarrow_\beta f(\texttt{Y}\, f)$.

Exemple 15 Sans se soucier des détails d'implémentation, on peut construire une fonction factorielle ainsi:

$$\text{F} := \lambda f.\lambda n.\text{si } n = 0 \text{ alors } 1 \text{ sinon } f(n-1)$$

$$\texttt{fact} := \texttt{Y}\, F$$

Alors, si on réduit schématiquement $\texttt{fact}\, 1$

$$\texttt{Y}\, F\, 1 \rightarrow F(\texttt{Y}\, F)\, 1 \rightarrow (\lambda n.\text{si } n = 0 \text{ alors } 1 \text{ sinon } (\texttt{Y}\, F)(n-1))\, 1$$

$$\rightarrow \text{si } 1 = 0 \text{ alors } 1 \text{ sinon } (\texttt{Y}\, F)(0) \rightarrow (\texttt{Y}\, F)\, 0 \rightarrow F(\texttt{Y}\, F)\, 0$$

$$\rightarrow \text{si } 0 = 0 \text{ alors } 1 \text{ sinon } f(0-1) \rightarrow 1$$

A l'aide de cette récursivité, on construit des fonctions pour gérer les listes et tableaux.

→ `l_getter` := $\lambda l.\lambda n.(n\,(\lambda p.p\,\texttt{false})\,l)\,\texttt{true}$

→ `l_setter` := $\texttt{Y}(\lambda F.\lambda l.\lambda n.\lambda x.\texttt{iszero}\,n\,(\lambda p.p\,x\,(\texttt{snd}\,l))\,(\lambda p.p(\texttt{fst}\,l)(F\,(l\,\texttt{false})(\texttt{pred}\,n)x)))$
Cette fonction fait un parcours en "profondeur" dans la liste, en la reconstruisant au fur et à mesure, elle encode l'idée que $\texttt{l\_setter}\,\langle a, \langle b, \langle c, \_\rangle\rangle\rangle\,1\,x \rightarrow \langle a, \texttt{l\_setter}\langle b, \langle c, \_\rangle\rangle\,0\,x\rangle$

Les fonctions a_getter et a_setter sur les tableaux relèvent de la même idée (mais en plus compliqué, pour gérer les $\langle true/false, \_\rangle$) Finalement, pour fold_left sur les tableaux (g représente une fonction de la forme $\lambda a.\lambda x.a'$ où a est l'accumulateur)

→ `a_fold_left` := $\texttt{Y}(\lambda F.\lambda l.\lambda g.\lambda a.(\texttt{fst}\,l)(F\,(\texttt{snd}(\texttt{snd}\,l))\,g\,(g\,a\,(\texttt{fst}(\texttt{snd}\,l)))))$

2 Fonctions d'état

Encodons à présent une logique impérative d'exécution, à l'aide d'un "objet d'état". Vu qu'il n'y a pas de mémoire mutable de disponible, on va construire une structure contenant une représentation de la mémoire et d'une trace de sortie (pour gérer les appels à print). On utilisera alors la paire $\langle [m_0, ..., m_n], [p_0, ..., p_k]\rangle$, où (m_i) sont les variables (on remplacera chaque nom de variable magma par un entier) et (p_j) les valeurs de sortie, dans l'ordre inverse (c'est un peu plus rapide à exécuter).

On construit alors des fonctions qui prennent en entrée un état et retournent un nouvel état auquel les bonnes modifications ont été faites. Notons $\texttt{set\_mem} := \lambda m.\lambda p.p\,m\,(\texttt{snd}\,st)$ et $\texttt{set\_prnt} := \lambda m.\lambda p.(\texttt{fst}\,st)\,m$ des petites fonctions qui supposent que st est défini dans le contexte et contient la variable d'état. Alors les fonctions d'état de base sont

→ $\texttt{setvar}_{i,v}$ change la valeur de la variable i à v: $\lambda st.\texttt{set\_mem}(\texttt{l\_setter}\,(\texttt{fst}\,st)\,i\,v)$

→ $\texttt{printer}_e$ ajoute la valeur de e à la print-liste: $\lambda st.\texttt{set\_prnt}\,(\lambda p.p\,e)\,(\text{fst}\,st)$

Définition 16 Si f et g sont des fonctions on note $f;g := x \mapsto g(f(x))$ la *composée impérative* de f et de g.

Lemme 17 Si f et g des fonctions d'états, et e une expression, alors on peut construire les fonctions d'états suivantes:

→ `If`(e, f, g): $\lambda st.e\ (f\ st)\,(g\ st)$

→ `While`(e, f): $\texttt{Y}(\lambda st.e\,(F\,(f\ st))\,st)$

→ `Repeat`(e, f): $\lambda st.e\ f\ st$

→ `For`(e, i, f): $\lambda st.\texttt{a\_fold\_left}\ e\ (\texttt{setvar}_{i,x}; f)\ st$

Lemme 18 Toute expression arithmétique Magma peut être calculé par un lambda terme.
Démonstration: Induction structurelle sur l'expression à l'aide de la librairie standard et du terme $\texttt{l\_getter}\,(\texttt{fst}\,st)\,i$ pour la variable i.

Lemme 19 Un programme Magma peut être calculé par un lambda terme.
Démonstration: On procède par induction structurelle sur l'arbre syntaxique du magma.

✿ Affectation de variable $x = e$, e est une expression. Alors $\texttt{setvar}_{x,\tilde{e}}$ convient, avec $\tilde{e}$ donné par le lemme précédent.

✿ Sortie `print` e, alors $\texttt{printer}_{\tilde{e}}$ convient.

✿ `If`, `While`, `Repeat`, `For`: Appliquer le **Lemme 18** aux expressions. Ensuite pour tout sous-bloc de code $\{l_1, ..., l_n\}$, par hypothèse d'induction, il existe $f_1, ...f_n$ des fonctions d'état associés. Alors $f_1; \cdots ; f_n$ est une fonction d'état associée au bloc. On peut alors appliquer le **Lemme 17** pour conclure.

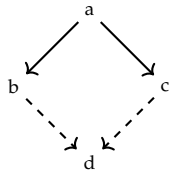
Le programme principal de ce TIPE prend en entrée un code source Magma, effectue l'analyse syntaxique à l'aide d'une grammaire LL1 pour obtenir l'arbre syntaxique, puis en déduit le programme en lambda calcul associé en suivant le procédé décrit auparavant.

IV Interprétation du Lambda Calcul

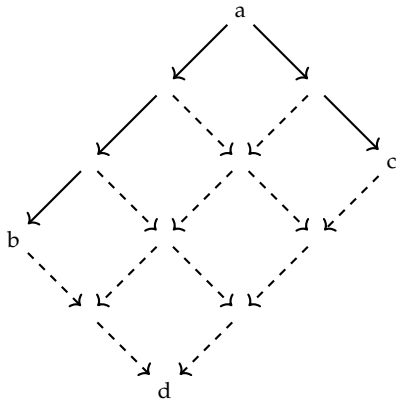
Pour finir notre démonstration, il faut que les machines de Turing puissent interpréter le lambda calcul, ce qui nécessite une stratégie de réduction.

Définition 20 Un lambda terme sera dit *en forme normale* lorsqu'aucune β-réduction n'est possible à l'intérieur du terme. On désignera sous le nom de réduction en forme normale le procédé d'application des β-réductions au sous-terme le plus à gauche parmi les moins profonds (leftmost outermost), jusqu'à atteindre la forme normale.

Définition 21 Une relation binaire $\rightarrow$ est dite *confluente* si $a \rightarrow b$ et $a \rightarrow c$ implique qu'il existe d tel que $b \rightarrow d$ et $c \rightarrow d$, pour tout a, b, c.



Lemme 22 Si une relation $\rightarrow$ est confluente, alors sa clôture réflexive transitive $\rightarrow^*$ est confluente aussi.
Démonstration: par récurrence forte sur (n, m) (muni de l'ordre lexicographique) où $a \rightarrow^n b$ et $a \rightarrow^m c$. Le schéma suivant donne l'intuition de la preuve.



Définition 23 On définit la relation de *réduction parallèle* $\Rightarrow$ par:

✦ $M \Rightarrow M$
✦ Si $M \Rightarrow M'$ alors $\lambda x.M \Rightarrow \lambda x.M'$

✦ Si $M \Rightarrow M'$ et $N \Rightarrow N'$ alors $MN \Rightarrow M'N'$ et $(\lambda x.M)N \Rightarrow M'[N'/x]$

Théorème 24 (Church-Rosser) Le lambda calcul est confluent pour $\rightarrow_\beta^*$.
Démonstration: La relation $\rightarrow_\beta$ n'est pas confluente, mais $\Rightarrow$ l'est [**1**]. Or $\rightarrow_\beta \subset \Rightarrow \subset \rightarrow_\beta^*$ donc $\rightarrow_\beta^* \subset \Rightarrow^* \subset \rightarrow_\beta^*$ d'où $\Rightarrow^* = \rightarrow_\beta^*$. D'après **Lemme 22**, $\Rightarrow^*$ est confluente donc $\rightarrow_\beta^*$ aussi.

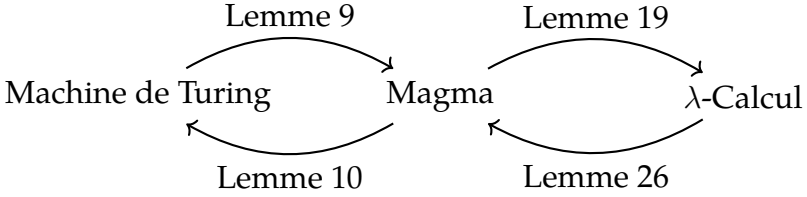
Théorème 25 Si un lambda terme admet une forme normale, alors elle est unique et la réduction en forme normale l'atteindra.
Démonstration (Unicité): Si $M \rightarrow_\beta^* A$ et $M \rightarrow_\beta^* B$ avec A et B en forme normale, alors d'après **Church-Rosser**, il existe C tel que $A \rightarrow_\beta^* C$ et $B \rightarrow_\beta^* C$. Or par normalité, $A = C$ et $B = C$ donc $A = B$.
Le deuxième point est technique, on admet donc le théorème de standardisation [**2**]*: si $M \rightarrow_\beta N$, alors il existe une suite de réductions qui sont "de moins en moins leftmost-outermost" vers N. On remarque de plus que la réduction leftmost-outermost suit exactement cette réduction standard lorsque la forme normale existe, ce qui permet de conclure.*

Lemme 26 Tout lambda terme est équivalent à un programme Magma.
Démonstration: Dans l'annexe, un programme python est fourni qui construit l'arbre de syntaxe d'un lambda terme donné, puis effectue itérativement des réductions leftmost-outermost, ce qui d'après le Théorème 23, est un algorithme correct. On pourrait interpréter cet algorithme en magma, grâce aux idées suivantes (l'implémentation n'a pas été faite par souci de testabilité: le code est extrêmement lent):

- ✿ Au lieu de parser le code source lambda, l'arbre peut être fourni directement sous forme d'arbre syntaxique, et V étant dénombrable, avec les variables déjà transformées en entiers
- ✿ Représenter l'arbre du lambda terme sous forme de tableaux imbriqués (en représentant $\lambda n.f$ comme $[0, n, f]$, n: $[1, n]$ et $f\ g$: $[2, f, g]$)
- ✿ Au lieu de pointeurs (Node), garder un chemin jusqu'au nœud intéressant
- ✿ Utiliser des tableaux de paires au lieu de tables de hachage (on ne se soucie pas du temps d'exécution)
- ✿ Utiliser une pile (tableau) pour simuler les appels récursifs

V Conclusion

D'après le diagramme suivant, on a bien une équivalence entre "fonction calculable par une machine de Turing" et "fonction calculable par un lambda terme", ce qui conclut la preuve du **Théorème de Church-Turing**.



References

[1] University of Waterloo *Proof of the Church-Rosser Theorem*
https://student.cs.uwaterloo.ca/~cs442/W25/extras/c-r-thm-proof.pdf

[2] Ryo Kashima *A Proof of the Standardization Theorem in λ-Calculus*
https://www.is.c.titech.ac.jp/~kashima/pub/C-145.pdf