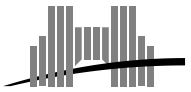
# *Sémantiques opérationnelles du lambda-calcul*

*version du 9 novembre 2004 – 18 : 02*

# *Les substitutions explicites*



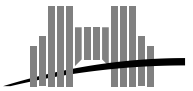


*The synthetic theory of combinators "gives the ultimate analysis of substitutions in terms of a system of extreme simplicity.*

*The theory of lambda-conversion is intermediate in character between synthetic theories and ordinary logic ... and it has the advantage of departing less radically from our intuition."*

Curry and Feys,
in the introduction of
*Combinatory Logic*, (1958) p. 6



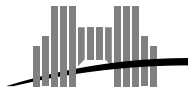


# Qu'est-ce que la contraction $\beta$ ?

La définition habituelle de la contraction $\beta$ en $\lambda$-calcul est

$$C[\![(\lambda x.M)\ P]\!] \to C[\![M[P/x]]\!].$$



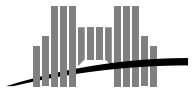


# Qu'est-ce qu'une substitution ?

La substitution $[_/_]$ est une opération atomique

qui consiste à remplacer une variable par un terme.





# Qu'est-ce qu'une substitution ?

La substitution $[_/_]$ est une opération atomique

qui consiste à remplacer une variable par un terme.

La substitution est une opération complexe

– dans son calcul

la même notation peut évoquer une opération coûteuse

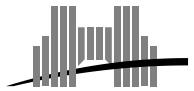
ou une opération bon marché,

ça n'est pas réaliste,

– et dans son implantation,

les captures sont difficiles à prendre en compte.



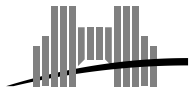


# Qu'est-ce qu'une substitution ?

Traditionnellement, la substitution n'est pas définie dans la théorie,

mais dans l'épithéorie.





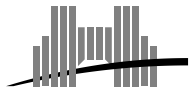
# Qu'est-ce qu'une substitution ?

Traditionnellement, la substitution n'est pas définie dans la théorie,

mais dans l'épithéorie.

Pourquoi ne pas en faire un citoyen de première classe ?





# Qu'est-ce qu'une substitution ?

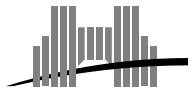
Traditionnellement, la substitution n'est pas définie dans la théorie,

mais dans l'épithéorie.

Pourquoi ne pas ne faire un citoyen de première classe ?

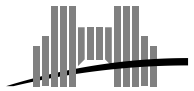
Pourquoi ne pas l'internaliser ?





# Le calcul $\lambda\mathbf{x}_{gc}$

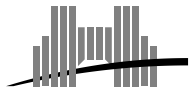




On fait un usage intensif de la convention de Barendregt sur les variables.

Dans un même contexte, une même variable n'est jamais à la fois libre et liée.





# La syntaxe de $\lambda\mathbf{x}_{gc}$

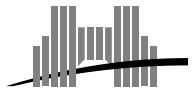
La syntaxe

$$M, N \quad ::= \quad x \mid \lambda x.M \mid M\ N \mid M\langle x = N\rangle$$

Un terme de la forme $M\langle x = N\rangle$ s'appelle une clôture.

Dans $M\langle x = N\rangle$ la variable $x$ est liée dans un $M$.



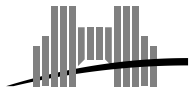


# La disponibilité

La notion de variable libre doit être généralisée avec précaution.

On parle de «variable disponible».





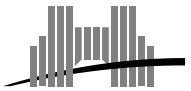
# La disponibilité

La notion de variable libre doit être généralisée avec précaution.

On parle de «variable disponible».

Une variable est non disponible si elle apparaît dans une clôture destinée à disparaître.





# La disponibilité

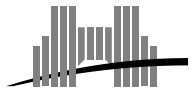
Si dans $M\langle x = N\rangle$

– la variable $y$ «apparaît» dans $N$

– mais si $x$ n'«apparaît» pas dans $M$

alors $y$ n'est pas disponible dans $M\langle x = N\rangle$.





# La disponibilité

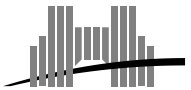
Plus formellement

$$\begin{cases} AV(x) & = & \{x\} & \\ AV(\lambda x.M) & = & AV(M)\backslash\{x\} & \\ AV(M\ N) & = & AV(M) \cup AV(N) & \\ AV(M\langle x = N\rangle) & = & (AV(M)\backslash\{x\}) \cup AV(N) & \text{if } x \in AV(M) \\ AV(M\langle x = N\rangle) & = & AV(M) & \text{if } x \notin AV(M) \end{cases}$$

# Les règles de $\lambda\mathbf{x}_{gc}$

$$
\begin{array}{lrcll}
(B) & (\lambda x.M)\,P & \rightarrow & M\langle x = P\rangle & \\
(App) & (MN)\langle x = A\rangle & \rightarrow & M\langle x = A\rangle\, N\langle x = A\rangle & \\
(Abs) & (\lambda y.M)\langle x = A\rangle & \rightarrow & \lambda y.(M\langle x = A\rangle) & \\
(VarI) & x\langle x = A\rangle & \rightarrow & A & \\
(VarK) & y\langle x = A\rangle & \rightarrow & y & \\
(gc) & M\langle x = A\rangle & \rightarrow & M & \text{if } x \notin AV(M)
\end{array}
$$





# Les règles de $\lambda\mathbf{x}_{gc}$

$$
\begin{array}{lrcll}
(B) & (\lambda x.M)\,P & \to & M\langle x = P\rangle & \\
(App) & (MN)\langle x = A\rangle & \to & M\langle x = A\rangle\, N\langle x = A\rangle & \\
(Abs) & (\lambda y.M)\langle x = A\rangle & \to & \lambda y.(M\langle x = A\rangle) & \\
(VarI) & x\langle x = A\rangle & \to & A & \\
(VarK) & y\langle x = A\rangle & \to & y & \\
(gc) & M\langle x = A\rangle & \to & M & \text{if } x \notin AV(M)
\end{array}
$$

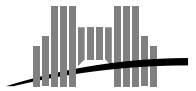
Noter l'importance de la convention de Barendregt dans la règle $(Abs)$.

# La règle (gc)

La règle $(VarK)$ est un cas particulier de la règle $(gc)$.

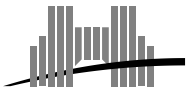
$(gc)$ a un effet global sur un terme qui rappelle le glanage de cellule (garbarge collection) dans les langages de programmation (notamment les langages de programmation fonctionnelle).





# Les indices de de Bruijn





# Notations Standard

$\lambda \; x \,.\, x$

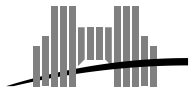
$\lambda \; x \; \lambda \, y \,.\, x$

$\lambda \, x \; \lambda \, y \,.\, y$

$\lambda \, f. \;\; \lambda \, x. \; f \, (f \, x)$

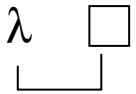
$\lambda \, x \,.\, x \;\; (\lambda \; y. \; x \, y)$



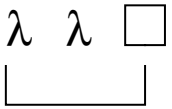


# Notations de Bourbaki

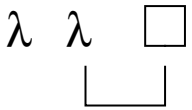
$\lambda\ x\,.\,x$



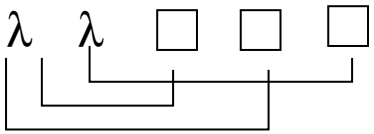
$\lambda\ x\ \lambda\ y\,.\,x$



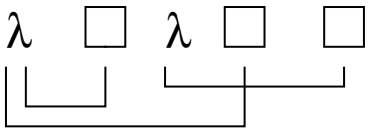
$\lambda\ x\ \lambda\ y\,.\,y$



$\lambda\ f.\ \ \lambda\ x.\ f\ (f\ x)$

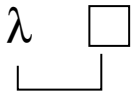


$\lambda\ x\,.\,x\ \ (\lambda\ y.\ x\ y)$



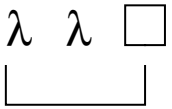
# Les indices de de Bruijn

$\lambda\ x\,.\,x$



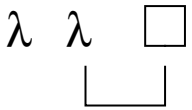
$\lambda\,\underline{0}$

$\lambda\ x\ \lambda\ y\,.\,x$



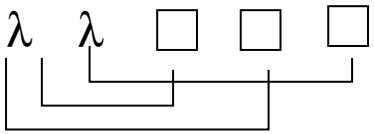
$\lambda\ \lambda\,\underline{1}$

$\lambda\,x\ \lambda\,y\,.\,y$



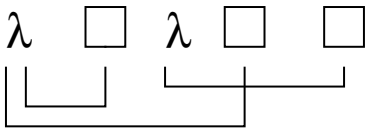
$\lambda\,\lambda\,\underline{0}$

$\lambda\,f.\ \ \lambda\,x.\,f\,(f\,x)$



$\lambda\,\lambda\,\underline{1}\ \ (\underline{1}\ \ \underline{0})$

$\lambda\,x\,.\,x\ \ (\lambda\ y.\,x\,y)$



$\lambda\,\underline{0}\,\lambda\,\underline{1}\,\underline{0}$

# Les indices de de Bruijn

Dans chaque terme, chaque variable est remplacée par sa profondeur de lien (son indice de de Bruijn).

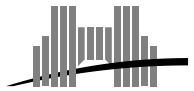
En d'autres mots.

$$I \equiv \lambda x.x \rightsquigarrow \lambda(\underline{0})$$

$$K \equiv \lambda c.\lambda x.c \equiv \lambda cx.c \rightsquigarrow \lambda(\lambda(\underline{1}))$$

$$two \equiv \lambda f.\lambda x.f(fx) \equiv \lambda fx.f(fx) \rightsquigarrow \lambda(\lambda(\underline{1}\,(\underline{1}\,\underline{0})))$$

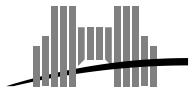




# *Présentation de $\lambda\upsilon$*

Un peu d'anglais ne fait pas de mal.





# Règle Beta

$$(B) \quad (\lambda M)\ P \quad \rightarrow \quad M[P/]$$

(B) introduit deux nouveaux opérateurs.

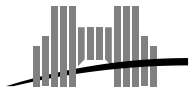
La syntaxe est étendue.

$$/ \quad : \quad \Lambda v \rightarrow Substitution$$

$$_[_] \quad : \quad \Lambda v \times Substitution \rightarrow \Lambda v$$

/ et _[_] introduisent une nouvelle sorte $Substitution$.



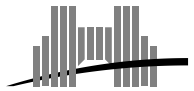


# Clôtures

_[_] doit être défini par deux règles nouvelles, à savoir.,

– une distribution à travers les applications les abstractions,
– une distribution sur les variables (indices de de Bruijn).





# Clôtures

$_[_]$ doit être défini par deux règles nouvelles, à savoir.,

– une distribution à travers les applications les abstractions,

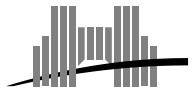
– une distribution sur les variables (indices de de Bruijn).

Les termes de la forme $M[s]$ sont appelés des clôtures.

Ils forment véritablement le calcul de substitution explicites.

Les termes sans clôtures sont appelés des termes purs.



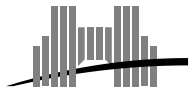


# Distribution des substitutions

**À travers les applications**

$$(App) \quad (M\ P)[s] \quad \rightarrow \quad M[s]\ P[s]$$





# Distribution des substitutions

**À travers les applications**

$$(App) \quad (M\ P)[s] \quad \rightarrow \quad M[s]\ P[s]$$

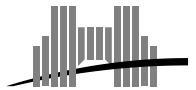
**À travers les abstractions**

$$(Lambda) \quad (\lambda M)[s] \quad \rightarrow \quad \lambda(M[\Uparrow(s)])$$

(Lambda) introduit un nouvel opérateur

$$\Uparrow(s) : Substitution \rightarrow Substitution.$$



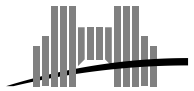


# Définir / et ⇑

$$(FVar) \qquad \underline{0}[M/] \quad \rightarrow \quad M$$

$$(RVar) \qquad \underline{n+1}[M/] \quad \rightarrow \quad \underline{n}$$



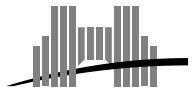


# Définir / et $\Uparrow$

$$
\begin{array}{lrcl}
(FVar) & \underline{0}[M/] & \rightarrow & M \\
(RVar) & \underline{n+1}[M/] & \rightarrow & \underline{n} \\
 & & & \\
(FVarLift) & \underline{0}[\Uparrow(s)] & \rightarrow & \underline{0} \\
(RVarLift) & \underline{n+1}[\Uparrow(s)] & \rightarrow & \underline{n}[s][\uparrow]
\end{array}
$$

(RVarLift) introduit une nouvelle constante $\uparrow$ : $Substitution$

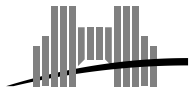




# Définir ↑

$$(VarShift) \quad \underline{n}[\uparrow] \quad \longrightarrow \quad \underline{n+1}$$

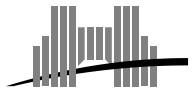




# $\lambda\upsilon$

| | | | |
|---|---|---|---|
| $(B)$ | $(\lambda M)\ P$ | $\rightarrow$ | $M[P/]$ |
| $(App)$ | $(M\ P)[s]$ | $\rightarrow$ | $M[s]\ P[s]$ |
| $(Lambda)$ | $(\lambda M)[s]$ | $\rightarrow$ | $\lambda(M[\Uparrow(s)])$ |
| $(FVar)$ | $\underline{0}[M/]$ | $\rightarrow$ | $M$ |
| $(RVar)$ | $\underline{n+1}[M/]$ | $\rightarrow$ | $\underline{n}$ |
| $(FVarLift)$ | $\underline{0}[\Uparrow(s)]$ | $\rightarrow$ | $\underline{0}$ |
| $(RVarLift)$ | $\underline{n+1}[\Uparrow(s)]$ | $\rightarrow$ | $\underline{n}[s][\uparrow]$ |
| $(VarShift)$ | $\underline{n}[\uparrow]$ | $\rightarrow$ | $\underline{n+1}$ |

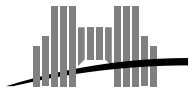




# Calculs dans $\lambda\upsilon$

$$
\begin{aligned}
two\ I \quad &\equiv \quad (\lambda\lambda\underline{1}\,(\underline{1}\,\underline{0}))\ (\lambda\underline{0}) \\
&\rightarrow \quad (\lambda\underline{1}\,(\underline{1}\,\underline{0}))[\lambda\underline{0}/]
\end{aligned}
$$

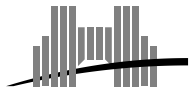




# Calculs dans $\lambda\upsilon$

$$
\begin{aligned}
two\ I \quad &\equiv \quad (\lambda\lambda\underline{1}\,(\underline{1}\,\underline{0}))\ (\lambda\underline{0}) \\
&\rightarrow \quad (\lambda\underline{1}\,(\underline{1}\,\underline{0}))[\lambda\underline{0}/] \\
&\rightarrow \quad \lambda((\underline{1}\,(\underline{1}\,\underline{0}))[\Uparrow(\lambda\underline{0}/)])
\end{aligned}
$$

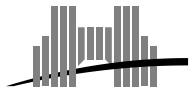




# Calculs dans $\lambda\upsilon$

$$
\begin{aligned}
two\ I \quad &\equiv \quad (\lambda\lambda\underline{1}\,(\underline{1}\,\underline{0}))\ (\lambda\underline{0}) \\
&\rightarrow \quad (\lambda\underline{1}\,(\underline{1}\,\underline{0}))[\lambda\underline{0}/] \\
&\rightarrow \quad \lambda((\underline{1}\,(\underline{1}\,\underline{0}))[\Uparrow(\lambda\underline{0}/)]) \\
&\rightarrow \quad \lambda(\underline{1}[\Uparrow(\lambda\underline{0}/)]\ (\underline{1}\,\underline{0})[\Uparrow(\lambda\underline{0}/)])
\end{aligned}
$$

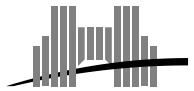




# Calculs dans $\lambda\upsilon$

$$
\begin{aligned}
two\ I &\equiv (\lambda\lambda\underline{1}\,(\underline{1}\,\underline{0}))\ (\lambda\underline{0}) \\
&\to (\lambda\underline{1}\,(\underline{1}\,\underline{0}))[\lambda\underline{0}/] \\
&\to \lambda((\underline{1}\,(\underline{1}\,\underline{0}))[\Uparrow(\lambda\underline{0}/)]) \\
&\to \lambda(\underline{1}[\Uparrow(\lambda\underline{0}/)]\ (\underline{1}\,\underline{0})[\Uparrow(\lambda\underline{0}/)]) \\
&\to \lambda(\underline{1}[\Uparrow(\lambda\underline{0}/)]\ (\underline{1}[\Uparrow(\lambda\underline{0}/)]\ \underline{0}[\Uparrow(\lambda\underline{0}/)]))
\end{aligned}
$$

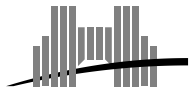




# Calculs dans $\lambda\upsilon$

$$
\begin{aligned}
two\ I &\equiv (\lambda\lambda \underline{1}\ (\underline{1}\ \underline{0}))\ (\lambda \underline{0}) \\
&\to (\lambda \underline{1}\ (\underline{1}\ \underline{0}))[\lambda \underline{0}/] \\
&\to \lambda((\underline{1}\ (\underline{1}\ \underline{0}))[\Uparrow(\lambda \underline{0}/)]) \\
&\to \lambda(\underline{1}[\Uparrow(\lambda \underline{0}/)]\ (\underline{1}\ \underline{0})[\Uparrow(\lambda \underline{0}/)]) \\
&\to \lambda(\underline{1}[\Uparrow(\lambda \underline{0}/)]\ (\underline{1}[\Uparrow(\lambda \underline{0}/)]\ \underline{0}[\Uparrow(\lambda \underline{0}/)])) \\
&\to \lambda(\underline{1}[\Uparrow(\lambda \underline{0}/)]\ (\underline{1}[\Uparrow(\lambda \underline{0}/)]\ \underline{0}))
\end{aligned}
$$

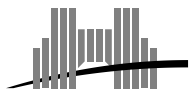




# Calculs dans $\lambda\upsilon$

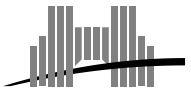
$$
\begin{array}{lcl}
two\ I & \equiv & (\lambda\lambda\underline{1}\ (\underline{1}\ \underline{0}))\ (\lambda\underline{0}) \\
 & \to & (\lambda\underline{1}\ (\underline{1}\ \underline{0}))[\lambda\underline{0}/] \\
 & \to & \lambda((\underline{1}\ (\underline{1}\ \underline{0}))[\Uparrow(\lambda\underline{0}/)]) \\
 & \to & \lambda(\underline{1}[\Uparrow(\lambda\underline{0}/)]\ (\underline{1}\ \underline{0})[\Uparrow(\lambda\underline{0}/)]) \\
 & \to & \lambda(\underline{1}[\Uparrow(\lambda\underline{0}/)]\ (\underline{1}[\Uparrow(\lambda\underline{0}/)]\ \underline{0}[\Uparrow(\lambda\underline{0}/)])) \\
 & \to & \lambda(\underline{1}[\Uparrow(\lambda\underline{0}/)]\ (\underline{1}[\Uparrow(\lambda\underline{0}/)]\ \underline{0})) \\
 & \to \cdot \to & \lambda(\underline{0}[\lambda\underline{0}/][\uparrow]\ (\underline{0}[\lambda\underline{0}/][\uparrow]\ \underline{0}))
\end{array}
$$





$$
\begin{aligned}
&\longrightarrow \cdot \longrightarrow \quad \lambda((\lambda\underline{0})[\uparrow]\,((\lambda\underline{0})[\uparrow]\,\underline{0})) \\
&\longrightarrow \cdot \longrightarrow \quad \lambda(\lambda(\underline{0}[\Uparrow(\uparrow)])\,(\lambda(\underline{0}[\Uparrow(\uparrow)])\,\underline{0})) \\
&\longrightarrow \cdot \longrightarrow \quad \lambda(\lambda\underline{0}\,((\lambda\underline{0})\underline{0})) \\
&\qquad\vdots \\
&\quad\longrightarrow\!\!\!\!\!\rightarrow \quad \lambda\underline{0}
\end{aligned}
$$

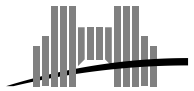




# Un calcul qui ne termine pas

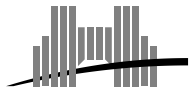
Exercice : Montrer que $(\lambda \underline{00})\ (\lambda \underline{00}) \xrightarrow[\lambda v]{} (\lambda \underline{00})\ (\lambda \underline{00})$.





# Propriétés de $\lambda\upsilon$





# Faits

– $\lambda v = (B) \oplus v$.

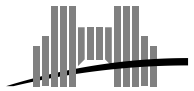
– $v$ est orthogonal

i.e., linéaire à gauche et sans paires critiques.

– $v$ termine

i.e., fortement normalisable.

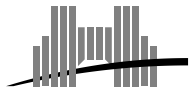




# Questions

**Est-ce que ce calcul représente le $\lambda$-calcul classique ?**





# Questions

**Est-ce que ce calcul représente le $\lambda$-calcul classique ?**

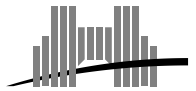
Oui

$M \xrightarrow[\beta]{} P$ est $M \xrightarrow[B]{} N$ et $N \xrightarrow[\upsilon]{} \upsilon(N)$.

où $\upsilon(N)$ représente la forme normale par $\upsilon$ de $N$,

qu'on note aussi $N \xrightarrow[\upsilon]{} P$ où $P \equiv \upsilon(N)$ .

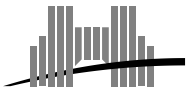




# Questions

**Est-ce que la spécification est complète ?**





# Questions

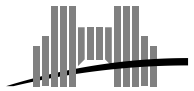
**Est-ce que la spécification est complète ?**

Que penser d'une règle

$$M[s][t] \longrightarrow M[\ldots s \ldots t \ldots]$$

où $[\ldots s \ldots t \ldots]$ est $[s \circ t]$ par exemple ?





# Questions

**Est-ce que la spécification est complète ?**

Que penser d'une règle

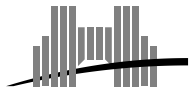
$$M[s][t] \longrightarrow M[s \circ t$$

par exemple ?

En fait, on n'en a a pas besoin,

puisque que chaque clôture disparaît dans les termes clos.



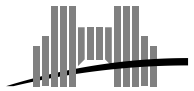


# Le lemme de substitution

$\lambda\upsilon$ admet une paire critique

$$\langle M[\Uparrow(s)][P[s]/] \, , \, M[P/][s] \rangle$$





# Le lemme de substitution

$\lambda\upsilon$ admet une paire critique

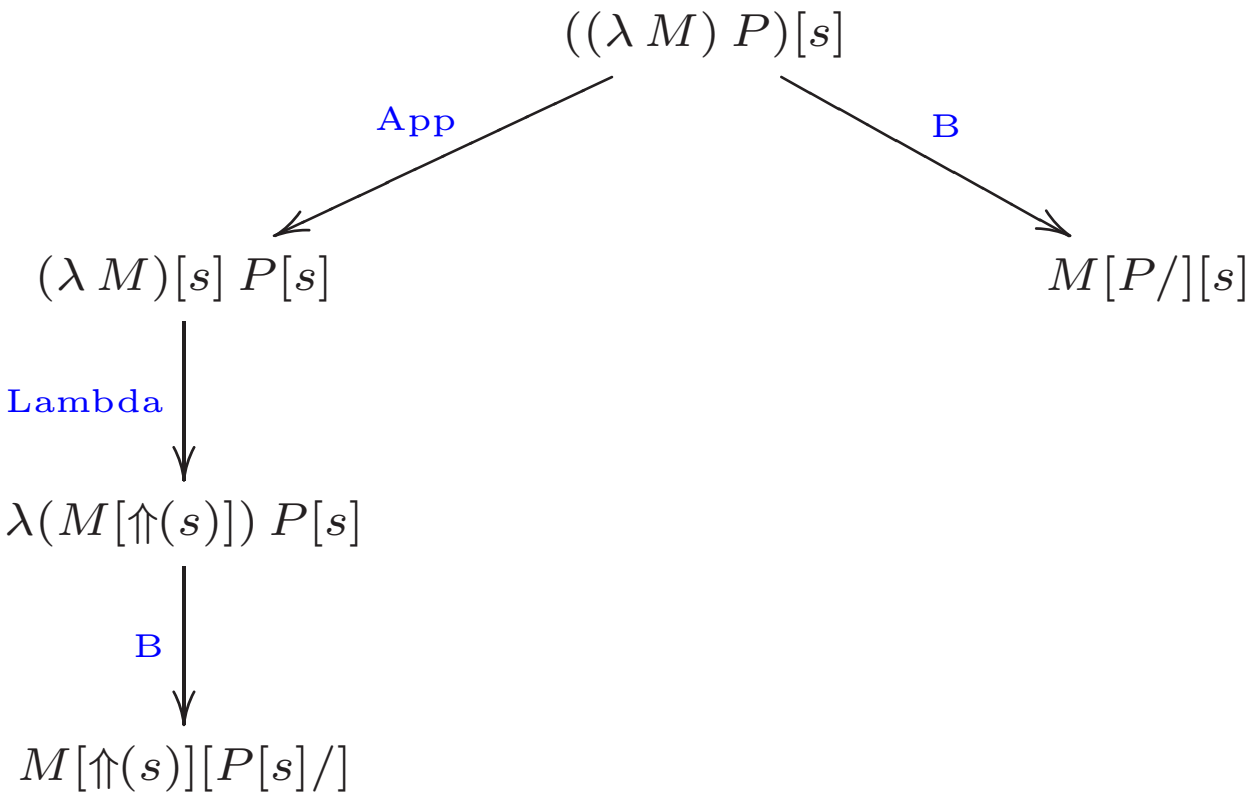
$$\langle M[\Uparrow(s)][P[s]/] \, , \, M[P/][s] \rangle$$

En général, l'identité

$$M[\Uparrow(s)][P[s]/] = M[P/][s]$$

est appelée le lemme de substitution.

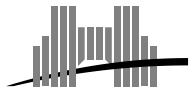
# Le lemme de substitution



# Le lemme de substitution

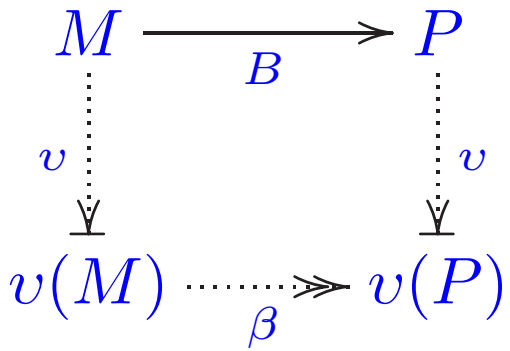
Le lemme de substitution est un théorème inductive dans $\lambda\upsilon$.





# le Lemme de projection

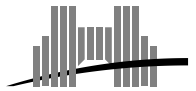
Si $M \xrightarrow[B]{} P$ alors $v(M) \xrightarrow[\beta]{} v(P)$.




Le lemme de projection est une conséquence

du lemme de substitution.

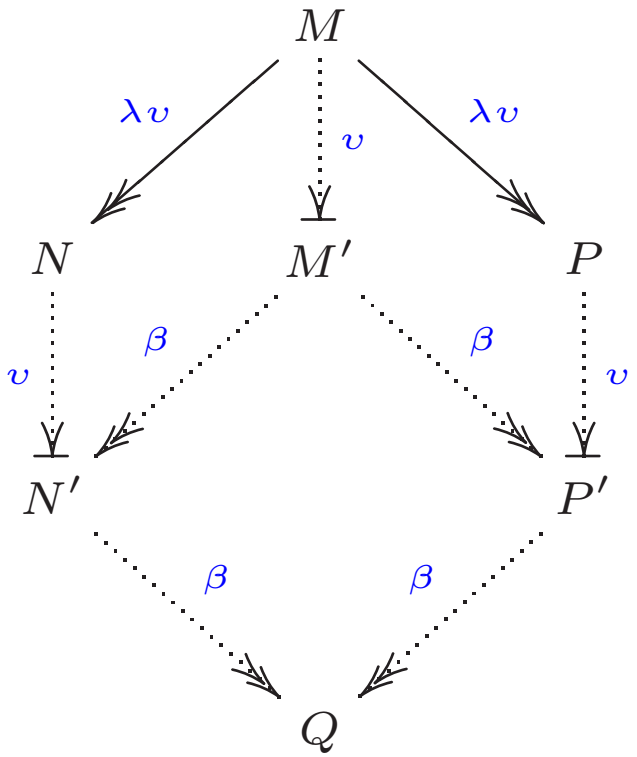




# Confluence de $\lambda\upsilon$

$\lambda\upsilon$ est confluent sur les termes clos.

La confluence est conséquence du lemme de projection.



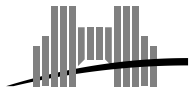
# $\lambda\upsilon$ préserve la normalisation forte

Si un terme $M$ est fortement $\beta$ normalisable,

alors $M$ est fortement $\lambda\upsilon$ normalisable.

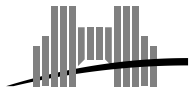
La preuve est par contradiction, fondée sur l'existence d'un plus petit contre-exemple.





# *Présentation de $\lambda\sigma$ faible*



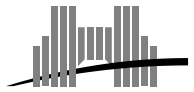


Le but est d'implanter la normalisation faible de tête :

– on ne réduit que le redex de tête,

– on ne réduit pas sous les $\lambda$.

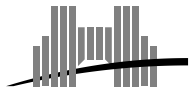




# Les opérateurs

$$
\begin{array}{rcll}
U, V & ::= & U\ V \mid M[s] & \textit{(Environnement d'évaluation)} \\
M, N & ::= & \underline{n} \mid \lambda M \mid M\ N & \textit{(Code)} \\
s, t & ::= & id \mid U \cdot s & \textit{(Substitutions)}
\end{array}
$$

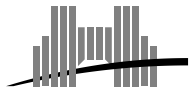




# Les règles

$$
\begin{array}{lrcl}
(B_\sigma) & (\lambda M)[s]\ U & \rightarrow & M[U \cdot s] \\
(App_\sigma) & (M\ N)[s] & \rightarrow & M[s]\ N[s] \\
(Fvar_\sigma) & \underline{0}[U \cdot s] & \rightarrow & U \\
(RVar_\sigma) & \underline{n+1}[U \cdot s] & \rightarrow & \underline{n}[s]
\end{array}
$$



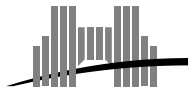


# Les règles

$$
\begin{array}{lrcl}
(B_\sigma) & (\lambda M)[s]\ U & \to & M[U \cdot s] \\
(App_\sigma) & (M\ N)[s] & \to & M[s]\ N[s] \\
(Fvar_\sigma) & \underline{0}[U \cdot s] & \to & U \\
(RVar_\sigma) & \underline{n+1}[U \cdot s] & \to & \underline{n}[s]
\end{array}
$$

Le formes normales sont $(\lambda M)[s]$, on les appelle aussi les valeurs.





## Comment coder $\lambda\sigma_w$ dans $\lambda\upsilon$ ?

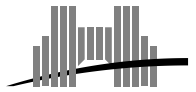
On voit que tout environnement d'évaluation de $\lambda\sigma_w$ s'écrit

– soit $M[U_1 \cdot \ldots \cdot U_p \cdot id]$,

où $M$ est un terme pur

– soit c'est combinaison par applications de tels environnements d'évaluation.





Définissons $\mathcal{T} : \lambda\sigma_w \mapsto \Lambda\upsilon$ par $\mathcal{T}(U) = \mathcal{T}'(U, 0)$ où

– $\mathcal{T}'(U\ V, n) = \mathcal{T}'(U, n)\,\mathcal{T}'(V, n)$

– $\mathcal{T}'(M[id], n) = M$,

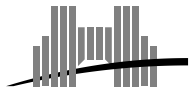
– $\mathcal{T}'(M[U \cdot s], n) = \mathcal{T}'(M[s], n+1)[\Uparrow^n(\mathcal{T}(U))/]$

Autrement dit :

$$\mathcal{T}(M[U_0 \cdot \ldots \cdot U_p \cdot id]) = M[\Uparrow^p(\mathcal{T}(U_p))/] \ldots [\mathcal{T}(U_0)/].$$

$M$ est un terme pur donc il est laissé intact par $T$.





Montrons que si $U \xrightarrow[B_\sigma]{} V$ alors $\mathcal{T}(U) \xrightarrow[Lambda_\upsilon]{} \cdot \xrightarrow[B_\upsilon]{} \mathcal{T}(V)$.

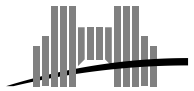
On peut supposer que cette réduction a lieu à la racine.

On a $U \equiv (\lambda M)[U_1 \cdot \ldots \cdot U_p \cdot id]\, U_0$
et $V \equiv M[U_0 \cdot U_1 \cdot \ldots \cdot U_p \cdot id]$.

On a donc

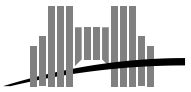
$$\begin{array}{lll}
\mathcal{T}(U) & = & (\lambda M)[\Uparrow^{p-1}(\mathcal{T}(U_p/))] \ldots [\mathcal{T}(U_1/)]\, \mathcal{T}(U_O) \\
& \xrightarrow[Lambda_\upsilon]{} & (\lambda M[\Uparrow^{p}(\mathcal{T}(U_p)/))] \ldots [\Uparrow(\mathcal{T}(U_1)/)])\, \mathcal{T}(U_0) \\
& \xrightarrow[B_\upsilon]{} & M[\Uparrow^{p}(\mathcal{T}(U_p)/)] \ldots [\Uparrow(\mathcal{T}(U_1)/)][\mathcal{T}(U_0)/] \\
& = & \mathcal{T}(V)
\end{array}$$





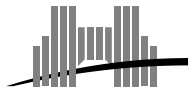
# *Les machines abstraites*





# *La machine de Krivine*





# Les états

Une machine abstraite comporte des états et des transitions.

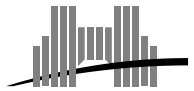
L'état $c$ , $e$ , $s$ de la machine de Krivine comporte trois composantes :

– le code $c$,

– l'environnement $e$,

– la pile $s$.





# Les états

Une machine abstraite comporte des états et des transitions.

L'état $c$ , $e$ , $s$ de la machine de Krivine comporte trois composantes :

– le code $c$,

– l'environnement $e$,

– la pile $s$.

$$
\begin{array}{rcll}
M, N & ::= & \lambda M \mid M\ N \mid \underline{n} & \textit{(Code)} \\
e & ::= & \langle M, e\rangle \cdot e \mid \mathsf{nil} & \textit{(Environnement)} \\
s & ::= & \langle M, e\rangle :: s \mid [\,] & \textit{(Pile)}
\end{array}
$$

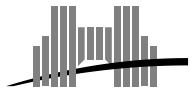
# Les transitions

On peut voir la machine de Krivine comme un moyen de décrire la réduction la plus à gauche et la plus externe dans le système $\lambda\sigma_w$.

On a quatre transitions qui correspondent aux quatre instructions de $\lambda\sigma_w$.

$$
\begin{aligned}
\lambda M\ ,\ e\ ,\ \langle N, f\rangle :: s &\ \rightarrow\ M\ ,\ \langle N, f\rangle \cdot e\ ,\ s \\
M\ N\ ,\ e\ ,\ s &\ \rightarrow\ M\ ,\ e\ ,\ \langle N, e\rangle :: s \\
0\ ,\ \langle N, f\rangle \cdot e\ ,\ s &\ \rightarrow\ N\ ,\ f\ ,\ s \\
n+1\ ,\ \langle N, f\rangle \cdot e\ ,\ s &\ \rightarrow\ n\ ,\ e\ ,\ s
\end{aligned}
$$

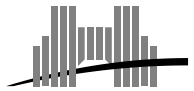




# Un exemple

$$
\begin{aligned}
(\lambda\underline{0})\ (\lambda\underline{0})\ ,\ \mathsf{nil}\ ,\ [\,] \quad &\rightarrow \quad (\lambda\underline{0})\ ,\ \mathsf{nil}\ ,\ [\langle\lambda\underline{0}, \mathsf{nil}\rangle] \\
&\rightarrow \quad \underline{0}\ ,\ \langle\lambda\underline{0}, \mathsf{nil}\rangle \cdot \mathsf{nil}\ ,\ [\,] \\
&\rightarrow \quad \lambda\underline{0}\ ,\ \mathsf{nil}\ ,\ [\,]
\end{aligned}
$$





# La sémantique

La machine de Krivine implante l'appel par nom.

La partie du corps de la fonction est évaluée avant les arguments.

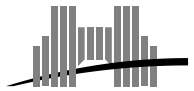
L'état de départ est $M$ , nil , $[\,]$.

Les états irréductibles sont

– $\lambda M$ , $e$ , $[\,]$ le code est une abstraction, mais il n'y a rien sur la pile,

– $\underline{n}$ , nil , $s$ le code est une variable mais l'environnement est vide.

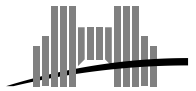




# Lien avec $\lambda\sigma_w$

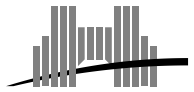
Exercice : Montrer la correspondance entre les réductions de la machine de Krivine et les réductions les plus à gauche et les plus externes dans le système $\lambda\sigma_w$.





# *Une machine pour l'appel par nécessité*





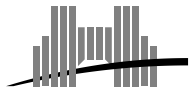
# Comment cela fonctionne

On utilise des places en mémoire repérées par des adresses pour stocker des clôtures.

On ajoute donc un état mémoire le tas («heap» en anglais) $h$ qui fait correspondre à chaque adresse une clôture.

Lorsqu'on réduit une clôture, il faut l'isoler dans son contexte.





# Les états

Les états sont de la forme $M$ , $e$ , $s$ , $u$ , $h$.

$$
\begin{array}{rcll}
M, N & ::= & \lambda M \mid M\ N \mid \underline{n} & \textit{(Code)} \\
e & ::= & a \cdot e \mid \mathsf{nil} & \textit{(Env)} \\
s & ::= & a :: s \mid [\,] & \textit{(Pile)} \\
u & ::= & (s, a) :: u \mid [\,] & \textit{(Pile de mise à jour)} \\
h & ::= & \{\,\} \mid h\{a \mapsto \langle M, e \rangle\} & \textit{(Tas)} \\
a & & \textit{les adresses de clôtures.} &
\end{array}
$$

# Les transitions

$$M\ N\ ,\ e\ ,\ s\ ,\ u\ ,\ h \quad \rightarrow \quad M\ ,\ e\ ,\ a \cdot s\ ,\ u\ ,\ h\{a \mapsto \langle N, e\rangle\} \qquad \textit{(App)}$$

$a$ fraîche

$$\lambda M\ ,\ e\ ,\ a \cdot s\ ,\ u\ ,\ h \quad \rightarrow \quad M\ ,\ a \cdot e\ ,\ s\ ,\ u\ ,\ h \qquad \textit{(Lam)}$$

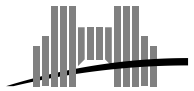
$$\underline{n+1}\ ,\ a \cdot e\ ,\ s\ ,\ u\ ,\ h \quad \rightarrow \quad \underline{n}\ ,\ e\ ,\ s\ ,\ u\ ,\ h \qquad \textit{(Skip)}$$

$$\underline{0}\ ,\ a \cdot e\ ,\ s\ ,\ u\ ,\ h \quad \rightarrow \quad N\ ,\ e'\ ,\ [\,]\ ,\ (s, a) :: u\ ,\ h \qquad \textit{(Access)}$$

où $h\,a = \langle N, e'\rangle$

$$\lambda M\ ,\ e\ ,\ [\,]\ ,\ (s, a) :: u\ ,\ h \quad \rightarrow \quad \lambda M\ ,\ e\ ,\ s\ ,\ u\ ,\ h\{a \mapsto \langle \lambda M, e\rangle\} \qquad \textit{(Update)}$$

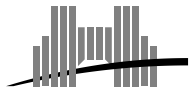




## Théorème

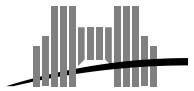
Cette machine réduit les termes clos jusqu'à leur forme normale de tête.





# *Sémantiques à grandes étapes*





# La sémantique à grandes étapes

Un système de réécriture donne typiquement une sémantique par petites étapes de calcul *small steps*.

C'est-à-dire qu'on décrit les étapes élémentaires du calcul.

Une sémantique qui décrit une relation donnée–résultat est dite sémantique à grandes étapes *(big steps)*.

La notation habituelle est $M \downarrow N$.

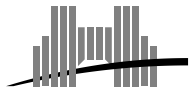
On parle aussi de sémantique opérationnelle structurée

(en abrégé et en anglais comme en français : SOS)

(due à Plotkin)

ou de sémantique naturelle (due à Kahn).





# L'appel par nom en sémantique à grandes étapes

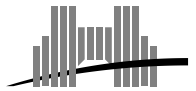
On se fonde sur le calcul de substitution explicite $\lambda\sigma_w$.

On verra que dans les applications

on réduit d'abord le corps (partie gauche)

avant le paramètre (partie droite).





# L'appel par nom en sémantique à grandes étapes

$$\frac{M[U \cdot s] \downarrow_n V}{(\lambda M)[s]\, U \downarrow_n V}$$

$$\frac{M_1[s] \downarrow_n V \qquad V\ M_2[s] \downarrow_n W}{(M_1\ M_2)[s] \downarrow_n W}$$

$$\frac{U \downarrow_n U'}{\underline{0}[U \cdot s] \downarrow_n U'} \qquad \frac{\underline{n}[s] \downarrow_n U'}{\underline{n+1}[U \cdot s] \downarrow_n U'}$$

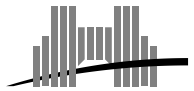
$$(\lambda M)[s] \downarrow_n (\lambda M)[s]$$

# L'appel par nom en sémantique à grandes étapes

$$\dfrac{\dfrac{}{(\lambda 1)[((\lambda 00)\ (\lambda 00))[id] \cdot id] \downarrow_n (\lambda 1)[((\lambda 00)\ (\lambda 00))[id] \cdot id]} \qquad \dfrac{}{(\lambda\lambda 1)[id] \downarrow_n (\lambda\lambda 1)[id]} \qquad \dfrac{}{(\lambda\lambda 1)[id]\ ((\lambda 00)\ (\lambda 00))[id] \downarrow_n (\lambda 1)[((\lambda 00)\ (\lambda 00))[id] \cdot id]}$$

$$(\lambda\lambda 1)\ ((\lambda 00)\ (\lambda 00))[id] \downarrow_n (\lambda 1)[(\lambda 00)\ (\lambda 00)[id] \cdot id]$$





# L'appel par valeur en sémantique à grandes étapes

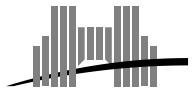
Dans la suite une valeur est une clôture contenant une abstraction dans sa partie terme,

c'est-à-dire de la forme $(\lambda x.M)[s]$.

On réduit d'abord le paramètre (partie droite),

avant le corps (partie gauche).





# L'appel par valeur en sémantique à grandes étapes

$$\frac{M[U \cdot s] \downarrow_v V}{(\lambda M)[s]\, U \downarrow_v V}$$

$$\frac{M_2[s] \downarrow_v V \qquad M_1[s]\, V \downarrow_v W \qquad M_2[s] \text{ n'est pas une valeur}}{(M_1\ M_2)[s] \downarrow_v W}$$

$$\frac{M_1[s] \downarrow_v V \qquad V\ M_2[s] \downarrow_v W \qquad M_2[s] \text{ est une valeur}}{(M_1\ M_2)[s] \downarrow_v W}$$

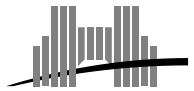
$$\frac{U \downarrow_v U'}{\underline{0}[U \cdot s] \downarrow_v U'} \qquad \frac{\underline{n}[s] \downarrow_v U'}{\underline{n+1}[U \cdot s] \downarrow_v U'}$$

$$(\lambda M)[s] \downarrow_v (\lambda M)[s]$$

# L'appel par valeur en sémantique à grandes étapes

$$\dfrac{\dfrac{\dfrac{(\lambda 00)[id] \downarrow_v (\lambda 00)[id]}{0[(\lambda 00)[id].id] \downarrow_v (\lambda 00)[id]}}{((\lambda 0)\ (\lambda 00))[id] \downarrow_v (\lambda 00)[id]} \qquad \dfrac{(\lambda 1)[(\lambda 00)[id] \cdot id] \downarrow_v (\lambda 1)[(\lambda 00)[id] \cdot id]}{(\lambda\lambda 1)[id]\ (\lambda 00)[id] \downarrow_v (\lambda 1)[(\lambda 00)[id] \cdot id]}}{(\lambda\lambda 1)\ ((\lambda 0)\ (\lambda 00))[id] \downarrow_v (\lambda 1)[(\lambda 00)[id] \cdot id]}$$



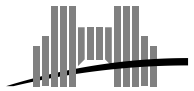


# L'appel par valeur en sémantique à grandes étapes

Exercice :

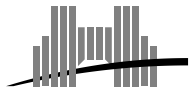
1. Dans quel ordre réduit-on les arguments dans cette définition de l'appel par valeur ?
2. Comment changer les règles pour obtenir un autre ordre d'évaluation des arguments ?





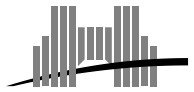
# *La sémantique par continuation*





# *Les origines et les motivations*





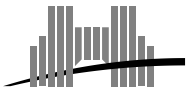
# Les origines

La méthode des continuations a été introduite par Strachey et Wadsworth comme un outil pour formaliser la notion de flot de contrôle dans les langages de programmation.

Dans cette méthode un terme est évalué dans un contexte qui représente le «reste du calcul».

Les termes, sont à proprement parler évalués de gauche à droite, mais surtout en manipulant ce contexte.



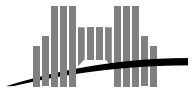


# Les origines

Il s'agit d'une technique fondée sur les fonctions d'ordre supérieur, qui peut-être utilisée

– directement par le programmeur,
– dans la transformation de programme,
– ou dans l'optimisation de compilateur.





# Les origines

Si le terme implique l'évaluation d'un sous-terme,

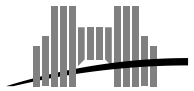
**alors** ce sous-terme est évalué dans un nouveau contexte qui évalue le reste du terme

**et alors** il prend en compte l'ancien contexte.

Si le terme peut être évalué immédiatement alors sa valeur est passée au contexte.

Un tel contexte est appelé une continuation.





# Un exemple

Soit la fonction OCAML

```
let f(x,y) = y +. 3. *. x +. (2. +. 1./.x)/.(1. +. 2./.x)
```

On peut considérer le calcul qui se fait après la division `1./.x`

et qui est donné par

```
let continue quotient =
     avant +. (2. +. quotient)/.(1. +. 2./.x)
```

ou `avant` est donné par

```
let avant = y +. 3. *. x
```

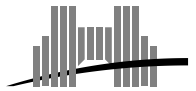
et où `quotient` est le résultat d'une division qui peut conduire à une erreur.

# Un exemple

On souhaite «naturellement» que dans le cas où `x` est égal à zéro, la fonction `f(x,y)` rende le résultat `y +. 3. *. x +. 0.5` :

```
let f(x,y) =
  let divide(numerateur, denominateur, continuation, autre_valeur) =
    if denominateur > 0.00001
    then continuation(numerateur /. denominateur)
    else autre_valeur
  in let avant = y +. 3. *. x
  in let continue quotient =
              avant +. (2. +. quotient) /. (1. +. 2. /. x)
  in divide(1., x, continue, y +. 3. *. x +. 0.5)
```





# IMP revisité

Si on reprend la sémantique de IMP avec continuation, il faut ajouter une structure de données **Réponses**.

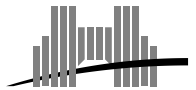
$\mathcal{C}'[\![c]\!] : (\Sigma \rightharpoonup \textbf{Réponses}) \rightharpoonup \Sigma \rightharpoonup \textbf{Réponses}$

On a une nouvelle définition de $\mathcal{C}'[\![\ ]\!]$ sur la composition séquentielle.

$$\mathcal{C}'[\![c_0; c_1]\!] \quad = \quad \mathcal{C}'[\![c_0]\!] \circ \mathcal{C}'[\![c_1]\!]$$

$\mathcal{C}'[\![c_1]\!]$ transforme la continuation, puis c'est au tour de $\mathcal{C}'[\![c_0]\!]$.





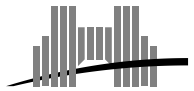
# Les continuations et la sémantique

La sémantique par continuation explique des phénomènes que la sémantique directe ne peut pas expliquer,

comme les sauts dans les langages impératifs.

Elle rend compte et permet de définir des implantations efficaces et souples.





# Les continuations et la sémantique

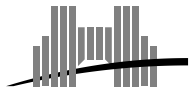
La sémantique par continuation est aussi appelée style de passage de continuations, en anglais continuation passing style, ou pour faire court CPS.

En lambda-calcul

Elle fait une traduction du lambda-calcul vers lui-même.

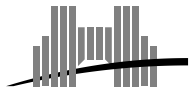
C'est même une véritable compilation du lambda-calcul dans le lambda-calcul.





# *Un résultat d'équivalence*





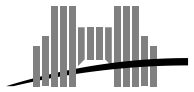
# Objectifs

Dans ce qui suit

– on va définir la sémantique par continuations,

– on va montrer l'équivalence de la sémantique par continuation avec la sémantique directe.



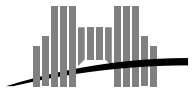


# La source d'inspiration

Notre inspiration vient d'un article d'Albert Meyer et Mitchell Wand, intitulé, *Continuation Semantics in Typed Lambda-Calculi*

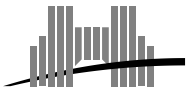
in Logic of Programs, LNCS 193, Springer (1985), pp :219-224.





# *La transformation*





# Le langage

Le lambda calcul est simplement typé.

Les types sont alors

– des types de base $\sigma_1, \sigma_2, \ldots$

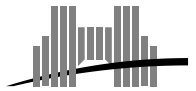
– ou des types fonctionnels $\alpha \longrightarrow \beta$.

Il y a un type particulier $o$ (pas nécessairement de base)

qui est le type des réponses.

Ce sont les seuls types.





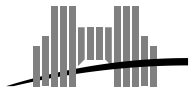
# Interprétation des types

La sémantique par continuation manipule des représentations d'objets qui apparaissent dans la sémantique directe.

On assigne à chaque type $\alpha$ un type $\alpha'$ des représentations des objets de type $\alpha$.

Les types de base sont représentés par eux mêmes.





# Interprétation des types

A une fonction de type $\alpha \to \beta$, nous faisons correspondre dans la sémantique par continuation une fonction qui prend deux arguments :

– une représentation de $\alpha$,

– et une continuation qui attend une représentation de $\beta$.

Munie de cette information la fonction calcule une réponse.

Ainsi nous avons :

$$\begin{aligned} \sigma' &\triangleq \sigma \\ (\alpha \to \beta)' &\triangleq \alpha' \to (\beta' \to o) \to o \end{aligned}$$

# La transformation

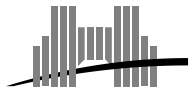
Pour chaque terme $M$ de type $\alpha$, nous construisons un terme $\overline{M}$ de type $(\alpha' \to o) \to o$ comme suit :

$$\begin{aligned} \overline{x} &\triangleq \lambda\kappa.\kappa x \\ \overline{\lambda x.M} &\triangleq \lambda\kappa.\kappa(\lambda x\overline{M}) \\ \overline{M\ N} &\triangleq \lambda\kappa.\overline{M}(\lambda m.\overline{N}(\lambda n.mn\kappa)) \end{aligned}$$

Exercice : J'ai enlevé les annotations de types pour la clarté.

Rétablissez les.





# La transformation

Montrons que si $M : \alpha$ alors $\overline{M} : (\alpha' \to o) \to o$

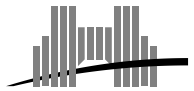
c'est-à-dire :

$$x : \alpha \quad \Rightarrow \quad \overline{x} : (\alpha' \to o) \to o$$

$$\lambda x.M : \alpha \to \beta \quad \Rightarrow \quad \overline{\lambda x.M} : ((\alpha \to \beta)' \to o) \to o$$

$$M\ N : \beta \quad \text{avec } M : \alpha \to \beta \text{ et } N : \alpha \quad \Rightarrow \quad \overline{M\ N} : (\beta' \to o) \to o$$





# La transformation

En bref,

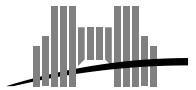
– Si on a une variable, on envoie le résultat à la continuation $\kappa$.

  $\kappa : \alpha' \to o$ et $\lambda\kappa.\kappa x : (\alpha' \to o) \to o$

– Si on a une abstraction, on fournit à la continuation une fonction appropriée.

  – $\lambda x.M : \alpha \to \beta$,
  – $M : \beta$,
  – $\lambda x.\overline{M} : \alpha' \to (\beta' \to o) \to o \equiv (\alpha \to \beta)'$,
  – $\kappa : (\alpha \to \beta)' \to o$,
  – $\kappa(\lambda x.\overline{M}) : o$,
  – $\lambda\kappa.\kappa(\lambda x.\overline{M}) : ((\alpha \to \beta)' \to o) \to o$,





– Si on a une application, on évalue l'opérateur dans une continuation qui consiste à évaluer l'opérande dans une continuation qui elle-même consiste à appliquer la valeur de l'opérateur à la valeur de l'opérande et à la continuation courante.

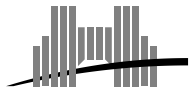
cela s'accorde joliment avec la définition de $(_)'$ : Si $M$ est de type $\alpha \to \beta$, alors

⋆ $m$ doit être de type $(\alpha \to \beta)' \triangleq \alpha' \to (\beta' \to o) \to o$,

⋆ $n$ doit être de type $\alpha'$,

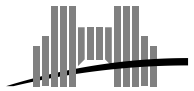
⋆ $\kappa$ doit être de type $\beta' \to o$.





- $M : \alpha \to \beta$,
- $N : \alpha$,
- $\overline{M} : ((\alpha \to \beta)' \to o) \to o$
  $\equiv ((\alpha' \to (\beta' \to o) \to o) \to o) \to o$
- $\overline{N} : (\alpha' \to o) \to o$,
- $m\ n\ \kappa : o$,
- $\lambda n.m\ n\ \kappa : \alpha' \to o$,
- $\overline{N}(\lambda n.m\ n\ \kappa) : o$,
- $\lambda m.\overline{N}(\lambda n.m\ n\ \kappa) : (\alpha \to \beta)' \to o$
  $\equiv (\alpha' \to (\beta' \to o) \to o) \to o$
- $\overline{M}(\lambda m.\overline{N}(\lambda n.m\ n\ \kappa)) : o$,
- $\lambda \kappa.\overline{M}(\lambda m.\overline{N}(\lambda n.m\ n\ \kappa)) : (\beta' \to o) \to o$





# La transformation

Une propriété intéressante de $\overline{M}$ est qu'elle est récursive terminale (tail-recursive).

Les sous-termes de la forme $\overline{M}$ qui apparaissent dans un terme $P\,Q$ apparaissent toujours dans $P$.

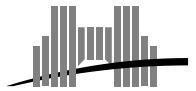
Cette propriété se préserve par $\beta$-réduction.

Il y a au plus un redex sous chaque lambda.

Ainsi (sur $\overline{M}$) la réduction par appel par valeur coïncide avec la réduction par appel par nom.

Les implantations sur les machines standards sont plus faciles.



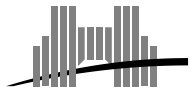


# La transformation

Exercice :

Montrer que $\overline{\lambda x.M} = \overline{[x]M}$ où $[x]M$ est l'«abstraction-crochet» définie sur les combinateurs (voir mon cours de logique sur la logique combinatoire).



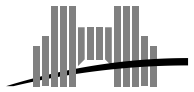


# La transformation

L'évaluation d'un terme se fait en l'appliquant à la continuation identité.

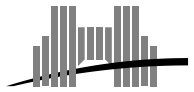
C'est-à-dire en l'appliquant à la continuation qui rend comme réponse la valeur qu'on lui a donnée.





# *La correction*



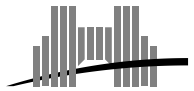


# Question

Comment définir la transformation inverse ?

$$\overline{M} \quad \rightarrow \quad M$$





# Rétraction

On dit que $\alpha$ est un retract de $\beta$ et on écrit $\alpha \lhd \beta$

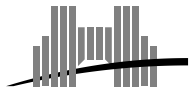
s'il existe des applications lambda-définissables

$i : \alpha \to \beta$ et $j : \beta \to \alpha$

telles que $j \circ i$ soit l'identité sur $\alpha$.

Le couple $(i, j)$ forme la rétraction.

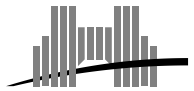




## Théorème

Si $\sigma \lhd o$ pour chaque type de base $\sigma$, alors $\alpha \lhd (\alpha \to o) \to o$.





# Théorème

Si $\sigma \lhd o$ pour chaque type de base $\sigma$, alors $\alpha \lhd (\alpha \to o) \to o$.

Démonstration

Définissons $I_\alpha : \alpha \to (\alpha \to o) \to o$ par $I_\alpha \triangleq \lambda x \kappa . \kappa x$.

Pour définir l'application inverse, observons que $\alpha$ doit être de la forme $\alpha_1 \to \alpha_2 \to \ldots \to \gamma$ où $\gamma$ est un type de base.

Soit $r : \gamma \to o$ l'injection de la rétraction de $\gamma$ vers $o$ avec comme inverse à gauche $l$.

Alors nous définissons

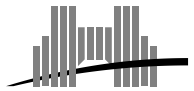
$$J_\alpha \triangleq \lambda u : (\alpha \to o) \to o . \lambda x_1 : \alpha_1 \ldots \lambda x_n : \alpha_n . l(u(\lambda a : \alpha . r(a x_1 \ldots x_n))).$$

On voit que $J_\alpha \circ I_\alpha = id_\alpha$.

En effet

$$
\begin{aligned}
J_\alpha(I_\alpha(a')) &\rightarrow \lambda x_1 ... \lambda x_n . l(I_\alpha(a')(\lambda a . r(a x_1 ... x_n))) \\
&= \lambda x_1 ... \lambda x_n . l((\lambda x \kappa . \kappa x)(a')(\lambda a . r(a x_1 ... x_n))) \\
&\rightarrow \lambda x_1 ... \lambda x_n . l((\lambda \kappa . \kappa a')(\lambda a . r(a x_1 ... x_n))) \\
&\rightarrow \lambda x_1 ... \lambda x_n . l((\lambda a . r(a x_1 ... x_n))\, a') \\
&\rightarrow \lambda x_1 ... \lambda x_n . l(r(a' x_1 ... x_n)) \\
&= \lambda x_1 ... \lambda x_n . a' x_1 ... x_n \\
&\underset{\eta}{\rightarrow} a'
\end{aligned}
$$





# Pseudo-application

Notons que $I_\alpha$ n'est pas le combinateur habituel $\lambda x : \alpha . x$.

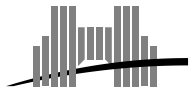
Notons aussi qu'on peut «appliquer»

– un élément $m$ de type $(\alpha \to \beta)'$
– à un élément $n$ de type $\alpha'$
– et obtenir un élément de type $\beta'$
– en écrivant $J_{\beta'}(m\ n)$.

Nous utiliserons la notation $m \bullet n$ pour $J_{\beta'}(m\ n)$

et nous parlerons de pseudo-application.

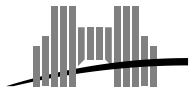




## Théorème

Pour tout type $\alpha$, $\alpha \lhd \alpha'$.





## Théorème

Pour tout type $\alpha$, $\alpha \lhd \alpha'$.

### Démonstration

Pour les types de base la rétraction est l'identité,

c'est-à-dire $i_\sigma \triangleq id_\sigma$ et $j_\sigma \triangleq id_\sigma$.

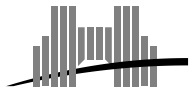
Pour les types fonctionnels, on définit

$$\begin{aligned} i_{\alpha\to\beta} &\triangleq \lambda f : \alpha \to \beta . I_{\beta'} \circ i_\beta \circ f \circ j_\alpha \\ j_{\alpha\to\beta} &\triangleq \lambda f' : (\alpha \to \beta)' . j_\beta \circ J_{\beta'} \circ f' \circ i_\alpha . \end{aligned}$$

On a bien

$$i_{\alpha\to\beta} \quad : \quad (\alpha \to \beta) \to (\alpha \to \beta)'$$
$$j_{\alpha\to\beta} \quad : \quad (\alpha \to \beta)' \to (\alpha \to \beta)$$



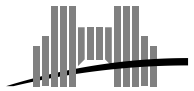


# Théorème principal

Soit $M$ un terme clos de type $\alpha$.

Alors $M = j_\alpha(J_{\alpha'}\overline{M})$.





# Théorème principal

Soit $M$ un terme clos de type $\alpha$.

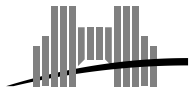
Alors $M = j_\alpha(J_{\alpha'}\overline{M})$.

Sa preuve nécessite d'introduire quelques concepts.

Vérifions cependant que les types fonctionnent.

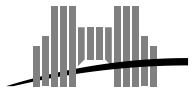
$$
\begin{array}{rcl}
M & : & \alpha \\
\overline{M} & : & \alpha' \to o \to o \\
J_{\alpha'} & : & (\alpha' \to o \to o) \to \alpha' \\
J_{\alpha'}\overline{M} & : & \alpha' \\
j_\alpha & : & \alpha' \to \alpha \\
j_\alpha(J_{\alpha'}\overline{M}) & : & \alpha
\end{array}
$$





# *Preuve du théorème principal*





# Interprétation d'un terme dans une valuation

Pour démontrer le théorème on a besoin de considérer aussi les termes avec des variables libres.

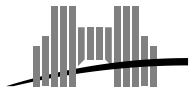
Si on dit comment instancier les variables, on sait interpréter les termes ouverts par des termes clos

On a besoin de «valuations».

Dans la suite, $\rho$ est une valuation

c'est-à-dire une fonction qui associe des termes clos à des variables.



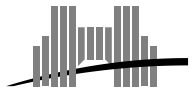


# Interprétation d'un terme dans une valuation

On suppose que les termes ont été traduits sous forme de combinateurs en utilisant l'abstraction-crochet.

| $M$ | $[\![M]\!]_\rho$ |
|---|---|
| $x$ | $\rho(x)$ |
| $K$ | $K$ |
| $S$ | $S$ |
| $P\,Q$ | $[\![P]\!]_\rho \bullet [\![Q]\!]_\rho$ |



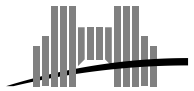


# Invariants concrets et prédicats acceptables

Pour pouvoir démontrer le théorème on a besoin de caractériser quels éléments de $\alpha'$ sont des représentants «légaux» d'éléments de $\alpha$..

C'est-à-dire qu'on doit trouver un invariant concret (i. e., expressible) approprié pour ce schéma de représentation.



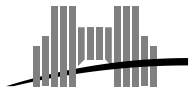


# Prédicats acceptables

$P$ est un prédicat acceptable si

1. Pour tous types $\alpha$ et $\beta$, ainsi que pour tout $x : \alpha$
   - $P_\alpha(i_\alpha x)$,
   - $P_{\alpha\to(\beta\to\alpha)}(J_{(\alpha\to(\beta\to\alpha))'}\overline{K})$,
   - $P_{(\alpha\to\beta\to\gamma)\to(\beta\to\gamma)\to(\alpha\to\gamma)}(J_{((\alpha\to\beta\to\gamma)\to(\beta\to\gamma)\to(\alpha\to\gamma))'}\overline{S})$.
2. Si $P_{\alpha\to\beta}(m)$ et $P_\alpha(n)$ , alors
   - $P_\beta(m \bullet n)$,
   - $(j_{\alpha\to\beta}\ m)(j_\alpha\ n) = j_\beta(m \bullet n)$,
   - $mn = I_{\beta'}(m \bullet n)$.





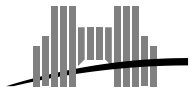
# Prédicats acceptables

La propriété 1. dit que les représentations canoniques et les images de combinateurs sont «légales».

La propriété 2. dit

- le prédicat est clos par pseudo-application,
- les applications conventionnelles sont des images homomorphes de pseudo-applications,
- l'application de représentation se comporte bien, elle envoie une valeur à sa continuation.





## Théorème

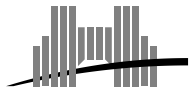
Soit $P$ un prédicat acceptable.

Pour tout $M$ et toute valuation $\rho$ tel que pour tout $x$, $P(\rho(x))$,

on a

$$[\![M]\!]_{j\circ\rho} \quad = \quad j(J([\![\overline{M}]\!]_\rho))$$





## Théorème

Soit $P$ un prédicat acceptable.

Pour tout $M$ et toute valuation $\rho$ tel que pour tout $x$, $P(\rho(x))$,

on a

$$[\![M]\!]_{j\circ\rho} \quad = \quad j(J([\![\overline{M}]\!]_\rho))$$

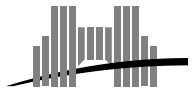
Démonstration

D'après l'exercice ci-dessus, on peut supposer que le terme est une composition des combinateurs $S$ et $K$.

Et alors, il suffit d'utiliser la définition et une hypothèse d'induction.

A faire complètement comme exercice.





# Commentaires sur le théorème

Que dit le théorème ?

Il dit que si $M$ est un terme clos,

$$[\![M]\!]_{j\circ\rho} \quad = \quad j(J([\![\overline{M}]\!]_\rho))$$

or dans ce cas $[\![M]\!]_{j\circ\rho} = M$ et $[\![\overline{M}]\!]_\rho = \overline{M}$

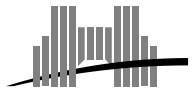
L'énoncé du théorème repose sur l'existence d'un prédicat acceptable.

Existe-t-il un prédicat acceptable ?

# Théorème

Il existe un prédicat acceptable.





## Théorème

Il existe un prédicat acceptable.

Esquisse de la démonstration

Le prédicat $R$

– qui contient $i\ x$ pour tout $x$

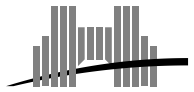
– qui contient $J\overline{S}$ et $J\overline{K}$

– et qui est clos par $\bullet$

est acceptable.

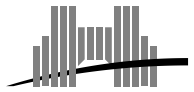
# *Les réseaux de Lamping*





# *L'optimalité de la réduction*





# Le problème de l'optimalité

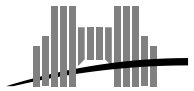
On revient au problème de la normalisation forte dans le lambda-calcul.

Comment réduire un terme à sa forme normale en effectuant le moins possible de réductions ?

En effet, si une contraction duplique un redex,
il faut d'abord réduire ce redex avant que la contraction ne le duplique.

Si une réduction fait disparaître le résidu de la contaction d'un redex
il n'aurait pas fallu contracter ce redex.





# Le problème de l'optimalité

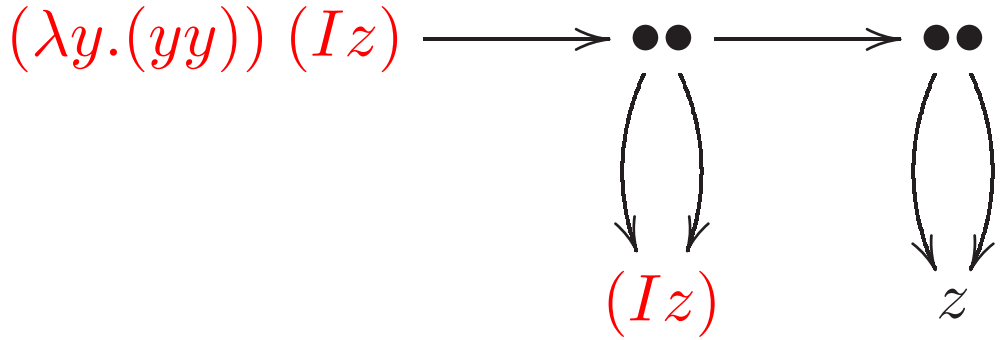
Considérons le terme $(\lambda y.(yy))\ (I\,z)$ où $I \equiv \lambda x.x$.

$$(\lambda y.(yy))\ (I\,z) \quad \rightarrow \quad (I\,z)\ (I\,z) \quad \rightarrow \quad z\ (I\,z) \quad \rightarrow \quad z\ z$$

$$(\lambda y.(yy))\ (I\,z) \quad \rightarrow \quad (I\,z)\ (I\,z) \quad \rightarrow \quad (I\,z)\ z \quad \rightarrow \quad z\ z$$

$$(\lambda y.(yy))\ (I\,z) \quad \rightarrow \quad (\lambda y.(yy))\ z \quad \rightarrow \quad z\ z$$

On peut faire du partage.



# Le problème de l'optimalité

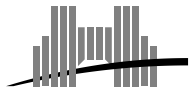
Réduire les redex internes n'est pas la solution.

Mais le partage associé à une réduction du redex le plus à gauche ne suffit pas.

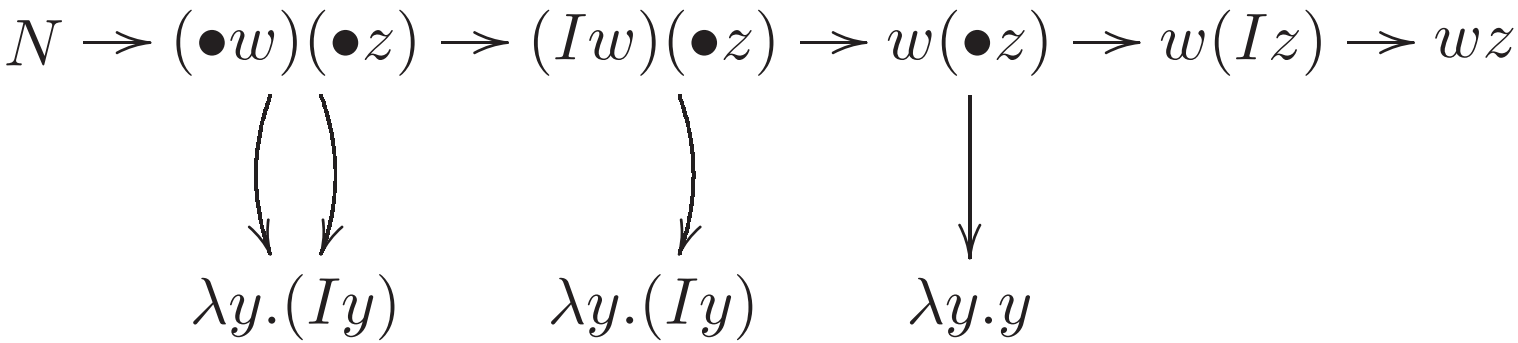
Considérons $N \equiv N_1\ N_2$ avec

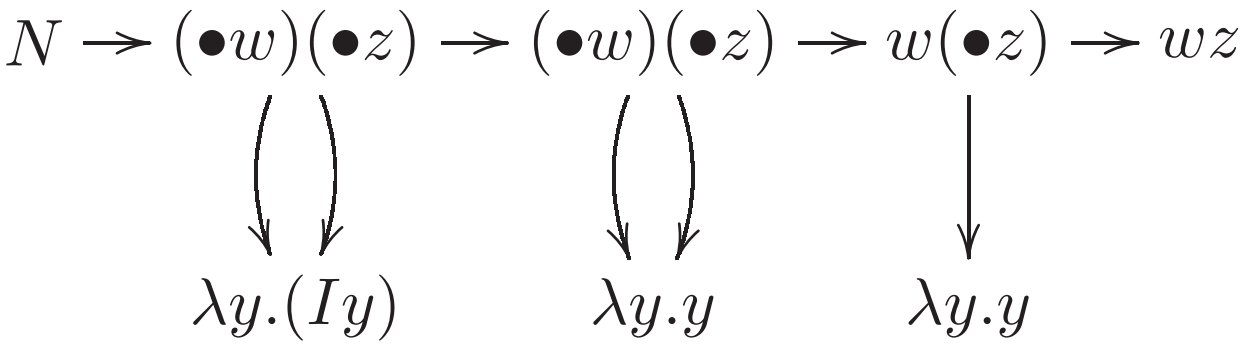
$$N_1 \equiv \lambda x.(xw\ (xz))$$
$$N_2 \equiv \lambda y.(Iy).$$





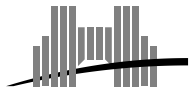
# Le problème de l'optimalité



# *Les réseaux d'interaction*

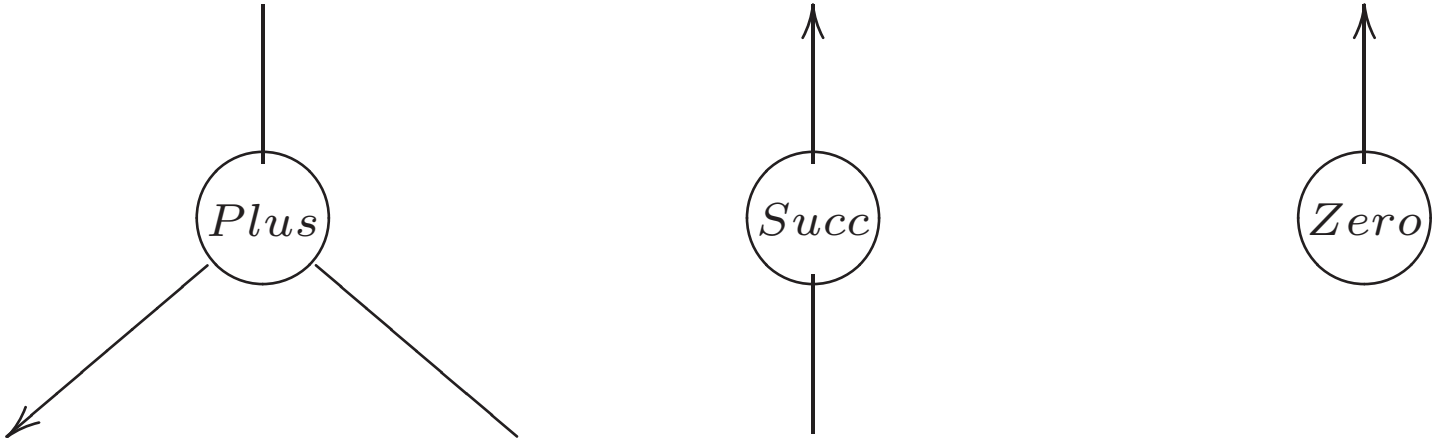




# Les réseaux d'interaction

Les réseaux d'interaction sont des manières graphiques de représenter les termes.

Sur les entiers



# Les réseaux d'interaction

Un réseau est un graphe (cyclique) non orienté dont les noeuds sont étiquetés.

Un ensemble fini et fixé de ports est attribué à chaque étiquette.

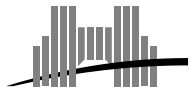
L'un des ports est appelé port principal.

Les noeuds sont connectés les uns aux autres par leurs ports.

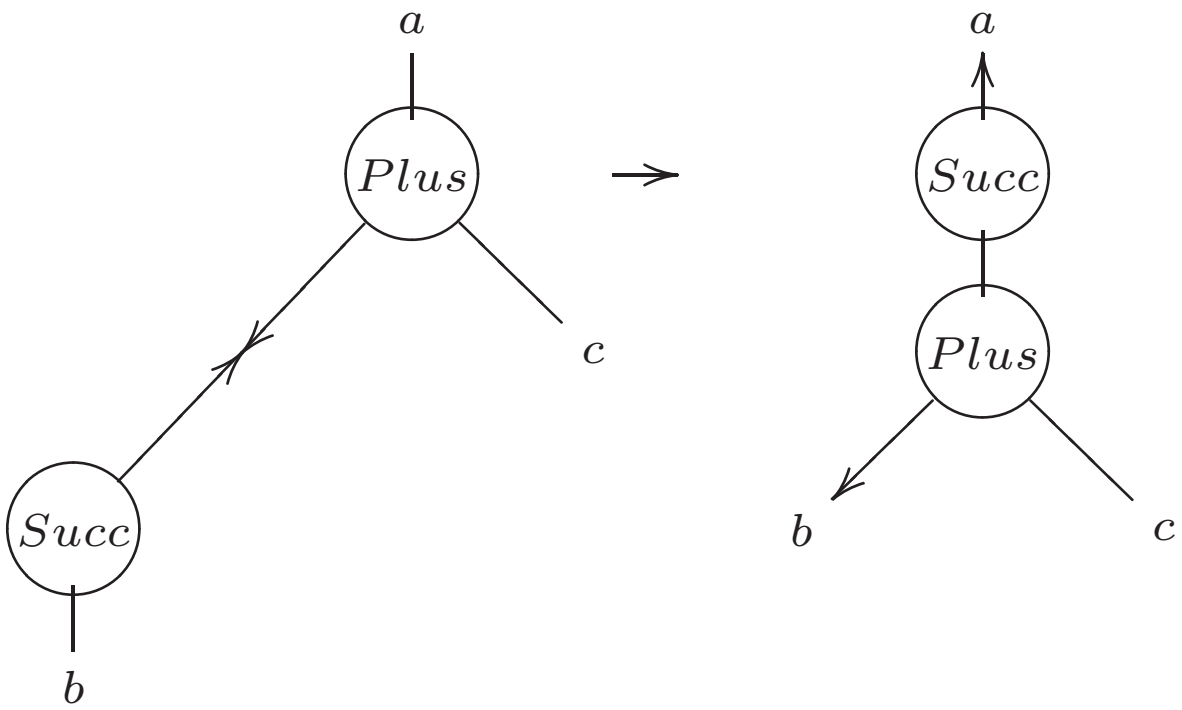
Chaque port est connecté à un et un seul autre port.

Si deux noeuds sont connectés par leur port principal, ils forment une paire active .

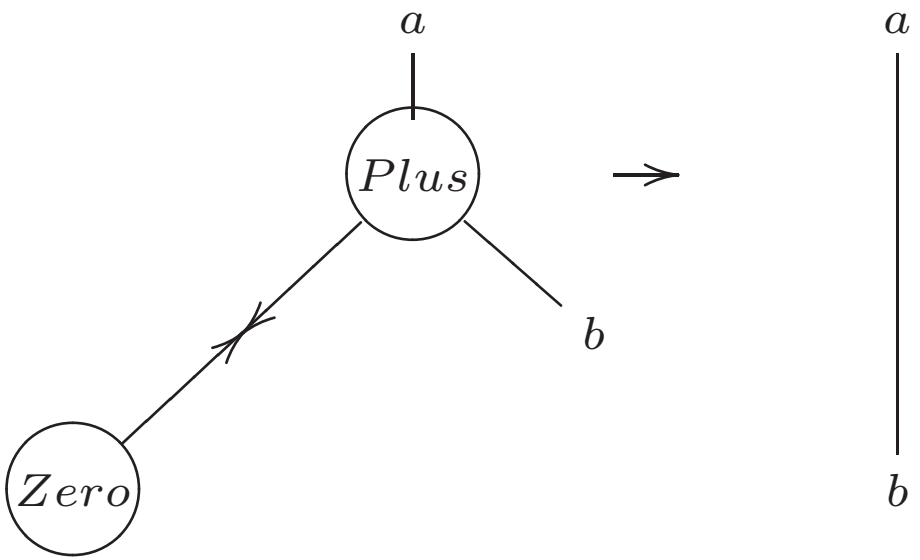




# Les réseaux d'interaction



# Les réseaux d'interaction



# Les réseaux d'interaction

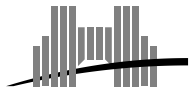
Une règle d'interaction est un couple de paires actives qui satisfont les conditions 1, 2 et 3.

Propriété 1, Linéarité : Dans chaque règle, chaque variable (associée à un port) a exactement deux occurrences : l'un dans le membre gauche et l'autre dans le membre droit.

Propriété 2, Non ambiguïté : Il y a au plus une règle pour chaque paire active.

Propriété 3, Optimisation : Les membres droits ne contiennent pas de paires actives.



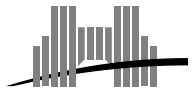


# Les réseaux d'interaction

Due à la localité des interactions et à leur non interférence avec une quelconque autre interaction, on voit que :

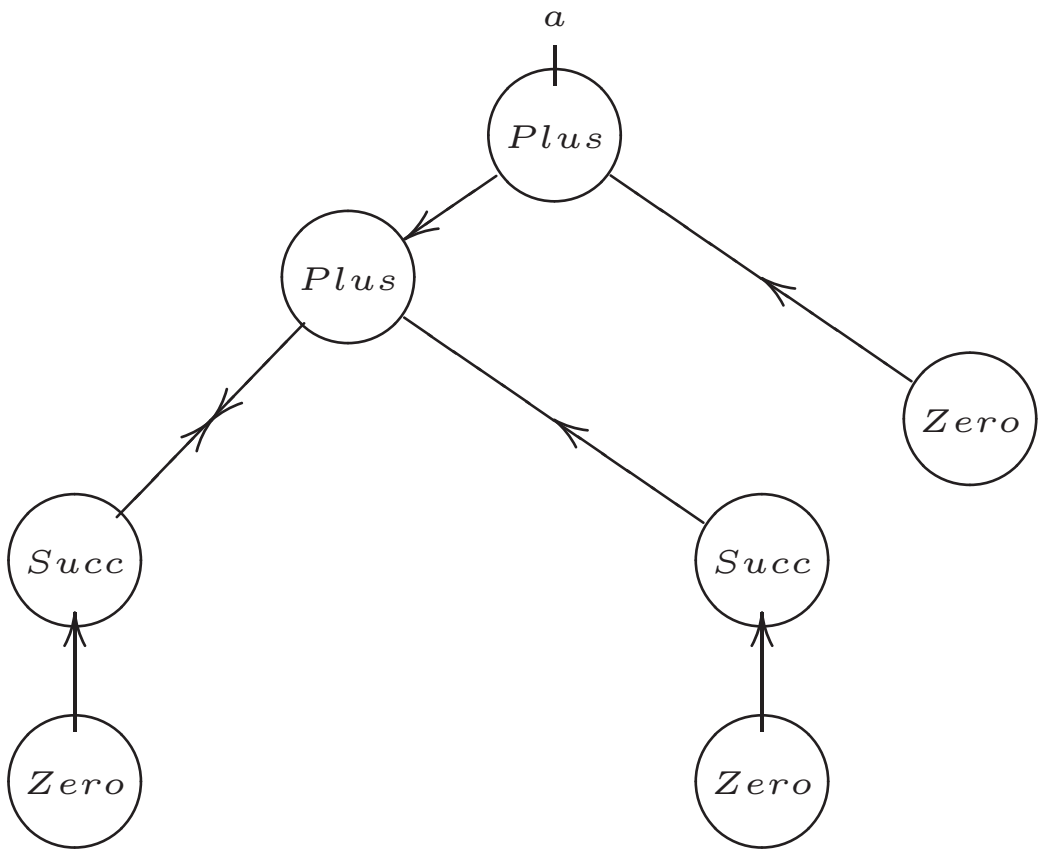
les réductions sont confluentes.





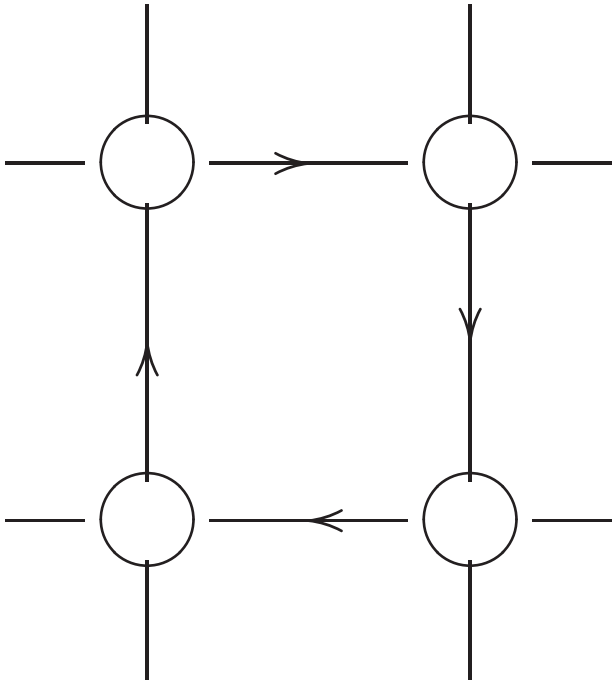
# Les réseaux d'interaction

Exercice : Réduire le réseau.



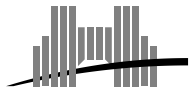
# Les interblocages

Il faut éviter les situations d'interblocage.



# *Le réseau d'interaction universel*





# Universalité des réseaux d'interaction

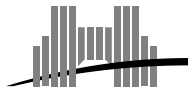
On peut coder par des réseaux d'interaction :

– les machines de Turing,

– les automates cellulaires,

– et, nous allons le voir, le $\lambda$-calcul.





# Universalité des réseaux d'interaction

On peut coder par des réseaux d'interaction :

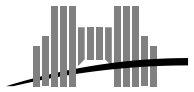
– les machines de Turing,

– les automates cellulaires,

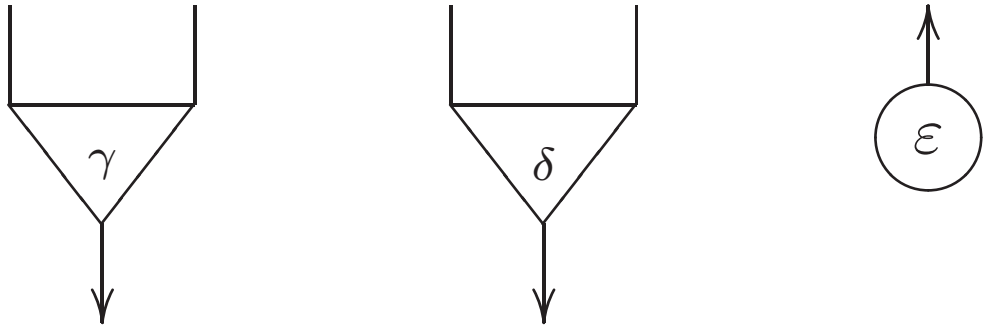
– et, nous allons le voir, le $\lambda$-calcul.

Nous allons maintenant définir un réseau universel.

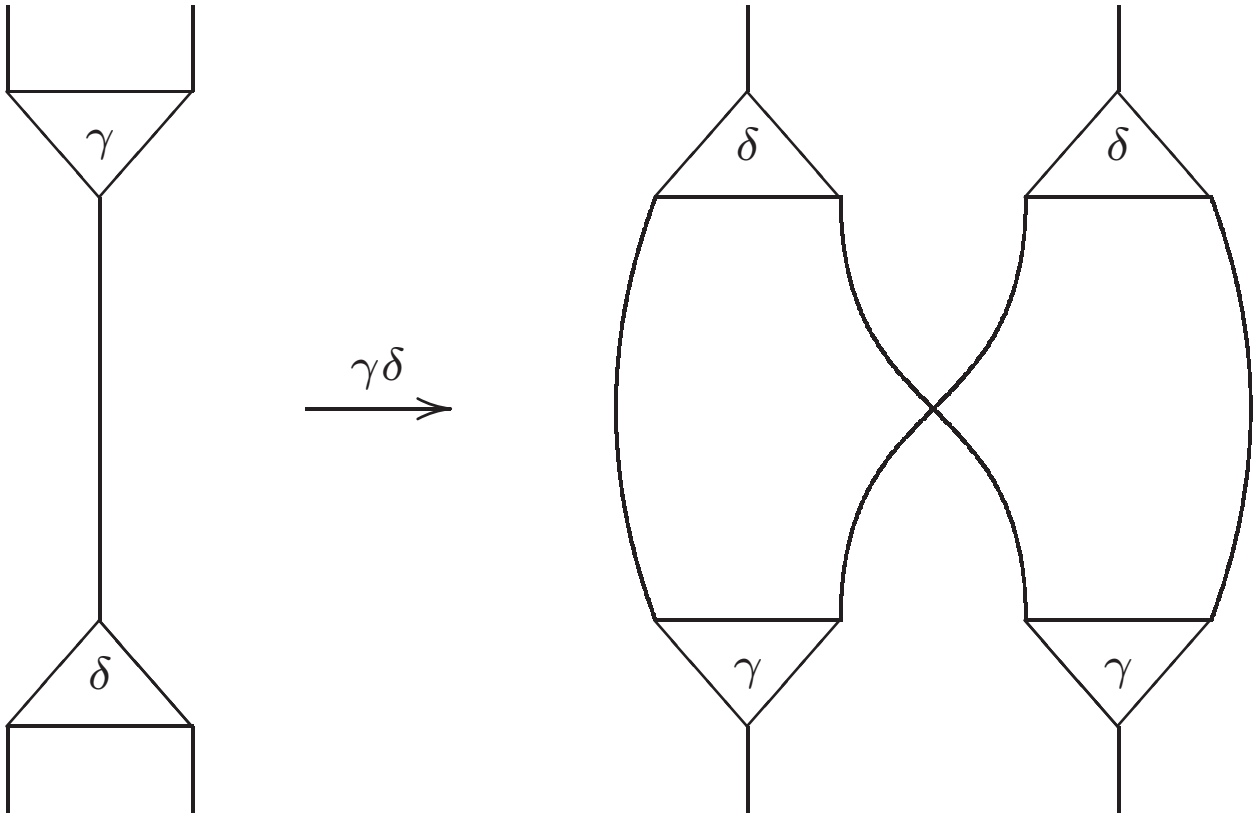




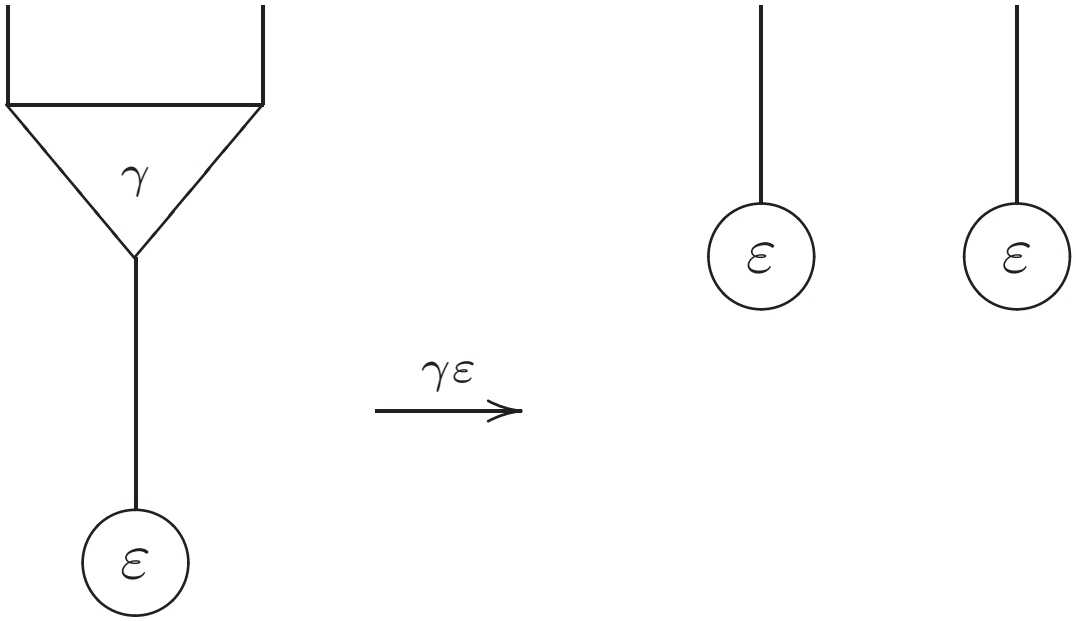
# Les noeuds



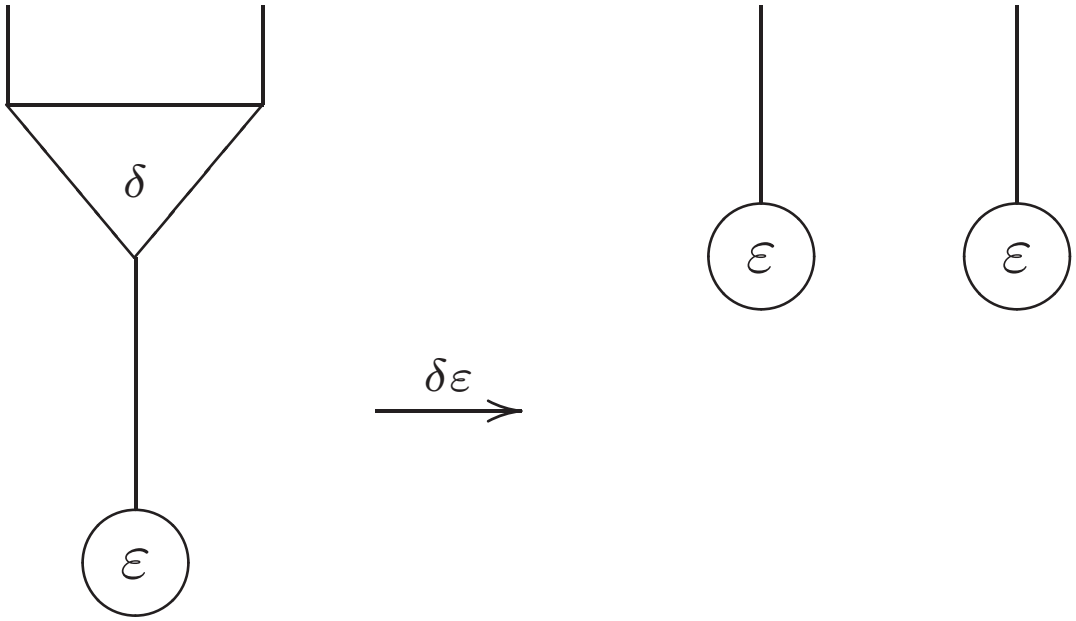
# Les règles



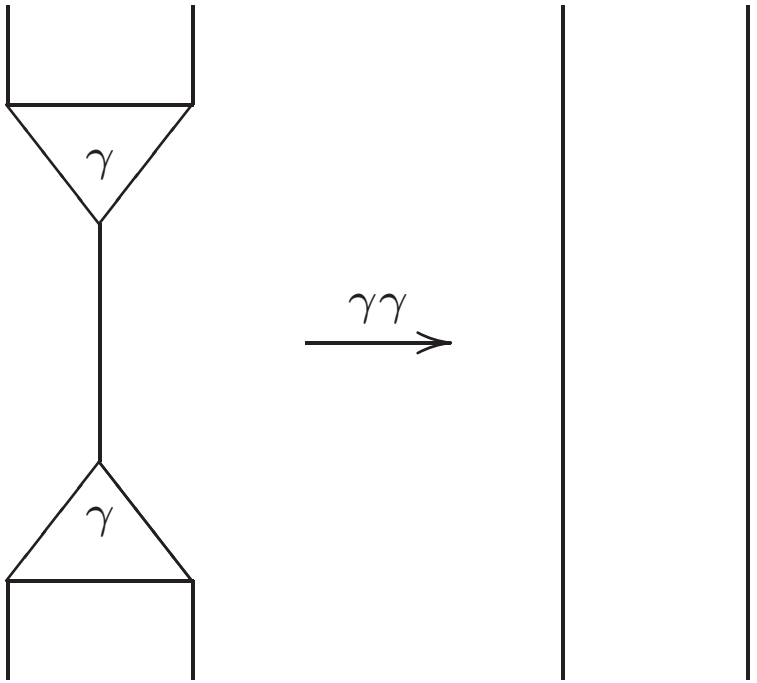
# Les règles



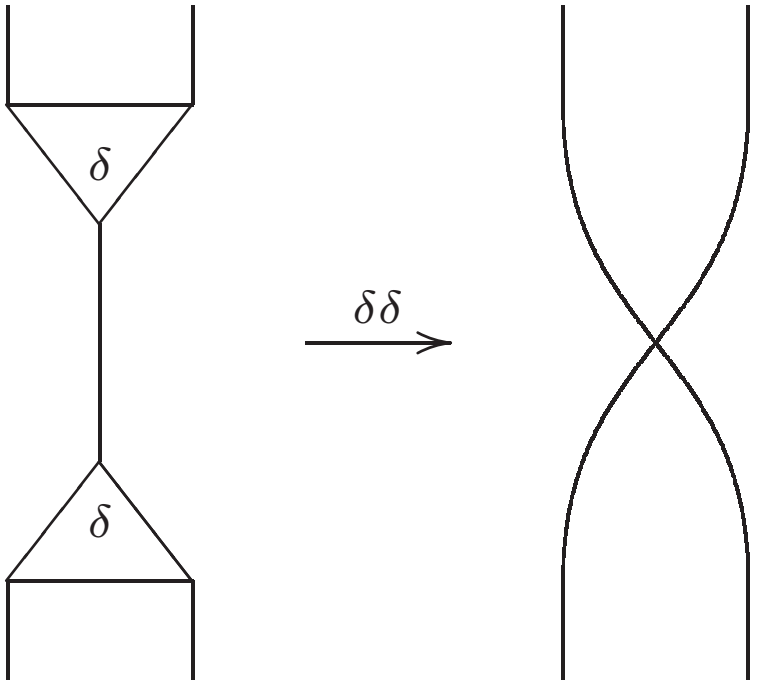
# Les règles



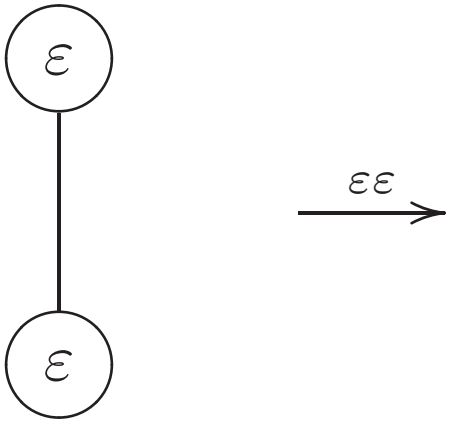
# Les règles



# Les règles



# Les règles

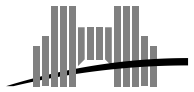


# Le résultat

Yves Lafont a montré que ce réseau (très simple !) est universel,

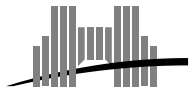
en ce sens que l'on peut y coder n'importe quel autre réseau.



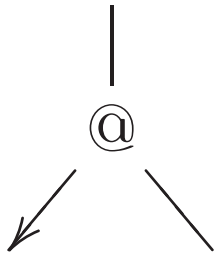


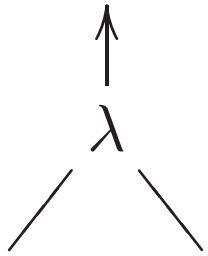
# *Le réseau d'interaction pour le lambda calcul*

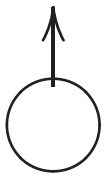


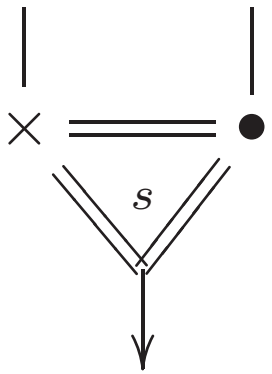


# Les noeuds

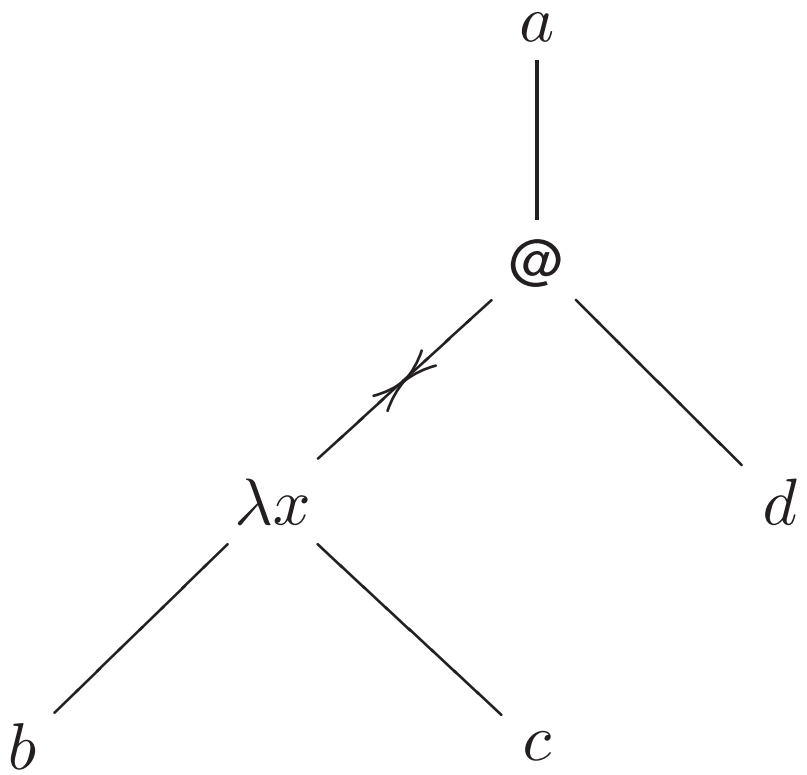


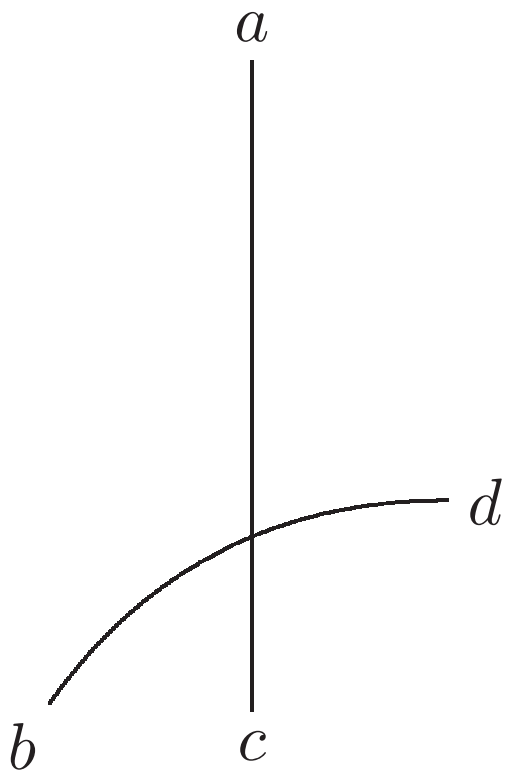



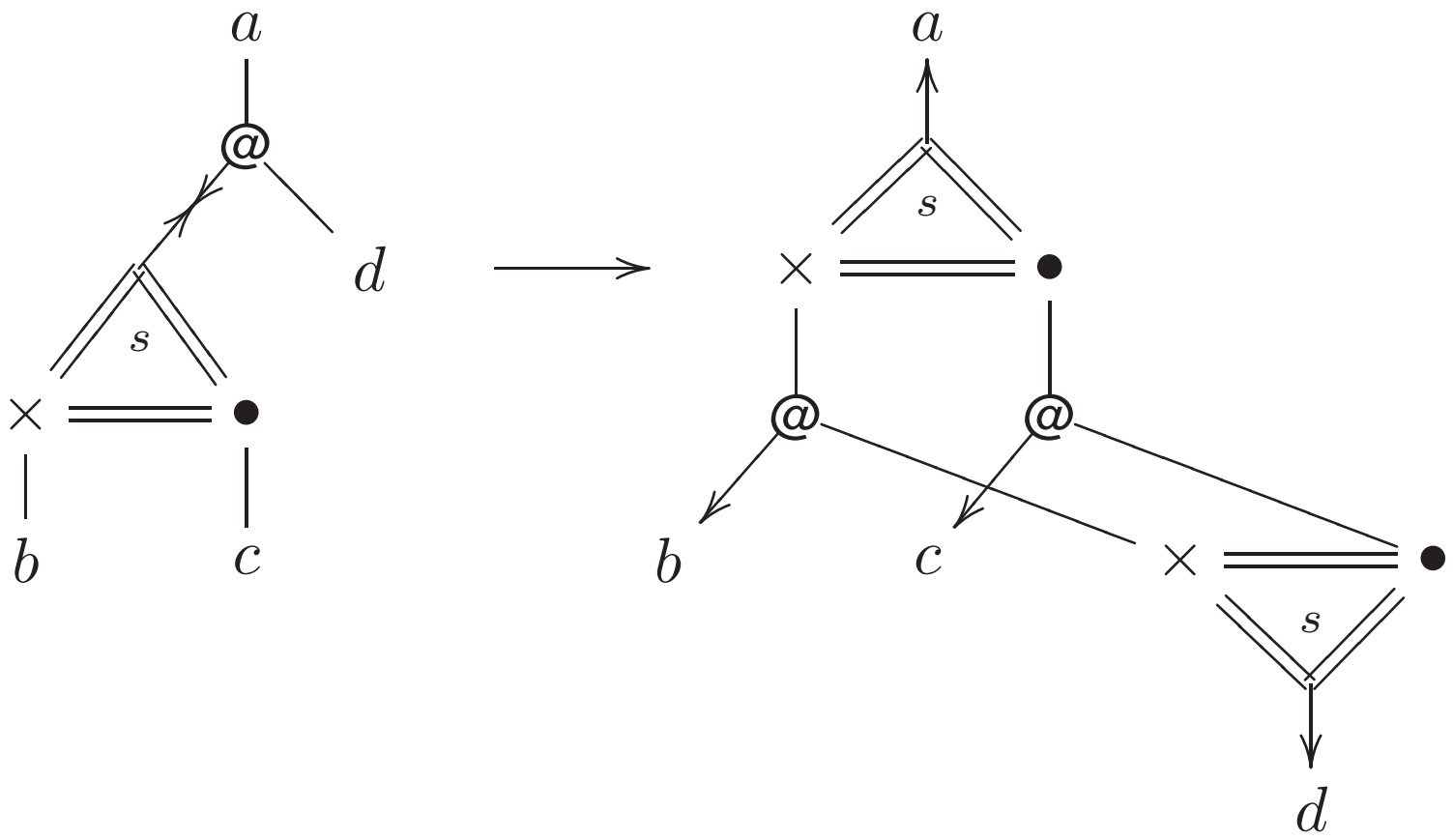


# L'élimination de $\beta$-redex

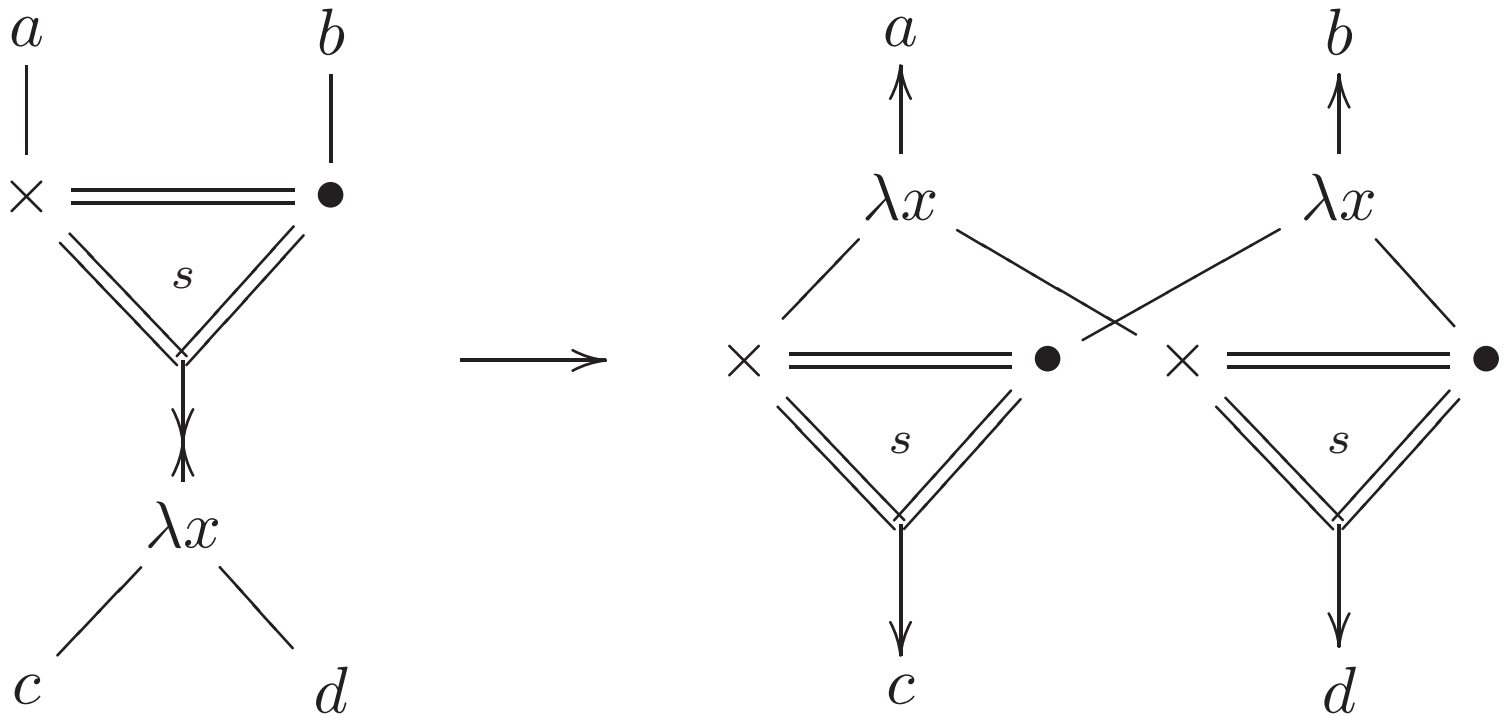


# @ contre fans

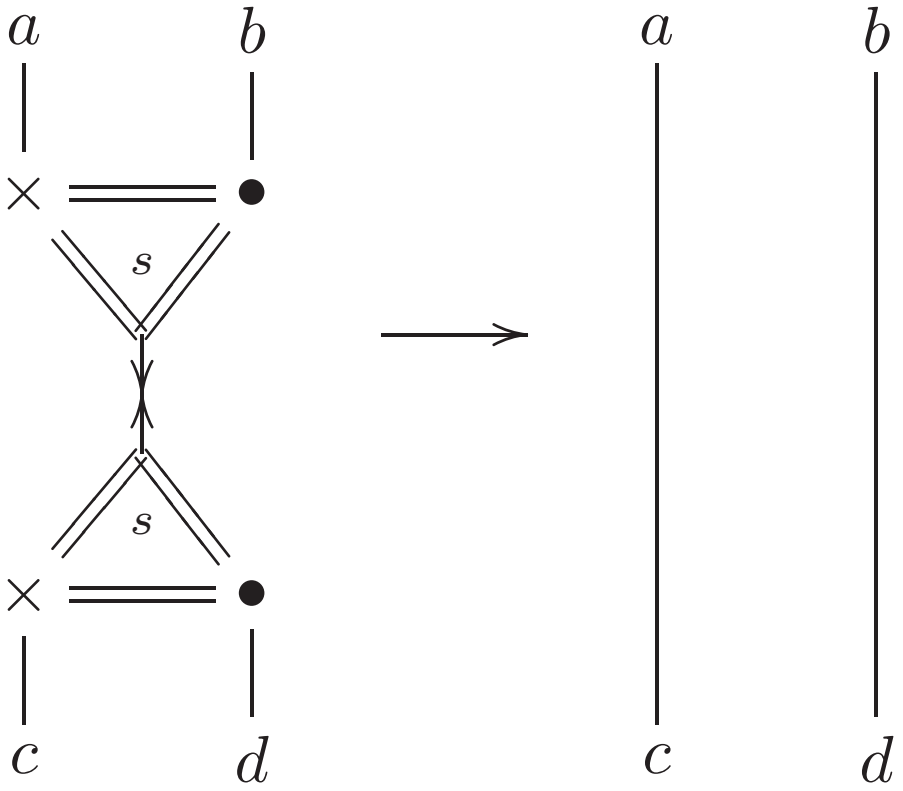


# $\lambda$ contre fans



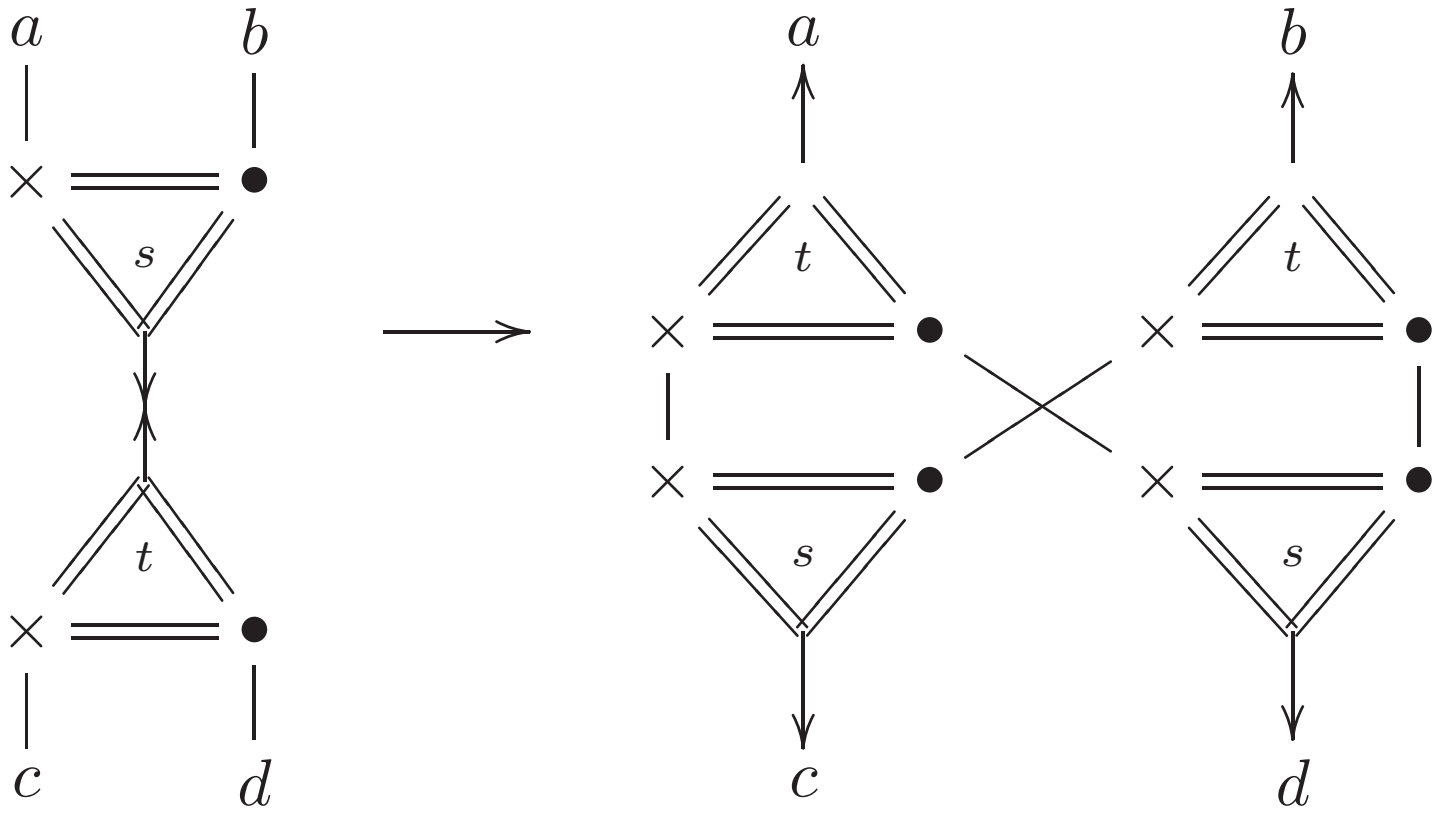
# Fans contre fans (même noms)

Les fans de même nom s'annihilent.



# Fans contre fans (noms différents)

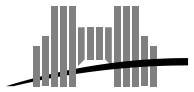
Les fans de noms différents se dupliquent.



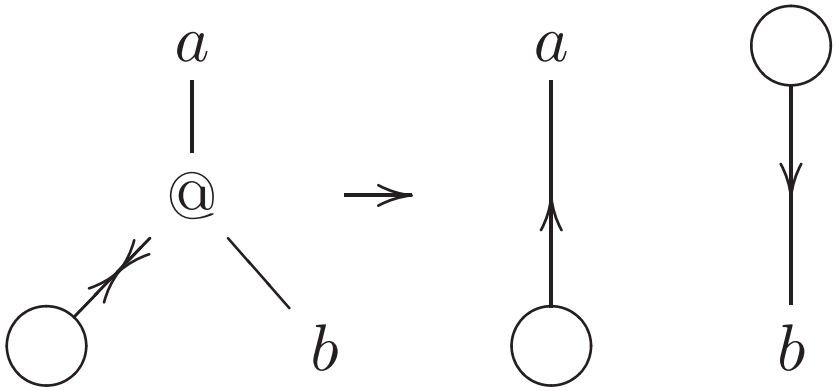
# Les noms des fans

Préserver correctement les noms des fans est un problème difficile qui n'est pas abordé dans ce cours.

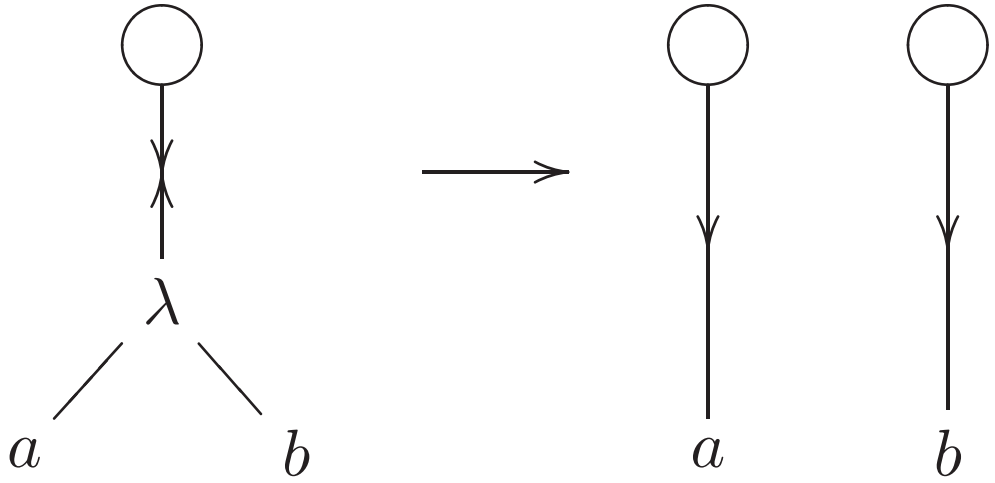




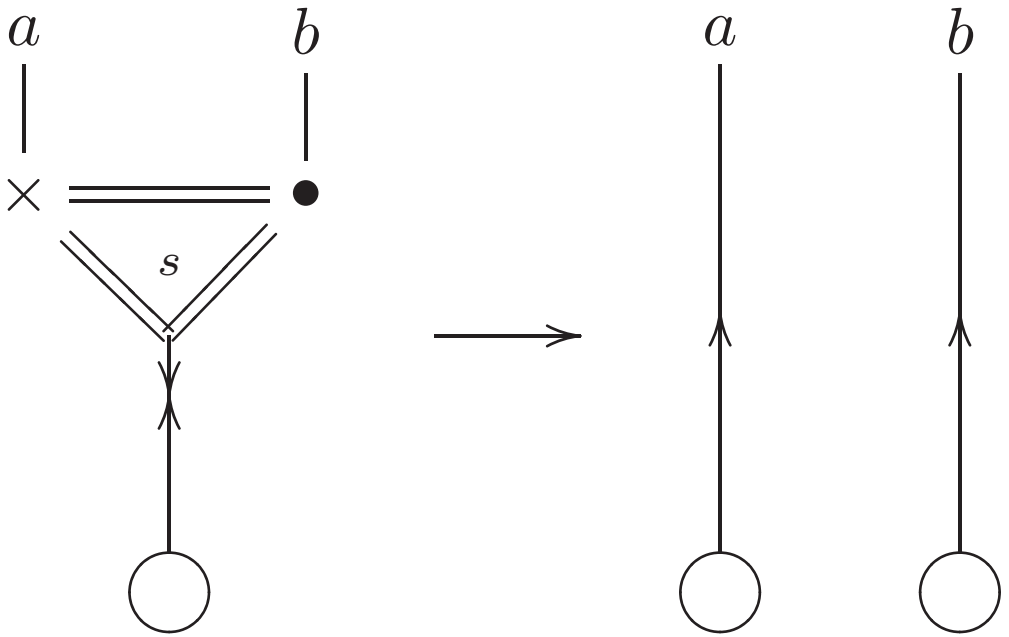
# Effaceur contre @



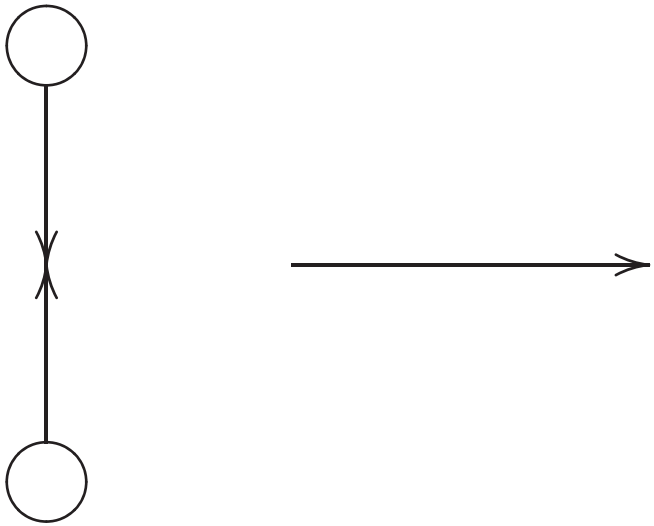
# Effaceur contre $\lambda$



# Effaceur contre fan

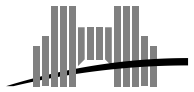


# Effaceur contre effaceur

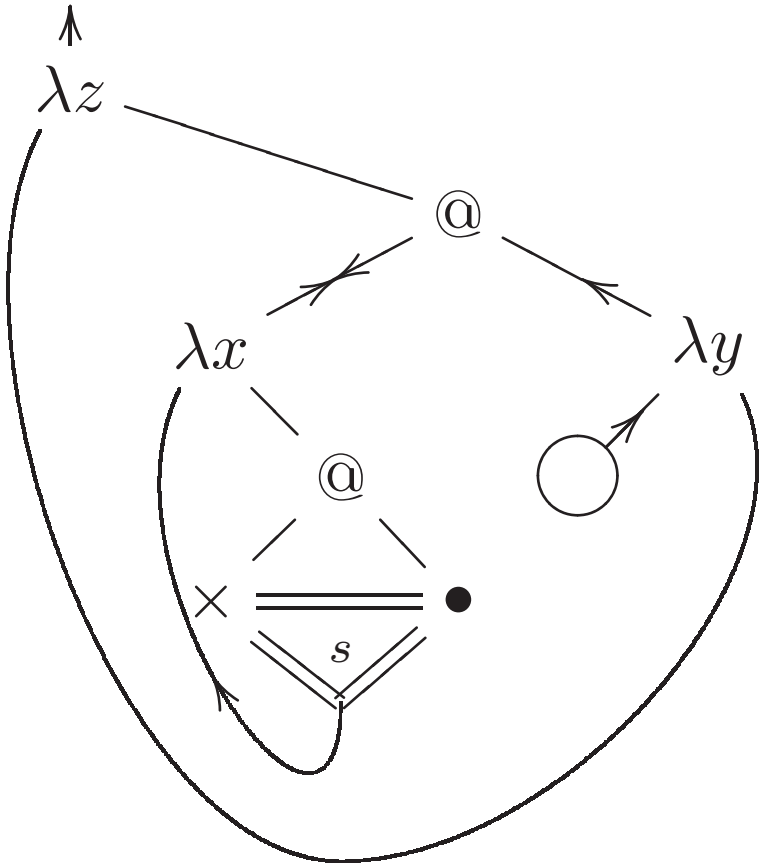


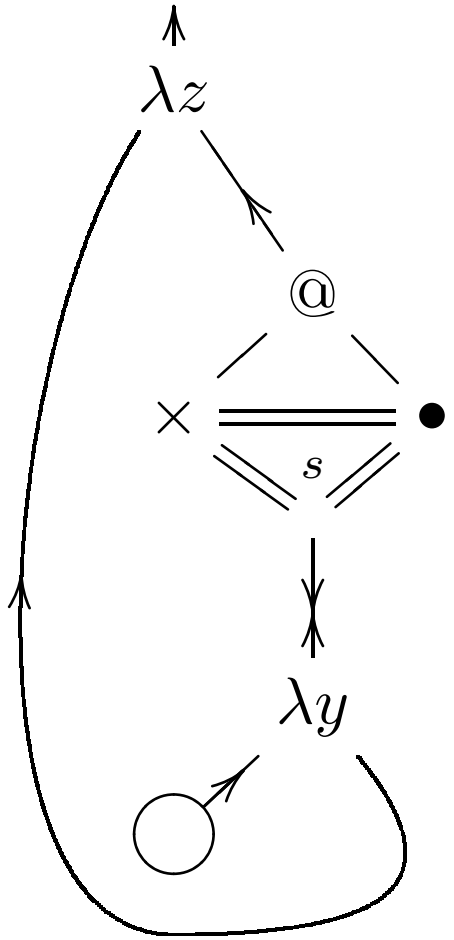
# *Un exemple*

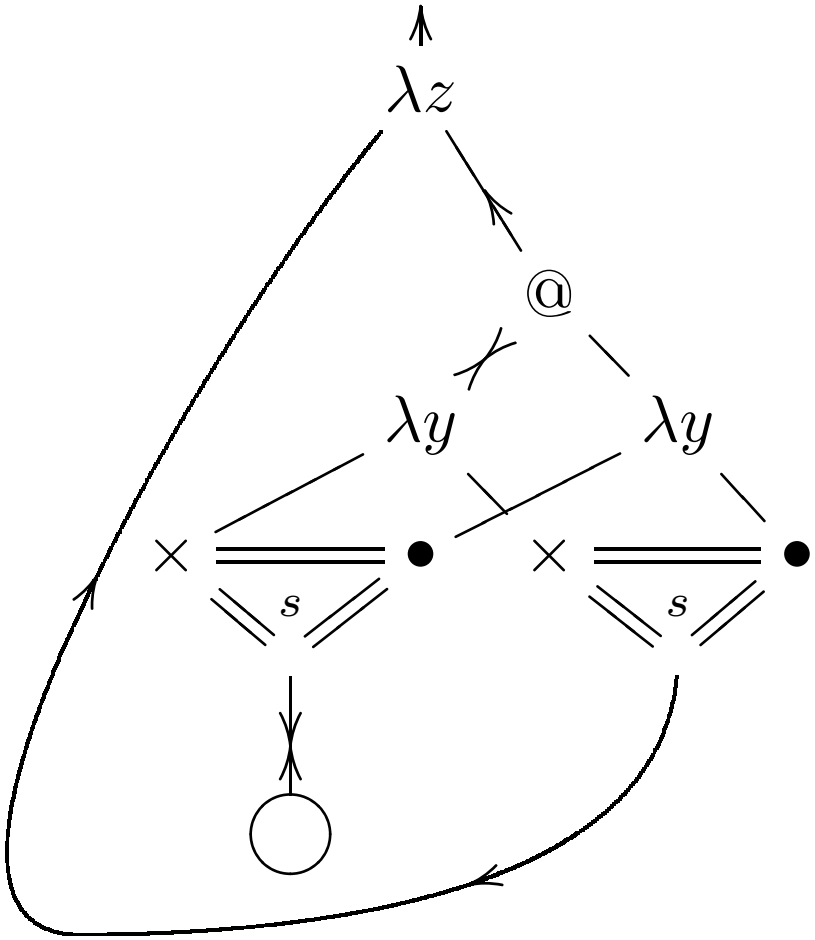


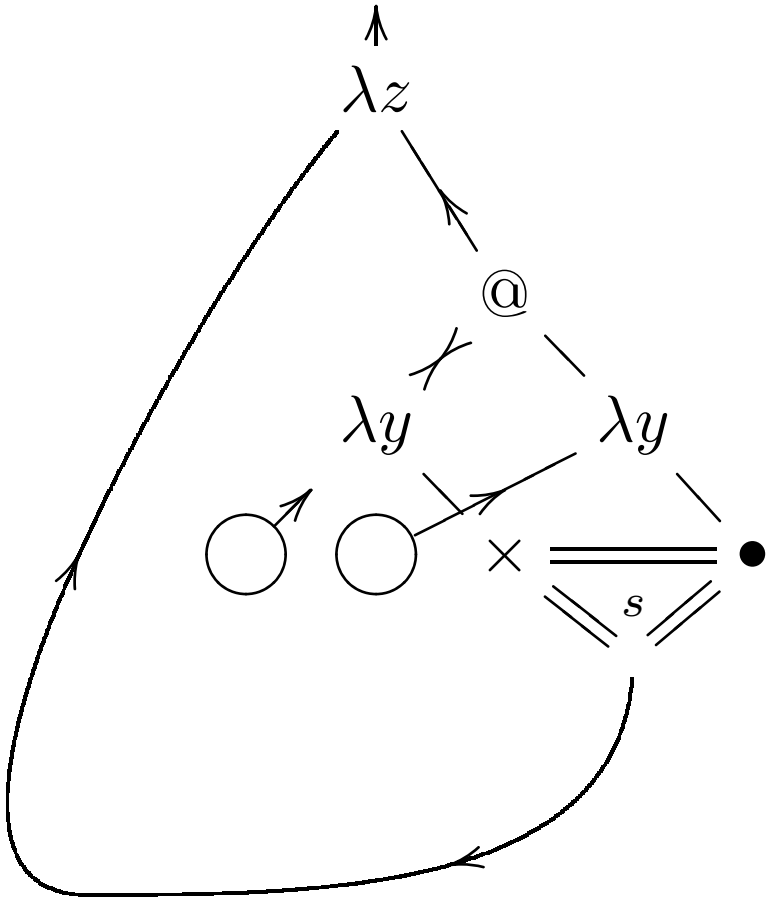


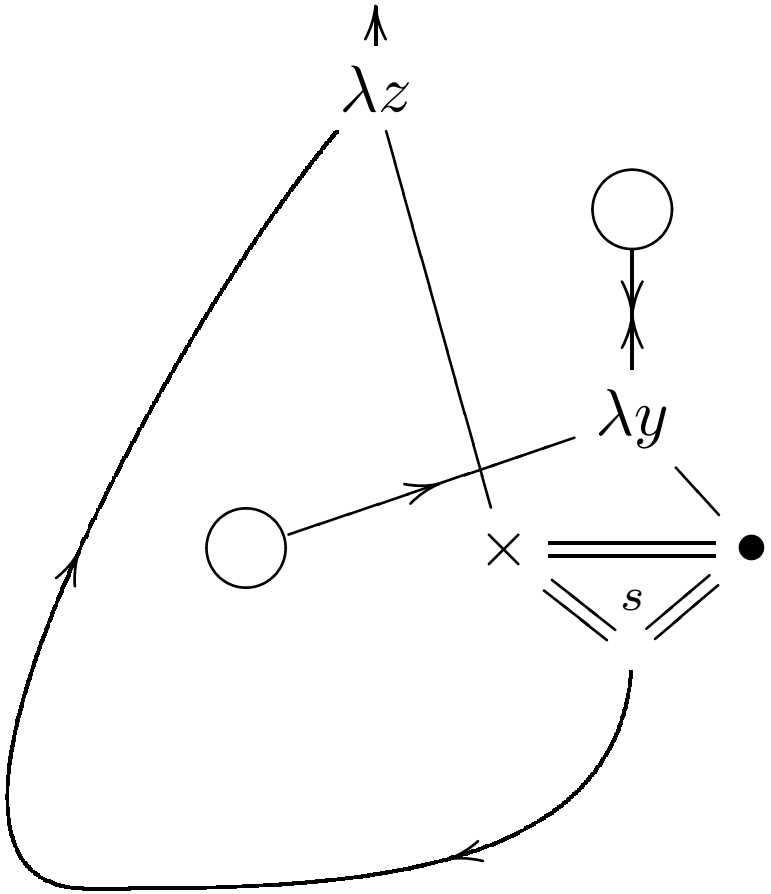
# Le terme $\lambda z.(\lambda x.x\ x)\ (\lambda y.z)$

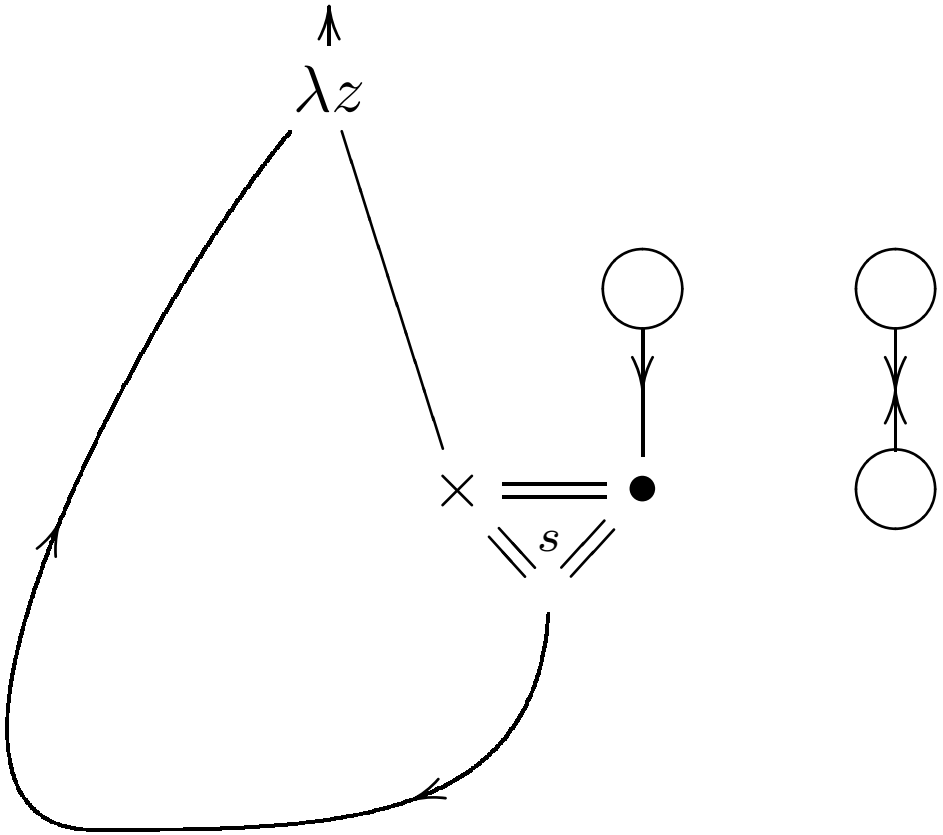


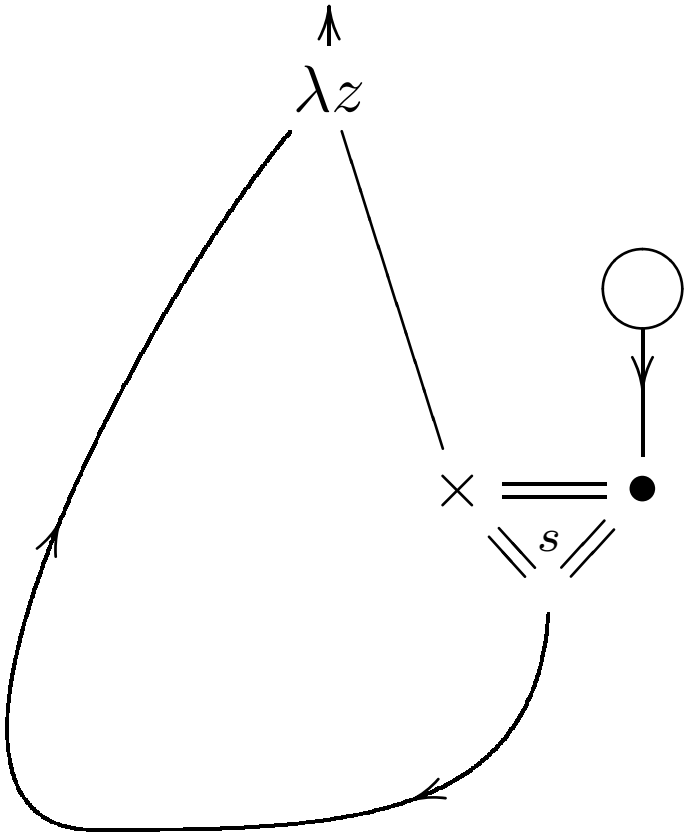

$\lambda z$
@
$\times$
$s$
$\lambda y$


λz
@
λy
λy
s
s


λz
@
λy
λy
s


$\lambda z$
$\lambda y$
$s$

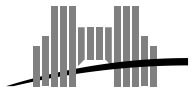

$\lambda z$
$s$


$\lambda z$
$s$

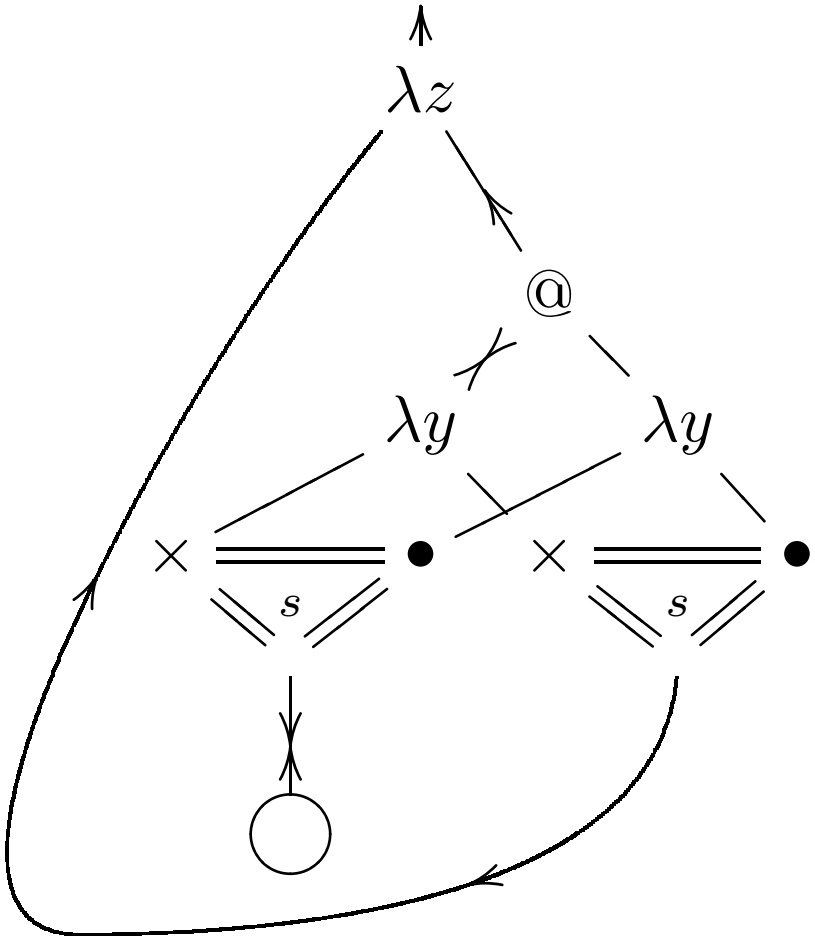
Ce qui est une des représentations possibles de $\lambda z.z$.

Le $z$ n'est partagé avec rien.

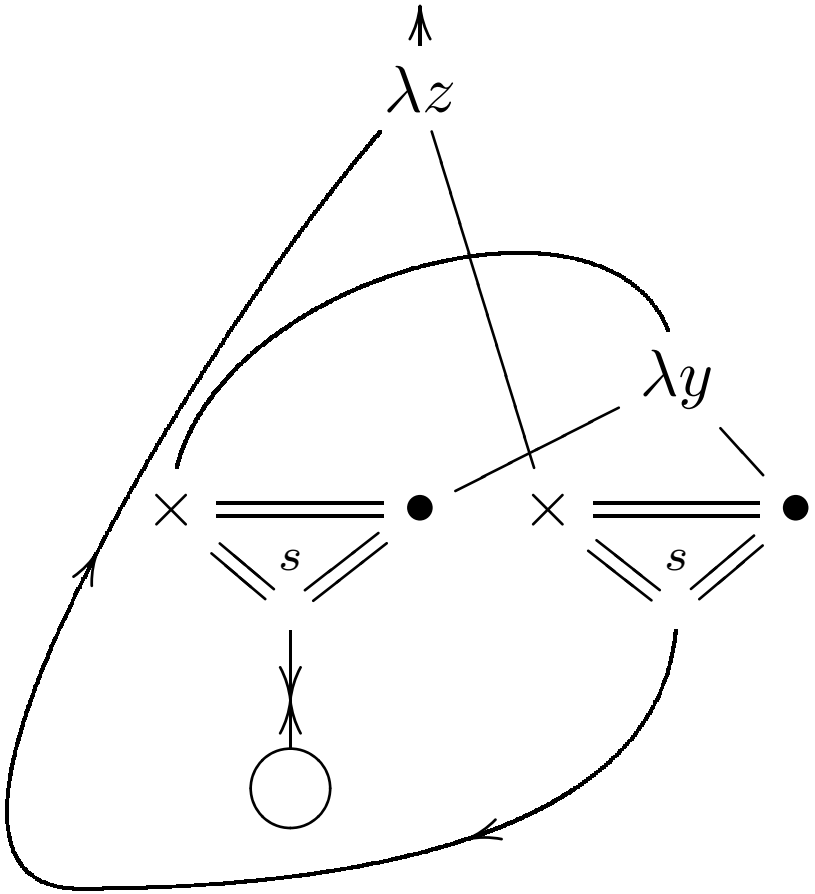




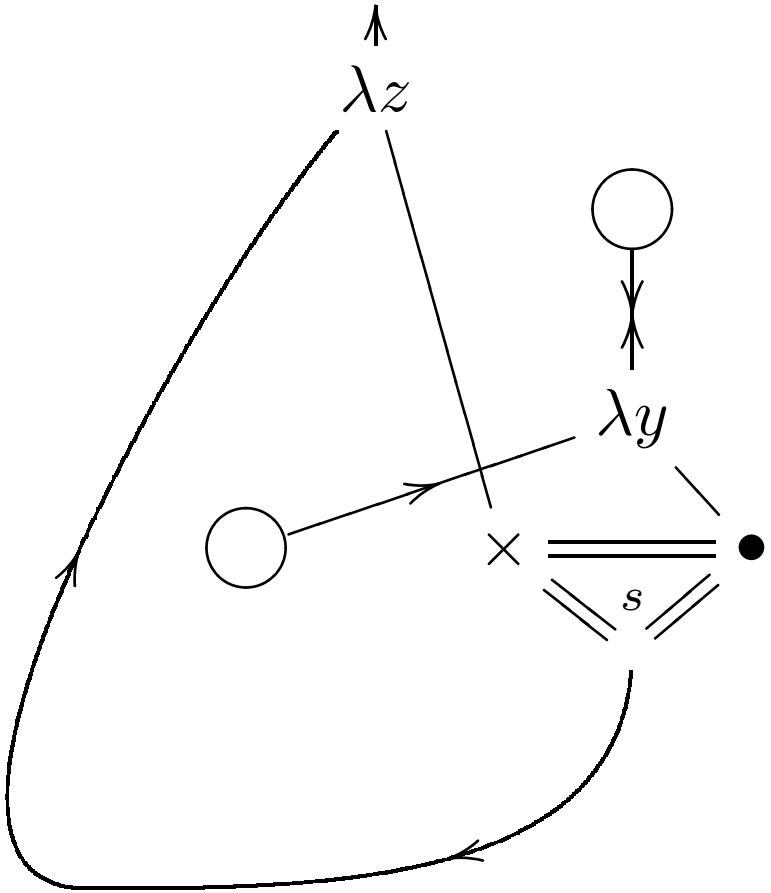
Notons qu'à partir du réseau de la page 167



on obtient aussi le réseau



et on retrouve par contraction de la paire active le réseau de la page 169.



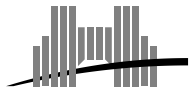

Mettant en évidence la confluence !

# La philosophie

Les réseaux d'interaction font du partage de code et du partage des paramètres.

Partage le code suppose de savoir «départager» quand on fait des calculs.

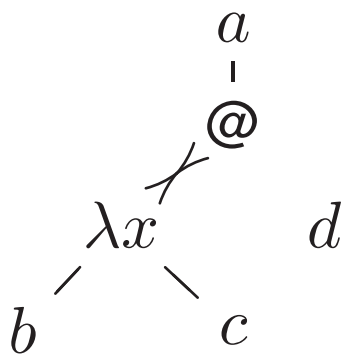




# La philosophie

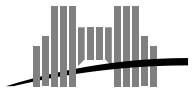
Dans un cadre théorique qui sait identifier les «clones» de fans, on peut montrer que les réseaux implantent une réduction optimale du lambda calcul, c'est-à-dire,

– qu'ils minimisent le nombre de réductions de $\beta$-redex,




– mais en revanche ils augmentent considérablement la bureaucratie. C'est-à-dire les réductions qui concernent les fans.



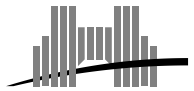


# La décompilation

A la fin du processus, il faut décompiler le réseau en un terme du lambda-calcul.

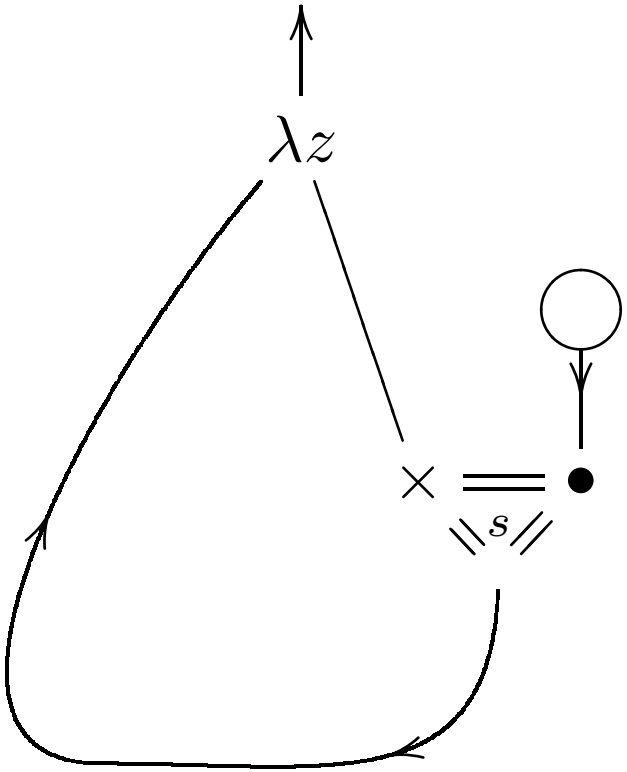
On s'aperçoit que plusieurs réseaux peuvent correspondre à un même terme.





Ainsi

et




correspondent au même terme $\lambda z.z$.