

Jean Goubault-Larrecq

λ -calcul

2. Pouvoir expressif,
stratégies, standardisation

Fonctions calculables

- ❖ Plusieurs modèles:
 - ❖ Machines de Turing
 - ❖ Machines à compteurs (Marvin Minsky)
 - ❖ Fonctions récursives (Kurt Gödel)
 - ❖ λ -calcul
- ❖ Ils sont tous équivalents (« thèse de Church »)

Fonctions calculables

- ❖ On admettra que machines de Turing \equiv fonctions récurrentives (voir cours de calculabilité)
- ❖ On peut implémenter la β -réduction du λ -calcul sur une machine de Turing (pour les détails, voir partie 3 du cours)
- ❖ On va montrer que toutes les **fonctions récurrentives** (générales) se codent en λ -calcul; d'abord, les fonctions **primitives récurrentives**

Fonctions primitives récuratives, et récuratives (générales)

Les fonctions primitives récursives

- ❖ L'ensemble **PR** des fonctions **primitives récursives** est le plus petit contenant:
 - ❖ les fonctions **constantes** : $\mathbf{N}^k \rightarrow \mathbf{N}$ (pour tout k)
 - ❖ la fonction **successeur** $S : m \in \mathbf{N} \mapsto m+1$
 - ❖ les **projections** $\pi_i^k : (m_1, \dots, m_k) \in \mathbf{N}^k \mapsto m_i \ (1 \leq i \leq k)$
- ❖ et clos par les opérations de...

Les fonctions primitives récursives

❖ L'ensemble **PR** des fonctions **primitives récursives** est le plus petit contenant les **constantes**, le **successeur**, les **projections** et clos par les opérations de:

❖ **composition**: si $g_1, \dots, g_k : \mathbf{N}^\ell \rightarrow \mathbf{N}$ sont dans **PR**, et $f : \mathbf{N}^k \rightarrow \mathbf{N}$ est dans **PR**, alors

$$f \circ \langle g_1, \dots, g_k \rangle : \underline{m} \in \mathbf{N}^\ell \mapsto f(g_1(\underline{m}), \dots, g_k(\underline{m}))$$

est dans **PR**

❖ **récurrence primitive**: si $f : \mathbf{N}^k \rightarrow \mathbf{N}$ et $g : \mathbf{N}^{k+2} \rightarrow \mathbf{N}$ sont dans **PR**, alors $R_{f,g} : \mathbf{N}^{k+1} \rightarrow \mathbf{N}$ est dans **PR**, où [déf. par réc. sur le 1er arg.]

$$R_{f,g}(0, \underline{m}) \stackrel{\text{def}}{=} f(\underline{m}) \quad \text{[cas de base]}$$

$$R_{f,g}(n+1, \underline{m}) \stackrel{\text{def}}{=} g(n, R_{f,g}(n, \underline{m}), \underline{m}) \quad \text{[cas de récurrence]}$$

Les fonctions récursives générales

- ❖ Toutes les fonctions primitives récursives sont **totales**.
Mais ce n'est pas le cas de toutes les fonctions récursives (calculables).
Il nous manque un équivalent de la boucle `while` générale.

Les fonctions récursives générales

- ❖ Toutes les fonctions primitives récursives sont **totales**.
Mais ce n'est pas le cas de toutes les fonctions récursives (calculables).
Il nous manque un équivalent de la boucle `while` générale.
- ❖ Traditionnellement, on utilise la **minimisation**:
pour toute fonction (partielle) $f: \mathbf{N}^{k+1} \rightarrow \mathbf{N}$,
on note $\mu f: \mathbf{N}^k \rightarrow \mathbf{N}$ la fonction (partielle) telle que:
 - $\mu f(\underline{m}) \stackrel{\text{def}}{=} l'$ unique entier n tel que $f(n, \underline{m})=0$
et pour tout entier $i < n$, $f(i, \underline{m})$ est défini et $\neq 0$
(« le premier entier n tel que $f(n, \underline{m})=0$ »)
... s'il existe
 - $\mu f(\underline{m})$ indéfini sinon

Les fonctions récursives

Informatiquement:

```
let i = ref 0 in (  
  while f(!i, m) != 0 do i := !i + 1;  
  !i)
```

- ❖ Toutes les fonctions récursives sont calculables.
Mais ce n'est pas réciproque.
(calculables).

Il nous manque un équivalent de la boucle `while` générale.

- ❖ Traditionnellement, on utilise la **minimisation**:
pour toute fonction (partielle) $f: \mathbf{N}^{k+1} \rightarrow \mathbf{N}$,
on note $\mu f: \mathbf{N}^k \rightarrow \mathbf{N}$ la fonction (partielle) telle que:
 - $\mu f(\underline{m}) \stackrel{\text{def}}{=} n$ l'unique entier n tel que $f(n, \underline{m}) = 0$
et pour tout entier $i < n$, $f(i, \underline{m})$ est défini et $\neq 0$
(« le premier entier n tel que $f(n, \underline{m}) = 0$ »)
... s'il existe
 - $\mu f(\underline{m})$ indéfini sinon

Les fonctions récursives

Informatiquement:

```
let i = ref 0 in (  
  while f(!i, m) != 0 do i := !i + 1;  
  !i)
```

- ❖ Toutes les fonctions récursives sont calculables.
Mais ce n'est pas réciproque.
(calculables).

Il nous manque un équivalent de la boucle `while` générale.

- ❖ Traditionnellement, on utilise la **minimisation**:
pour toute fonction (partielle) $f: \mathbf{N}^{k+1} \rightarrow \mathbf{N}$,
on note $\mu f: \mathbf{N}^k \rightarrow \mathbf{N}$ la fonction (partielle) telle que:

— $\mu f(\underline{m}) \stackrel{\text{def}}{=} n$ l'unique entier n tel que $f(n, \underline{m}) = 0$

et pour tout entier $i < n$, $f(i, \underline{m})$ est défini et $\neq 0$

(« le premier entier n tel que $f(n, \underline{m}) = 0$ »)

... s'il existe

— $\mu f(\underline{m})$ indéfini sinon

à condition que
tous les $f(i, \underline{m})$, $i < n$,
soit **définis**

Les fonctions récursives générales

- ❖ L'ensemble **Rec** des fonctions **récursives** est le plus petit contenant les **constantes**, le **successeur**, les **projections** et clos par les opérations de:

Les fonctions récursives générales

- ❖ L'ensemble **Rec** des fonctions **récursives** est le plus petit contenant les **constantes**, le **successeur**, les **projections** et clos par les opérations de:
 - ❖ **composition**: si $g_1, \dots, g_k : \mathbf{N}^\ell \rightarrow \mathbf{N} \in \mathbf{Rec}$, et $f : \mathbf{N}^k \rightarrow \mathbf{N} \in \mathbf{Rec}$, alors
$$f \circ \langle g_1, \dots, g_k \rangle : \underline{m} \in \mathbf{N}^\ell \rightarrow f(g_1(\underline{m}), \dots, g_k(\underline{m}))$$
est dans **Rec**

Les fonctions récursives générales

- ❖ L'ensemble **Rec** des fonctions **récursives** est le plus petit contenant les **constantes**, le **successeur**, les **projections** et clos par les opérations de:
 - ❖ **composition**: si $g_1, \dots, g_k : \mathbf{N}^\ell \rightarrow \mathbf{N} \in \mathbf{Rec}$, et $f : \mathbf{N}^k \rightarrow \mathbf{N} \in \mathbf{Rec}$, alors
$$f \circ \langle g_1, \dots, g_k \rangle : \underline{m} \in \mathbf{N}^\ell \rightarrow f(g_1(\underline{m}), \dots, g_k(\underline{m}))$$
est dans **Rec**
 - ❖ **récurrence primitive**: si $f : \mathbf{N}^k \rightarrow \mathbf{N}$ et $g : \mathbf{N}^{k+2} \rightarrow \mathbf{N} \in \mathbf{Rec}$, alors
$$R_{f,g} : \mathbf{N}^{k+1} \rightarrow \mathbf{N} \in \mathbf{Rec}, \text{ où}$$
$$R_{f,g}(0, \underline{m}) \stackrel{\text{def}}{=} f(\underline{m})$$
$$R_{f,g}(n+1, \underline{m}) \stackrel{\text{def}}{=} g(n, R_{f,g}(n, \underline{m}), \underline{m})$$

Les fonctions récursives générales

- ❖ L'ensemble **Rec** des fonctions **récursives** est le plus petit contenant les **constantes**, le **successeur**, les **projections** et clos par les opérations de:
 - ❖ **composition**: si $g_1, \dots, g_k : \mathbf{N}^\ell \rightarrow \mathbf{N} \in \mathbf{Rec}$, et $f : \mathbf{N}^k \rightarrow \mathbf{N} \in \mathbf{Rec}$, alors
$$f \circ \langle g_1, \dots, g_k \rangle : \underline{m} \in \mathbf{N}^\ell \rightarrow f(g_1(\underline{m}), \dots, g_k(\underline{m}))$$
est dans **Rec**
 - ❖ **récurrence primitive**: si $f : \mathbf{N}^k \rightarrow \mathbf{N}$ et $g : \mathbf{N}^{k+2} \rightarrow \mathbf{N} \in \mathbf{Rec}$, alors
$$R_{f,g} : \mathbf{N}^{k+1} \rightarrow \mathbf{N} \in \mathbf{Rec}, \text{ où}$$
$$R_{f,g}(0, \underline{m}) \stackrel{\text{def}}{=} f(\underline{m})$$
$$R_{f,g}(n+1, \underline{m}) \stackrel{\text{def}}{=} g(n, R_{f,g}(n, \underline{m}), \underline{m})$$
 - ❖ **minimisation**: si $f : \mathbf{N}^{k+1} \rightarrow \mathbf{N} \in \mathbf{Rec}$, alors $\mu f : \mathbf{N}^k \rightarrow \mathbf{N} \in \mathbf{Rec}$

Nouveau!

Les fonctions récursives générales

- ❖ L'ensemble **Rec** des fonctions **récursives** est le plus petit contenant les **constantes**, le **successeur**, les **projections** et clos par les opérations de:

$(f \circ \langle g_1, \dots, g_k \rangle) (\underline{m})$ défini ssi
côté droit $f(g_1(\underline{m}), \dots, g_k(\underline{m}))$ **défini**

- ❖ **composition**: si $g_1, \dots, g_k : \mathbf{N}^\ell \rightarrow \mathbf{N} \in \mathbf{Rec}$, et $f : \mathbf{N}^k \rightarrow \mathbf{N} \in \mathbf{Rec}$, alors

$$f \circ \langle g_1, \dots, g_k \rangle : \underline{m} \in \mathbf{N}^\ell \rightarrow f(g_1(\underline{m}), \dots, g_k(\underline{m}))$$

est dans **Rec**

- ❖ **récurrence primitive**: si $f : \mathbf{N}^k \rightarrow \mathbf{N}$ et $g : \mathbf{N}^{k+2} \rightarrow \mathbf{N} \in \mathbf{Rec}$, alors

$R_{f,g} : \mathbf{N}^{k+1} \rightarrow \mathbf{N} \in \mathbf{Rec}$, où

$$R_{f,g}(0, \underline{m}) \stackrel{\text{def}}{=} f(\underline{m})$$

$$R_{f,g}(n+1, \underline{m}) \stackrel{\text{def}}{=} g(n, R_{f,g}(n, \underline{m}), \underline{m})$$

Nouveau!

- ❖ **minimisation**: si $f : \mathbf{N}^{k+1} \rightarrow \mathbf{N} \in \mathbf{Rec}$, alors $\mu f : \mathbf{N}^k \rightarrow \mathbf{N} \in \mathbf{Rec}$

Les fonctions récursives générales

- ❖ L'ensemble **Rec** des fonctions **récursives** est le plus petit contenant les **constantes**, le **successeur**, les **projections** et clos par les opérations de:

$(f \circ \langle g_1, \dots, g_k \rangle) (\underline{m})$ défini ssi
côté droit $f(g_1(\underline{m}), \dots, g_k(\underline{m}))$ **défini**

- ❖ **composition**: si $g_1, \dots, g_k : \mathbf{N}^\ell \rightarrow \mathbf{N} \in \mathbf{Rec}$, et $f : \mathbf{N}^k \rightarrow \mathbf{N} \in \mathbf{Rec}$, alors

$$f \circ \langle g_1, \dots, g_k \rangle : \underline{m} \in \mathbf{N}^\ell \rightarrow f(g_1(\underline{m}), \dots, g_k(\underline{m}))$$

est dans **Rec**

- ❖ **récurrence primitive**: si $f : \mathbf{N}^k \rightarrow \mathbf{N}$ et $g : \mathbf{N}^{k+2} \rightarrow \mathbf{N} \in \mathbf{Rec}$, alors

$$R_{f,g} : \mathbf{N}^{k+1} \rightarrow \mathbf{N} \in \mathbf{Rec}, \text{ où}$$

$$R_{f,g}(0, \underline{m}) \stackrel{\text{def}}{=} f(\underline{m})$$

$$R_{f,g}(n+1, \underline{m}) \stackrel{\text{def}}{=} g(n, R_{f,g}(n, \underline{m}), \underline{m})$$

Nouveau!

ceci voulant maintenant dire que
le côté gauche est défini
ssi le côté droit est **défini**

- ❖ **minimisation**: si $f : \mathbf{N}^{k+1} \rightarrow \mathbf{N} \in \mathbf{Rec}$, alors $\mu f : \mathbf{N}^k \rightarrow \mathbf{N} \in \mathbf{Rec}$

Les entiers de Church

- ❖ Avant de commencer, il va falloir trouver une représentation des **entiers**
- ❖ Il y en a plusieurs, mais l'une des plus naturelles est:
$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . \underbrace{f(f(\dots(fx)))}_{n \text{ fois } f} \quad [\text{entier de Church}]$$

Les entiers de Church

- ❖ Avant de commencer, il va falloir trouver une représentation des **entiers**
- ❖ Il y en a plusieurs, mais l'une des plus naturelles est:
$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . \underbrace{f(f(\dots(fx)))}_{n \text{ fois } f} \quad [\text{entier de Church}]$$
- ❖ i.e., $\ulcorner n \urcorner$ est « la fonctionnelle $f \mapsto f^n$ » [itérateur]

Les entiers de Church

- ❖ Avant de commencer, il va falloir trouver une représentation des **entiers**
- ❖ Il y en a plusieurs, mais l'une des plus naturelles est:
$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . \underbrace{f(f(\dots(fx)))}_{n \text{ fois } f} \quad [\text{entier de Church}]$$
- ❖ i.e., $\ulcorner n \urcorner$ est « la fonctionnelle $f \mapsto f^n$ » [itérateur]
- ❖ Je noterai donc, abusivement:
$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But du codage

- ❖ On cherche, pour toute fonction récursive $f: \mathbf{N}^k \rightarrow \mathbf{N}$, un λ -terme clos $\ulcorner f \urcorner$ tel que, pour tous $n_1, \dots, n_k \in \mathbf{N}$, si $f(n_1, \dots, n_k)$ est définie (et vaut, disons, n), alors

$$\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner =_{\beta} \ulcorner n \urcorner$$

But du codage

sans variable libre

- ❖ On cherche, pour toute fonction récursive $f: \mathbf{N}^k \rightarrow \mathbf{N}$, un λ -terme clos $\ulcorner f \urcorner$ tel que, pour tous $n_1, \dots, n_k \in \mathbf{N}$, si $f(n_1, \dots, n_k)$ est définie (et vaut, disons, n), alors

$$\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner =_{\beta} \ulcorner n \urcorner$$

But du codage

sans variable libre

- ❖ On cherche, pour toute fonction récursive $f: \mathbf{N}^k \rightarrow \mathbf{N}$, un λ -terme clos $\ulcorner f \urcorner$ tel que, pour tous $n_1, \dots, n_k \in \mathbf{N}$, si $f(n_1, \dots, n_k)$ est définie (et vaut, disons, n), alors

$$\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner =_{\beta} \ulcorner n \urcorner$$

- ❖ **Note:** ceci est équivalent à:

$$\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner \rightarrow^* \ulcorner n \urcorner \quad (\text{pourquoi?})$$

Codage des fonctions de base

❖ Fonctions **constantes**:

si $f(n_1, \dots, n_k) = n$ la même constante n
pour tous n_1, \dots, n_k ,

$$\ulcorner f \urcorner \stackrel{\text{def}}{=} \lambda z_1 . \dots . \lambda z_k . \ulcorner n \urcorner$$

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Codage des fonctions de base

❖ Fonctions constantes:

si $f(n_1, \dots, n_k) = n$ la même constante n
pour tous n_1, \dots, n_k ,

$$\lceil f \rceil \stackrel{\text{def}}{=} \lambda z_1 . \dots . \lambda z_k . \lceil n \rceil$$

❖ Projections:

$$\lceil \pi_i^k \rceil \stackrel{\text{def}}{=} \lambda z_1 . \dots . \lambda z_k . z_i$$

$$\lceil n \rceil \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\lceil f \rceil \lceil n_1 \rceil \dots \lceil n_k \rceil$
 $=_{\beta} \lceil f(n_1, \dots, n_k) \rceil$
si $f(n_1, \dots, n_k)$ défini

Codage des fonctions de base

❖ Fonctions constantes:

si $f(n_1, \dots, n_k)$ = la même constante n
pour tous n_1, \dots, n_k ,

$$\ulcorner f \urcorner \stackrel{\text{def}}{=} \lambda z_1 . \dots . \lambda z_k . \ulcorner n \urcorner$$

❖ Projections:

$$\ulcorner \pi_i^k \urcorner \stackrel{\text{def}}{=} \lambda z_1 . \dots . \lambda z_k . z_i$$

❖ Successeur: plusieurs possibilités:

$$\ulcorner S \urcorner \stackrel{\text{def}}{=} \lambda z . \lambda f . \lambda x . f(zfx) \quad \dots \text{ par exemple}$$

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Le successeur

- ❖ **Successeur:** plusieurs possibilités:

$$\ulcorner S \urcorner \stackrel{\text{def}}{=} \lambda z . \lambda f . \lambda x . f(zfx)$$

- ❖ Vérifions:

$$\begin{aligned} \ulcorner S \urcorner \ulcorner n \urcorner &\rightarrow \lambda f . \lambda x . f(\ulcorner n \urcorner f x) \\ &\rightarrow^2 \lambda f . \lambda x . f(f^n(x)) \\ &= \ulcorner n+1 \urcorner \end{aligned}$$

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Le successeur

- ❖ **Successeur:** plusieurs possibilités:

$$\ulcorner S \urcorner \stackrel{\text{def}}{=} \lambda z . \lambda f . \lambda x . f(zfx)$$

- ❖ Vérifions:

$$\begin{aligned} \ulcorner S \urcorner \ulcorner n \urcorner &\rightarrow \lambda f . \lambda x . f(\ulcorner n \urcorner f x) \\ &\rightarrow^2 \lambda f . \lambda x . f(f^n(x)) \\ &= \ulcorner n+1 \urcorner \end{aligned}$$

deux étapes ici,
d'accord?

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Le successeur

- ❖ **Successeur:** plusieurs possibilités:

$$\ulcorner S \urcorner \stackrel{\text{def}}{=} \lambda z . \lambda f . \lambda x . f(zfx)$$

- ❖ Vérifions:

$$\ulcorner S \urcorner \ulcorner n \urcorner \rightarrow \lambda f . \lambda x . f(\ulcorner n \urcorner f x)$$

deux étapes ici,
d'accord?

$$\begin{aligned} &\rightarrow^2 \lambda f . \lambda x . f(f^n(x)) \\ &= \ulcorner n+1 \urcorner \end{aligned}$$

- ❖ Voyez-vous d'autres définitions possibles de $\ulcorner S \urcorner$?

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Le successeur

- ❖ **Successeur:** plusieurs possibilités:

$$\ulcorner S \urcorner \stackrel{\text{def}}{=} \lambda z . \lambda f . \lambda x . f(zfx)$$

- ❖ Vérifions:

$$\begin{aligned} \ulcorner S \urcorner \ulcorner n \urcorner &\rightarrow \lambda f . \lambda x . f(\ulcorner n \urcorner f x) \\ &\rightarrow^2 \lambda f . \lambda x . f(f^n(x)) \\ &= \ulcorner n+1 \urcorner \end{aligned}$$

deux étapes ici,
d'accord?

- ❖ Voyez-vous d'autres définitions possibles de $\ulcorner S \urcorner$?
- ❖ $\lambda z . \lambda f . \lambda x . zf(fx)$, par exemple

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Le successeur

- ❖ **Successeur:** plusieurs possibilités:

$$\ulcorner S \urcorner \stackrel{\text{def}}{=} \lambda z . \lambda f . \lambda x . f(zfx)$$

- ❖ Vérifions:

$$\begin{aligned} \ulcorner S \urcorner \ulcorner n \urcorner &\rightarrow \lambda f . \lambda x . f(\ulcorner n \urcorner f x) \\ &\rightarrow^2 \lambda f . \lambda x . f(f^n(x)) \\ &= \ulcorner n+1 \urcorner \end{aligned}$$

deux étapes ici,
d'accord?

- ❖ Voyez-vous d'autres définitions possibles de $\ulcorner S \urcorner$?
- ❖ $\lambda z . \lambda f . \lambda x . zf(fx)$, par exemple

au fait, pourquoi ce terme-ci n'est-il
pas β -équivalent à $\ulcorner S \urcorner$ ci-dessus?

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Une parenthèse: opérations arithmétiques

- ❖ Toutes les opérations arithmétiques sont primitives récursives, mais on peut aussi les définir directement

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Une parenthèse: opérations arithmétiques

❖ Toutes les opérations arithmétiques sont primitives récursives, mais on peut aussi les définir directement

❖ $\ulcorner + \urcorner \stackrel{\text{def}}{=} \lambda z_1 . \lambda z_2 . \lambda f . \lambda x . z_1 f(z_2 f x)$

... ou $\lambda z_1 . \lambda z_2 . \lambda f . \lambda x . z_2 f(z_1 f x)$

... ou $\lambda z_1 . \lambda z_2 . z_1 \ulcorner S \urcorner z_2$

... ou $\lambda z_1 . z_1 \ulcorner S \urcorner$

[tous β -inéquivalents!]

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Une parenthèse: opérations arithmétiques

❖ Toutes les opérations arithmétiques sont primitives récurrentes, mais on peut aussi les définir directement

❖ $\ulcorner + \urcorner \stackrel{\text{def}}{=} \lambda z_1 . \lambda z_2 . \lambda f . \lambda x . z_1 f(z_2 f x)$

... ou $\lambda z_1 . \lambda z_2 . \lambda f . \lambda x . z_2 f(z_1 f x)$

... ou $\lambda z_1 . \lambda z_2 . z_1 \ulcorner S \urcorner z_2$

... ou $\lambda z_1 . z_1 \ulcorner S \urcorner$

[tous β -inéquivalents!]

❖ $\ulcorner \times \urcorner \stackrel{\text{def}}{=} \lambda z_1 . \lambda z_2 . \lambda f . z_1(z_2 f)$

[... par exemple]

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Une parenthèse: opérations arithmétiques

❖ Toutes les opérations arithmétiques sont primitives récurrentes, mais on peut aussi les définir directement

❖ $\ulcorner + \urcorner \stackrel{\text{def}}{=} \lambda z_1 . \lambda z_2 . \lambda f . \lambda x . z_1 f(z_2 f x)$

... ou $\lambda z_1 . \lambda z_2 . \lambda f . \lambda x . z_2 f(z_1 f x)$

... ou $\lambda z_1 . \lambda z_2 . z_1 \ulcorner S \urcorner z_2$

... ou $\lambda z_1 . z_1 \ulcorner S \urcorner$

[tous β -inéquivalents!]

❖ $\ulcorner \times \urcorner \stackrel{\text{def}}{=} \lambda z_1 . \lambda z_2 . \lambda f . z_1(z_2 f)$

[... par exemple]

❖ $\ulcorner \text{pow} \urcorner \stackrel{\text{def}}{=} \lambda z_1 . \lambda z_2 . z_1 z_2$

[... ou juste $\lambda z . z!$]

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Composition etc.

composition: si $g_1, \dots, g_k : \mathbf{N}^l \rightarrow \mathbf{N} \in \mathbf{Rec}$,
et $f : \mathbf{N}^k \rightarrow \mathbf{N} \in \mathbf{Rec}$, alors
 $f \circ \langle g_1, \dots, g_k \rangle : \underline{m} \in \mathbf{N}^l \mapsto f(g_1(\underline{m}), \dots, g_k(\underline{m}))$
est dans **Rec**

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

$$\diamond \ulcorner f \circ \langle g_1, \dots, g_k \rangle \urcorner \stackrel{\text{def}}{=} \lambda z_1 . \dots . \lambda z_l .$$
$$\ulcorner f \urcorner (\ulcorner g_1 \urcorner z_1 \dots z_l) \dots (\ulcorner g_k \urcorner z_1 \dots z_l)$$

Composition etc.

composition: si $g_1, \dots, g_k : \mathbf{N}^l \rightarrow \mathbf{N} \in \mathbf{Rec}$,
et $f : \mathbf{N}^k \rightarrow \mathbf{N} \in \mathbf{Rec}$, alors
 $f \circ \langle g_1, \dots, g_k \rangle : \underline{m} \in \mathbf{N}^l \mapsto f(g_1(\underline{m}), \dots, g_k(\underline{m}))$
est dans **Rec**

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

$$\diamond \ulcorner f \circ \langle g_1, \dots, g_k \rangle \urcorner \stackrel{\text{def}}{=} \lambda z_1 . \dots . \lambda z_l .$$

$$\ulcorner f \urcorner (\ulcorner g_1 \urcorner z_1 \dots z_l) \dots (\ulcorner g_k \urcorner z_1 \dots z_l)$$

- ❖ La récurrence primitive = minimisation + prédécesseurs
(voir annexe A pour définir prédécesseur, i.e., la fonction $_ - 1$)

Composition etc.

composition: si $g_1, \dots, g_k : \mathbf{N}^l \rightarrow \mathbf{N} \in \mathbf{Rec}$,
et $f : \mathbf{N}^k \rightarrow \mathbf{N} \in \mathbf{Rec}$, alors
 $f \circ \langle g_1, \dots, g_k \rangle : \underline{m} \in \mathbf{N}^l \mapsto f(g_1(\underline{m}), \dots, g_k(\underline{m}))$
est dans **Rec**

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

- ❖ $\ulcorner f \circ \langle g_1, \dots, g_k \rangle \urcorner \stackrel{\text{def}}{=} \lambda z_1 . \dots . \lambda z_l .$
 $\ulcorner f \urcorner (\ulcorner g_1 \urcorner z_1 \dots z_l) \dots (\ulcorner g_k \urcorner z_1 \dots z_l)$
- ❖ La récurrence primitive = minimisation + prédécesseurs
(voir annexe A pour définir prédécesseur, i.e., la fonction $_ - 1$)
- ❖ L'important sera la minimisation

letrec et points fixes

❖ On va coder la récurrence primitive $R_{f,g}$ par:

letrec $R_{f,g}(z, \underline{m}) =$ (informellement)
if $z=0$ then $f(\underline{m})$
else $g(P(z), R_{f,g}(P(z), \underline{m}), \underline{m})$

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

letrec et points fixes

- ❖ On va coder la récurrence primitive $R_{f,g}$ par:

`letrec $R_{f,g}(z, \underline{m}) =$ (informellement)
 if $z=0$ then $f(\underline{m})$
 else $g(P(z), R_{f,g}(P(z), \underline{m}), \underline{m})$`

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

- ❖ On a tout (if, =0, P, composition)... reste letrec

letrec et points fixes

- ❖ On va coder la récurrence primitive $R_{f,g}$ par:

`letrec $R_{f,g}(z, \underline{m}) =$ (informellement)
 if $z=0$ then $f(\underline{m})$
 else $g(P(z), R_{f,g}(P(z), \underline{m}), \underline{m})$`

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

- ❖ On a tout (if, =0, P, composition)... reste `letrec`
- ❖ `letrec $f(\underline{z}) = e(f, \underline{z})$` , c'est trouver f tel que $f =_{\beta} \lambda \underline{z} . e(f, \underline{z})$

letrec et points fixes

- ❖ On va coder la récurrence primitive $R_{f,g}$ par:

`letrec $R_{f,g}(z, \underline{m}) =$ (informellement)
 if $z=0$ then $f(\underline{m})$
 else $g(P(z), R_{f,g}(P(z), \underline{m}), \underline{m})$`

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

- ❖ On a tout (if, =0, P, composition)... reste `letrec`
- ❖ `letrec $f(\underline{z}) = e(f, \underline{z})$` , c'est trouver f tel que $f =_{\beta} \lambda \underline{z} . e(f, \underline{z})$
- ❖ autrement dit un **point fixe** de $\lambda f . \lambda \underline{z} . e(f, \underline{z})$ (mod $=_{\beta}$)

Points fixes

- ❖ **Proposition.** Tout λ -terme u a un point fixe Yu mod $=_{\beta}$ (i.e., $u(Yu) =_{\beta} Yu$).

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Points fixes

- ❖ **Proposition.** Tout λ -terme u a un point fixe $Yu \text{ mod } =_{\beta}$ (i.e., $u(Yu) =_{\beta} Yu$).
- ❖ En fait, il existe un λ -terme clos Y tel que, pour tout λ -terme u , $u(Yu) =_{\beta} Yu$.

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Points fixes

un combinateur
(de point fixe, ici)

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

- ❖ **Proposition.** Tout λ -terme u a un point fixe $Yu \text{ mod } =_\beta$ (i.e., $u(Yu) =_\beta Yu$).
- ❖ En fait, il existe un λ -terme clos Y tel que, pour tout λ -terme u , $u(Yu) =_\beta Yu$.

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_\beta \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Points fixes

un combinateur
(de point fixe, ici)

- ❖ **Proposition.** Tout λ -terme u a un point fixe Yu mod $=_{\beta}$ (i.e., $u(Yu) =_{\beta} Yu$).
- ❖ En fait, il existe un λ -terme clos Y tel que, pour tout λ -terme u , $u(Yu) =_{\beta} Yu$.
- ❖ Combinateur de point fixe... **de Church** [surprise!]
$$Y \stackrel{\text{def}}{=} \lambda f . (\lambda x . f(xx)) (\lambda x . f(xx))$$

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Points fixes

un combinateur
(de point fixe, ici)

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

- ❖ **Proposition.** Tout λ -terme u a un point fixe $Y u \text{ mod } =_{\beta}$ (i.e., $u(Y u) =_{\beta} Y u$).
- ❖ En fait, il existe un λ -terme clos Y tel que, pour tout λ -terme u , $u(Y u) =_{\beta} Y u$.
- ❖ Combinateur de point fixe... **de Church** [surprise!]
$$Y \stackrel{\text{def}}{=} \lambda f . (\lambda x . f(x x)) (\lambda x . f(x x))$$
- ❖ $Y u \rightarrow (\lambda x . u(x x)) (\lambda x . u(x x))$ [x fraîche, $\notin \text{fv}(u)$]
 $\rightarrow u (A_u A_u)$
 $\leftarrow u(Y u)$
$$\underbrace{\hspace{10em}}_{A_u}$$

Récurrance primitive

récurrance primitive: si $f : \mathbf{N}^k \rightarrow \mathbf{N}$
et $g : \mathbf{N}^{k+2} \rightarrow \mathbf{N} \in \mathbf{Rec}$,
alors $R_{f,g} : \mathbf{N}^{k+1} \rightarrow \mathbf{N} \in \mathbf{Rec}$, où
 $R_{f,g}(0, \underline{m}) \stackrel{\text{def}}{=} f(\underline{m})$
 $R_{f,g}(n+1, \underline{m}) \stackrel{\text{def}}{=} g(n, R_{f,g}(n, \underline{m}), \underline{m})$

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

$$\begin{aligned} \diamond \ulcorner R_{f,g} \urcorner &\stackrel{\text{def}}{=} Y(\lambda R . \lambda z . \lambda z_1 \dots \lambda z_k . \\ &\ulcorner \text{if} \urcorner (\ulcorner =0 \urcorner z) \\ &(\ulcorner f \urcorner z_1 \dots z_k) \\ &(\ulcorner g \urcorner (\ulcorner P \urcorner z) (R (\ulcorner P \urcorner z) z_1 \dots z_k) z_1 \dots z_k)) \end{aligned}$$

Récurrance primitive

récurrance primitive: si $f : \mathbf{N}^k \rightarrow \mathbf{N}$
 et $g : \mathbf{N}^{k+2} \rightarrow \mathbf{N} \in \mathbf{Rec}$,
 alors $R_{f,g} : \mathbf{N}^{k+1} \rightarrow \mathbf{N} \in \mathbf{Rec}$, où
 $R_{f,g}(0, \underline{m}) \stackrel{\text{def}}{=} f(\underline{m})$
 $R_{f,g}(n+1, \underline{m}) \stackrel{\text{def}}{=} g(n, R_{f,g}(n, \underline{m}), \underline{m})$

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
 si $f(n_1, \dots, n_k)$ défini

- $\diamond \ulcorner R_{f,g} \urcorner \stackrel{\text{def}}{=} Y(\lambda R . \lambda z . \lambda z_1 \dots \lambda z_k .$
 $\ulcorner \text{if} \urcorner (\ulcorner =0 \urcorner z)$
 $(\ulcorner f \urcorner z_1 \dots z_k)$
 $(\ulcorner g \urcorner (\ulcorner P \urcorner z) (R (\ulcorner P \urcorner z) z_1 \dots z_k) z_1 \dots z_k))$
- $\diamond \ulcorner R_{f,g} \urcorner \ulcorner 0 \urcorner u_1 \dots u_k =_{\beta} \ulcorner f \urcorner u_1 \dots u_k$
 $\ulcorner R_{f,g} \urcorner \ulcorner n+1 \urcorner u_1 \dots u_k =_{\beta} \ulcorner g \urcorner \ulcorner n \urcorner (\ulcorner R_{f,g} \urcorner \ulcorner n \urcorner u_1 \dots u_k) u_1 \dots u_k$
 (exercice!)

Minimisation

minimisation: si $f: \mathbf{N}^{k+1} \rightarrow \mathbf{N} \in \mathbf{Rec}$,
alors $\mu f: \mathbf{N}^k \rightarrow \mathbf{N} \in \mathbf{Rec}$

$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Minimisation

minimisation: si $f: \mathbf{N}^{k+1} \rightarrow \mathbf{N} \in \mathbf{Rec}$,
alors $\mu f: \mathbf{N}^k \rightarrow \mathbf{N} \in \mathbf{Rec}$

« let $\mu f(\underline{m}) =$
let rec search $z =$
if $f(z, \underline{m}) = 0$ then z
else search $(z+1)$
in search 0 »

$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Minimisation

minimisation: si $f: \mathbf{N}^{k+1} \rightarrow \mathbf{N} \in \mathbf{Rec}$,
alors $\mu f: \mathbf{N}^k \rightarrow \mathbf{N} \in \mathbf{Rec}$

❖ $\ulcorner \mu f \urcorner \stackrel{\text{def}}{=} \lambda z_1 \dots \lambda z_k .$

$Y(\lambda search . \lambda z .$

$\ulcorner \text{if} \urcorner (\ulcorner =0 \urcorner (\ulcorner f \urcorner z z_1 \dots z_k))$

z

$search (\ulcorner S \urcorner z))$

$\ulcorner 0 \urcorner$

« let $\mu f(\underline{m}) =$
letrec search $z =$
if $f(z, \underline{m})=0$ then z
else search $(z+1)$
in search 0 »

$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Minimisation

minimisation: si $f: \mathbf{N}^{k+1} \rightarrow \mathbf{N} \in \mathbf{Rec}$,
alors $\mu f: \mathbf{N}^k \rightarrow \mathbf{N} \in \mathbf{Rec}$

❖ $\ulcorner \mu f \urcorner \stackrel{\text{def}}{=} \lambda z_1 \dots \lambda z_k .$

$Y(\lambda search . \lambda z .$

$\ulcorner \text{if} \urcorner (\ulcorner =0 \urcorner (\ulcorner f \urcorner z z_1 \dots z_k))$

z

$search (\ulcorner S \urcorner z))$

$\ulcorner 0 \urcorner$

❖ Si $\ulcorner f \urcorner \ulcorner 0 \urcorner u_1 \dots u_k =_{\beta} \ulcorner m_0 \urcorner, \dots, \ulcorner f \urcorner \ulcorner n \urcorner u_1 \dots u_k =_{\beta} \ulcorner m_n \urcorner,$
avec m_0, \dots, m_{n-1} non nuls et $m_n=0$, alors
 $\ulcorner \mu f \urcorner u_1 \dots u_k =_{\beta} \ulcorner n \urcorner$ (exercice! implique une réc. sur n)

« let $\mu f(\underline{m}) =$
letrec search $z =$
if $f(z, \underline{m})=0$ then z
else search $(z+1)$
in search 0 »

$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Pouvoir expressif

❖ Au total, nous avons:

❖ **Théorème.** Pour toute fonction récursive $f: \mathbf{N}^k \rightarrow \mathbf{N}$,
il existe un λ -terme clos $\ulcorner f \urcorner$ tel que,

pour tous $n_1, \dots, n_k \in \mathbf{N}$,

si $f(n_1, \dots, n_k)$ est définie (et vaut, disons, n), alors

$$\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner =_{\beta} \ulcorner n \urcorner$$

Pouvoir expressif

❖ Au total, nous avons:

❖ **Théorème.** Pour toute fonction récursive $f: \mathbf{N}^k \rightarrow \mathbf{N}$,
il existe un λ -terme clos $\ulcorner f \urcorner$ tel que,

pour tous $n_1, \dots, n_k \in \mathbf{N}$,

si $f(n_1, \dots, n_k)$ est définie (et vaut, disons, n), alors

$$\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner =_{\beta} \ulcorner n \urcorner$$

❖ Kleene a montré encore plus fort:

on peut demander que si $f(n_1, \dots, n_k)$ n'est pas **définie**,

alors $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$ n'est **pas normalisable**

Pouvoir expressif

❖ Au total, nous avons:

❖ **Théorème.** Pour toute fonction récursive $f: \mathbf{N}^k \rightarrow \mathbf{N}$,
il existe un λ -terme clos $\ulcorner f \urcorner$ tel que,

pour tous $n_1, \dots, n_k \in \mathbf{N}$,

si $f(n_1, \dots, n_k)$ est définie (et vaut, disons, n), alors

$$\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner =_{\beta} \ulcorner n \urcorner$$

❖ Kleene a montré encore plus fort:

on peut demander que si $f(n_1, \dots, n_k)$ n'est pas **définie**,

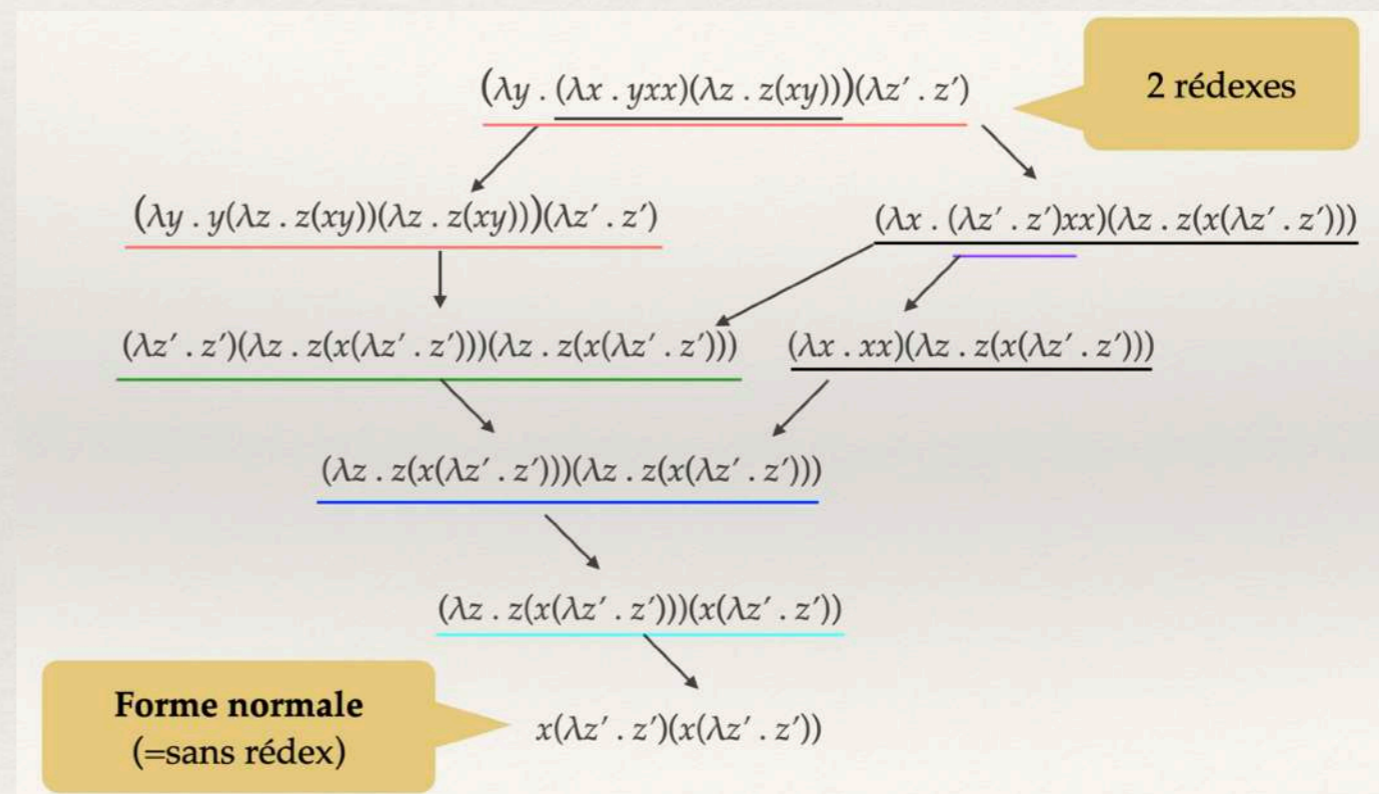
alors $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$ n'est **pas normalisable**

❖ C'est beaucoup plus dur, et utilise le **théorème de standardisation** (dans la suite), ou bien de démontrer qu'en fait $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$ n'a même pas de **forme normale de tête** (dans la suite)

Stratégies

Stratégies

- ❖ En général, on doit **choisir** un rédex parmi tous ceux que l'on peut contracter, à chaque étape de réduction



Stratégies

- ❖ En général, on doit **choisir** un rédex parmi tous ceux que l'on peut contracter, à chaque étape de réduction
- ❖ Une fonction qui à chaque terme non normal associe un de ses rédexes est une **stratégie**
- ❖ Quelques exemples de stratégies
- ❖ La stratégie gauche, et le théorème de standardisation
- ❖ Avant ça, les stratégies de quelques langages de programmation

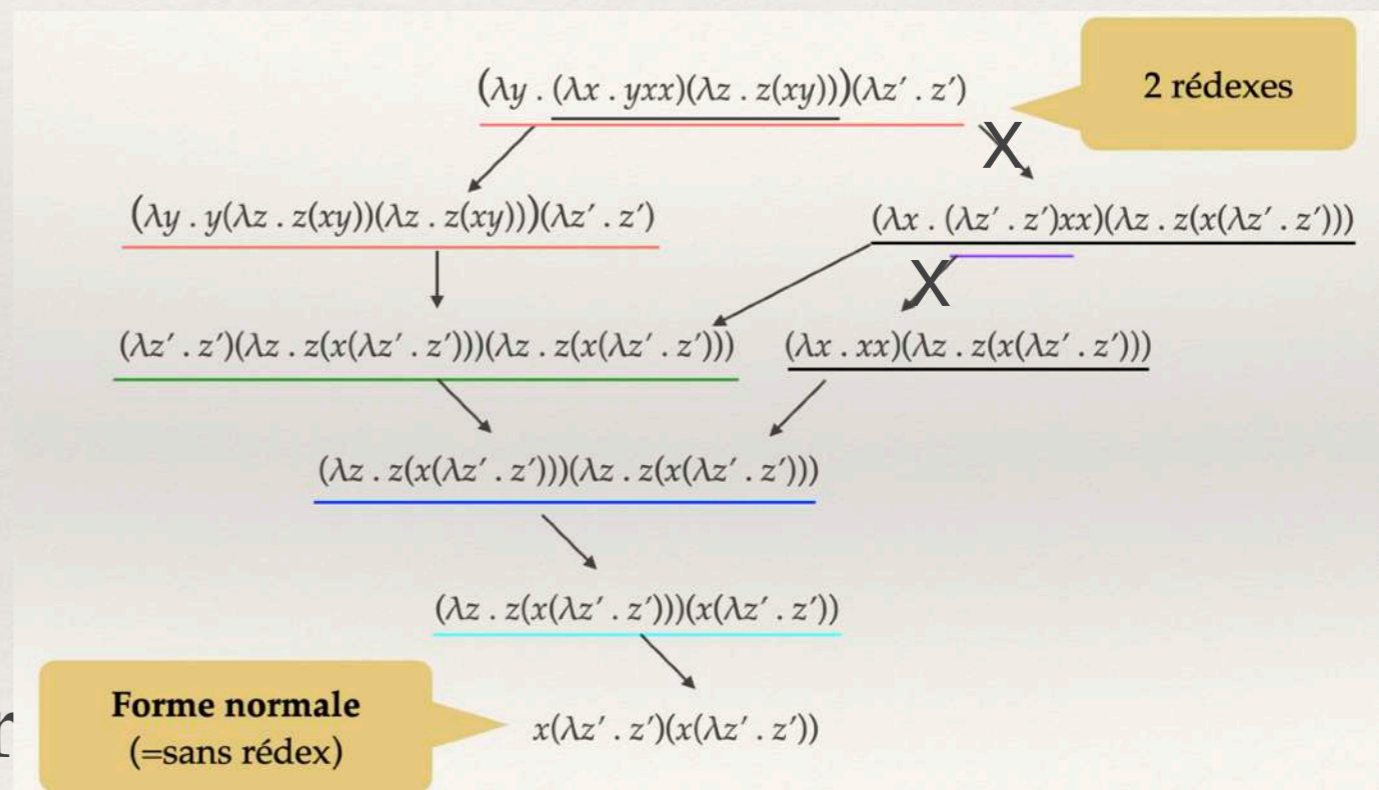
Stratégies faibles

- ❖ Les stratégies des langages de programmation sont des **stratégies faibles**: elles ne réduisent **pas sous les λ** .

```
> fn x => x+1+2;
it : int -> int
it = <fn>
>
```

- ❖ (Quoique... voir **inlining**.)

- ❖ Suffisent pour évaluer les programmes de type de base (`int`, `bool`, `string`).



Stratégies faibles

- ❖ Les stratégies des langages de programmation sont des **stratégies faibles**: elles ne réduisent **pas sous les λ** .

Contextes pour les stratégies faibles:

- ❖ $C ::= _$ trou (où le terme est inséré)
 $\frac{}{\lambda x . C}$ ~~réduction « sous la lambda »~~
 $| C v$ la réduction s'opère dans la fonction
 $| u C$ la réduction s'opère dans l'argument

- ❖ Une stratégie faible est en général **incomplète**: elle ne permet pas de trouver la forme normale
(Ex: $\lambda x . (\lambda y . y)x$ ne se réduit pas, faiblement)

Appel par valeur

- ❖ Comment est évaluée une application uv ?
- ❖ En Lisp, en ML (en C, en Java, etc.):
 1. on évalue d'abord à **la fois** u et v
 2. si u s'évalue à $\lambda x . s$, **et** v en une **valeur** V ,
on continue l'exécution en $s[x:=V]$

Appel par valeur

- ❖ Comment est évaluée une application uv ?
- ❖ En Lisp, en ML (en C, en Java, etc.):
 1. on évalue d'abord à la fois u et v
 2. si u s'évalue à $\lambda x . s$, et v en une valeur V , on continue l'exécution en $s[x:=V]$
- ❖ Une modélisation classique:
valeurs $V =$ variables ou abstractions
(pas applications)
- ❖ $(\beta_V) \quad (\lambda x . u) V \rightarrow u[x:=V]$

Theoretical Computer Science 1 (1975) 125-159. © North-Holland Publishing Company

CALL-BY-NAME, CALL-BY-VALUE AND THE λ -CALCULUS

G. D. PLOTKIN

Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh,
Edinburgh, United Kingdom

Communicated by R. Milner
Received 1 August 1974

Appel par valeur

- ❖ Comment est évaluée une application uv ?
- ❖ En Lisp, en ML (en C, en Java, etc.):
 1. on évalue d'abord à la fois u et v
 2. si u s'évalue à $\lambda x . s$, et v en une valeur V , on continue l'exécution en $s[x:=V]$
- ❖ Une modélisation classique:
valeurs V = variables ou abstractions
(pas applications)
- ❖ $(\beta_v) \quad (\lambda x . u) V \rightarrow u[x:=V]$

Dans ces langages, la stratégie est par valeur et faible

Theoretical Computer Science 1 (1975) 125-159. © North-Holland Publishing Company

CALL-BY-NAME, CALL-BY-VALUE AND THE λ -CALCULUS

G. D. PLOTKIN

Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh, United Kingdom

Communicated by R. Milner
Received 1 August 1974

Appel par nom (évaluation paresseuse)

❖ Comment est évaluée uv ?

❖ En Haskell (et Miranda):

1. on évalue d'abord **uniquement** u

2. si u s'évalue à $\lambda x . s$, ~~et v en une valeur V ,~~

on continue l'exécution en $s[x:=v]$

Evaluation **paresseuse**:

On n'évaluera v que si s
a besoin de la valeur de x

Appel par nom (évaluation paresseuse)

- ❖ Comment est évaluée uv ?
- ❖ En Haskell (et Miranda):
 1. on évalue d'abord **uniquement** u
 2. si u s'évalue à $\lambda x . s$, ~~et v en une valeur V .~~
on continue l'exécution en $s[x:=v]$
- ❖ Modélisé par la **réduction gauche**:
toujours réduire le redex le plus à gauche dans l'écriture du terme

Evaluation **paresseuse**:
On n'évaluera v que si s
a besoin de la valeur de x

Theoretical Computer Science 1 (1975) 125-159. © North-Holland Publishing Company

CALL-BY-NAME, CALL-BY-VALUE AND THE λ -CALCULUS

G. D. PLOTKIN

Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh, United Kingdom

Communicated by R. Milner
Received 1 August 1974

Appel par nom (évaluation paresseuse)

- ❖ Comment est évaluée uv ?
- ❖ En Haskell (et Miranda):
 1. on évalue d'abord **uniquement** u
 2. si u s'évalue à $\lambda x . s$, ~~et v en une valeur V .~~
on continue l'exécution en $s[x:=v]$
- ❖ Modélisé par la **réduction gauche**:
toujours réduire le redex le plus à gauche dans l'écriture du terme

Evaluation **paresseuse**:
On n'évaluera v que si s
a besoin de la valeur de x

Dans ces langages,
la stratégie est par
nom **et** faible

Theoretical Computer Science 1 (1975) 125-159. © North-Holland Publishing Company

CALL-BY-NAME, CALL-BY-VALUE AND THE λ -CALCULUS

G. D. PLOTKIN

Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh, United Kingdom

Communicated by R. Milner
Received 1 August 1974

Classification des stratégies en λ -calcul

Classification des stratégies

- ❖ **Stratégies internes:** le rédex contracté est le plus bas possible (dans $(\lambda x . u)v$, u et v sont normaux)

Classification des stratégies

- ❖ Stratégies **internes**: le rédex contracté est le plus bas possible (dans $(\lambda x . u)v$, u et v sont normaux)
- ❖ Stratégies **externes**: le rédex contracté est le plus haut possible (on ne réduit pas sous un rédex)

Classification des stratégies

- ❖ Stratégies **internes**: le rédex contracté est le plus bas possible (dans $(\lambda x . u)v$, u et v sont normaux)
- ❖ Stratégies **externes**: le rédex contracté est le plus haut possible (on ne réduit pas sous un rédex)
- ❖ Evaluation des arguments **de gauche à droite** (comme en Java, Python, etc.) ou **de droite à gauche** (comme en Caml)

Classification des stratégies

- ❖ Stratégies **internes**: le rédex contracté est le plus bas possible (dans $(\lambda x . u)v$, u et v sont normaux)
- ❖ Stratégies **externes**: le rédex contracté est le plus haut possible (on ne réduit pas sous un rédex)
- ❖ Evaluation des arguments **de gauche à droite** (comme en Java, Python, etc.) ou **de droite à gauche** (comme en Caml)
- ❖ Stratégies **faibles**: on ne contracte pas sous une λ -abstraction (comme en Caml, Haskell, etc.)

Stratégies optimales?

- ❖ Y a-t-il une stratégie (calculable) **optimale**?

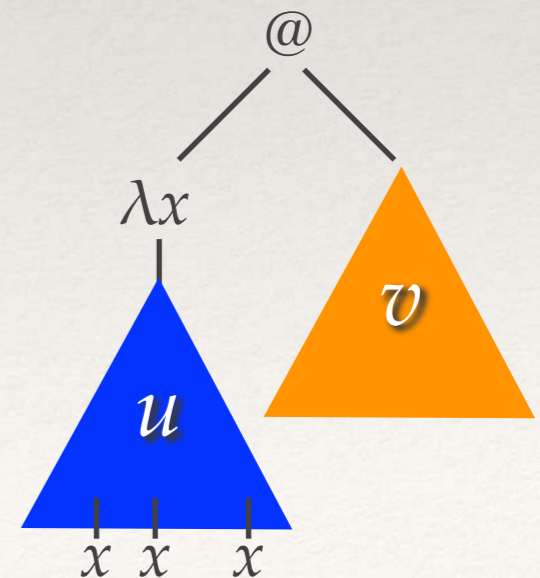
Stratégies optimales?

- ❖ Y a-t-il une stratégie (calculable) **optimale**?
- ❖ Si optimale = **standardisante**
(= « trouve une forme normale s'il en existe une »)
la réponse est **oui**: la **stratégie gauche**
(voir transparents suivants) est standardisante.

Stratégie gauche,
réductions de tête

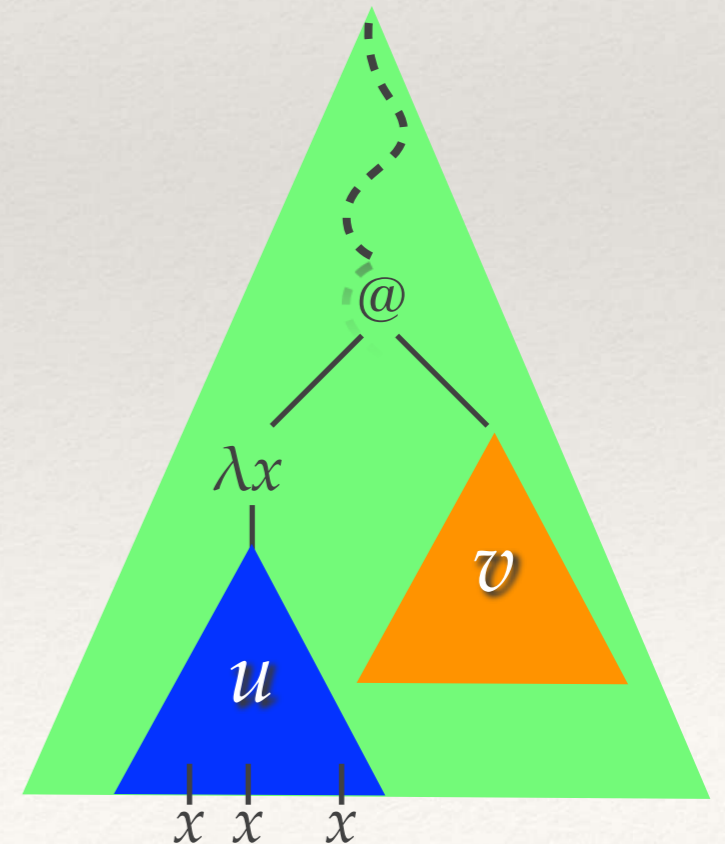
Stratégie gauche

- ❖ La stratégie **gauche** consiste à toujours sélectionner le **rédex le plus à gauche** dans l'écriture textuelle d'un terme
- ❖ De façon équivalente, dans la représentation en arbre:
 - parmi tous les **rédexes externes**
 - on choisit le **plus à gauche**(anglais: **leftmost-outermost**)



Stratégie gauche

- ❖ La stratégie **gauche** consiste à toujours sélectionner le **rédex le plus à gauche** dans l'écriture textuelle d'un terme
- ❖ De façon équivalente, dans la représentation en arbre:
 - parmi tous les **rédexes externes**
 - on choisit le **plus à gauche**
(anglais: **leftmost-outermost**)



Théorème de standardisation

- ❖ Le théorème de **standardisation**:
- ❖ **Théorème.** Pour tout λ -terme u , et toute **forme normale** v , les affirmations suivantes sont équivalentes:
 1. $u \rightarrow^* v$
 2. $u \rightarrow^* v$ par réduction gauche.

Théorème de standardisation

- ❖ Le théorème de **standardisation**:
- ❖ **Théorème.** Pour tout λ -terme u , et toute **forme normale** v , les affirmations suivantes sont équivalentes:
 1. $u \rightarrow^* v$
 2. $u \rightarrow^* v$ par réduction gauche.
- ❖ Autrement dit, pour trouver la forme normale de u , il suffit d'effectuer des **réductions gauches**.
Si les réductions gauches ne terminent pas, u n'a pas de forme normale.

Les preuves du thm. de standardisation

- ❖ La preuve **classique**: on définit une relation de réécriture **entre réécritures**.
- ❖ Les réécritures normales pour cette relation sont les réécritures gauches.
- ❖ La terminaison est assurée par le théorème des **développements finis** (voir exercice)
+ une notion de résidu (ou radicaux) de rédex.
[Hardin, Section 7, 7 pages]

Les preuves du thm. de standardisation

- ❖ La preuve **de René David**: une preuve élémentaire (mais longue), aperçu en annexe B.
- ❖ On définit une relation \Rightarrow_s par une définition astucieuse.
- ❖ On montre que $\Rightarrow_s = \rightarrow^*$ (voir exercices pour les détails).
- ❖ Si $u \Rightarrow_s v$, et si v est **normal**, alors on verra que u se réduit en v par **réduction gauche**.
- ❖ ... aussi une excuse pour examiner les notions fondamentales de **réduction de tête, formes de tête**, etc.

Semi-calculable
=récurisif
= λ -définissable

λ -définissable \Rightarrow semi-calculable

- ❖ Semi-calculable = réalisée par une machine de Turing
- Calculable = ... par une machine de Turing **qui termine**.
- ❖ On peut implémenter la réduction gauche
(voir la fonction **red** de l'annexe C), donc:
- ❖ **Prop.** Il existe une machine de Turing *Eval* qui,
étant donné tout (mot représentant) un λ -terme u
en entrée, calcule
la **forme normale** de u en sortie si elle existe,
et ne termine pas sinon.

récurusif \Rightarrow λ -définissable

❖ Rappel:

Théorème. Pour toute fonction récurusif $f: \mathbf{N}^k \rightarrow \mathbf{N}$,
il existe un λ -terme clos $\ulcorner f \urcorner$ tel que,
pour tous $n_1, \dots, n_k \in \mathbf{N}$,
si $f(n_1, \dots, n_k)$ est définie (et vaut, disons, n), alors
$$\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner =_{\beta} \ulcorner n \urcorner$$

- ❖ Nous pouvons maintenant montrer la partie manquante:
si $f(n_1, \dots, n_k)$ n'est pas **définie**,
alors $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$ n'est **pas normalisable**
- ❖ Mieux: ... n'a **pas de forme normale de tête**
(je vous le laisse en exercice [« La fin du théorème de Kleene »])

That's all, folks!

Annexe A: prédécesseur

Détour 1: booléens

❖ Booléens de Church:

$\ulcorner \text{vrai} \urcorner \stackrel{\text{def}}{=} \lambda \text{then} . \lambda \text{else} . \text{then}$

$\ulcorner \text{faux} \urcorner \stackrel{\text{def}}{=} \lambda \text{then} . \lambda \text{else} . \text{else}$

❖ $\ulcorner \text{if} \urcorner \stackrel{\text{def}}{=} \lambda \text{test} . \lambda \text{then} . \lambda \text{else} . \text{test then else}$
(ou bien $\lambda \text{test} . \text{test}!$)

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Détour 2: test à zéro

❖ Test d'égalité d'un entier à 0:

$$\ulcorner =0 \urcorner \stackrel{\text{def}}{=} \lambda z . z (\lambda _ . \ulcorner \text{faux} \urcorner) \ulcorner \text{vrai} \urcorner$$

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Détour 2: test à zéro

❖ Test d'égalité d'un entier à 0:

$$\ulcorner =0 \urcorner \stackrel{\text{def}}{=} \lambda z . z (\lambda _ . \ulcorner \text{faux} \urcorner) \ulcorner \text{vrai} \urcorner$$

❖ $\ulcorner =0 \urcorner \ulcorner 0 \urcorner \rightarrow \ulcorner 0 \urcorner (\lambda _ . \ulcorner \text{faux} \urcorner) \ulcorner \text{vrai} \urcorner$
 $\rightarrow (\lambda x . x) \ulcorner \text{vrai} \urcorner \rightarrow \ulcorner \text{vrai} \urcorner$

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Détour 2: test à zéro

❖ Test d'égalité d'un entier à 0:

$$\ulcorner =0 \urcorner \stackrel{\text{def}}{=} \lambda z . z (\lambda _ . \ulcorner \text{faux} \urcorner) \ulcorner \text{vrai} \urcorner$$

❖ $\ulcorner =0 \urcorner \ulcorner 0 \urcorner \rightarrow \ulcorner 0 \urcorner (\lambda _ . \ulcorner \text{faux} \urcorner) \ulcorner \text{vrai} \urcorner$
 $\rightarrow (\lambda x . x) \ulcorner \text{vrai} \urcorner \rightarrow \ulcorner \text{vrai} \urcorner$

❖ $\ulcorner =0 \urcorner \ulcorner n+1 \urcorner \rightarrow \ulcorner n+1 \urcorner (\lambda _ . \ulcorner \text{faux} \urcorner) \ulcorner \text{vrai} \urcorner$
 $\rightarrow^2 (\lambda _ . \ulcorner \text{faux} \urcorner)^{n+1} \ulcorner \text{vrai} \urcorner$
 $= (\lambda _ . \ulcorner \text{faux} \urcorner) ((\lambda _ . \ulcorner \text{faux} \urcorner)^n \ulcorner \text{vrai} \urcorner)$
 $\rightarrow \ulcorner \text{faux} \urcorner$

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Détour 2: test à zéro

- ❖ Test d'égalité d'un entier à 0:

$$\ulcorner =0 \urcorner \stackrel{\text{def}}{=} \lambda z . z (\lambda _ . \ulcorner \text{faux} \urcorner) \ulcorner \text{vrai} \urcorner$$

- ❖ $\ulcorner =0 \urcorner \ulcorner 0 \urcorner \rightarrow \ulcorner 0 \urcorner (\lambda _ . \ulcorner \text{faux} \urcorner) \ulcorner \text{vrai} \urcorner$
 $\rightarrow (\lambda x . x) \ulcorner \text{vrai} \urcorner \rightarrow \ulcorner \text{vrai} \urcorner$

- ❖ $\ulcorner =0 \urcorner \ulcorner n+1 \urcorner \rightarrow \ulcorner n+1 \urcorner (\lambda _ . \ulcorner \text{faux} \urcorner) \ulcorner \text{vrai} \urcorner$
 $\rightarrow^2 (\lambda _ . \ulcorner \text{faux} \urcorner)^{n+1} \ulcorner \text{vrai} \urcorner$
 $= (\lambda _ . \ulcorner \text{faux} \urcorner) ((\lambda _ . \ulcorner \text{faux} \urcorner)^n \ulcorner \text{vrai} \urcorner)$
 $\rightarrow \ulcorner \text{faux} \urcorner$

- ❖ Le terme réellement difficile à trouver, c'est le **prédécesseur**

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Détour 3: couples

❖ Formation de couples:

$$\ulcorner \text{pair} \urcorner \stackrel{\text{def}}{=} \lambda z_1 . \lambda z_2 . \lambda p . p z_1 z_2$$

$$\ulcorner \text{pair} \urcorner uv \rightarrow^2 \langle u, v \rangle \stackrel{\text{def}}{=} \lambda p . p uv$$

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Détour 3: couples

❖ Formation de couples:

$$\ulcorner \text{pair} \urcorner \stackrel{\text{def}}{=} \lambda z_1 . \lambda z_2 . \lambda p . p z_1 z_2$$

$$\ulcorner \text{pair} \urcorner uv \rightarrow^2 \langle u, v \rangle \stackrel{\text{def}}{=} \lambda p . p uv$$

❖ Projections:

$$\ulcorner p_1 \urcorner \stackrel{\text{def}}{=} \lambda c . c(\lambda z_1 . \lambda z_2 . z_1) \quad (\text{oui, } = \lambda c . c \ulcorner \text{vrai} \urcorner)$$

$$\ulcorner p_2 \urcorner \stackrel{\text{def}}{=} \lambda c . c(\lambda z_1 . \lambda z_2 . z_2) \quad (\text{oui, } = \lambda c . c \ulcorner \text{faux} \urcorner)$$

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Détour 3: couples

❖ Formation de couples:

$$\ulcorner \text{pair} \urcorner \stackrel{\text{def}}{=} \lambda z_1 . \lambda z_2 . \lambda p . p z_1 z_2$$

$$\ulcorner \text{pair} \urcorner uv \rightarrow^2 \langle u, v \rangle \stackrel{\text{def}}{=} \lambda p . p uv$$

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

❖ Projections:

$$\ulcorner p_1 \urcorner \stackrel{\text{def}}{=} \lambda c . c(\lambda z_1 . \lambda z_2 . z_1) \quad (\text{oui, } = \lambda c . c \ulcorner \text{vrai} \urcorner)$$

$$\ulcorner p_2 \urcorner \stackrel{\text{def}}{=} \lambda c . c(\lambda z_1 . \lambda z_2 . z_2) \quad (\text{oui, } = \lambda c . c \ulcorner \text{faux} \urcorner)$$

$$\ulcorner p_1 \urcorner \langle u, v \rangle \rightarrow \langle u, v \rangle (\lambda z_1 . \lambda z_2 . z_1) \rightarrow (\lambda z_1 . \lambda z_2 . z_1) uv \rightarrow^2 u$$

$$\ulcorner p_2 \urcorner \langle u, v \rangle \rightarrow \langle u, v \rangle (\lambda z_1 . \lambda z_2 . z_2) \rightarrow (\lambda z_1 . \lambda z_2 . z_2) uv \rightarrow^2 v$$

Détour 4: prédécesseur

- ❖ Soit $P(n) \stackrel{\text{def}}{=} n-1$ si $n \geq 1$,
0 si $n=0$ (prédécesseur)

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

But: $\ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner$
 $=_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner$
si $f(n_1, \dots, n_k)$ défini

Détour 4: prédécesseur

- ❖ Soit $P(n) \stackrel{\text{def}}{=} n-1$ si $n \geq 1$,
0 si $n=0$ (prédécesseur)

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

- ❖ Idée de codage:

$c := \langle 0,0 \rangle$; répéter n fois: remplacer $\langle m,n \rangle$ par $\langle m+1,m \rangle$

Ainsi c passe par $\langle 0,0 \rangle, \langle 1,0 \rangle, \langle 2,1 \rangle, \dots, \langle n,n-1 \rangle$ (si $n \geq 1$)

Finalelement, prendre la 2ème projection.

$$\begin{aligned} \text{But: } \ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner \\ =_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner \\ \text{si } f(n_1, \dots, n_k) \text{ défini} \end{aligned}$$

Détour 4: prédécesseur

- ❖ Soit $P(n) \stackrel{\text{def}}{=} n-1$ si $n \geq 1$,
 0 si $n=0$ (prédécesseur)

$$\ulcorner n \urcorner \stackrel{\text{def}}{=} \lambda f . \lambda x . f^n(x)$$

- ❖ Idée de codage:

$c := \langle 0,0 \rangle$; répéter n fois: remplacer $\langle m,n \rangle$ par $\langle m+1,m \rangle$

Ainsi c passe par $\langle 0,0 \rangle, \langle 1,0 \rangle, \langle 2,1 \rangle, \dots, \langle n,n-1 \rangle$ (si $n \geq 1$)

Enfin, prendre la 2ème projection.

- ❖ $\ulcorner P \urcorner \stackrel{\text{def}}{=} \lambda z . \ulcorner p_2 \urcorner (z(\lambda c . \langle \ulcorner S \urcorner (\ulcorner p_1 \urcorner c), \ulcorner p_1 \urcorner c \rangle) \langle \ulcorner 0 \urcorner, \ulcorner 0 \urcorner \rangle))$

Vérification: exercice!

$$\text{But: } \ulcorner f \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_k \urcorner \\ =_{\beta} \ulcorner f(n_1, \dots, n_k) \urcorner \\ \text{si } f(n_1, \dots, n_k) \text{ défini}$$

Annexe B: un aperçu de la preuve de standardisation, d'après R. David

Forme de tête

- ❖ Etant donné un λ -terme u , on peut énumérer tous les λx_i au début de l'écriture de u :

$$\lambda x_1 . \dots . \lambda x_n . v$$

de sorte que v ne soit **pas une λ -abstraction**

(n peut valoir 0)

Forme de tête

- ❖ Etant donné un λ -terme u , on peut énumérer tous les λx_i au début de l'écriture de u :

$$\lambda x_1 . \dots . \lambda x_n . v$$

de sorte que v ne soit **pas une λ -abstraction**

(n peut valoir 0)

- ❖ on peut maintenant écrire v comme une application:

$$h u_1 \dots u_m$$

où h n'est pas une application (m peut valoir 0)

Forme de tête

- ❖ Etant donné un λ -terme u , on peut énumérer tous les λx_i au début de l'écriture de u :

$$\lambda x_1 . \dots . \lambda x_n . v$$

de sorte que v ne soit **pas une λ -abstraction**

(n peut valoir 0)

- ❖ on peut maintenant écrire v comme une application:

$$h u_1 \dots u_m$$

où h n'est pas une application (m peut valoir 0)

- ❖ Donc...

Forme de tête

- ❖ On peut écrire tout λ -terme u de façon unique comme:

$$\lambda x_1 \dots \lambda x_n . h u_1 \dots u_m$$

avec $n, m \geq 0$, et:

Forme de tête

- ❖ On peut écrire tout λ -terme u de façon unique comme:

$$\lambda x_1 . \dots . \lambda x_n . h u_1 \dots u_m$$

avec $n, m \geq 0$, et:

- ❖ soit h est une **variable** (la **variable de tête** de u
on dit alors que u est en **forme normale de tête**)

Forme de tête

- ❖ On peut écrire tout λ -terme u de façon unique comme:

$$\lambda x_1 . \dots . \lambda x_n . h u_1 \dots u_m$$

avec $n, m \geq 0$, et:

« forme (normale de tête) »,
pas
« (forme normale) de tête »

- ❖ soit h est une **variable** (la **variable de tête** de u
on dit alors que u est en **forme normale de tête**)

Forme de tête

- ❖ On peut écrire tout λ -terme u de façon unique comme:

$$\lambda x_1 . \dots . \lambda x_n . h u_1 \dots u_m$$

avec $n, m \geq 0$, et:

« forme (normale de tête) »,
pas
« (forme normale) de tête »

- ❖ soit h est une **variable** (la **variable de tête** de u
on dit alors que u est en **forme normale de tête**)
- ❖ soit h est une λ -abstraction $\lambda x . s$ et $m \geq 1$
(alors $(\lambda x . s) u_1$ est le **rédex de tête** de u)

Forme de tête

- ❖ On peut écrire tout λ -terme u de façon unique comme:

$$\lambda x_1 \dots \lambda x_n . h u_1 \dots u_m$$

« forme (normale de tête) »,
pas
« (forme normale) de tête »

avec $n, m \geq 0$, et:

- ❖ soit h est une **variable** (la **variable de tête** de u
on dit alors que u est en **forme normale de tête**)
- ❖ soit h est une λ -abstraction $\lambda x . s$ et $m \geq 1$
(alors $(\lambda x . s) u_1$ est le **rédex de tête** de u)
- ❖ Dans tous les cas, cette forme est la **forme de tête** de u
 - h est la **tête** de u
 - $u_1 \dots u_m$ sont les **arguments** de u

Réduction de tête

- ❖ Une **réduction de tête** est une β -réduction qui contracte le rédex de tête (s'il existe)

Réduction de tête

- ❖ Une **réduction de tête** est une β -réduction qui contracte le rédex de tête (s'il existe)
- ❖
$$\lambda x_1 . \dots . \lambda x_n . (\lambda x . s) u_1 \dots u_m$$
$$\rightarrow_t \lambda x_1 . \dots . \lambda x_n . s[x:=u_1] u_2 \dots u_m$$

Réduction de tête

- ❖ Une **réduction de tête** est une β -réduction qui contracte le rédex de tête (s'il existe)
- ❖
$$\lambda x_1 . \dots . \lambda x_n . (\lambda x . s) u_1 \dots u_m$$
$$\rightarrow_t \lambda x_1 . \dots . \lambda x_n . s[x:=u_1] u_2 \dots u_m$$
- ❖ On ne demande (surtout) **pas** que \rightarrow_t soit compatible au contexte

Réduction de tête

- ❖ Une **réduction de tête** est une β -réduction qui contracte le rédex de tête (s'il existe)
- ❖ $\lambda x_1 . \dots . \lambda x_n . (\lambda x . s) u_1 \dots u_m$
 $\rightarrow_t \lambda x_1 . \dots . \lambda x_n . s[x:=u_1] u_2 \dots u_m$
- ❖ On ne demande (surtout) **pas** que \rightarrow_t soit compatible au contexte
- ❖ **Note:** on a une compatibilité partielle, en ce que
si $u \rightarrow_t v$ alors $\lambda x . u \rightarrow_t \lambda x . v$

Réduction de tête

- ❖ Une **réduction de tête** est une β -réduction qui contracte le rédex de tête (s'il existe)
- ❖ $\lambda x_1 . \dots . \lambda x_n . (\lambda x . s) u_1 \dots u_m$
 $\rightarrow_t \lambda x_1 . \dots . \lambda x_n . s[x:=u_1] u_2 \dots u_m$
- ❖ On ne demande (surtout) **pas** que \rightarrow_t soit compatible au contexte
- ❖ **Note:** on a une compatibilité partielle, en ce que
si $u \rightarrow_t v$ alors $\lambda x . u \rightarrow_t \lambda x . v$
- ❖ **Note:** la réduction de tête est **déterministe** (au plus 1 rédex de tête)
En particulier, la forme normale de tête est **unique** si elle existe.

Formes normales de tête

- ❖ Tout terme u en forme normale a pour forme de tête une **forme normale de tête**:

$$\lambda x_1 . \dots . \lambda x_n . h u_1 \dots u_m \quad (h \text{ variable})$$

... car u ne contient pas de rédex

Formes normales de tête

- ❖ Tout terme u en forme normale a pour forme de tête une **forme normale de tête**:

$$\lambda x_1 . \dots . \lambda x_n . h u_1 \dots u_m \quad (h \text{ variable})$$

... car u ne contient pas de rédex

- ❖ Il existe des termes en forme normale de tête, qui ne sont **pas normaux**, par exemple:

$$\lambda x . x \Omega$$

Formes normales de tête

- ❖ Tout terme u en forme normale a pour forme de tête une **forme normale de tête**:

$$\lambda x_1 . \dots . \lambda x_n . h u_1 \dots u_m \quad (h \text{ variable})$$

... car u ne contient pas de rédex

- ❖ Il existe des termes en forme normale de tête, qui ne sont **pas normaux**, par exemple:

$$\lambda x . x \Omega$$

- ❖ **Fait.** u est normal ssi il est en **forme normale de tête**:

$$\lambda x_1 . \dots . \lambda x_n . h u_1 \dots u_m \quad (h \text{ variable})$$

et, récursivement, **tous** ses arguments u_i sont **normaux**

Formes normales de tête

- ❖ Tout terme u en forme normale a pour forme de tête une **forme normale de tête**:

$$\lambda x_1 . \dots . \lambda x_n . h u_1 \dots u_m \quad (h \text{ variable})$$

... car u ne contient pas de rédex

- ❖ Il existe des termes en forme normale de tête, qui ne sont **pas normaux**, par exemple:

$$\lambda x . x \Omega$$

On dit parfois
« u est **héréditairement** en forme normale de tête »

- ❖ **Fait.** u est normal ssi il est en **forme normale de tête**:

$$\lambda x_1 . \dots . \lambda x_n . h u_1 \dots u_m \quad (h \text{ variable})$$

et, récursivement, **tous** ses arguments u_i sont **normaux**

Stratégies standard

- ❖ Une **stratégie standard** est, intuitivement, une stratégie qui réduit « du haut vers le bas »
- ❖ Toute stratégie externe (y compris la stratégie gauche) est standard

Stratégies standard

- ❖ Une **stratégie standard** est, intuitivement,
une stratégie qui réduit « du haut vers le bas »
- ❖ Toute stratégie externe (y compris la stratégie gauche) est standard
- ❖ Formellement, on donne une définition abstraite de la notion « se réduit par une stratégie standard [en 0, 1, ou plusieurs étapes] en »
via une relation \Rightarrow_s , due à René David
(ainsi que la preuve qui suit)

La relation de réduction standard \Rightarrow_s

- ❖ On dit que $u \Rightarrow_s v$ ssi (récurrence sur $|v|$)

La relation de réduction standard \Rightarrow_s

❖ On dit que $u \Rightarrow_s v$ ssi (récurrence sur $|v|$)

❖ v s'écrit sous forme de tête

$$\lambda x_1 . \dots . \lambda x_n . v_0 v_1 \dots v_m$$

La relation de réduction standard \Rightarrow_s

❖ On dit que $u \Rightarrow_s v$ ssi (récurrence sur $|v|$)

❖ v s'écrit sous forme de tête

$$\lambda x_1 . \dots . \lambda x_n . v_0 v_1 \dots v_m$$

❖ $u \rightarrow_t^* \lambda x_1 . \dots . \lambda x_n . u_0 u_1 \dots u_m$ (les mêmes x_1, \dots, x_n , merci le α -renommage)

La relation de réduction standard \Rightarrow_s

❖ On dit que $u \Rightarrow_s v$ ssi (récurrence sur $|v|$)

❖ v s'écrit sous forme de tête

$$\lambda x_1 . \dots . \lambda x_n . v_0 v_1 \dots v_m$$

❖ $u \rightarrow_t^* \lambda x_1 . \dots . \lambda x_n . u_0 u_1 \dots u_m$ (les mêmes x_1, \dots, x_n , merci le α -renommage)

❖ $u_1 \Rightarrow_s v_1, \dots, u_m \Rightarrow_s v_m$

La relation de réduction standard \Rightarrow_s

❖ On dit que $u \Rightarrow_s v$ ssi (récurrence sur $|v|$)

❖ v s'écrit sous forme de tête

$$\lambda x_1 . \dots . \lambda x_n . v_0 v_1 \dots v_m$$

❖ $u \rightarrow_t^* \lambda x_1 . \dots . \lambda x_n . u_0 u_1 \dots u_m$ (les mêmes x_1, \dots, x_n , merci le α -renommage)

❖ $u_1 \Rightarrow_s v_1, \dots, u_m \Rightarrow_s v_m$

❖ $u_0 \Rightarrow_s v_0$, ce par quoi j'entends:

— si v_0 est une variable x , alors $u_0 = x$

— si $v_0 = \lambda x . v'$, alors $u_0 = \lambda x . u'$ et $u' \Rightarrow_s v'$

La relation de réduction standard \Rightarrow_s

❖ On dit que $u \Rightarrow_s v$ ssi (récurrence sur $|v|$)

❖ v s'écrit sous forme de tête

$$\lambda x_1 . \dots . \lambda x_n . v_0 v_1 \dots v_m$$

❖ $u \rightarrow_t^* \lambda x_1 . \dots . \lambda x_n . u_0 u_1 \dots u_m$ (les mêmes x_1, \dots, x_n , merci le α -renommage)

❖ $u_1 \Rightarrow_s v_1, \dots, u_m \Rightarrow_s v_m$

❖ $u_0 \Rightarrow_s v_0$, ce par quoi j'entends:

— si v_0 est une variable x , alors $u_0 = x$

— si $v_0 = \lambda x . v'$, alors $u_0 = \lambda x . u'$ et $u' \Rightarrow_s v'$

❖ J'écrirai en abrégé:

$$\begin{array}{ccccccc} u \rightarrow_t^* \lambda x_1 . \dots . \lambda x_n . u_0 & u_1 & \dots & u_m & & & \\ & \Downarrow_s & & \Downarrow_s & & & \\ & \lambda x_1 . \dots . \lambda x_n . v_0 & v_1 & \dots & v_m & \stackrel{\text{tête}}{=} & v \end{array}$$

La proposition clé (R. David)

- ❖ **Proposition.** Si $u \rightarrow^* v$ alors $u \Rightarrow_s v$.
- ❖ La démonstration se fait par une série assez longue de lemmes
(voir exercices)

La proposition clé (R. David)

- ❖ **Proposition.** Si $u \rightarrow^* v$ alors $u \Rightarrow_s v$.
- ❖ La démonstration se fait par une série assez longue de lemmes
(voir exercices)
- ❖ C'est essentiellement un **tri**, qui réordonne les réductions de sorte à mettre les plus hautes d'abord

La proposition clé (R. David)

- ❖ **Proposition.** Si $u \rightarrow^* v$ alors $u \Rightarrow_s v$.
- ❖ La démonstration se fait par une série assez longue de lemmes
(voir exercices)
- ❖ C'est essentiellement un **tri**, qui réordonne les réductions de sorte à mettre les plus hautes d'abord
- ❖ Le lemme important dit:
Lemme de tri. si $u \Rightarrow_s w \rightarrow v$, alors $u \Rightarrow_s v$
(Intuition: si la réduction $w \rightarrow v$ était effectuée trop haut, on aurait pu la permuter avec des réductions antérieures de u à w opérant plus bas)

La proposition clé (R. David)

❖ On peut maintenant montrer:

Proposition. Si $u \rightarrow^* v$ alors $u \Rightarrow_s v$.

❖ Par récurrence sur la longueur k de la réduction $u \rightarrow^* v$

La proposition clé (R. David)

❖ On peut maintenant montrer:

Proposition. Si $u \rightarrow^* v$ alors $u \Rightarrow_s v$.

❖ Par récurrence sur la longueur k de la réduction $u \rightarrow^* v$

❖ Si $k=0$,

(a) $u \Rightarrow_s u$.

La proposition clé (R. David)

- ❖ On peut maintenant montrer:

Proposition. Si $u \rightarrow^* v$ alors $u \Rightarrow_s v$.

- ❖ Par récurrence sur la longueur k de la réduction $u \rightarrow^* v$
- ❖ Si $k=0$, (a) $u \Rightarrow_s u$.
- ❖ Sinon, il existe un w tel que $u \rightarrow^* w$ en $\leq k-1$ étapes
et $w \rightarrow v$

La proposition clé (R. David)

- ❖ On peut maintenant montrer:

Proposition. Si $u \rightarrow^* v$ alors $u \Rightarrow_s v$.

- ❖ Par récurrence sur la longueur k de la réduction $u \rightarrow^* v$
- ❖ Si $k=0$, (a) $u \Rightarrow_s u$.
- ❖ Sinon, il existe un w tel que $u \rightarrow^* w$ en $\leq k-1$ étapes
et $w \rightarrow v$
- ❖ Par hyp. réc., $u \Rightarrow_s w$

La proposition clé (R. David)

- ❖ On peut maintenant montrer:

Proposition. Si $u \rightarrow^* v$ alors $u \Rightarrow_s v$.

- ❖ Par récurrence sur la longueur k de la réduction $u \rightarrow^* v$

- ❖ Si $k=0$, (a) $u \Rightarrow_s u$.

- ❖ Sinon, il existe un w tel que $u \rightarrow^* w$ en $\leq k-1$ étapes
et $w \rightarrow v$

- ❖ Par hyp. réc., $u \Rightarrow_s w$

- ❖ Donc $u \Rightarrow_s v$.

Lemme de tri. Si $u \Rightarrow_s w \rightarrow v$ alors $u \Rightarrow_s v$.

La proposition clé (R. David)

- ❖ On peut maintenant montrer:

Proposition. Si $u \rightarrow^* v$ alors $u \Rightarrow_s v$.

- ❖ Par récurrence sur la longueur k de la réduction $u \rightarrow^* v$

- ❖ Si $k=0$, (a) $u \Rightarrow_s u$.

Je vous laisse démontrer que la réciproque est vraie aussi:
donc $u \rightarrow^* v$ si et seulement si $u \Rightarrow_s v$.

- ❖ Sinon, il existe un w tel que $u \rightarrow^* w$ en $\leq k-1$ étapes
et $w \rightarrow v$

- ❖ Par hyp. réc., $u \Rightarrow_s w$

- ❖ Donc $u \Rightarrow_s v$.

Lemme de tri. Si $u \Rightarrow_s w \rightarrow v$ alors $u \Rightarrow_s v$.

Réduction standard vers une forme normale

❖ Un point subtil:

si $u \Rightarrow_s v$,

alors $\lambda x_1 . \dots . \lambda x_n . u_0 u_1 \dots u_m$ n'est **pas nécessairement**

la forme normale de tête de u

($u_0 u_1$ peut être un redex)

Réduction standard vers une forme normale

❖ Un point subtil:

$$\text{si } u \Rightarrow_s v, \quad u \rightarrow_t^* \lambda x_1 . \dots . \lambda x_n . u_0 u_1 \dots u_m$$
$$\lambda x_1 . \dots . \lambda x_n . \begin{array}{c} \Downarrow_s \\ v_0 \end{array} \begin{array}{c} \Downarrow_s \\ v_1 \end{array} \dots \begin{array}{c} \Downarrow_s \\ v_m \end{array} = v$$

alors $\lambda x_1 . \dots . \lambda x_n . u_0 u_1 \dots u_m$ n'est **pas nécessairement**
la forme normale de tête de u
($u_0 u_1$ peut être un rédex)

Réduction standard vers une forme normale

- ❖ Un point subtil:

$$\text{si } u \Rightarrow_s v, \quad u \rightarrow_t^* \lambda x_1 . \dots . \lambda x_n . u_0 u_1 \dots u_m$$
$$\lambda x_1 . \dots . \lambda x_n . \begin{array}{c} \Downarrow_s \\ v_0 \end{array} \begin{array}{c} \Downarrow_s \\ v_1 \end{array} \dots \begin{array}{c} \Downarrow_s \\ v_m \end{array} = v$$

alors $\lambda x_1 . \dots . \lambda x_n . u_0 u_1 \dots u_m$ n'est **pas nécessairement**
la forme normale de tête de u

($u_0 u_1$ peut être un redex)

- ❖ Mais ceci n'arrive pas si v est en **forme normale**
(car alors v_0 est une variable, et donc $u_0=v_0$ par déf. de \Rightarrow_s)

Réductions standard, réductions gauches

- ❖ **Proposition.** Si $u \Rightarrow_s v$ et si v est normal, alors $u \rightarrow^* v$ par une réduction gauche
- ❖ Par récurrence sur $|v|$

Réductions standard, réductions gauches

❖ **Proposition.** Si $u \Rightarrow_s v$ et si v est normal, alors $u \rightarrow^* v$ par une réduction gauche

❖ Par récurrence sur $|v|$

❖ Comme v normal:

$$u \rightarrow_t^* \lambda x_1 . \dots . \lambda x_n . x \ u_1 \ \dots \ u_m$$
$$\lambda x_1 . \dots . \lambda x_n . x \ \begin{array}{c} \parallel \\ \downarrow \\ v_1^s \end{array} \ \dots \ \begin{array}{c} \parallel \\ \downarrow \\ v_m^s \end{array} = v$$

Réductions standard, réductions gauches

❖ **Proposition.** Si $u \Rightarrow_s v$ et si v est normal, alors $u \rightarrow^* v$ par une réduction gauche

❖ Par récurrence sur $|v|$

toute réduction de tête est gauche

❖ Comme v normal:

$$u \rightarrow_t^* \lambda x_1 . \dots . \lambda x_n . x u_1 \dots u_m$$
$$\lambda x_1 . \dots . \lambda x_n . x v_1^s \dots v_m^s = v$$

\parallel \downarrow \downarrow

Réductions standard, réductions gauches

❖ **Proposition.** Si $u \Rightarrow_s v$ et si v est normal, alors $u \rightarrow^* v$ par une réduction gauche

❖ Par récurrence sur $|v|$

toute réduction de tête est gauche

❖ Comme v normal:

$$u \rightarrow_t^* \lambda x_1 . \dots . \lambda x_n . x u_1 \dots u_m$$
$$\lambda x_1 . \dots . \lambda x_n . x v_1^s \dots v_m^s = v$$

\parallel \downarrow \downarrow

❖ Donc $u \rightarrow_{\text{gauche}}^* \lambda x_1 . \dots . \lambda x_n . x u_1 u_2 \dots u_m$

Réductions standard, réductions gauches

❖ **Proposition.** Si $u \Rightarrow_s v$ et si v est normal, alors $u \rightarrow^* v$ par une réduction gauche

❖ Par récurrence sur $|v|$

toute réduction de tête est gauche

❖ Comme v normal:

$$u \rightarrow_t^* \lambda x_1 . \dots . \lambda x_n . x u_1 \dots u_m$$

$$\lambda x_1 . \dots . \lambda x_n . x v_1^s \dots v_m^s = v$$

\parallel \downarrow \downarrow
 \downarrow \downarrow \downarrow

❖ Donc $u \rightarrow_{\text{gauche}}^* \lambda x_1 . \dots . \lambda x_n . x u_1 u_2 \dots u_m$

\downarrow gauche par hyp. réc.

❖ $\rightarrow_{\text{gauche}}^* \lambda x_1 . \dots . \lambda x_n . x v_1^* u_2 \dots u_m$

Réductions standard, réductions gauches

❖ **Proposition.** Si $u \Rightarrow_s v$ et si v est normal, alors $u \rightarrow^* v$ par une réduction gauche

❖ Par récurrence sur $|v|$

toute réduction de tête est gauche

❖ Comme v normal:

$$u \rightarrow_t^* \lambda x_1 . \dots . \lambda x_n . x u_1 \dots u_m$$

$$\lambda x_1 . \dots . \lambda x_n . x v_1^s \dots v_m^s = v$$

\parallel \Downarrow \Downarrow
 \Downarrow \Downarrow \Downarrow

❖ Donc $u \rightarrow_{\text{gauche}}^* \lambda x_1 . \dots . \lambda x_n . x u_1 u_2 \dots u_m$

gauche dans u_1 , donc dans tout le terme car pas de redex plus à gauche

$\downarrow_{\text{gauche}}^*$ par hyp. réc.

❖ $\rightarrow_{\text{gauche}}^* \lambda x_1 . \dots . \lambda x_n . x v_1 u_2 \dots u_m$

Réductions standard, réductions gauches

❖ **Proposition.** Si $u \Rightarrow_s v$ et si v est normal, alors $u \rightarrow^* v$ par une réduction gauche

❖ Par récurrence sur $|v|$

toute réduction de tête est gauche

❖ Comme v normal:

$$u \rightarrow_t^* \lambda x_1 . \dots . \lambda x_n . x u_1 \dots u_m$$

$$\lambda x_1 . \dots . \lambda x_n . x v_1^s \dots v_m^s = v$$

\parallel \Downarrow \Downarrow
 \Downarrow \Downarrow \Downarrow

❖ Donc $u \rightarrow_{\text{gauche}}^* \lambda x_1 . \dots . \lambda x_n . x u_1 u_2 \dots u_m$

gauche dans u_1 , donc dans tout le terme car pas de redex plus à gauche

$\downarrow_{\text{gauche}}^*$ par hyp. réc.

❖ $\rightarrow_{\text{gauche}}^* \lambda x_1 . \dots . \lambda x_n . x v_1 u_2 \dots u_m$

$\downarrow_{\text{gauche}}^*$ par hyp. réc.

❖ $\rightarrow_{\text{gauche}}^* \lambda x_1 . \dots . \lambda x_n . x v_1 v_2 \dots u_m$

Réductions standard, réductions gauches

❖ **Proposition.** Si $u \Rightarrow_s v$ et si v est normal, alors $u \rightarrow^* v$ par une réduction gauche

❖ Par récurrence sur $|v|$

toute réduction de tête est gauche

❖ Comme v normal:

$$u \rightarrow_t^* \lambda x_1 . \dots . \lambda x_n . x u_1 \dots u_m$$

$$\lambda x_1 . \dots . \lambda x_n . x v_1^s \dots v_m^s = v$$

\parallel \Downarrow \Downarrow
 \Downarrow \Downarrow \Downarrow

❖ Donc $u \rightarrow_{\text{gauche}}^* \lambda x_1 . \dots . \lambda x_n . x u_1 u_2 \dots u_m$

gauche dans u_1 , donc dans tout le terme car pas de redex plus à gauche

$\downarrow_{\text{gauche}}^*$ par hyp. réc.

❖ $\rightarrow_{\text{gauche}}^* \lambda x_1 . \dots . \lambda x_n . x v_1 u_2 \dots u_m$

gauche dans u_2 , donc dans tout le terme car pas de redex plus à gauche

$\downarrow_{\text{gauche}}^*$ par hyp. réc.

❖ $\rightarrow_{\text{gauche}}^* \lambda x_1 . \dots . \lambda x_n . x v_1 v_2 \dots u_m$

Réductions standard, réductions gauches

❖ **Proposition.** Si $u \Rightarrow_s v$ et si v est normal, alors $u \rightarrow^* v$ par une réduction gauche

❖ Par récurrence sur $|v|$

toute réduction de tête est gauche

❖ Comme v normal:

$$u \rightarrow_t^* \lambda x_1 . \dots . \lambda x_n . x u_1 \dots u_m$$

$$\lambda x_1 . \dots . \lambda x_n . x v_1^s \dots v_m^s = v$$

\parallel \downarrow \downarrow
 \downarrow \downarrow \downarrow

❖ Donc $u \rightarrow_{\text{gauche}}^* \lambda x_1 . \dots . \lambda x_n . x u_1 u_2 \dots u_m$

gauche dans u_1 , donc dans tout le terme car pas de redex plus à gauche

\downarrow gauche par hyp. réc.

❖ $\rightarrow_{\text{gauche}}^* \lambda x_1 . \dots . \lambda x_n . x v_1 u_2 \dots u_m$

gauche dans u_2 , donc dans tout le terme car pas de redex plus à gauche

\downarrow gauche par hyp. réc.

❖ $\rightarrow_{\text{gauche}}^* \lambda x_1 . \dots . \lambda x_n . x v_1 v_2 \dots u_m$

❖ ... $\rightarrow_{\text{gauche}}^* \lambda x_1 . \dots . \lambda x_n . x v_1 v_2 \dots v_m$

Théorème de standardisation

- ❖ **Théorème.** Pour tous λ -termes u, v , les affirmations:
1. $u \rightarrow^* v$ 2. $u \rightarrow^* v$ par réduction gauche 3. $u \Rightarrow_s v$
- sont telles que $2 \Rightarrow 1 \Leftrightarrow 3$.
- Si de plus v est **normal**, $1 \Leftrightarrow 2 \Leftrightarrow 3$.

Théorème de standardisation

- ❖ **Théorème.** Pour tous λ -termes u, v , les affirmations:
1. $u \rightarrow^* v$ 2. $u \rightarrow^* v$ par réduction gauche 3. $u \Rightarrow_s v$
sont telles que $2 \Rightarrow 1 \Leftrightarrow 3$.
Si de plus v est **normal**, $1 \Leftrightarrow 2 \Leftrightarrow 3$.
- ❖ On a donc un **algorithme** pour trouver la forme normale d'un terme u , si elle existe:
 - tant que u n'est pas normal, réduire son redex gauche

Théorème de standardisation

- ❖ **Théorème.** Pour tous λ -termes u, v , les affirmations:
1. $u \rightarrow^* v$ 2. $u \rightarrow^* v$ par réduction gauche 3. $u \Rightarrow_s v$
sont telles que $2 \Rightarrow 1 \Leftrightarrow 3$.
Si de plus v est **normal**, $1 \Leftrightarrow 2 \Leftrightarrow 3$.
- ❖ On a donc un **algorithme** pour trouver la forme normale d'un terme u , si elle existe:
— tant que u n'est pas normal, réduire son redex gauche
- ❖ (Bien entendu, si u n'est pas normalisable, cet algorithme ne termine pas.)

Remarques

- ❖ Une stratégie **standardisante** est une stratégie de réduction qui trouve la forme normale d'un λ -terme si elle existe.
- ❖ Nous venons de voir que la **stratégie (externe) gauche** est standardisante.
- ❖ Ce n'est pas la seule: par exemple, les stratégies **quasi-internes** sont aussi standardisantes (voir exercices).

*Annexe C: quelques
implémentations simples du λ -calcul*

Implémentations du λ -calcul

- ❖ Un interprète naïf en Caml
- ❖ Machines de Krivine
- ❖ Machines à environnement

Représentation des λ -termes

```
type lambda_term = VAR of string
                  | APPL of lambda_term * lambda_term
                  | ABS of string * lambda_term
```

Représentation des λ -termes

```
type lambda_term = VAR of string
                  | APPL of lambda_term * lambda_term
                  | ABS of string * lambda_term
```

Par exemple, $\lambda x . y(\lambda z . yxzx)$ serait:

```
ABS ("x", APPL (VAR "y",
                ABS ("z",
                    APPL (APPL (APPL (VAR "y", VAR "x"),
                                    VAR "z"),
                            VAR "x" )))
```

Un premier interprète

```
type lambda_term = VAR of string
                  | APPL of lambda_term * lambda_term
                  | ABS of string * lambda_term
```

```
let rec norm (t : lambda_term) = (* calcule la forme normale de t *)
  match t with
  | VAR x -> t
  | ABS (x, u) -> ABS (x, norm u)
  | APPL (u, v) ->
    let u' = norm u in
    match u' with
    | ABS (y, u1) -> norm (subst u1 y v)
    | _ -> APPL (u', norm v)
```


Un premier interprète

```
type lambda_term = VAR of string
                  | APPL of lambda_term * lambda_term
                  | ABS of string * lambda_term
```

Réduction complète.

Pour une réduction faible, écrire `t` à la place de `ABS (x, norm u)`

```
let rec norm (t : lambda_term) = (* calcule la forme normale de t *)
  match t with
  | VAR x -> t
  | ABS (x, u) -> ABS (x, norm u)
  | APPL (u, v) ->
    let u' = norm u in
    match u' with
    | ABS (y, u1) -> norm (subst u1 y v)
    | _ -> APPL (u', norm v)
```

Un premier interprète

```
type lambda_term = VAR of string
                  | APPL of lambda_term * lambda_term
                  | ABS of string * lambda_term
```

Réduction complète.

Pour une réduction faible, écrire `t` à la place de `ABS (x, norm u)`

```
let rec norm (t : lambda_term) = (* calcule la forme normale de t *)
  match t with
  | VAR x -> t
  | ABS (x, u) -> ABS (x, norm u)
  | APPL (u, v) ->
    let u' = norm u in
    match u' with
    | ABS (y, u1) -> norm (subst u1 y v)
    | _ -> APPL (u', norm v)
```

Réduction gauche

Un premier interprète

```
type lambda_term = VAR of string
                  | APPL of lambda_term * lambda_term
                  | ABS of string * lambda_term
```

Réduction complète.

Pour une réduction faible, écrire `t` à la place de `ABS (x, norm u)`

```
let rec norm (t : lambda_term) = (* calcule la forme normale de t *)
  match t with
  | VAR x -> t
  | ABS (x, u) -> ABS (x, norm u)
  | APPL (u, v) ->
    let u' = norm u in
    match u' with
    | ABS (y, u1) -> norm (subst u1 y v)
    | _ -> APPL (u', norm v)
```

À définir...

(attention au α -renommage!)

Réduction gauche

Un premier interprète

```
type lambda_term = VAR of string
                  | APPL of lambda_term * lambda_term
                  | ABS of string * lambda_term
```

Réduction complète.

Pour une réduction faible, écrire `t` à la place de `ABS (x, norm u)`

```
let rec norm (t : lambda_term) = (* calcule la forme normale de t *)
  match t with
  | VAR x -> t
  | ABS (x, u) -> ABS (x, norm u)
  | APPL (u, v) ->
    let u' = norm u in
    match u' with
    | ABS (y, u1) -> norm (subst u1 y v)
    | _ -> APPL (u', norm v)
```

À définir...

(attention au α -renommage!)

Réduction gauche

Efficacité?
(pas terrible...)

Un premier interprète

```
type lambda_term = VAR of string
                  | APPL of lambda_term * lambda_term
                  | ABS of string * lambda_term
```

Réduction complète.

Pour une réduction faible, écrire `t` à la place de `ABS (x, norm u)`

```
let rec norm (t : lambda_term) = (* calcule la forme normale de t *)
  match t with
  | VAR x -> t
  | ABS (x, u) -> ABS (x, norm u)
  | APPL (u, v) ->
    let u' = norm u in
    match u' with
    | ABS (y, u1) -> norm (subst u1 y v)
    | _ -> APPL (u', norm v)
```

Réduction gauche

À définir...

(attention au α -renommage!)

Efficacité?

`u1` est normal, mais on va quand même normaliser `subst u1 y v`

Efficacité?

(pas terrible...)

Un premier interprète

```
type lambda_term = VAR of string
                  | APPL of lambda_term * lambda_term
                  | ABS of string * lambda_term
```

Ce code est-il correct?

Réduction complète.

Pour une réduction faible, écrire `t` à la place de `ABS (x, norm u)`

```
let rec norm (t : lambda_term) = (* calcule la forme normale de t *)
  match t with
  | VAR x -> t
  | ABS (x, u) -> ABS (x, norm u)
  | APPL (u, v) ->
    let u' = norm u in
    match u' with
    | ABS (y, u1) -> norm (subst u1 y v)
    | _ -> APPL (u', norm v)
```

Réduction gauche

À définir...
(attention au α -renommage!)

Efficacité?

`u1` est normal, mais on va quand même normaliser `subst u1 y v`

Efficacité?
(pas terrible...)

L'astuce de Krivine



<https://www.irif.fr/~padovani/Images/Gandvine.jpg>

- ❖ Pour éviter la réévaluation, on garde une liste args des arguments en attente

```
let rec apply_head (t : lambda_term) (args : lambda_term list) =  
  match args with  
  | nil -> t  
  | u::rest -> apply_head (APPL (t, norm2 u)) rest
```

```
let rec norm2 (t : lambda_term) (args : lambda_term list) =  
  match t with  
  | VAR x -> apply_head t args  
  | ABS (x, u) -> (match args with  
                    | nil -> ABS (x, norm2 u nil)  
                    | v::rest -> norm2 (subst u y v) rest)  
  | APPL (u, v) -> norm2 u (v::args)
```

L'astuce de Krivine



<https://www.irif.fr/~padovani/Images/Gandvine.jpg>

- ❖ Pour éviter la réévaluation, on garde une liste `args` des arguments en attente

Si $\text{args}=[v_1; \dots; v_n]$,
calcule la forme
normale
de $t \ v_1 \ \dots \ v_n$

```
let rec apply_head (t : lambda_term) (args : lambda_term list) =  
  match args with  
  | nil -> t  
  | u::rest -> apply_head (APPL (t, norm2 u)) rest
```

```
let rec norm2 (t : lambda_term) (args : lambda_term list) =  
  match t with  
  | VAR x -> apply_head t args  
  | ABS (x, u) -> (match args with  
                    | nil -> ABS (x, norm2 u nil)  
                    | v::rest -> norm2 (subst u y v) rest)  
  | APPL (u, v) -> norm2 u (v::args)
```


L'astuce de Krivine



- ❖ Pour éviter la réévaluation, on garde une liste `args` des arguments en attente

Si $\text{args}=[v_1; \dots; v_n]$,
calcule la forme
normale
de $t \ v_1 \ \dots \ v_n$

Réduction complète.
Pour une réduction faible,
écrire t à la place
de `ABS (x, norm2 u nil)`
(+enlever `norm2` du code de `apply_head`)

```
let rec apply_head (t : lambda_term) (args : lambda_term list) =  
  match args with  
  | nil -> t  
  | u::rest -> apply_head (APPL (t, norm2 u)) rest  
  
let rec norm2 (t : lambda_term) (args : lambda_term list) =  
  match t with  
  | VAR x -> apply_head t args  
  | ABS (x, u) -> (match args with  
                    | nil -> ABS (x, norm2 u nil)  
                    | v::rest -> norm2 (subst u y v) rest)  
  | APPL (u, v) -> norm2 u (v::args)
```

L'astuce de Krivine



- ❖ Pour éviter la réévaluation, on garde une liste `args` des arguments en attente

Si $\text{args}=[v_1; \dots; v_n]$,
calcule la forme
normale
de $t \ v_1 \ \dots \ v_n$

Réduction complète.
Pour une réduction faible,
écrire t à la place
de `ABS (x, norm2 u nil)`
(+enlever `norm2` du code de `apply_head`)

```
let rec apply_head (t : lambda_term) (args : lambda_term list) =  
  match args with  
  | nil -> t  
  | u::rest -> apply_head (APPL (t, norm2 u)) rest
```

```
let rec norm2 (t : lambda_term) (args : lambda_term list) =  
  match t with  
  | VAR x -> apply_head t args  
  | ABS (x, u) -> (match args with  
                    | nil -> ABS (x, norm2 u nil)  
                    | v::rest -> norm2 (subst u y v) rest)  
  | APPL (u, v) -> norm2 u (v::args)
```

La β -réduction est ici

L'astuce de Krivine



- ❖ Pour éviter la réévaluation, on garde une liste `args` des arguments en attente

Si $\text{args}=[v_1; \dots; v_n]$,
calcule la forme normale
de $t \ v_1 \ \dots \ v_n$

Réduction complète.
Pour une réduction faible,
écrire t à la place
de `ABS (x, norm2 u nil)`
(+enlever `norm2` du code de `apply_head`)

```
let rec apply_head (t : lambda_term) (args : lambda_term list) =  
  match args with  
  | nil -> t  
  | u::rest -> apply_head (APPL (t, norm2 u)) rest
```

```
let rec norm2 (t : lambda_term) (args : lambda_term list) =  
  match t with  
  | VAR x -> apply_head t args  
  | ABS (x, u) -> (match args with  
    | nil -> ABS (x, norm2 u nil)  
    | v::rest -> norm2 (subst u y v) rest)  
  | APPL (u, v) -> norm2 u (v::args)
```

La β -réduction est ici

Exploration du
terme:
recherche du
rédex de tête

Une machine de Krivine

❖ On abstrait tout ça sous forme de **règles** de sémantique opérationnelle

❖ (explore) $uv, args \rightsquigarrow u, v::args$

(β) $\lambda x . u, v::rest \rightsquigarrow u[x:=v], rest$

pour la réduction de tête faible:

$$(\lambda x . s) u_1 \dots u_m$$
$$\rightarrow_{\text{tf}} s[x:=u_1] u_2 \dots u_m$$

Une **configuration**

$u, [v_1; \dots; v_m]$ représente le λ -terme $uv_1 \dots v_m$

Une machine de Krivine

❖ On abstrait tout ça sous forme de **règles** de sémantique opérationnelle

❖ (explore) $uv, args \rightsquigarrow u, v::args$

(β) $\lambda x . u, v::rest \rightsquigarrow u[x:=v], rest$

❖ **Thm (correction).** Si $u, [v_1; \dots; v_n] \rightsquigarrow^* u', [v'_1; \dots; v'_m]$

alors $u v_1 \dots v_n \rightarrow_{\text{tf}}^* u' v'_1 \dots v'_m$ (réduction de tête faible)

pour la réduction de tête faible:

$$(\lambda x . s) u_1 \dots u_m$$
$$\rightarrow_{\text{tf}} s[x:=u_1] u_2 \dots u_m$$

Une **configuration**

$u, [v_1; \dots; v_m]$ représente le λ -terme $uv_1 \dots v_m$

Une machine de Krivine

- ❖ On abstrait tout ça sous forme de **règles** de sémantique opérationnelle

pour la réduction de tête faible:

$$\begin{aligned} & (\lambda x . s) u_1 \dots u_m \\ & \rightarrow_{\text{tf}} s[x:=u_1] u_2 \dots u_m \end{aligned}$$

- ❖ (explore) $uv, args \rightsquigarrow u, v::args$

Une **configuration**

$u, [v_1; \dots; v_m]$ représente le λ -terme $uv_1 \dots v_m$

(β) $\lambda x . u, v::rest \rightsquigarrow u[x:=v], rest$

- ❖ **Thm (correction).** Si $u, [v_1; \dots; v_n] \rightsquigarrow^* u', [v'_1; \dots; v'_m]$
alors $u v_1 \dots v_n \rightarrow_{\text{tf}}^* u' v'_1 \dots v'_m$ (réd. de tête faible)

- ❖ **Thm (progrès).** Les configurations stoppées
sont celles de la forme $x, [v_1; \dots; v_n]$ ou $\lambda x . u, []$

(NB: $x v_1 \dots v_n$ et $\lambda x . u$ sont les formes normales de tête faibles; exercice.)

Une machine de Krivine

pour la réduction de tête faible:

$$\begin{aligned} & (\lambda x . s) u_1 \dots u_m \\ & \rightarrow_{\text{tf}} s[x:=u_1] u_2 \dots u_m \end{aligned}$$

❖ On abstrait tout ça sous forme de **règles** de sémantique opérationnelle

❖ (explore) $uv, args \rightsquigarrow u, v::args$

Une **configuration**

$u, [v_1; \dots; v_m]$ représente le λ -terme $uv_1 \dots v_m$

(β) $\lambda x . u, v::rest \rightsquigarrow u[x:=v], rest$

❖ **Thm (correction).** Si $u, [v_1; \dots; v_n] \rightsquigarrow^* u', [v'_1; \dots; v'_m]$
alors $u v_1 \dots v_n \rightarrow_{\text{tf}}^* u' v'_1 \dots v'_m$ (réd. de tête faible)

❖ **Thm (progrès).** Les configurations stoppées
sont celles de la forme $x, [v_1; \dots; v_n]$ ou $\lambda x . u, []$

(NB: $x v_1 \dots v_n$ et $\lambda x . u$ sont les formes normales de tête faibles; exercice.)

Reste quand même à définir la
substitution... ou à la remplacer (efficacité)

Réduction de tête (pas faible)

❖ On peut étendre le principe à d'autres stratégies de réduction:

❖ Réduction de tête (pas faible)

$u, [v_1; \dots; v_m] / [x_1; \dots; x_n]$ représente
 $\lambda x_n, \dots, x_1 . uv_1 \dots v_m$

❖ (explore-lam) $\lambda x . u, [] / vars \rightsquigarrow u, [] / x::vars$

(explore-app) $uv, args/vars \rightsquigarrow u, v::args / vars$

(β) $\lambda x . u, v::rest / vars \rightsquigarrow u[x:=v], rest / vars$

(Voir exercices)

Reste quand même à définir la
[substitution](#)... ou à la remplacer (efficacité)

Réduction de tête en OCaml

```
❖ let rec red_t t args vars =  
  match t with  
  | ABS (x, u) ->  
    (match args with  
     | [] ->  
       red_t u [] (x::vars) (* explore-lam *)  
     | v::rest ->  
       red_t (subst u x v) rest vars (* beta *)  
    )  
  | APPL (u, v) ->  
    red_t u (v::args) vars (* explore-app *)  
  | _ -> (t, args, vars)  
  (* forme normale de tête λ vars . t args *)
```

```
(explore-lam)  $\lambda x . u, [] / vars \rightsquigarrow u, [] / x::vars$   
(explore-app)  $uv, args / vars \rightsquigarrow u, v::args / vars$   
( $\beta$ )  $\lambda x . u, v::rest / vars \rightsquigarrow u[x:=v], rest / vars$ 
```

```
❖ let rec subst u x v = ... (* exercice! *)
```

Réduction gauche en OCaml

- ❖

```
let rec red_t t args vars = ...  
    (* retourne la forme normale de tête  
    de  $\lambda$  vars . t args [si elle existe]  
    sous la forme (x, args', vars') *)
```
- ❖

```
let rec reconstruct x args' vars' = ...  
    (* reconstruit le  $\lambda$ -terme  $\lambda$  vars' . t args'  
    (exercice) *)
```
- ❖

```
let rec red t =  
    let (x, args', vars') = red_t t [] [] in  
    let args'_norm = List.map red args' in  
    reconstruct x args'_norm vars'
```

Principe:
on met t en forme **normale de tête**
 $\lambda x_1, \dots, x_n . uv_1 \dots v_m$
puis on réduit chaque v_i ,
récursivement