λ-calcul pour l'agrégation

Jean Goubault-Larrecq

Pour l'agrégation

- Ce que dit le programme (2022, 2023):

  « Lambda-calcul pur comme modèle de calcul :
  définition, propriétés (dont confluence), stratégies.
  Équivalence avec les machines de Turing et les fonctions récursives. »

- Pas de leçon (oral) sur le sujet.

- Le λ-calcul **typé** est hors-sujet.

- Liens avec les leçons:

2. Paradigmes de programmation : impératif, fonctionnel, objet. Exemples et applications.
30. Décidabilité et indécidabilité. Exemples.

⚠️ Vous pouvez y mentionner éventuellement un point de λ-calcul, mais pas plus (ce ne sont pas des leçons de λ-calcul).

L'essentiel

- **Motivation**: λ-calcul = cœur des langages fonctionnels
- **Définition**: α-équivalence, β-réduction, [η-réduction]
- **Propriétés** fondamentales:
  confluence, non-terminaison, développements finis, standardisation
- **Implémentations**: interprètes simples, machines de Krivine, [combinateurs]
- et **stratégies**:
  par valeur/nom, interne/externe, gauche/droite, faibles ou non,
  réductions de tête et réductions standard
- **Équivalence** avec les machines de Turing et les fonctions récursives;
  combinateurs de point fixe (Y)

Références bibliographiques

- 📖 Henk Barendregt
  The Lambda Calculus — Its Syntax and Semantics
  North-Holland (revised ed., 1984)
  l'encyclopédie! contient tout

- 📖 Jean-Louis Krivine
  Lambda-calcul, types et modèles [seuls les chapitres 1 et 2 sont au programme]
  Masson (1991)
  version anglaise en ligne: https://www.irif.fr/~krivine/articles/Lambda.pdf

- 📖 Richard Lassaigne et Michel de Rougemont
  Logique et fondements de l'informatique:
  logique du 1er ordre, calculabilité et lambda-calcul
  Hermès (1993)

- 📁 Thérèse Hardin-Accart, *cours de lambda-calcul*, DEA SPP, 2003-04
  http://www-spi.lip6.fr/~hardin/DEA/poly.pdf

- 📁 JGL, *cours de L3 « logique informatique — lambda-calcul »*, ENS Paris-Saclay,
  https://projects.lsv.fr/agreg/?page_id=113

Motivation

Alonzo Church

- Le λ-calcul a été inventé par…

Alonzo Church

Historiquement

- … en 1932 par A. Church
pour donner un fondement aux mathématiques basé sur la notion de fonction plutôt que d’ensemble
- (s’est révélé contradictoire plus tard…)

A SET OF POSTULATES FOR THE FOUNDATION OF LOGIC.[1]

BY ALONZO CHURCH.[2]

1. **Introduction.** In this paper we present a set of postulates for the foundation of formal logic, in which we avoid use of the free, or real, variable, and in which we introduce a certain restriction on the law of excluded middle as a means of avoiding the paradoxes connected with the mathematics of the transfinite.

Our reason for avoiding use of the free variable is that we require that every combination of symbols belonging to our system, if it represents a proposition at all, shall represent a particular proposition, unambiguously, and without the addition of verbal explanations. That the use of the free variable involves violation of this requirement, we believe is readily seen. For example, the identity

$$a(b+c) = ab+ac \quad (1)$$

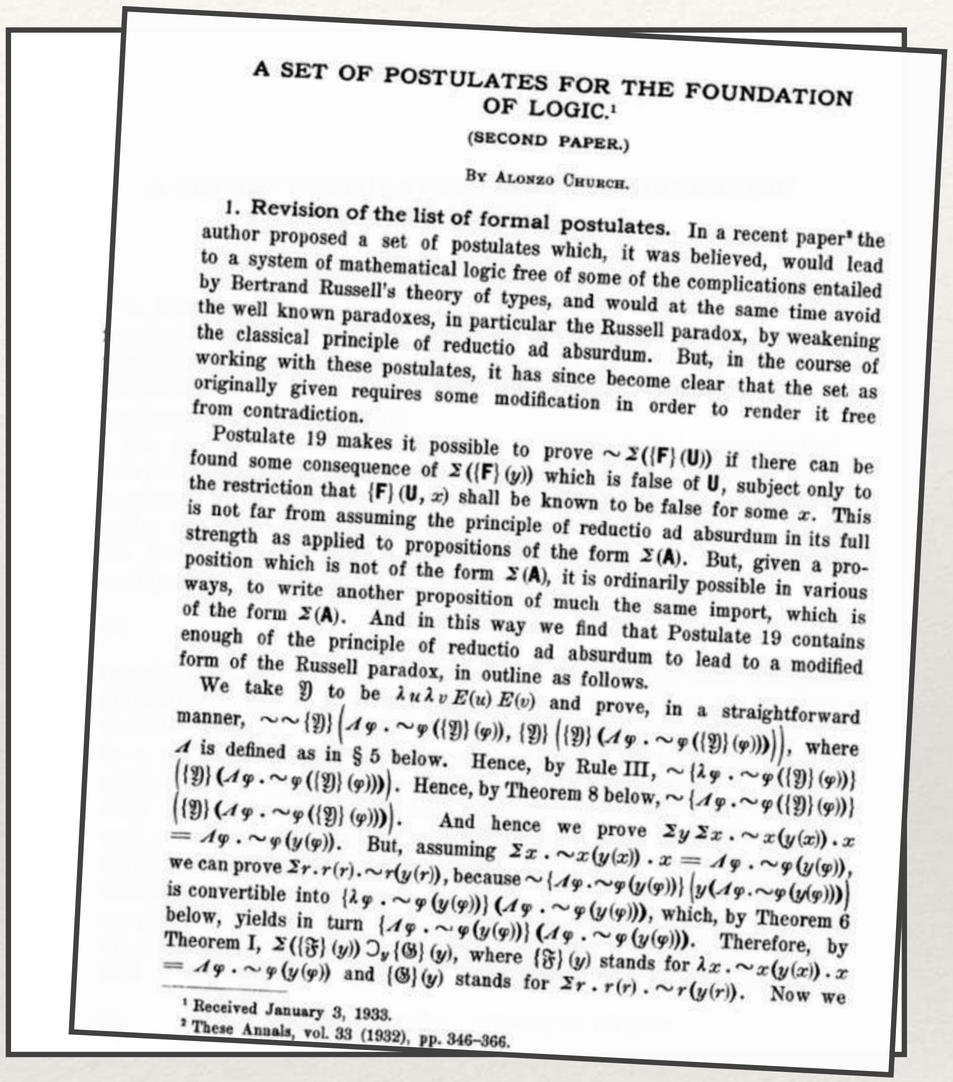
in which a, b, and c are used as free variables, does note state a definite proposition unless it is known what values may be taken on by these variables, and this information, if not implied in the context, must be given by a verbal addition. The range allowed to the variables a, b, and c might consist of all real numbers, or of all complex numbers, or of some other set, or the ranges allowed to the variables might differ, and for each possibility equation (1) has a different meaning. Clearly, when this equation is written alone, the proposition intended has not been completely translated into symbolic language, and, in order to make the translation complete, the necessary verbal addition must be expressed by means of the symbols of formal logic and included, with the equation, in the formula used to represent the proposition. When this is done we obtain, say,

$$R(a)\,R(b)\,R(c) \supset_{abc} \cdot\, a(b+c) = ab+ac \quad (2)$$

Annals of Mathematics, 2nd series
33(2), April 1932, pages 346-366
https://doi.org/10.2307/1968337

Historiquement

- …puis en 1933 par A. Church pour donner un fondement aux mathématiques basé sur la notion de fonction plutôt que d'ensemble
- (toujours contradictoire…
  détecté par J. B. Rosser en 1936)



A SET OF POSTULATES FOR THE FOUNDATION OF LOGIC.[1]

(SECOND PAPER.)

BY ALONZO CHURCH.

1. Revision of the list of formal postulates. In a recent paper[2] the author proposed a set of postulates which, it was believed, would lead to a system of mathematical logic free of some of the complications entailed by Bertrand Russell's theory of types, and would at the same time avoid the well known paradoxes, in particular the Russell paradox, by weakening the classical principle of reductio ad absurdum. But, in the course of working with these postulates, it has since become clear that the set as originally given requires some modification in order to render it free from contradiction.

Postulate 19 makes it possible to prove $\sim\Sigma(\{\mathbf{F}\}(\mathbf{U}))$ if there can be found some consequence of $\Sigma(\{\mathbf{F}\}(y))$ which is false of $\mathbf{U}$, subject only to the restriction that $\{\mathbf{F}\}(\mathbf{U}, x)$ shall be known to be false for some x. This is not far from assuming the principle of reductio ad absurdum in its full strength as applied to propositions of the form $\Sigma(\mathbf{A})$. But, given a proposition which is not of the form $\Sigma(\mathbf{A})$, it is ordinarily possible in various ways, to write another proposition of much the same import, which is of the form $\Sigma(\mathbf{A})$. And in this way we find that Postulate 19 contains enough of the principle of reductio ad absurdum to lead to a modified form of the Russell paradox, in outline as follows.

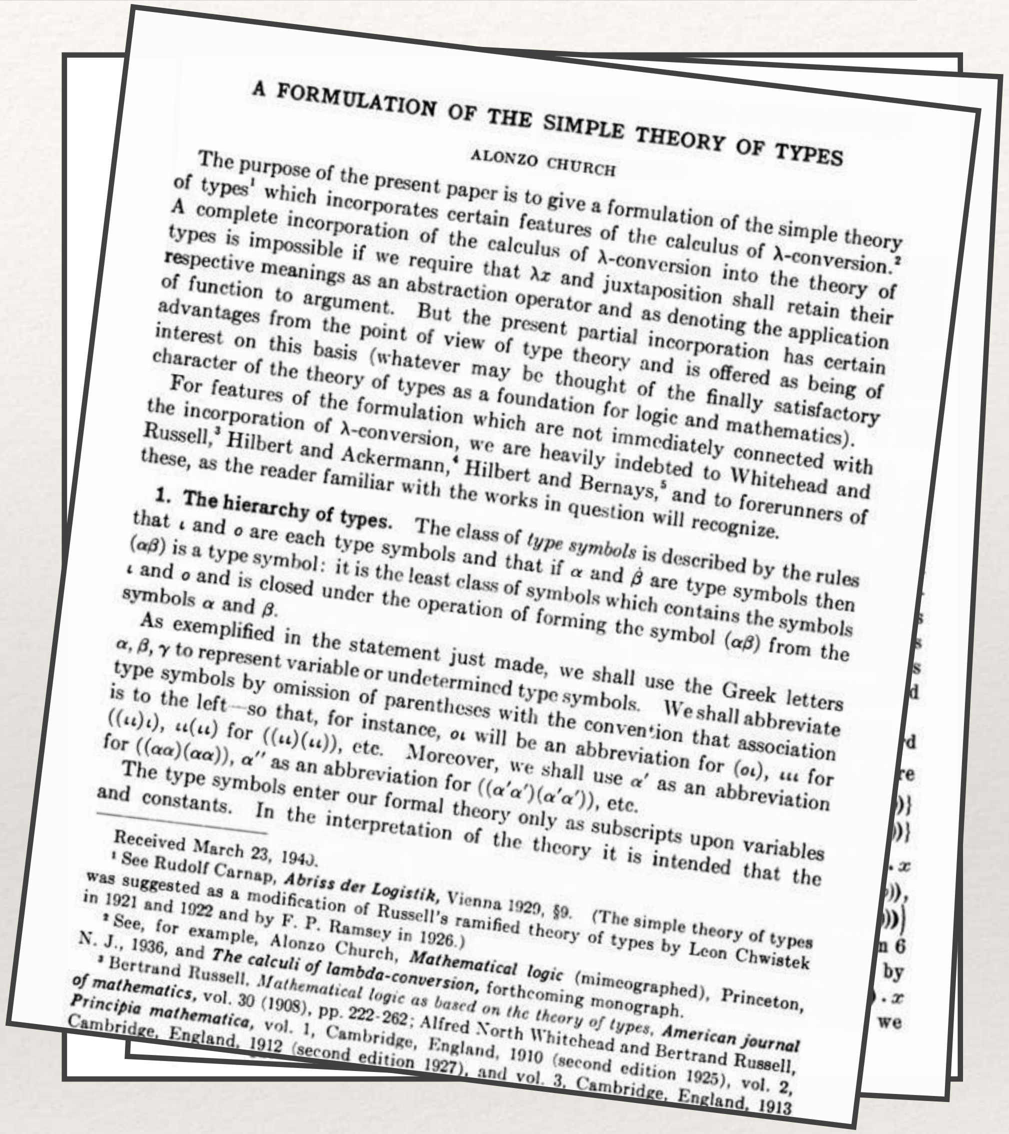
We take $\mathfrak{Y}$ to be $\lambda u \lambda v E(u) E(v)$ and prove, in a straightforward manner, $\sim\sim\{\mathfrak{Y}\}\big(\Lambda\varphi\,.\sim\varphi(\{\mathfrak{Y}\}(\varphi)), \{\mathfrak{Y}\}\big(\{\mathfrak{Y}\}(\Lambda\varphi\,.\sim\varphi(\{\mathfrak{Y}\}(\varphi)))\big)\big)$, where Λ is defined as in § 5 below. Hence, by Rule III, $\sim\{\lambda\varphi\,.\sim\varphi(\{\mathfrak{Y}\}(\varphi))\}\big(\{\mathfrak{Y}\}(\Lambda\varphi\,.\sim\varphi(\{\mathfrak{Y}\}(\varphi)))\big)$. Hence, by Theorem 8 below, $\sim\{\Lambda\varphi\,.\sim\varphi(\{\mathfrak{Y}\}(\varphi))\}\big(\{\mathfrak{Y}\}(\Lambda\varphi\,.\sim\varphi(\{\mathfrak{Y}\}(\varphi)))\big)$. And hence we prove $\Sigma y\,\Sigma x\,.\sim x(y(x))\,.\,x = \Lambda\varphi\,.\sim\varphi(y(\varphi))$. But, assuming $\Sigma x\,.\sim x(y(x))\,.\,x = \Lambda\varphi\,.\sim\varphi(y(\varphi))$, we can prove $\Sigma r\,.\,r(r)\,.\sim r(y(r))$, because $\sim\{\Lambda\varphi\,.\sim\varphi(y(\varphi))\}\big(y(\Lambda\varphi\,.\sim\varphi(y(\varphi)))\big)$ is convertible into $\{\lambda\varphi\,.\sim\varphi(y(\varphi))\}(\Lambda\varphi\,.\sim\varphi(y(\varphi)))$, which, by Theorem 6 below, yields in turn $\{\Lambda\varphi\,.\sim\varphi(y(\varphi))\}(\Lambda\varphi\,.\sim\varphi(y(\varphi)))$. Therefore, by Theorem I, $\Sigma(\{\mathfrak{F}\}(y)) \supset_y \{\mathfrak{G}\}(y)$, where $\{\mathfrak{F}\}(y)$ stands for $\lambda x\,.\sim x(y(x))\,.\,x = \Lambda\varphi\,.\sim\varphi(y(\varphi))$ and $\{\mathfrak{G}\}(y)$ stands for $\Sigma r\,.\,r(r)\,.\sim r(y(r))$. Now we

[1] Received January 3, 1933.
[2] These Annals, vol. 33 (1932), pp. 346–366.

Annals of Mathematics, 2nd series
34(4), October 1933, pages 839-864
https://doi.org/10.2307/1968702

Historiquement

- …puis en 1940 par A. Church pour donner un fondement à la logique d'ordre supérieur



A FORMULATION OF THE SIMPLE THEORY OF TYPES

ALONZO CHURCH

The purpose of the present paper is to give a formulation of the simple theory of types[1] which incorporates certain features of the calculus of λ-conversion.[2] A complete incorporation of the calculus of λ-conversion into the theory of types is impossible if we require that λx and juxtaposition shall retain their respective meanings as an abstraction operator and as denoting the application of function to argument. But the present partial incorporation has certain advantages from the point of view of type theory and is offered as being of interest on this basis (whatever may be thought of the finally satisfactory character of the theory of types as a foundation for logic and mathematics).

For features of the formulation which are not immediately connected with the incorporation of λ-conversion, we are heavily indebted to Whitehead and Russell,[3] Hilbert and Ackermann,[4] Hilbert and Bernays,[5] and to forerunners of these, as the reader familiar with the works in question will recognize.

1. The hierarchy of types. The class of *type symbols* is described by the rules that ι and o are each type symbols and that if α and β are type symbols then (αβ) is a type symbol: it is the least class of symbols which contains the symbols ι and o and is closed under the operation of forming the symbol (αβ) from the symbols α and β.

As exemplified in the statement just made, we shall use the Greek letters α, β, γ to represent variable or undetermined type symbols. We shall abbreviate type symbols by omission of parentheses with the convention that association is to the left—so that, for instance, oι will be an abbreviation for (oι), ιιι for ((ιι)ι), ι(ιι) for ((ιι)(ιι)), etc. Moreover, we shall use α′ as an abbreviation for ((αα)(αα)), α″ as an abbreviation for ((α′α′)(α′α′)), etc.

The type symbols enter our formal theory only as subscripts upon variables and constants. In the interpretation of the theory it is intended that the

Received March 23, 1940.

[1] See Rudolf Carnap, *Abriss der Logistik*, Vienna 1929, §9. (The simple theory of types was suggested as a modification of Russell's ramified theory of types by Leon Chwistek in 1921 and 1922 and by F. P. Ramsey in 1926.)

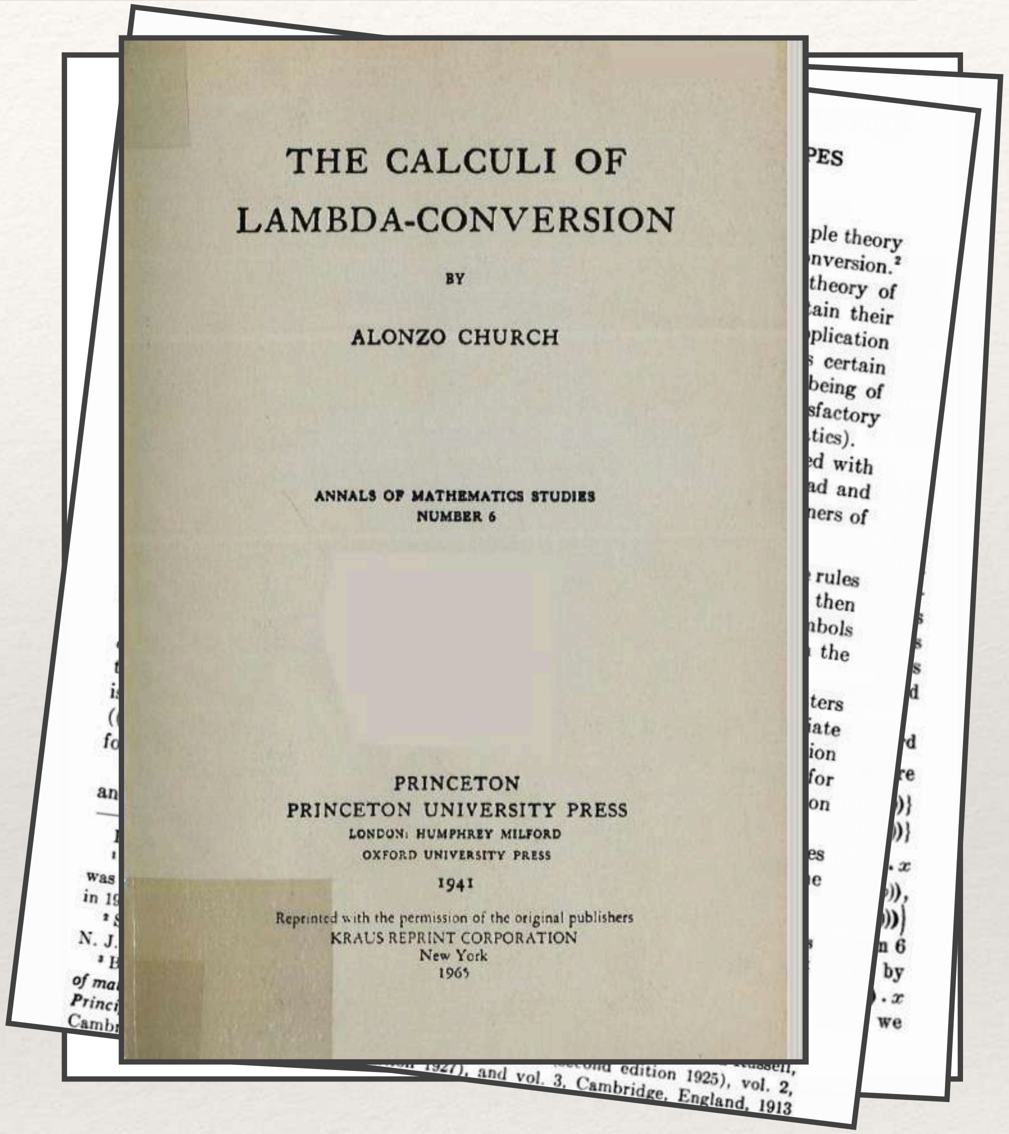
[2] See, for example, Alonzo Church, *Mathematical logic* (mimeographed), Princeton, N. J., 1936, and *The calculi of lambda-conversion*, forthcoming monograph.

[3] Bertrand Russell, *Mathematical logic as based on the theory of types*, *American journal of mathematics*, vol. 30 (1908), pp. 222-262; Alfred North Whitehead and Bertrand Russell, *Principia mathematica*, vol. 1, Cambridge, England, 1910 (second edition 1925), vol. 2, Cambridge, England, 1912 (second edition 1927), and vol. 3, Cambridge, England, 1913

The Journal of Symbolic Logic 5(2), June 1940, pages 56-68
http://links.jstor.org/sici?sici=0022-4812%28194006%295%3A2%3C56%3AAFOTST%3E2.0.CO%3B2-Q

Historiquement

- …puis en 1941 par A. Church (d'après ses notes de cours)


THE CALCULI OF LAMBDA-CONVERSION

BY

ALONZO CHURCH

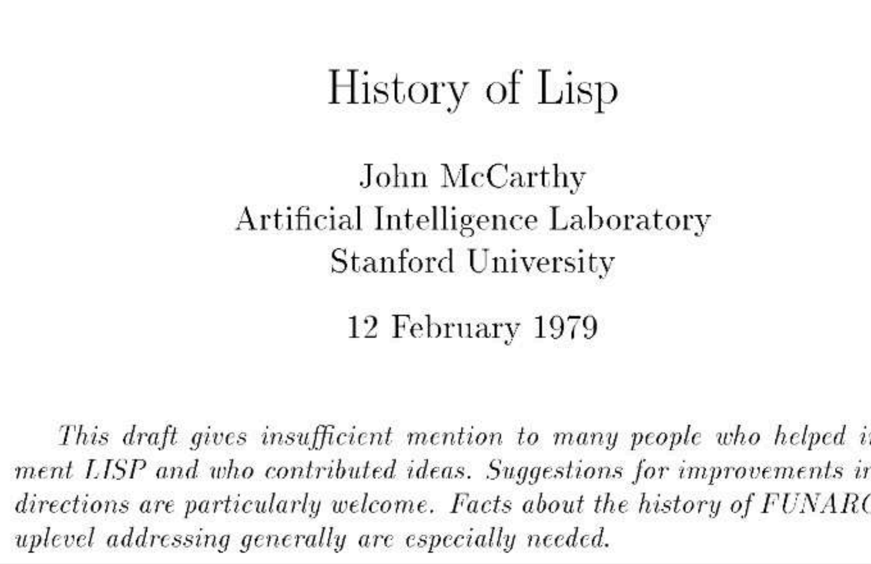
ANNALS OF MATHEMATICS STUDIES
NUMBER 6

PRINCETON
PRINCETON UNIVERSITY PRESS
LONDON: HUMPHREY MILFORD
OXFORD UNIVERSITY PRESS
1941

Reprinted with the permission of the original publishers
KRAUS REPRINT CORPORATION
New York
1965

Annals of Mathematics, series 6,
Princeton University Press, 1941, 77 pages.
https://archive.org/details/AnnalsOfMathematicalStudies6ChurchAlonzoTheCalculiOfLambdaConversionPrincetonUniversityPress1941/mode/2up

Lisp


History of Lisp

John McCarthy
Artificial Intelligence Laboratory
Stanford University

12 February 1979

This draft gives insufficient mention to many people who helped i ment LISP and who contributed ideas. Suggestions for improvements ir directions are particularly welcome. Facts about the history of FUNARC uplevel addressing generally are especially needed.

- Le tout premier langage fonctionnel:
  📖 John McCarthy *et al.*
  LISP 1.5 Programmer's Manual
  MIT Press (1962)

- `(define fact(x)`

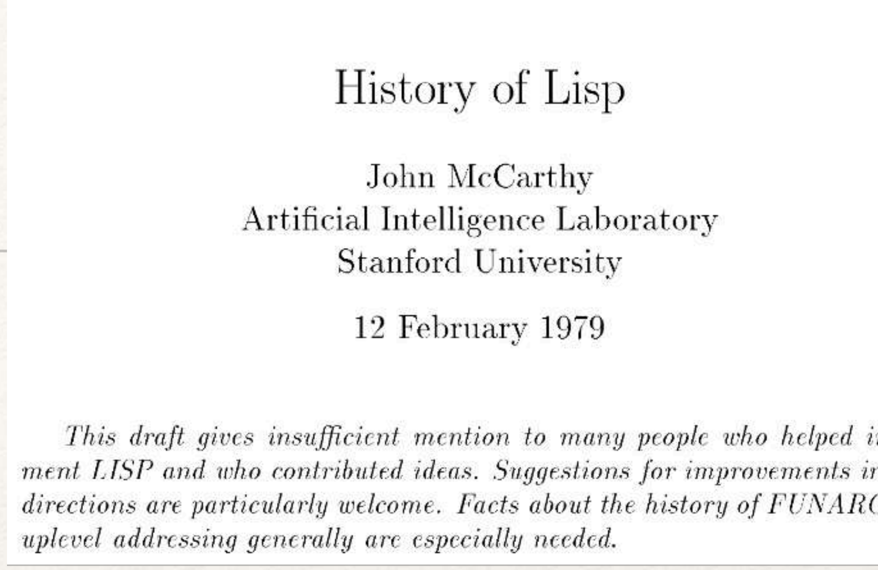
```
(define fact(x)
      (cond (eq x 0)
            1
            (* x (fact (- x 1))))))
```

Langage **fonctionnel**:
on calcule en **appliquant** des fonctions
à des arguments (`eq`, `cond`, `*`, `fact`, `-`)

On peut aussi
définir des fonctions

- Lisp est un lambda-calcul **enrichi**
  (avec des primitives: `eq`, `cond`, `*`, `0`, `1`, `-`, etc.)

Lisp


History of Lisp

John McCarthy
Artificial Intelligence Laboratory
Stanford University

12 February 1979

This draft gives insufficient mention to many people who helped i ment LISP and who contributed ideas. Suggestions for improvements ir directions are particularly welcome. Facts about the history of FUNARC uplevel addressing generally are especially needed.

- Le tout premier langage fonctionnel:
  📖 John McCarthy *et al.*
  LISP 1.5 Programmer's Manual, MIT Press (1962)

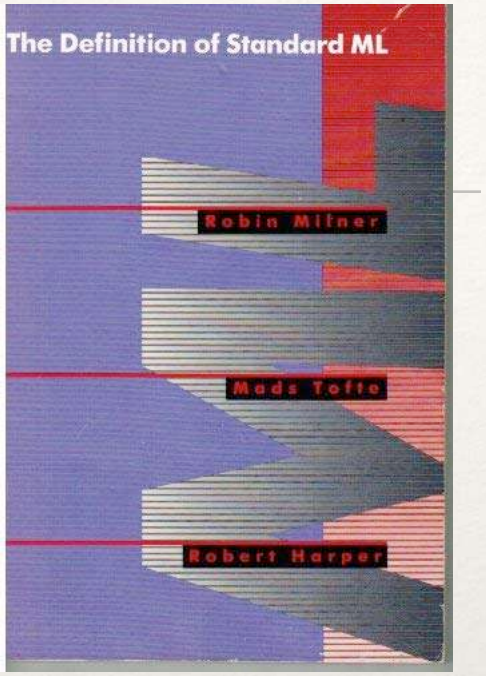
- ```
  (define fact
      (lambda (x)
          (cond (eq x 0)
                1
                (* x (fact (-x 1))))))
  ```

On peut même définir des fonctions **anonymes**

Ceci est directement inspiré de la notation λx . … du λ-calcul

- ```
  (mapcar (lambda (x) (+ x 1))
          (list 1 2 3))        ; calcule (2 3 4)
  ```

ML



https://pictures.abebooks.com/isbn/9780262631327-us.jpg

- Dû à Robin Milner (1978)

  λ-calc. → Hope → ML → CaML → CaML light → OCaML

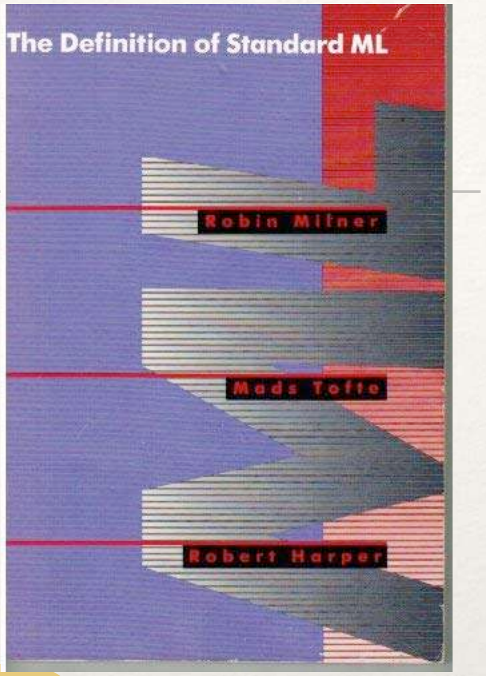
  … aussi → Standard ML (SML/NJ)

-
```
fun fact x =
    if x=0
       then 1
    else x*fact(x-1);
```

Langage **fonctionnel**:
on calcule en **appliquant** des fonctions
à des arguments (=, `if`,`*`, `fact`, `-`);
la syntaxe est simplement plus agréable

On peut aussi **définir** des fonctions — récursives, comme en Lisp

ML



ks.com/isbn/9780262631327-us.jpg

- Dû à Robin Milner (1978)

  λ-calc. → Hope → ML → CaML → CaML light → OCaML
  … aussi → Standard ML (SML/NJ)

- ```
  val rec fact =
      fn x => if x=0
                 then 1
              else x*fact(x-1);
  ```

On peut même définir des fonctions **anonymes**

Ceci est directement inspiré de la notation λx . … du λ-calcul

- ```
  map (fn x => x+1) [1, 2, 3];
      (* calcule [2, 3, 4] *)
  ```

Haskell

Par Thought up by Darrin Thompson
and produced by Jeff Wheeler —
Thompson-Wheeler logo on the haskell wiki,
Domaine public,
https://commons.wikimedia.org/w/index.php?curid=8479507

- (1990)
  λ-calc. → Miranda → Haskell

- ```
  fact 0 = 1
  fact x = x*fact(x-1)
  ```

Langage **fonctionnel**:
on calcule en **appliquant** des fonctions
à des arguments (`*`, `fact`, `-`)

On peut aussi **définir** des fonctions — récursives, comme en Lisp et en ML
  ```

Haskell

Par Thought up by Darrin Thompson
eler –
kell wiki,
hp?curid=8479507

- (1990)

  λ-calc. → Miranda → Haskell

On peut même définir des fonctions **anonymes**

Ceci est directement inspiré de la notation λx . … du λ-calcul

- ```
  fact = \x | x=0 -> 1
            | otherwise -> x*fact(x-1)
  ```

- ```
  map (\x -> x+1) [1,2,3]
        -- calcule [2,3,4]
  ```

- ```
  nat = 0:map (\x -> x+1) nat
        -- calcule [0, 1, 2, …]
  ```

Evaluation **paresseuse**:
(appel par nom, voir stratégies, plus tard dans le cours)
les arguments de fonction (ici, :) ne sont évalués que si on a besoin de les connaître
  ```

La syntaxe du λ-calcul

La syntaxe du λ-calcul

- Très très simple! Les termes sont:

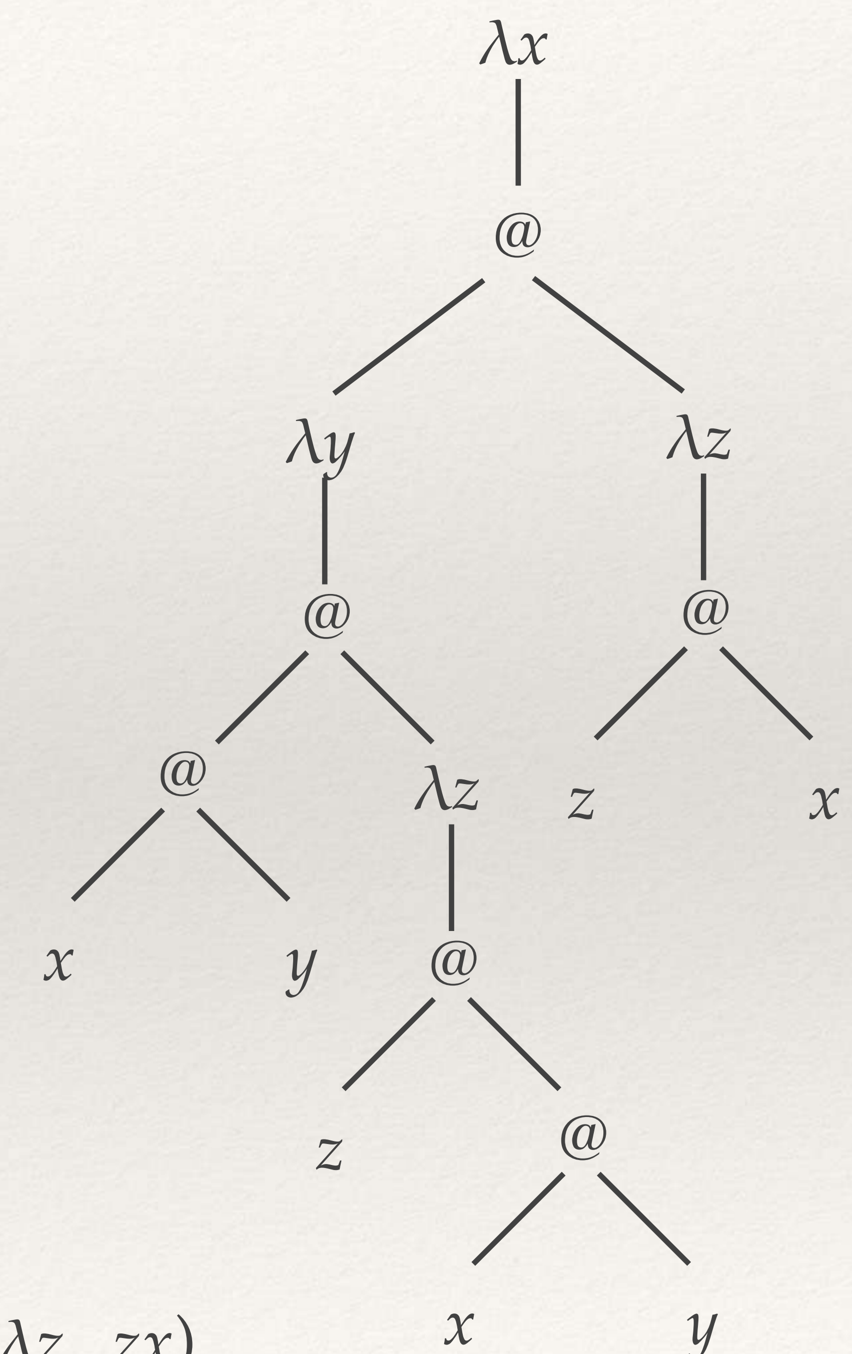
$s,t,u,v,\ldots ::=$

$x,y,z,\ldots$	variables (en nb. ∞ dénombrable)
$\mid st$	application (de s à t)
$\mid \lambda x\,.\,s$	λ-abstraction (`fun` x `->` s, en Caml)

- C'est tout! Pas d'entiers, pas de listes, pas de récursion, pas de types, pas de modules, rien d'autre…
- Et pourtant, on verra que le langage est Turing-complet

La syntaxe du λ-calcul

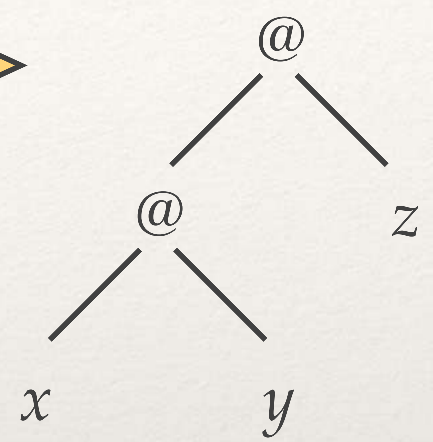
- Un terme est réellement un **arbre**.
  La syntaxe représente ces arbres,
  modulo les conventions usuelles
  de **parenthésage**
  et de **priorités** (à la Caml)
- Quelques exemples…



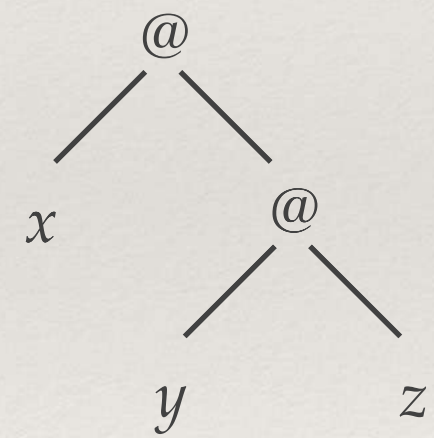
$\lambda x \,.\, (\lambda y \,.\, xy(\lambda z \,.\, z(xy)))\ (\lambda z \,.\, zx)$

La syntaxe du λ-calcul

- xyz dénote $(xy)z$,

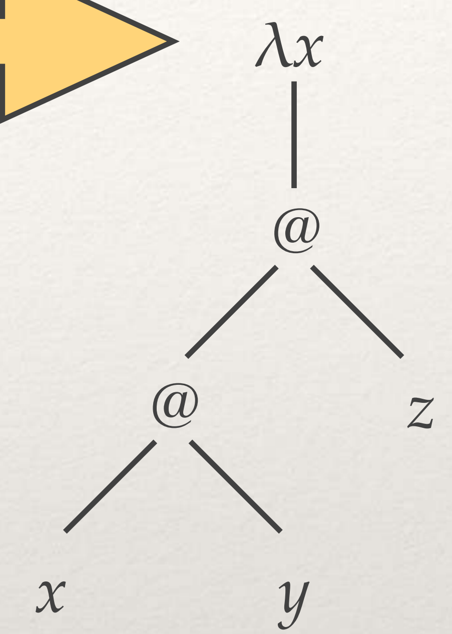


- pas $x(yz)$
- L'application n'est **pas** associative



La syntaxe du λ-calcul

- $\lambda x \,.\, xyz$ dénote $(\lambda x \,.\, (xyz))$
- … autrement dit la portée de λx s'étend aussi loin à droite que possible



Calcul: la β-réduction

La β-réduction

rédex

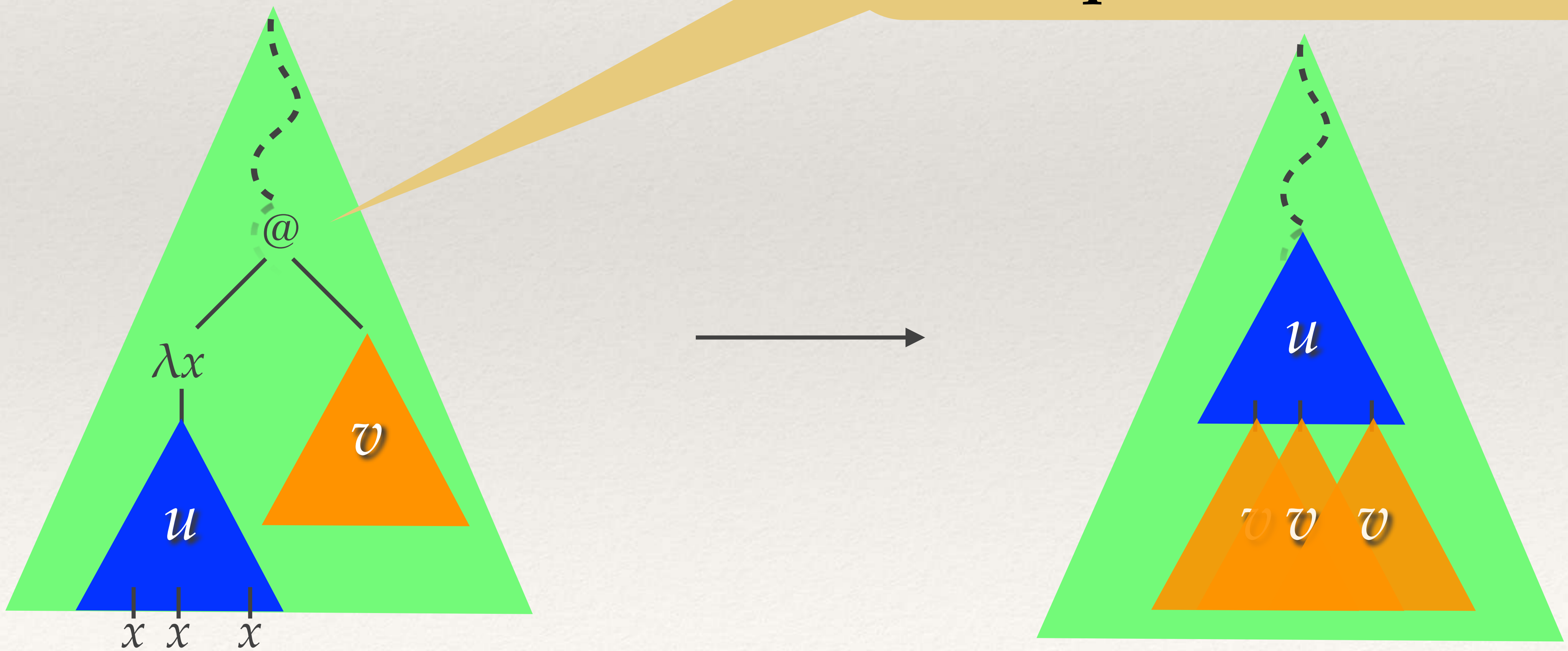
contractum

- Une seule règle de calcul:

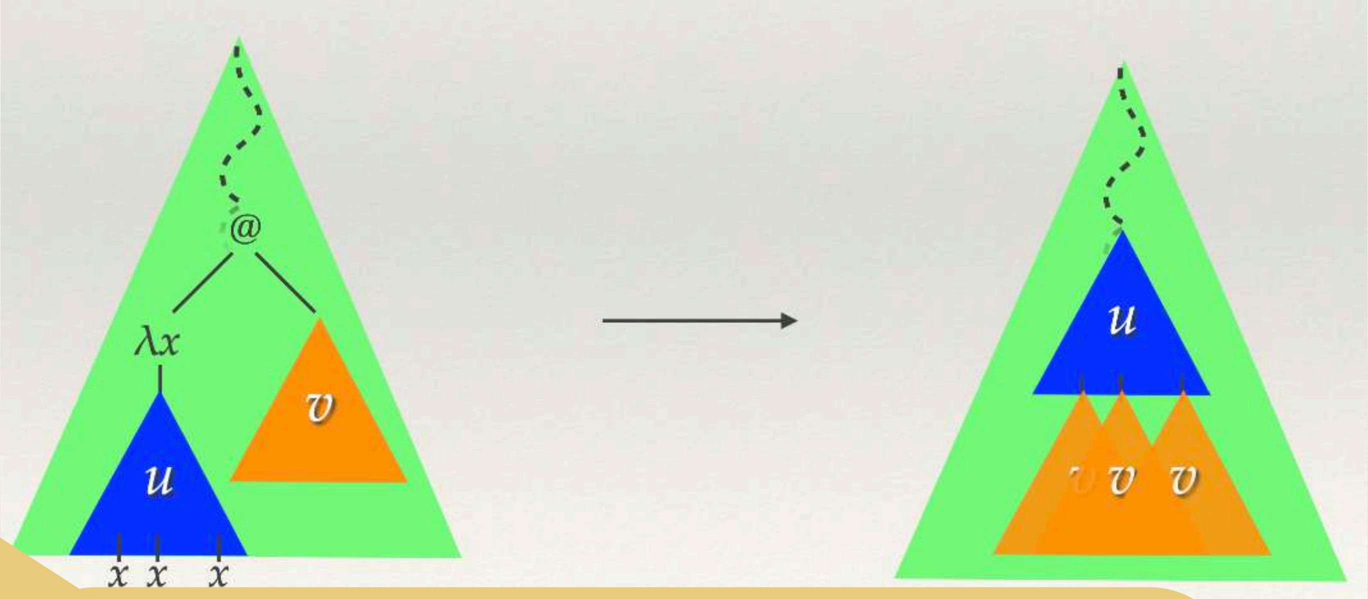
$$(\beta) \quad (\lambda x \,.\, u)\, v \to u[x{:=}v]$$

- applicable n'importe où dans un terme

Il peut y avoir plusieurs rédexes dans un terme, mais on n'en contracte **qu'un** à la fois



Une autre présentation de la β-réduction



- (β) $(\lambda x\ .\ u)\ v \to u[x{:=}v]$
- On dit que $s \to t$ ssi
  il existe un **contexte** C
  et un rédex $(\lambda x\ .\ u)\ v$
  tels que $s = C[(\lambda x\ .\ u)\ v]$
  et $t = C[u[x{:=}v]]$

Oralement: « *s* se **contracte** en *t* »
ou « *s* se **réduit en une étape** en *t* »

La β-réduction est la plus petite relation contenant β et **compatible aux contextes**

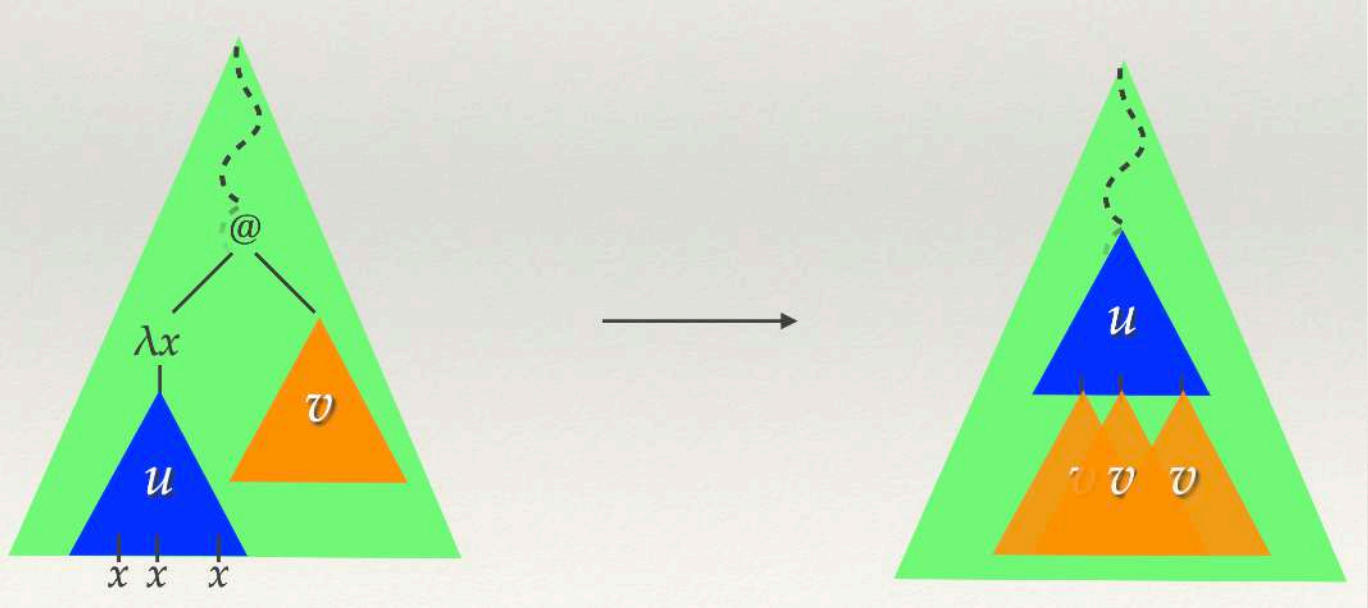
- $C ::= \_$ trou (où le terme est inséré)
  $\mid \lambda x\ .\ C$ réduction « sous la lambda »
  $\mid C\ v$ la réduction s'opère dans la fonction
  $\mid u\ C$ la réduction s'opère dans l'argument

Encore une autre présentation

$$\frac{}{(\lambda x\,.\,u)\,v \to u[x{:=}v]}$$

$$\frac{u \to u'}{\lambda x\,.\,u \to \lambda x\,.\,u'}$$

$$\frac{u \to u'}{uv \to u'v} \qquad \frac{v \to v'}{uv \to uv'}$$



- Alors $s \to t$ si et seulement le jugement « $s \to t$ » est dérivable

Autres relations

- On écrira $\to$ pour la β-réduction, mais parfois aussi pour n'importe quelle autre relation de réduction (relation binaire…)
- Si ambiguïté, on écrira $\to_\beta$ pour la β-réduction
- **Exemple:** on peut ajouter la règle

  $$(\eta) \quad \lambda x\,.\,ux \to u \qquad\qquad \text{(si } x \text{ pas libre dans } u\text{)}$$

  et considérer la $\beta\eta$-réduction $\to_{\beta\eta}$

  (Non, on ne peut pas simuler (η) par (β), cf. $\lambda x\,.\,yx$ où y est une variable $\neq x$)

Clôtures

- $\to^*$ = clôture **réflexive-transitive** de $\to$ (étoile de Kleene)
  = plus petit préordre contenant $\to$
  $u \to^* v$ ssi il existe une **réduction** (un chemin)
  $$u=u_0 \to u_1 \to \ldots\ u_{n-1} \to u_n=v, \text{ avec } n\geq 0$$
- $\to^+$ = clôture **transitive** de $\to$
  = plus petite relation transitive contenant $\to$
  $u \to^+ v$ ssi il existe une **réduction non vide**
  $$u=u_0 \to u_1 \to \ldots\ u_{n-1} \to u_n=v, \text{ avec } n\geq 1$$
- $\leftrightarrow^* = (\leftarrow \cup \to)^*$ est la **β-équivalence** (parfois notée $=_\beta$)

Substitution

Substitution?

- (β) $(\lambda x \,.\, u)\, v \to u[x{:=}v]$
- D'accord, mais comment définit-on **formellement** la substitution $u[x{:=}v]$ de v pour x dans u?
- Bizarrement, c'est une question très compliquée
  (et très enquiquinante — on s'empressera d'ignorer les difficultés à l'avenir)

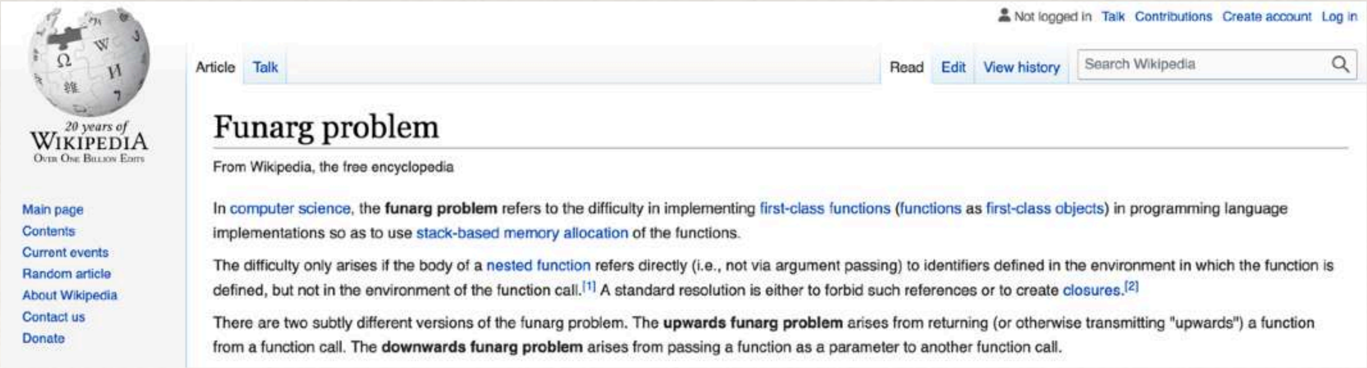
Substitution: 1er essai

- Substitution textuelle: non, donnerait:
  $(\lambda x\ .\ u)\ [x{:=}v] = \lambda v\ .\ (u[x{:=}v])$

n'est même pas syntaxiquement correct
(sauf si v est une variable)

Substitution: 2ème essai

- Définition par récurrence sur u:
  $x\,[x{:=}v] \stackrel{\text{def}}{=} v$
  $y\,[x{:=}v] \stackrel{\text{def}}{=} y \quad (y{\neq}x)$
  $(st)\,[x{:=}v] \stackrel{\text{def}}{=} (s\,[x{:=}v])\,(t\,[x{:=}v])$
  $(\lambda z\,.\,u)\,[x{:=}v] \stackrel{\text{def}}{=} \lambda z\,.\,(u[x{:=}v])$
- A l'air bien défini, mais souffre de quelques problèmes…


Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

20 years of WIKIPEDIA Over One Billion Edits

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Funarg problem

From Wikipedia, the free encyclopedia

In computer science, the **funarg problem** refers to the difficulty in implementing first-class functions (functions as first-class objects) in programming language implementations so as to use stack-based memory allocation of the functions.

The difficulty only arises if the body of a nested function refers directly (i.e., not via argument passing) to identifiers defined in the environment in which the function is defined, but not in the environment of the function call.[1] A standard resolution is either to forbid such references or to create closures.[2]

There are two subtly different versions of the funarg problem. The **upwards funarg problem** arises from returning (or otherwise transmitting "upwards") a function from a function call. The **downwards funarg problem** arises from passing a function as a parameter to another function call.

Le problème avec la substitution

- D'après la définition,
  $(\lambda x\ .\ x)\ [x{:=}y] = \lambda x\ .\ y$
  $(\lambda y\ .\ x)\ [x{:=}y] = \lambda y\ .\ y$

$$x\ [x{:=}v] \stackrel{\text{def}}{=} v$$
$$y\ [x{:=}v] \stackrel{\text{def}}{=} y \quad (y{\neq}x)$$
$$(st)\ [x{:=}v] \stackrel{\text{def}}{=} (s\ [x{:=}v])\ (t\ [x{:=}v])$$
$$(\lambda z\ .\ u)\ [x{:=}v] \stackrel{\text{def}}{=} \lambda z\ .\ (u[x{:=}v])$$

- Pour corriger ça, on va:
- restreindre la substitution par des conditions sur les variables **libres** et **liées**
- autoriser à **renommer** les variables liées (**α-renommage**)
- autre solution: indices de **de Bruijn** (omis ici; voir exercices)

Variables libres, variables liées

- $\mathrm{fv}(u) = \{$variables **libres** de $u\}$
  « qui apparaissent dans u à une position où on peut les remplacer »
- $\mathrm{bv}(u) = \{$variables **liées** de $u\}$

$\mathrm{fv}(x) \stackrel{\text{def}}{=} \{x\}$	$\mathrm{bv}(x) \stackrel{\text{def}}{=} \varnothing$
$\mathrm{fv}(uv) \stackrel{\text{def}}{=} \mathrm{fv}(u) \cup \mathrm{fv}(v)$	$\mathrm{bv}(uv) \stackrel{\text{def}}{=} \mathrm{bv}(u) \cup \mathrm{bv}(v)$
$\mathrm{fv}(\lambda x\,.\,u) \stackrel{\text{def}}{=} \mathrm{fv}(u) - \{x\}$	$\mathrm{bv}(\lambda x\,.\,u) \stackrel{\text{def}}{=} \mathrm{bv}(u) \cup \{x\}$

- Exemples.

u	$\mathrm{fv}(u)$	$\mathrm{bv}(u)$
$\lambda x\,.\,x$	?	?
$x(\lambda y\,.\,z)$	?	?
$x(\lambda x\,.\,x)$	?	?
$(\lambda x\,.\,xx)(y(\lambda z\,.\,yz)x)$	?	?

Variables libres, variables liées

- $\mathrm{fv}(u) = \{$variables **libres** de $u\}$
  « qui apparaissent dans u à une position où on peut les remplacer »
- $\mathrm{bv}(u) = \{$variables **liées** de $u\}$

$\mathrm{fv}(x) \stackrel{\text{def}}{=} \{x\}$	$\mathrm{bv}(x) \stackrel{\text{def}}{=} \varnothing$
$\mathrm{fv}(uv) \stackrel{\text{def}}{=} \mathrm{fv}(u) \cup \mathrm{fv}(v)$	$\mathrm{bv}(uv) \stackrel{\text{def}}{=} \mathrm{bv}(u) \cup \mathrm{bv}(v)$
$\mathrm{fv}(\lambda x\,.\,u) \stackrel{\text{def}}{=} \mathrm{fv}(u) - \{x\}$	$\mathrm{bv}(\lambda x\,.\,u) \stackrel{\text{def}}{=} \mathrm{bv}(u) \cup \{x\}$

- Exemples.

u	$\mathrm{fv}(u)$	$\mathrm{bv}(u)$
$\lambda x\,.\,x$	$\varnothing$	$\{x\}$
$x(\lambda y\,.\,z)$	?	?
$x(\lambda x\,.\,x)$	?	?
$(\lambda x\,.\,xx)(y(\lambda z\,.\,yz)x)$	?	?

Variables libres, variables liées

- $\mathrm{fv}(u)$ = {variables **libres** de u}
  « qui apparaissent dans u à une position où on peut les remplacer »
- $\mathrm{bv}(u)$ = {variables **liées** de u}

$\mathrm{fv}(x) \stackrel{\text{def}}{=} \{x\}$	$\mathrm{bv}(x) \stackrel{\text{def}}{=} \varnothing$
$\mathrm{fv}(uv) \stackrel{\text{def}}{=} \mathrm{fv}(u) \cup \mathrm{fv}(v)$	$\mathrm{bv}(uv) \stackrel{\text{def}}{=} \mathrm{bv}(u) \cup \mathrm{bv}(v)$
$\mathrm{fv}(\lambda x\,.\,u) \stackrel{\text{def}}{=} \mathrm{fv}(u) - \{x\}$	$\mathrm{bv}(\lambda x\,.\,u) \stackrel{\text{def}}{=} \mathrm{bv}(u) \cup \{x\}$

- Exemples.

u	$\mathrm{fv}(u)$	$\mathrm{bv}(u)$
$\lambda x\,.\,x$	$\varnothing$	$\{x\}$
$x(\lambda y\,.\,z)$	$\{x,z\}$	$\{y\}$
$x(\lambda x\,.\,x)$	?	?
$(\lambda x\,.\,xx)(y(\lambda z\,.\,yz)x)$	?	?

Variables libres, variables liées

- $\mathrm{fv}(u)$ = {variables **libres** de u}
  « qui apparaissent dans u à une position où on peut les remplacer »
- $\mathrm{bv}(u)$ = {variables **liées** de u}

$\mathrm{fv}(x) \stackrel{\text{def}}{=} \{x\}$	$\mathrm{bv}(x) \stackrel{\text{def}}{=} \varnothing$
$\mathrm{fv}(uv) \stackrel{\text{def}}{=} \mathrm{fv}(u) \cup \mathrm{fv}(v)$	$\mathrm{bv}(uv) \stackrel{\text{def}}{=} \mathrm{bv}(u) \cup \mathrm{bv}(v)$
$\mathrm{fv}(\lambda x\,.\,u) \stackrel{\text{def}}{=} \mathrm{fv}(u) - \{x\}$	$\mathrm{bv}(\lambda x\,.\,u) \stackrel{\text{def}}{=} \mathrm{bv}(u) \cup \{x\}$

- Exemples.

u	$\mathrm{fv}(u)$	$\mathrm{bv}(u)$
$\lambda x\,.\,x$	$\varnothing$	$\{x\}$
$x(\lambda y\,.\,z)$	$\{x,z\}$	$\{y\}$
$x(\lambda x\,.\,x)$	$\{x\}$	$\{x\}$
$(\lambda x\,.\,xx)(y(\lambda z\,.\,yz)x)$	?	?

oui, une variable peut être libre **et** liée!

Variables libres, variables liées

- $\mathrm{fv}(u)$ = {variables **libres** de u}
  « qui apparaissent dans u à une position où on peut les remplacer »
- $\mathrm{bv}(u)$ = {variables **liées** de u}

$\mathrm{fv}(x) \stackrel{\text{def}}{=} \{x\}$	$\mathrm{bv}(x) \stackrel{\text{def}}{=} \varnothing$
$\mathrm{fv}(uv) \stackrel{\text{def}}{=} \mathrm{fv}(u) \cup \mathrm{fv}(v)$	$\mathrm{bv}(uv) \stackrel{\text{def}}{=} \mathrm{bv}(u) \cup \mathrm{bv}(v)$
$\mathrm{fv}(\lambda x \,.\, u) \stackrel{\text{def}}{=} \mathrm{fv}(u) - \{x\}$	$\mathrm{bv}(\lambda x \,.\, u) \stackrel{\text{def}}{=} \mathrm{bv}(u) \cup \{x\}$

- Exemples.

u	$\mathrm{fv}(u)$	$\mathrm{bv}(u)$
$\lambda x \,.\, x$	$\varnothing$	$\{x\}$
$x(\lambda y \,.\, z)$	$\{x,z\}$	$\{y\}$
$x(\lambda x \,.\, x)$	$\{x\}$	$\{x\}$
$(\lambda x \,.\, xx)(y(\lambda z \,.\, yz)x)$	$\{x,y\}$	$\{x,z\}$

oui, une variable peut être libre **et** liée!

au fait, la variable z' ($\neq x,y,z$) est-elle libre ici?

Substitution, 3ème essai

- On définit une opération de substitution **partielle**

$$u\ [x{:=}v]$$

définie uniquement lorsque

$$x \notin \mathrm{bv}(u) \text{ et } \mathrm{fv}(v) \cap \mathrm{bv}(u) = \varnothing$$

$x\ [x{:=}v] \stackrel{\text{def}}{=} v$
$y\ [x{:=}v] \stackrel{\text{def}}{=} y \quad (y{\neq}x)$
$(st)\ [x{:=}v] \stackrel{\text{def}}{=} (s\ [x{:=}v])\ (t\ [x{:=}v])$
$(\lambda z\ .\ u)\ [x{:=}v] \stackrel{\text{def}}{=} \lambda z\ .\ (u[x{:=}v])$

on dit que x est **substituable** par v dans u

pour éviter le problème « $(\lambda x\ .\ x)\ [x{:=}y] = \lambda x\ .\ y$ »

pour éviter le problème « $(\lambda y\ .\ x)\ [x{:=}y] = \lambda y\ .\ y$ »

- On peut assurer cette condition en **renommant** x…

α-renommage

- On souhaite considérer que « $\lambda x\,.\,u(x)$ et $\lambda y\,.\,u(y)$ sont interchangeables »
- On définit la relation α par:

  $$\lambda x\,.\,u \quad \alpha \quad \lambda y\,.\,(u[x{:=}y])$$

  à condition que x soit substituable par y dans u

  et que y ne soit pas libre dans u

  (sinon on aurait $\lambda x\,.\,xy \quad \alpha \quad \lambda y\,.\,yy$)

  i.e., $x,y \notin \mathrm{bv}(u)$

  et $y \notin \mathrm{fv}(u)$
- La relation d'***α*-équivalence** $=_\alpha$ est la plus petite congruence (rel. d'équivalence compatible aux contextes) contenant α

Propriétés du α-renommage

$\lambda x \,.\, u \quad \alpha \quad \lambda y \,.\, (u[x{:=}y])$
$(x,y \notin \mathrm{bv}(u),\ y \notin \mathrm{fv}(u))$

- Exemple: $\lambda x \,.\, \lambda y \,.\, xy \quad =_\alpha \lambda x \,.\, \lambda z \,.\, xz$

  (avec $z{\neq}x,y$—noter le passage au contexte)

  $=_\alpha \lambda y \,.\, \lambda z \,.\, yz =_\alpha \lambda y \,.\, \lambda x \,.\, yx$

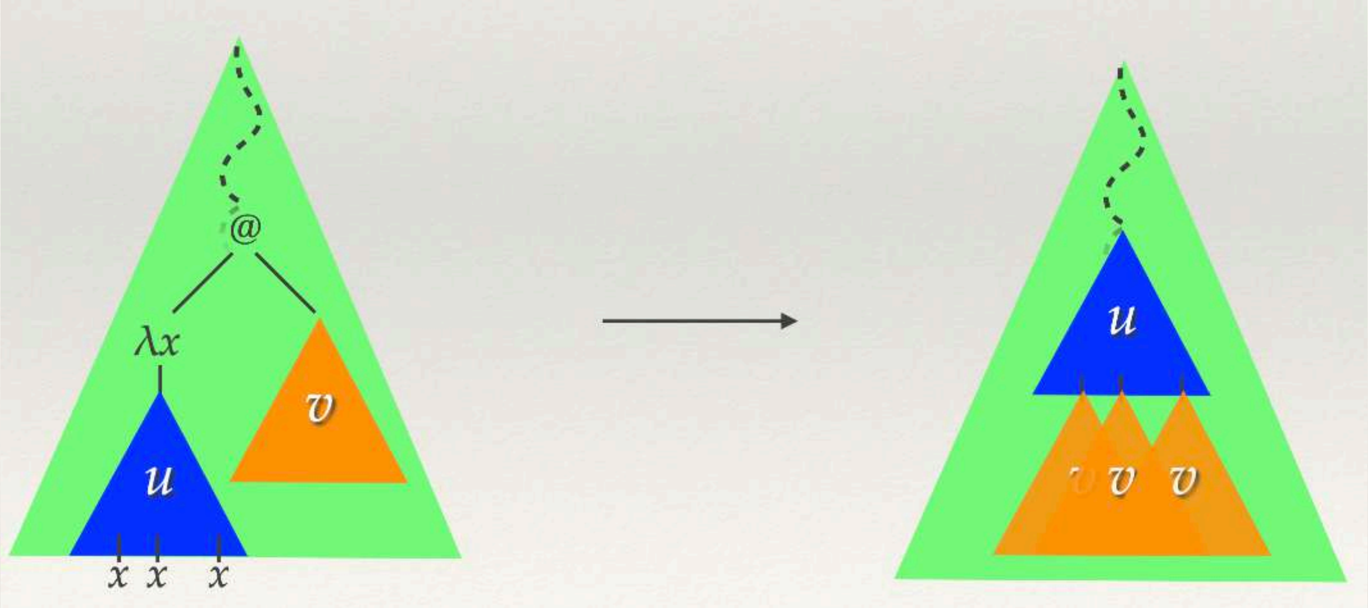
- On peut toujours α-renommer $\lambda x \,.\, u$
  de sorte que $u[x{:=}v]$ soit bien défini
  (= de sorte que $x \notin \mathrm{bv}(u)$ et $\mathrm{fv}(v) \cap \mathrm{bv}(u) = \varnothing$)

- Permet de (re)définir proprement la β-réduction!

La β-réduction… vraiment

La β-réduction, vraiment

- (β) $(\lambda x\ .\ u)\ v \to u[x{:=}v]$
- On dit que $s \to t$ ssi
  il existe un **contexte** C
  et un rédex $(\lambda x\ .\ u)\ v$
  tels que $s =_\alpha C[(\lambda x\ .\ u)\ v]$
  et $t =_\alpha C[u[x{:=}v]]$



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

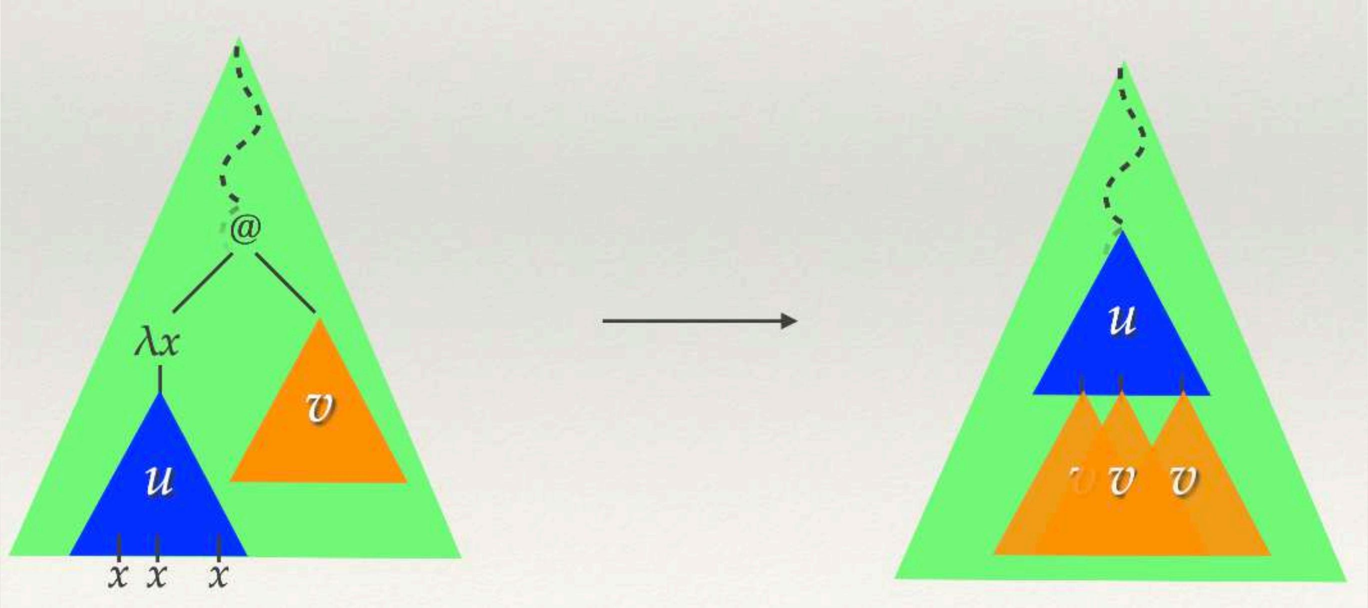
Ou, de façon équivalente

$$\frac{}{(\lambda x\,.\,u)\,v \to u[x{:=}v]}$$

$$\frac{u \to u'}{\lambda x\,.\,u \to \lambda x\,.\,u'}$$

$$\frac{u \to u'}{uv \to u'v} \qquad \frac{v \to v'}{uv \to uv'}$$

$$\frac{u =_\alpha v \quad v \to v'}{u \to v'} \qquad \frac{u \to u' \quad u' =_\alpha v'}{u \to v'}$$



- Alors $s \to t$ si et seulement le jugement « $s \to t$ » est dérivable

Un exemple de réductions

$$(\lambda y\,.\,(\lambda x\,.\,yxx)(\lambda z\,.\,z(xy)))(\lambda z'\,.\,z')$$

2 rédexes

$$(\lambda y\,.\,y(\lambda z\,.\,z(xy))(\lambda z\,.\,z(xy)))(\lambda z'\,.\,z')$$

$$(\lambda x\,.\,(\lambda z'\,.\,z')xx)(\lambda z\,.\,z(x(\lambda z'\,.\,z')))$$

$$(\lambda z'\,.\,z')(\lambda z\,.\,z(x(\lambda z'\,.\,z')))(\lambda z\,.\,z(x(\lambda z'\,.\,z')))$$

$$(\lambda x\,.\,xx)(\lambda z\,.\,z(x(\lambda z'\,.\,z')))$$

$$(\lambda z\,.\,z(x(\lambda z'\,.\,z')))(\lambda z\,.\,z(x(\lambda z'\,.\,z')))$$

$$(\lambda z\,.\,z(x(\lambda z'\,.\,z')))(x(\lambda z'\,.\,z'))$$

Les réductions **confluent**, et la forme normale est **unique**

$$x(\lambda z'\,.\,z')(x(\lambda z'\,.\,z'))$$

Forme normale (=sans rédex)

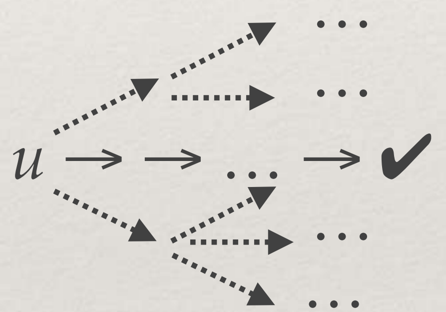
Sont-ce des phénomènes généraux?

Terminaison

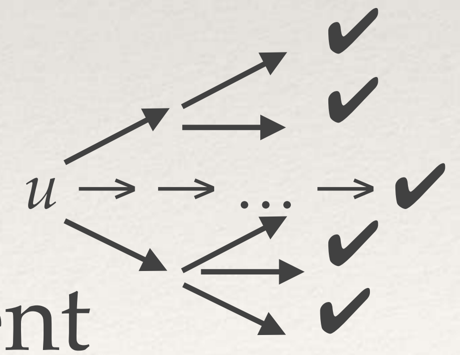
- Une **forme normale** est un terme u sans rédex,

  i.e., $u \not\to$

- Un terme u est **normalisable** (= **faiblement terminant**)
  ssi il a une forme normale
  ssi **il existe** une réduction partant
  de u qui termine



- Un terme u est
  fortement normalisable (=**terminant**)
  ssi **toutes** les réductions partant de u terminent



Le terme Ω

- Posons $\Omega \stackrel{\text{def}}{=} \delta\delta$, où $\delta \stackrel{\text{def}}{=} \lambda x\ .\ xx$ [**auto-application**]
- Son « arbre » de réductions est:
  ($\Omega = (\lambda x\ .\ xx)\ \delta \to \delta\delta = \Omega$,
  et c'est la seule réduction possible)

$$\begin{tikzcd} \Omega \arrow[loop below] \end{tikzcd}$$

- Ω est un terme qui n'est (même) **pas normalisable**
- Y a-t-il un terme normalisable mais pas fortement?
- On a donc tous les cas possibles:
  fortement norm. / norm. / pas normalisable

$$\begin{tikzcd} & (\lambda x\ .\ y)\ \Omega \arrow[loop right] \arrow[dl] \\ y & \end{tikzcd}$$

Confluence

confluence forte

confluence Church-Rosser

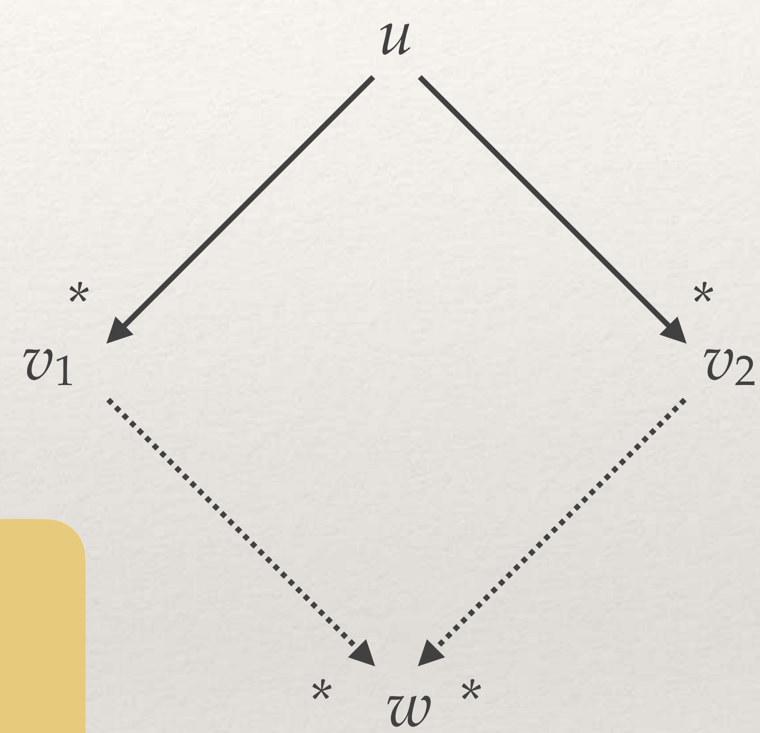
confluence locale forme normale unique

Propriété de forme normale unique

- On dit qu'une relation de réduction a la
  propriété de forme normale unique
  ssi tout terme a **au plus une** forme normale
- On verra que la β-réduction a cette propriété.

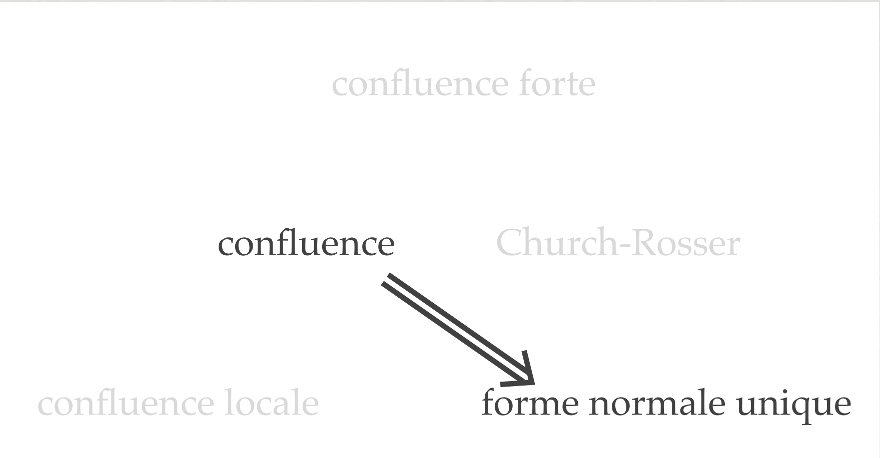
Confluence

- Une relation de réduction $\to$ est **confluente** ssi toutes les réductions $u \to^* v_1$ et $u \to^* v_2$ sont **joignables**, i.e. il existe des réductions $v_1 \to^* w$ et $v_2 \to^* w$ vers le même terme w



- **Fait.** Toute relation confluente a la propriété de forme normale unique.

- *Preuve.* Si v_1 et v_2 sont deux formes normales de u, les réductions $v_1 \to^* w$ et $v_2 \to^* w$ sont de longueur 0.

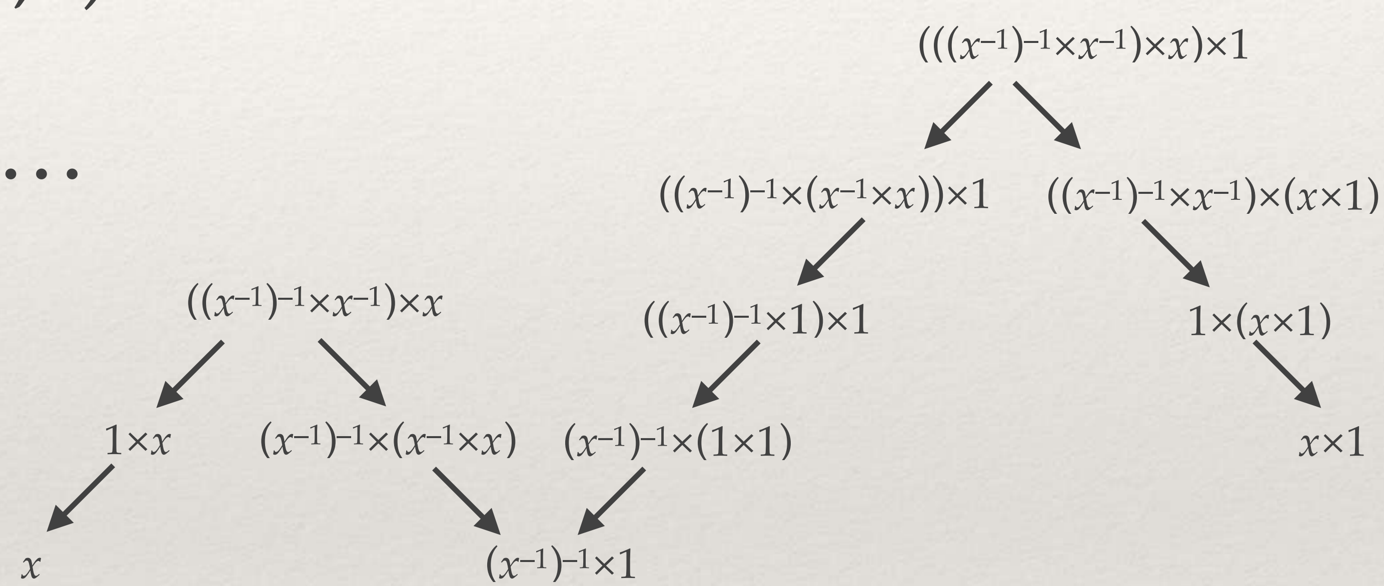


Propriété de Church-Rosser

- $\to$ a la propriété de **Church-Rosser** ssi $\leftrightarrow^* = \to^*; \leftarrow^*$
  i.e., pour tous u, v tels que $u \leftrightarrow^* v$, il existe w tel que
  $$u \to^* w \text{ et } w \leftarrow^* v$$
- A quoi ça sert?
  Un exemple en théorie des groupes…

Exemple: la théorie des groupes

- On vous dit: $1 \times x \to x \quad x^{-1} \times x \to 1 \quad (x \times y) \times z \to x \times (y \times z)$ (pour tous termes x,y,z)
- Démontrer $x \times 1 \leftrightarrow^* x$…
- Oui, on peut!



- mais ceci demande de l'inventivité
- Ça serait plus simple si on pouvait simpler réduire $x \times 1$ et x et vérifier qu'ils se réduisent aux mêmes termes…

La procédure de Knuth-Bendix (un aperçu)

- On cherche deux règles $\ell_1 \to r_1$ et $\ell_2 \to r_2$ telles que
  ℓ_2 s'unifie avec un sous-terme non var. t de ℓ_1
  (ℓ_1=C[t], σ=mgu(ℓ_2,t))

- Alors $\ell_1\sigma \to r_1\sigma$, mais aussi $\ell_1\sigma = C\sigma[\ell_2\sigma] \to C\sigma[r_2\sigma]$

- Le couple ($\ell_1\sigma\downarrow$, $C\sigma[r_2\sigma]\downarrow$) est
  une **paire critique**

$$\begin{tikzcd}
 & \ell_1\sigma \arrow[dl] \arrow[dr] & \\
r_1\sigma \arrow[d, dotted, "*"'] & & C\sigma[r_2\sigma] \arrow[d, dotted, "*"] \\
r_1\sigma\downarrow & & C\sigma[r_2\sigma]\downarrow \\
\checkmark & & \checkmark
\end{tikzcd}$$

- Si (u,v) est une paire critique avec $u \neq v$,
  rajouter la règle $u \to v$ ou $v \to u$… et recommencer.

Exemple: la théorie des groupes

- On vous dit: $1\times x \to x$ $\quad x^{-1}\times x \to 1$ $\quad (x\times y)\times z \to x\times(y\times z)$

$$
\begin{tikzcd}
& (x^{-1}\times x)\times z \arrow[dl] \arrow[dr] & \\
1\times z \arrow[d, dotted] & & x^{-1}\times(x\times z) \\
z & & \checkmark \\
\checkmark & &
\end{tikzcd}
$$

- Rajouter la règle $x^{-1}\times(x\times z) \to z$
- Ceci peut rajouter de nouvelles paires critiques, et on itère…

Exemple: la théorie des groupes

- On vous dit: $1 \times x \to x \quad x^{-1} \times x \to 1 \quad (x \times y) \times z \to x \times (y \times z)$
- Résultat (après effacement de règles inutiles):

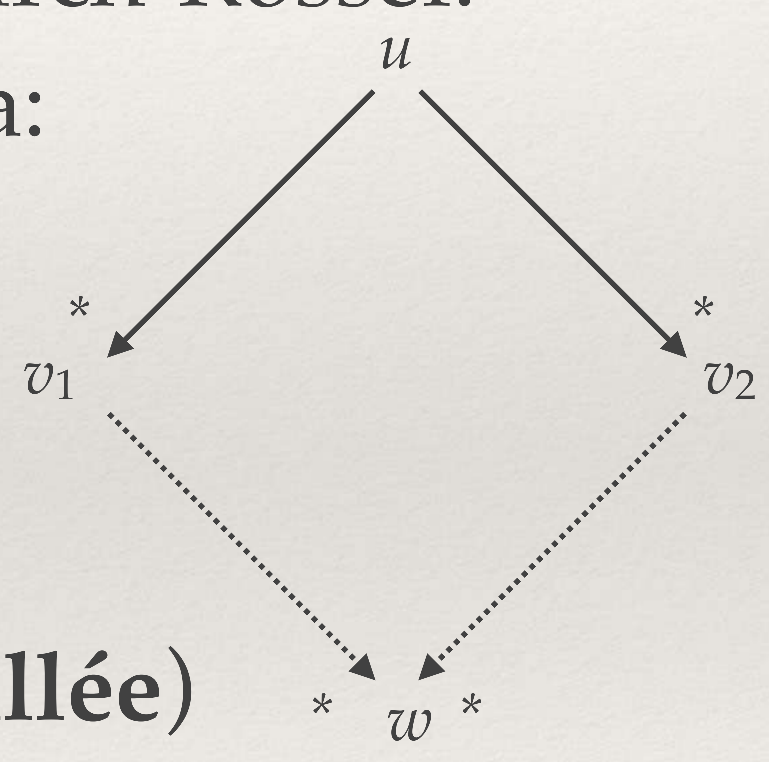
$$1 \times x \to x \qquad x \times 1 \to x \qquad 1^{-1} \to 1$$
$$(x^{-1})^{-1} \to x \qquad (x \times y)^{-1} \to y^{-1} \times x^{-1}$$
$$x \times x^{-1} \to 1 \qquad x^{-1} \times x \to 1 \qquad (x \times y) \times z \to x \times (y \times z)$$
$$x \times (x^{-1} \times z) \to z \qquad x^{-1} \times (x \times z) \to z$$

- Ce système de réécriture **est Church-Rosser**!
- **Corollaire:** la théorie des groupes est **décidable**.
(Pour savoir si $u \leftrightarrow^* v$, comparer leurs formes normales dans ce nouveau système.)

Confluence et Church-Rosser

- **Théorème.** Confluence = Church-Rosser.
- Preuve (1/2). Suppons $\to$ Church-Rosser.

  Si l'on a:

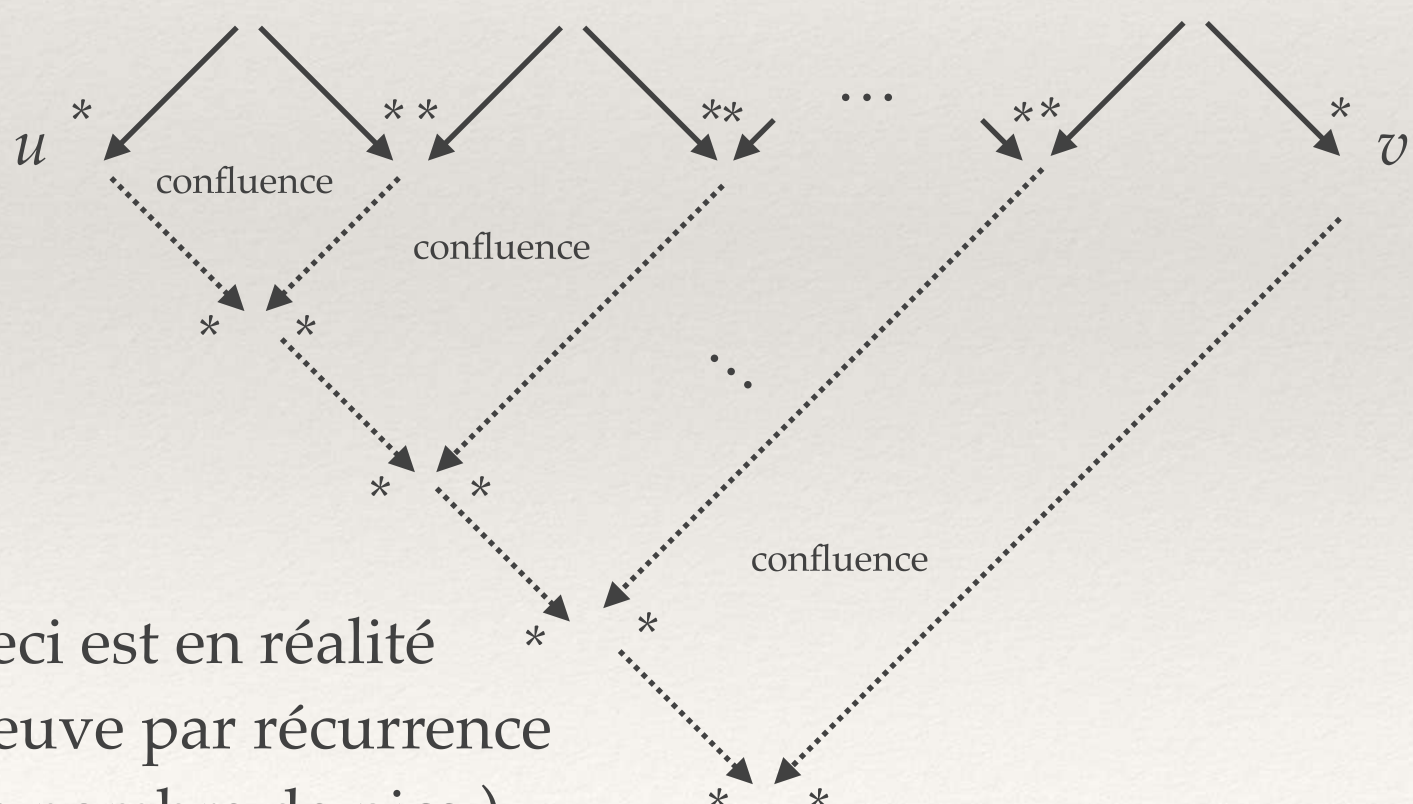
  

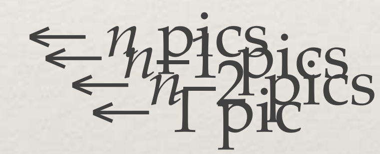
  en particulier $v_1 \longleftrightarrow^* v_2$
  (par une preuve dite en **pic**)
- Par Church-Rosser…
  (preuve de $v_1 \longleftrightarrow^* v_2$ dite en **vallée**)
- donc $\to$ est confluente.

Confluence et Church-Rosser

- **Théorème.** Confluence = Church-Rosser.
- Preuve (2/2). Suppons $\to$ confluente, et $u \longleftrightarrow^* v$

  On peut organiser cette preuve en pics et vallées:



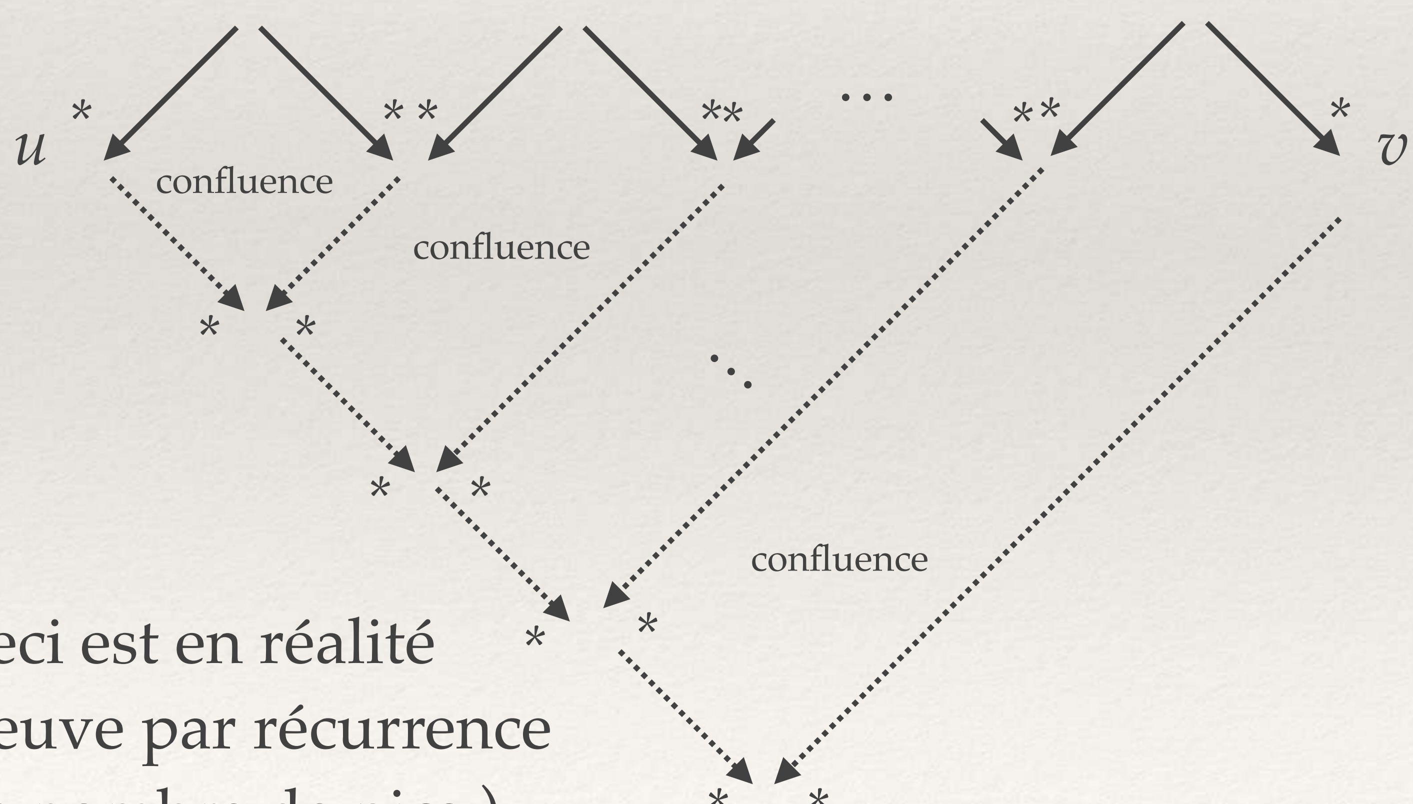


(Ceci est en réalité une preuve par récurrence sur le nombre de pics.)

Plus de pic!
On a obtenu une preuve en vallée de $u \longleftrightarrow^* v$.
Donc $\to$ est Church-Rosser.

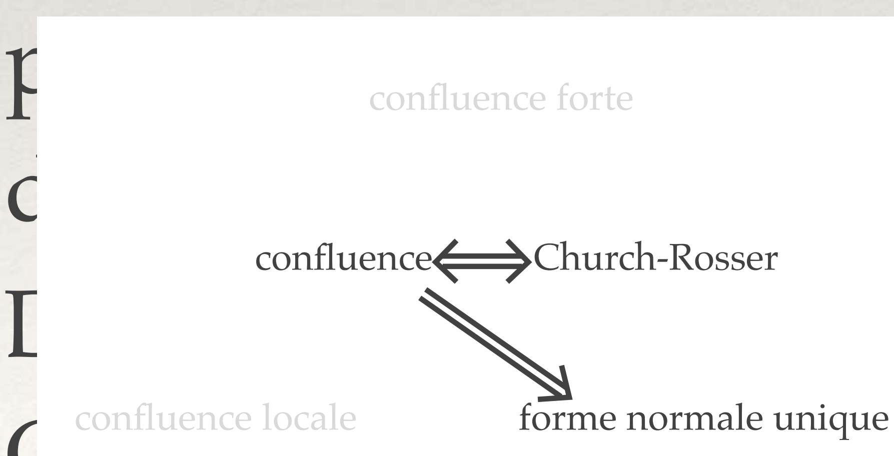
Confluence et Church-Rosser

- **Théorème.** Confluence = Church-Rosser.
- Preuve (2/2). Suppons $\to$ confluente, et $u \longleftrightarrow^* v$
  On peut organiser cette preuve en pics et vallées:



$\leftarrow n$ pics
$\leftarrow n-1$ pics
$\leftarrow n-2$ pics
$\leftarrow 1$ pic

Plus de pic!
On a obtenu une
p
c
D
Church-Rosser.

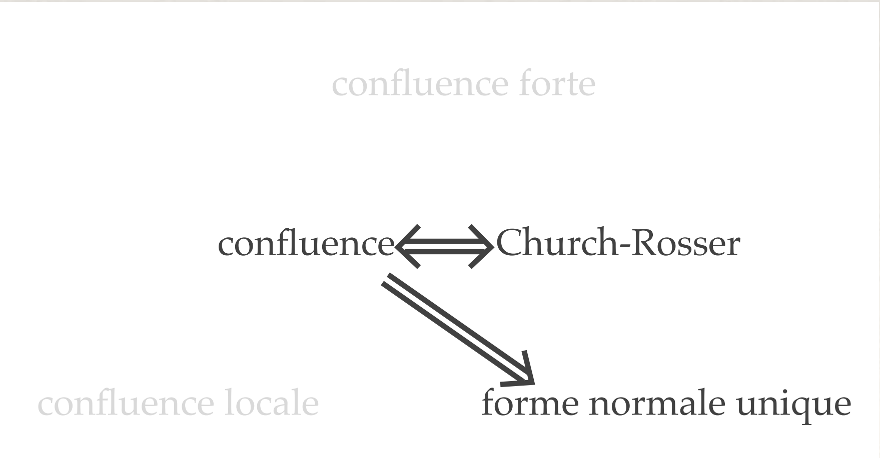


(Ceci est en réalité
une preuve par récurrence
sur le nombre de pics.)

Comment prouver la confluence?

- En principe, on devrait énumérer toutes les situations $v_1 \leftarrow^* u \rightarrow^* v_2$…
- … pour toutes les longueurs de réduction possibles de u à v_1 (resp. v_2)
- On va donc chercher des critères de confluence plus simples

$$\begin{tikzcd}
 & u \arrow[dl, "*"'] \arrow[dr, "*"] & \\
v_1 \arrow[dr, dotted, "*"'] & & v_2 \arrow[dl, dotted, "*"] \\
 & w &
\end{tikzcd}$$

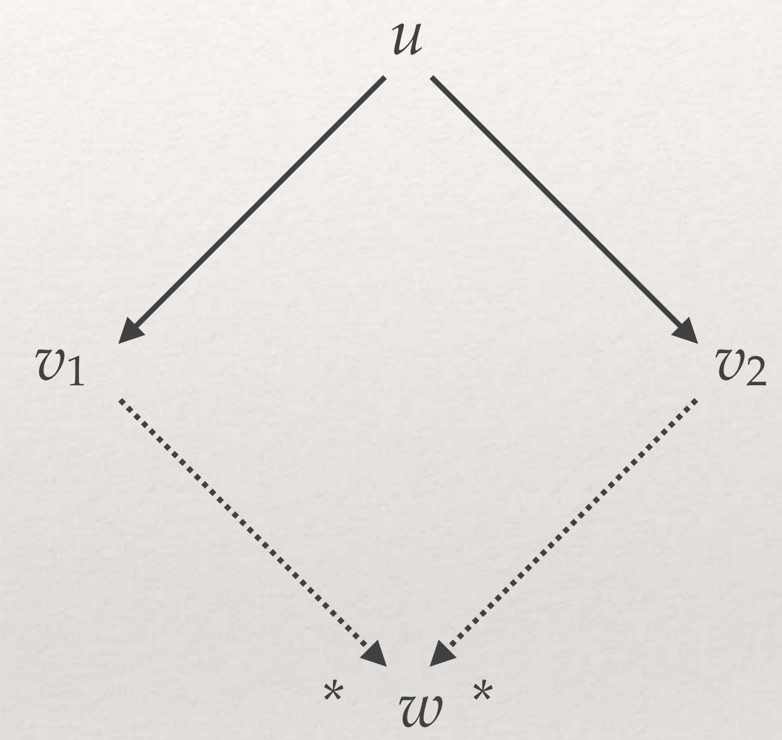


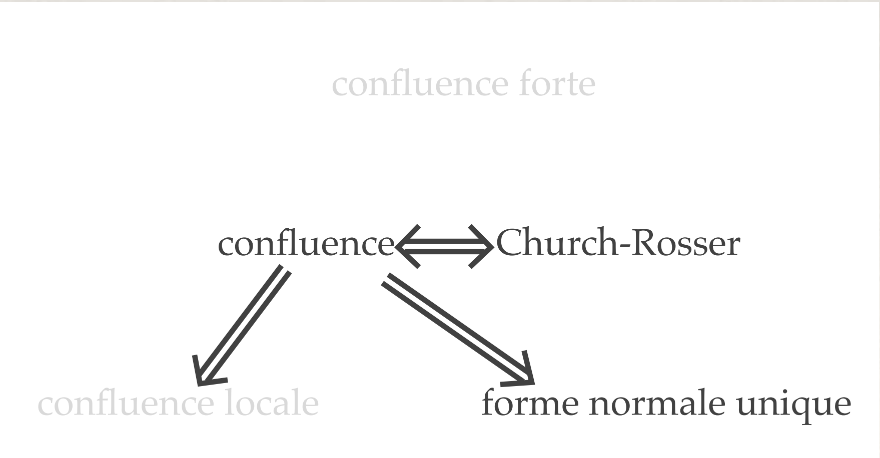
Confluence forte, confluence locale

Confluence locale

- $\rightarrow$ est **localement confluente** ssi:
- Voyez-vous la différence avec la confluence?
- Plus facile à vérifier (en fait, on n'a qu'à énumérer les paires critiques)
- **Fait.** Confluence implique confluence locale.

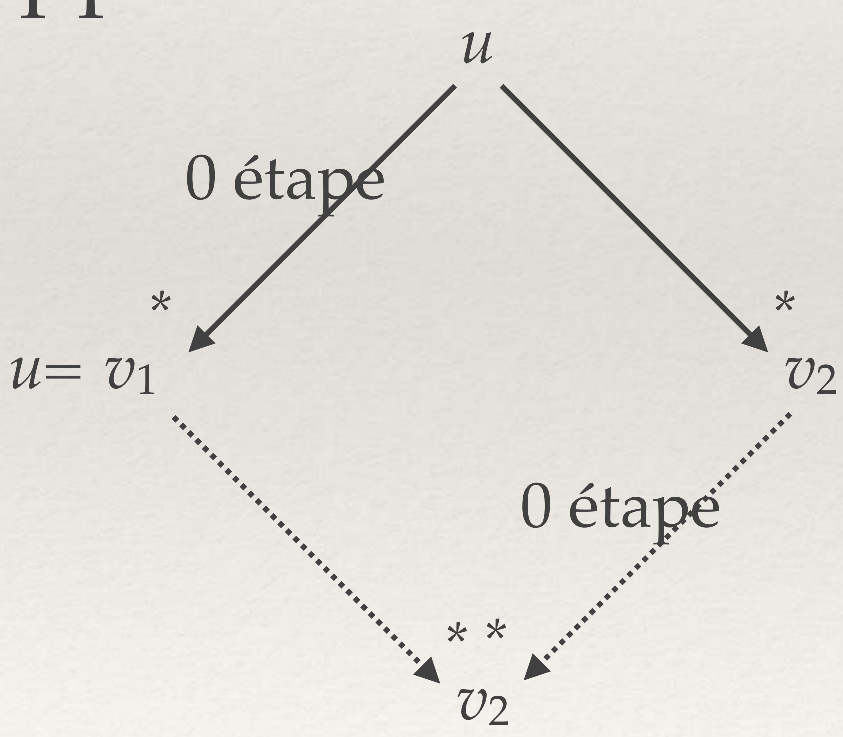
(L'implication n'est pas dans le sens souhaité… ça arrive.)





Confluence locale et confluence?

- La preuve suivante est fausse, dites-moi pourquoi.
- **Arnaque.** Si $\to$ loc. confluente alors $\to$ confluente (non).
- Supposons:



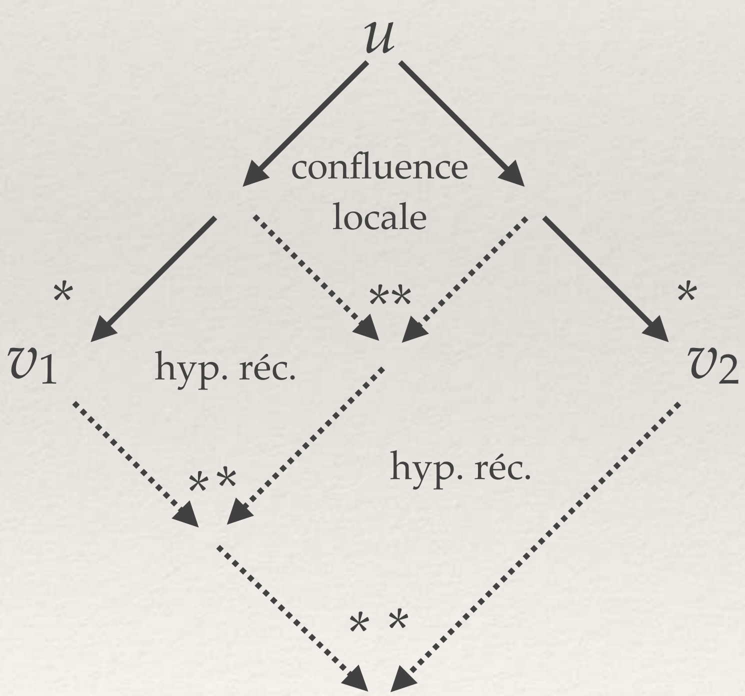
Si u=v_1…

Si u=v_2, on raisonne symétriquement.

Regardons donc ce qui se passe si ≥1 étape de u à v_1, resp. à v_2

Confluence locale et confluence?

- La preuve suivante est fausse, dites-moi pourquoi.
- **Arnaque.** Si $\rightarrow$ loc. confluente alors $\rightarrow$ confluente (non).
- Cas de récurrence:



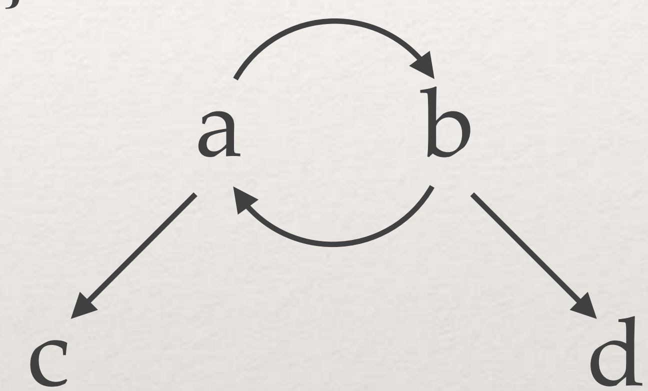
Où est l'erreur?

Le contre-exemple de Curry

Haskell B. Curry

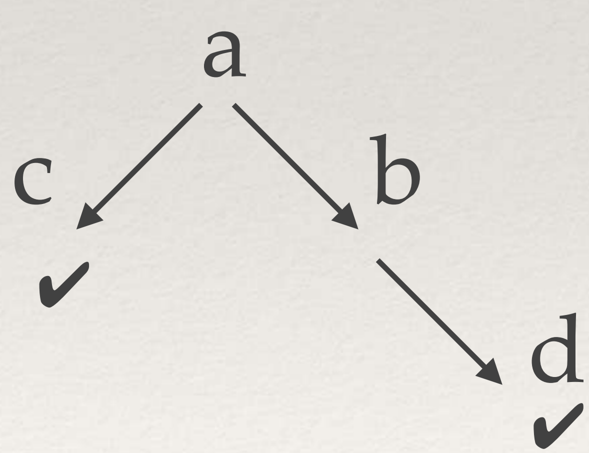
https://upload.wikimedia.org/wikipedia/commons/8/86/HaskellBCurry.jpg

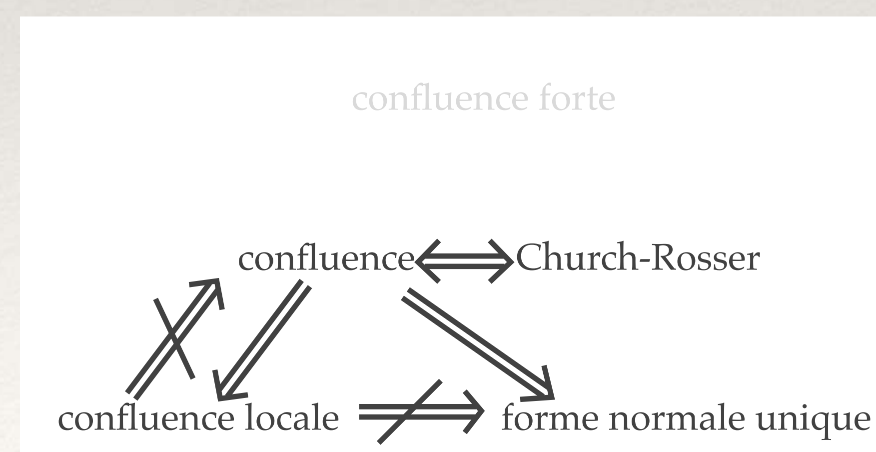
- Une relation → définie sur un ensemble à 4 éléments {a,b,c,d}



- Localement confluent:
  — c ← a → b joignable
  — a ← b → d joignable

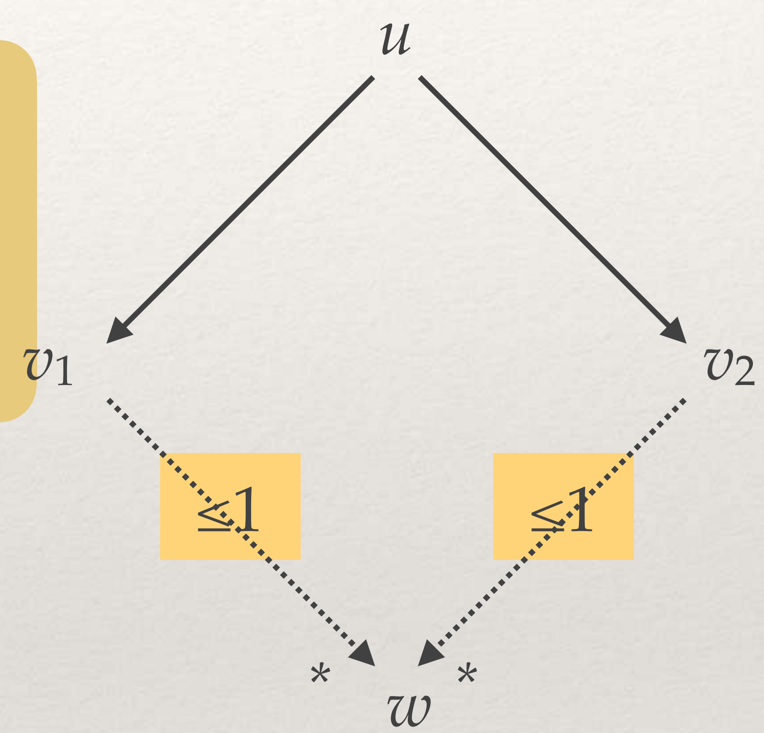
- **Pas** confluent (a n'a **pas** de forme normale **unique**)



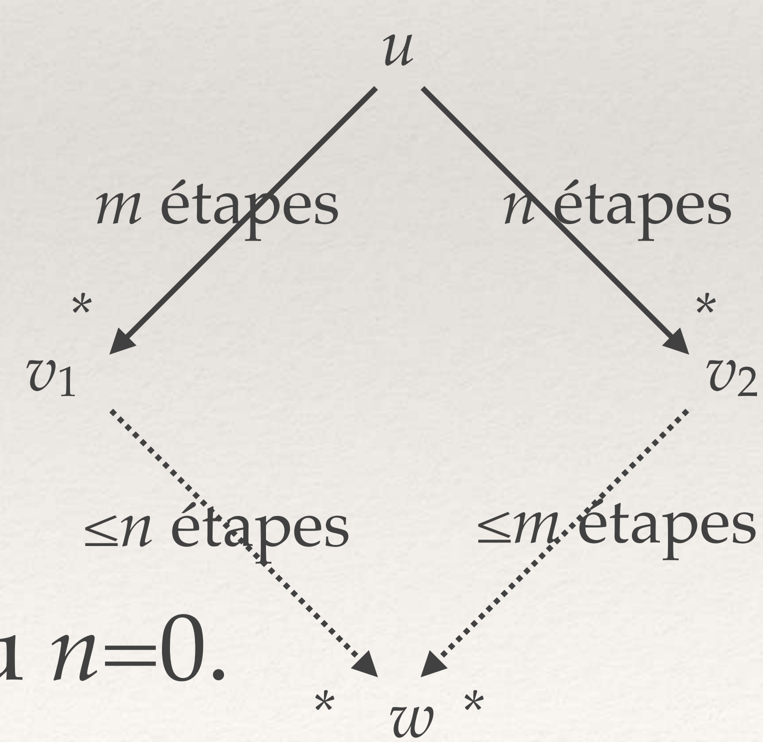


Confluence forte

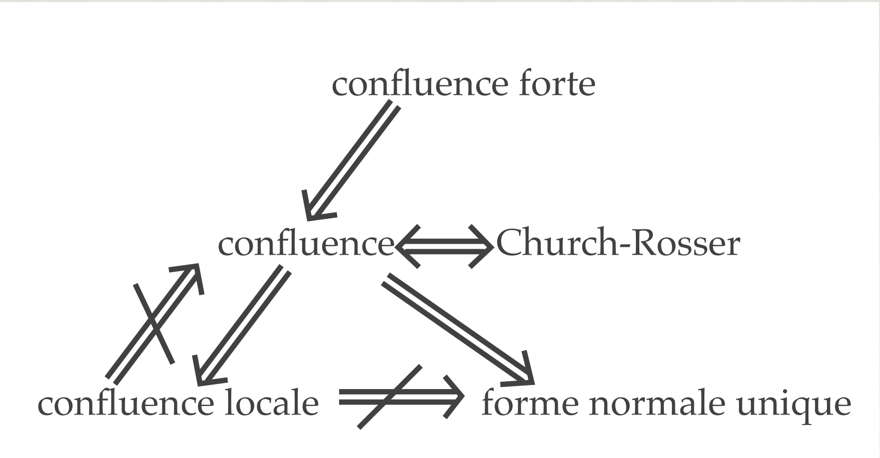
- $\to$ est **fortement confluente** ssi:
- **Lemme.** La confluence forte implique la confluence.



- Preuve: *quasiment* comme avant
- On montre:
- par récurrence sur m+n.

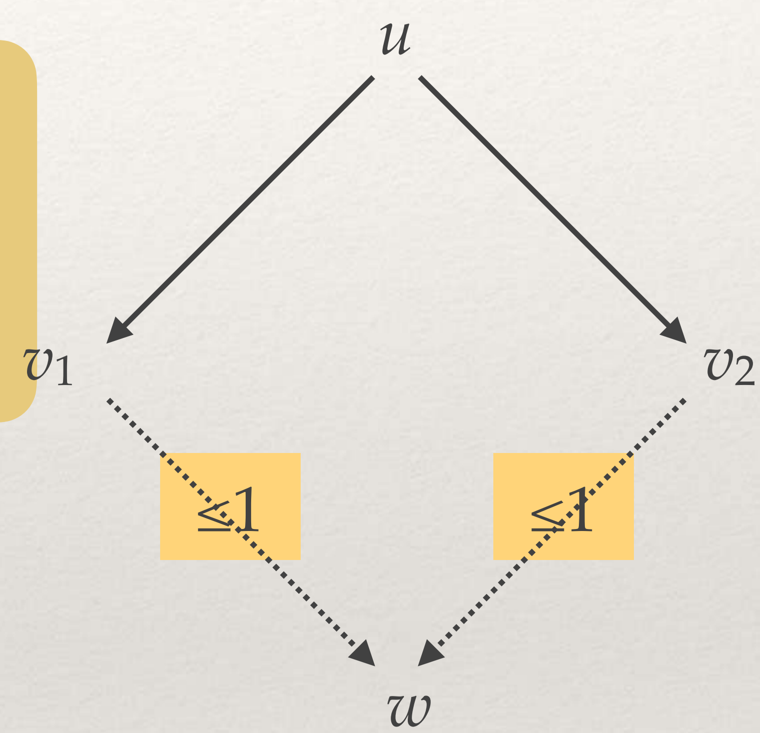


- Évident si m=0 ou n=0.



Confluence forte

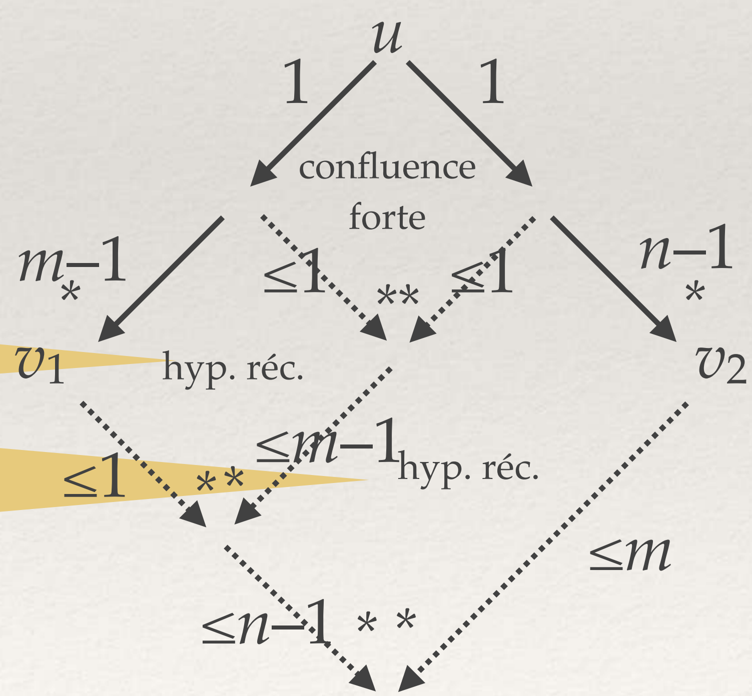
- $\rightarrow$ est **fortement confluente** ssi:

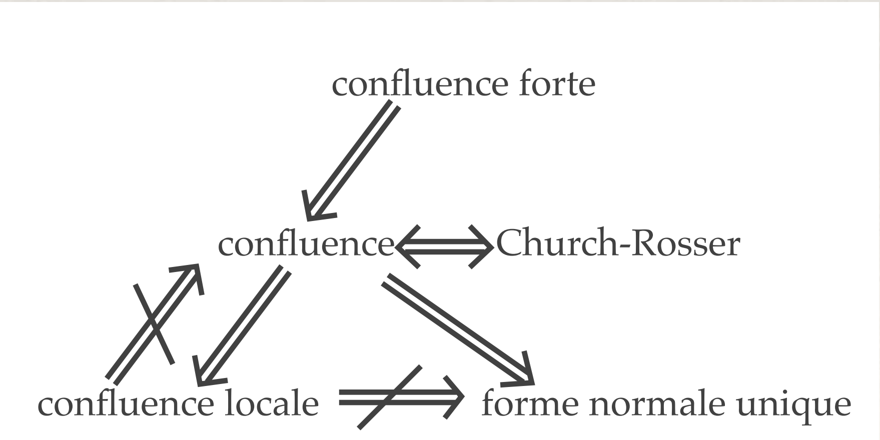


- **Lemme.** La confluence forte implique la confluence.
- Si $m \geq 1$ et $n \geq 1$:

s'applique car
$(m-1)+1 = m$
$< m+n$

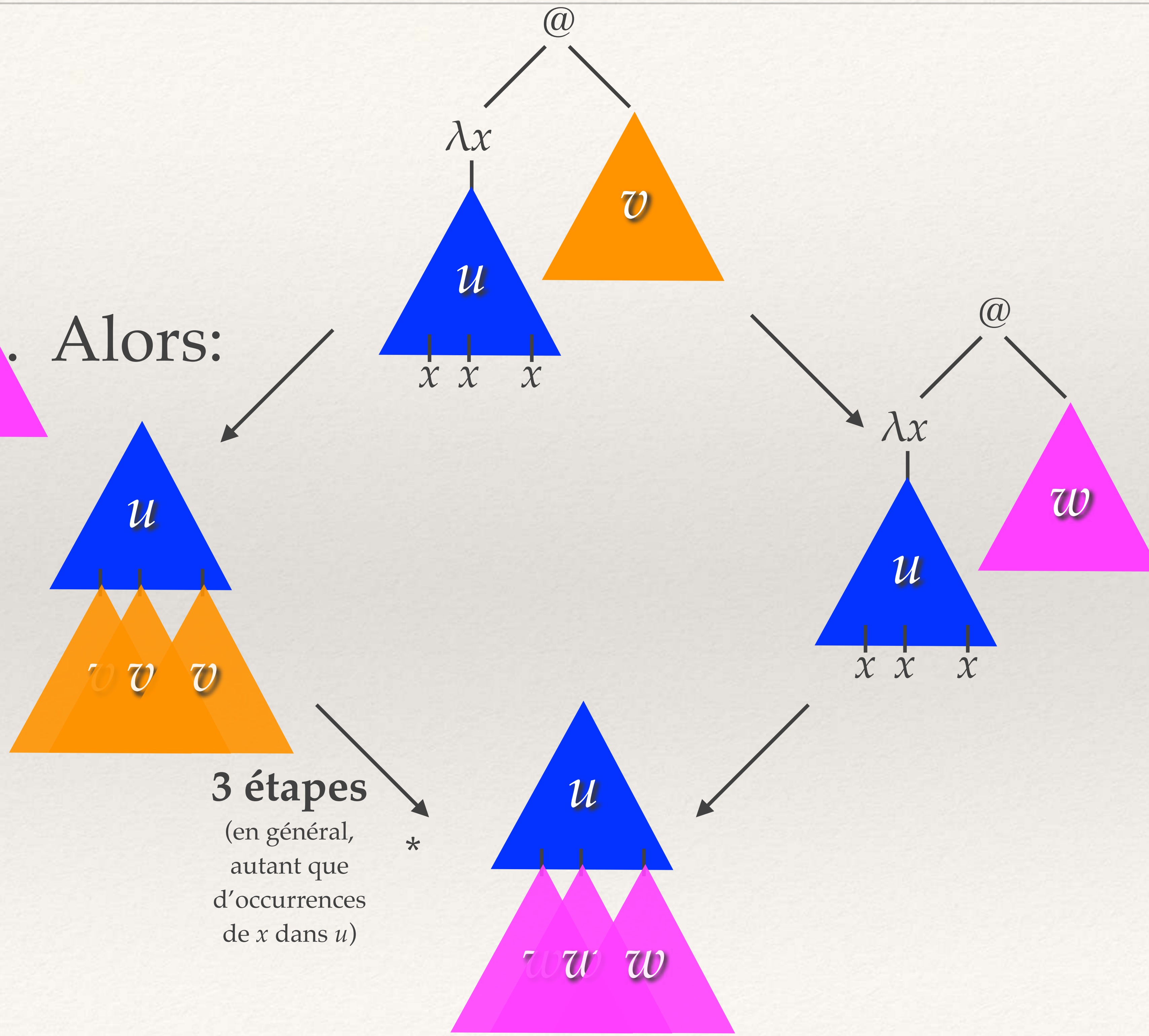
s'applique car
$(m-1+1)+(n-1)$
$= m+n-1$
$< m+n$





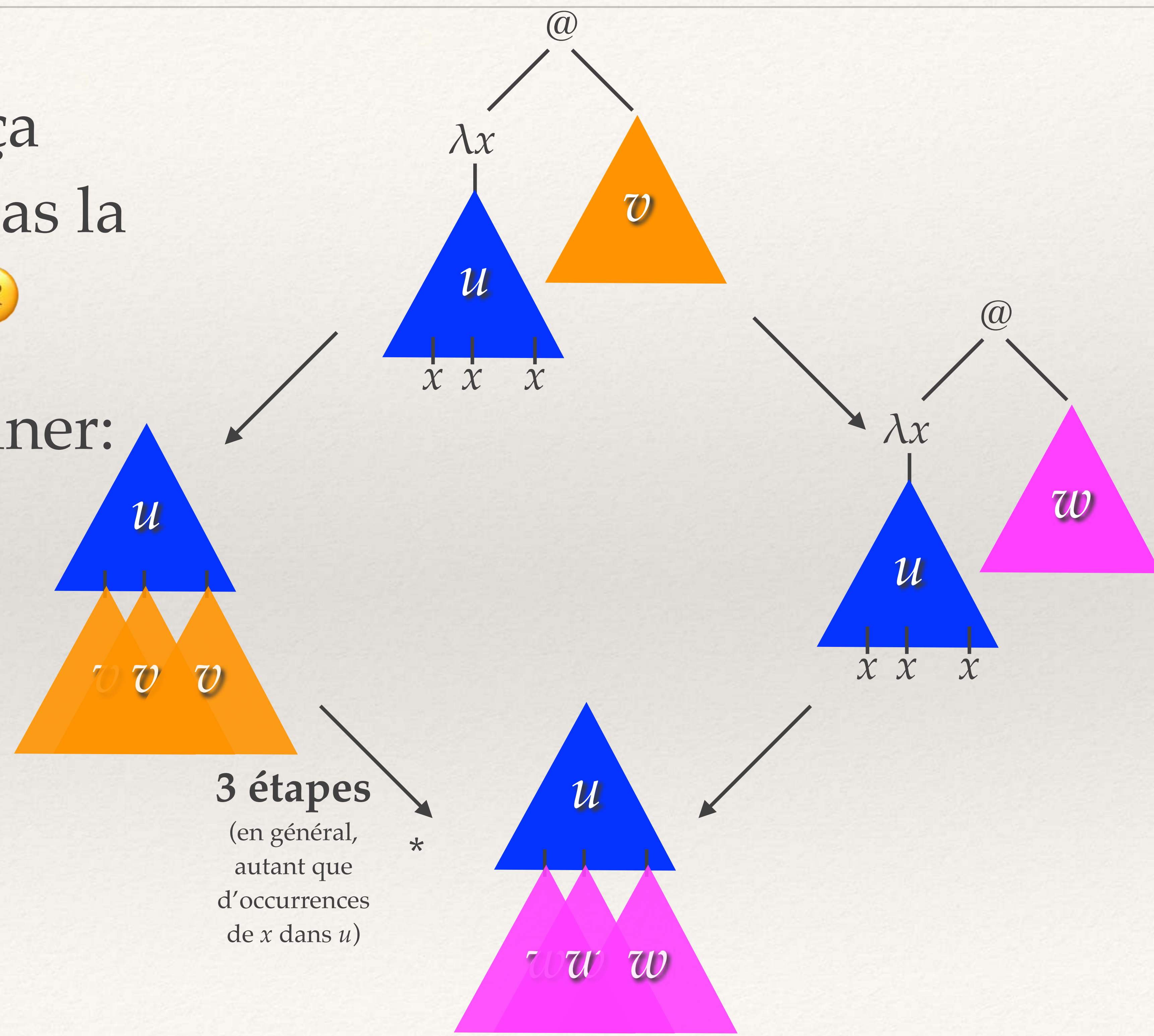
Le λ-calcul est-il fortement confluent?

- **Non…** 😢
- Imaginons que  . Alors:



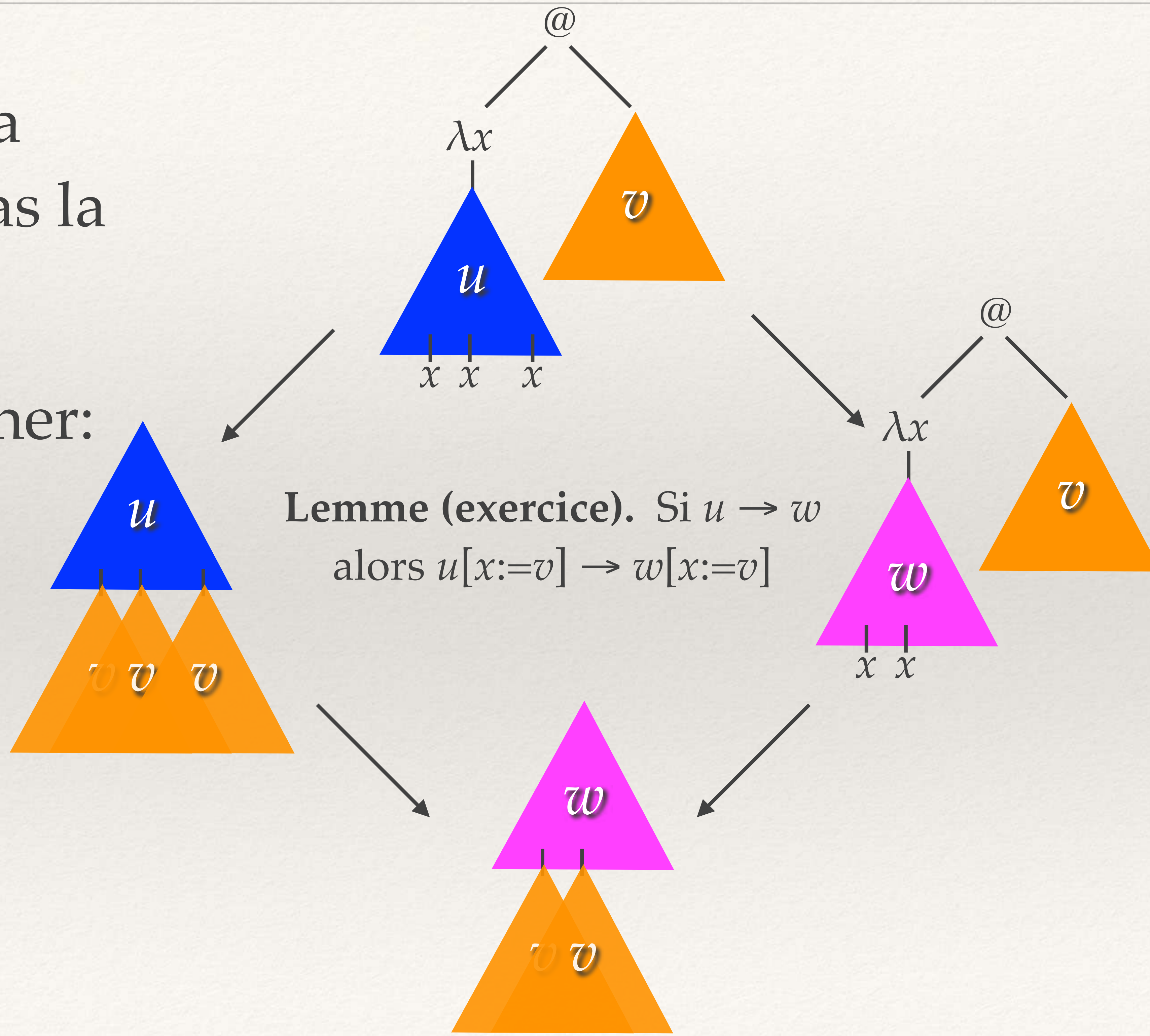
Le λ-calcul est-il localement confluent?

- **Oui**… mais ça n'implique pas la confluence 😕
- 3 cas à examiner:
- **Cas 1:** $v \longrightarrow w$, v argument d'un rédex



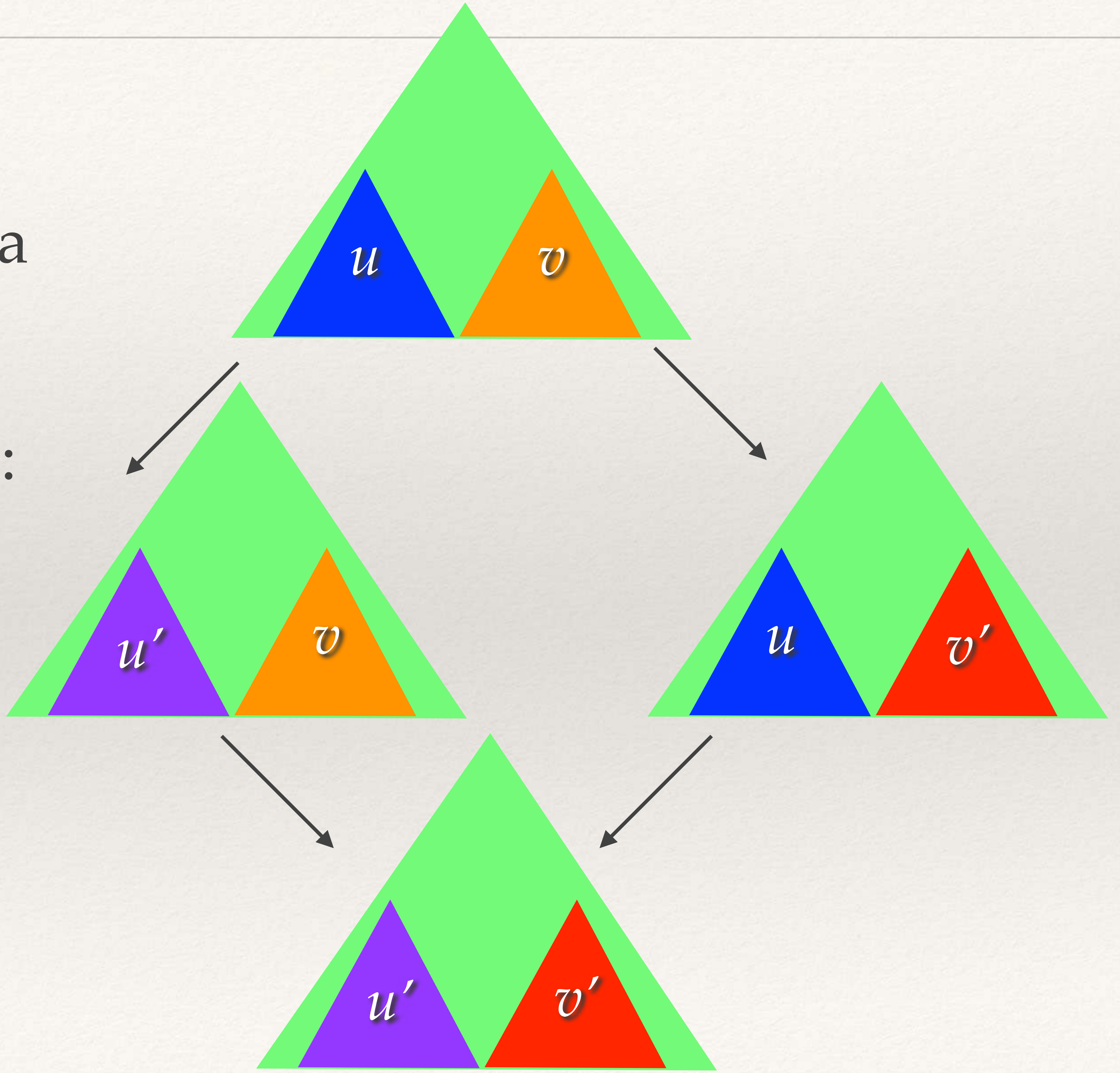
Le λ-calcul est-il localement confluent?

- **Oui**… mais ça n'implique pas la confluence 😕
- 3 cas à examiner:
- **Cas 2:** $u \to w$, u sous la λ-abstraction d'un rédex



Le λ-calcul est-il localement confluent?

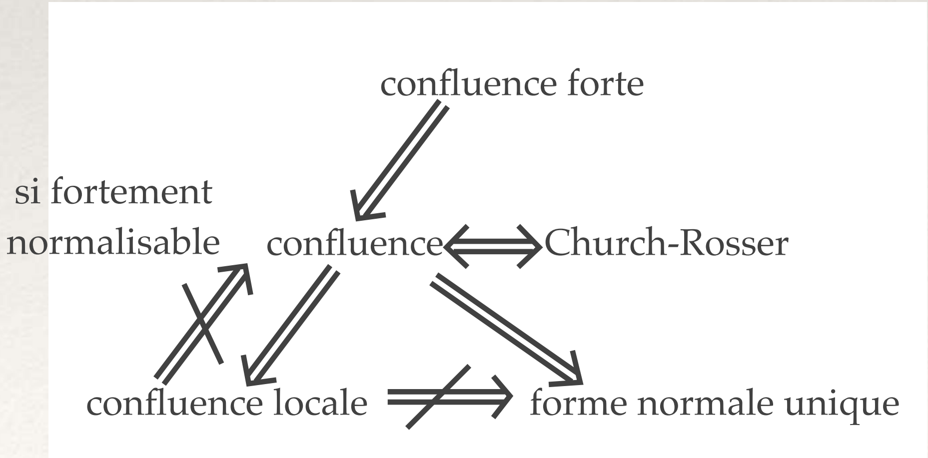
- **Oui**… mais ça n'implique pas la confluence 😕
- 3 cas à examiner:
- **Cas 3:** rédex **disjoints** $u \to u', v \to v'$



Le lemme de Newman

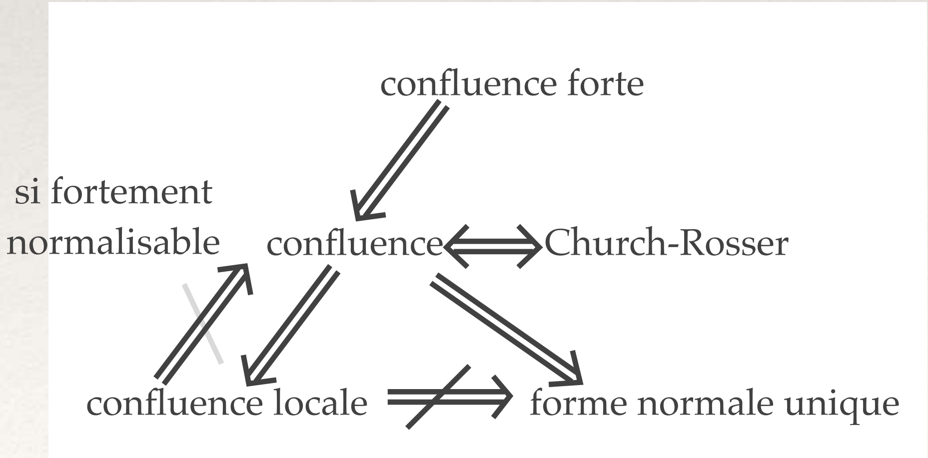
Le lemme de Newman

- **Lemme (Newman 1941).** Toute relation localement confluente
  et fortement normalisable
  est confluente.
- Preuve(s): transparents suivants.
- **Note 1:** le contre-ex. de Curry n'est **pas** fortement normalisable
- **Note 2:** la normalisabilité (faible) ne suffit pas (cf. Curry)
- **Note 3:** ne s'applique toujours pas au
  λ-calcul, qui n'est pas fortement
  normalisable…
  mais sera utile quand même!



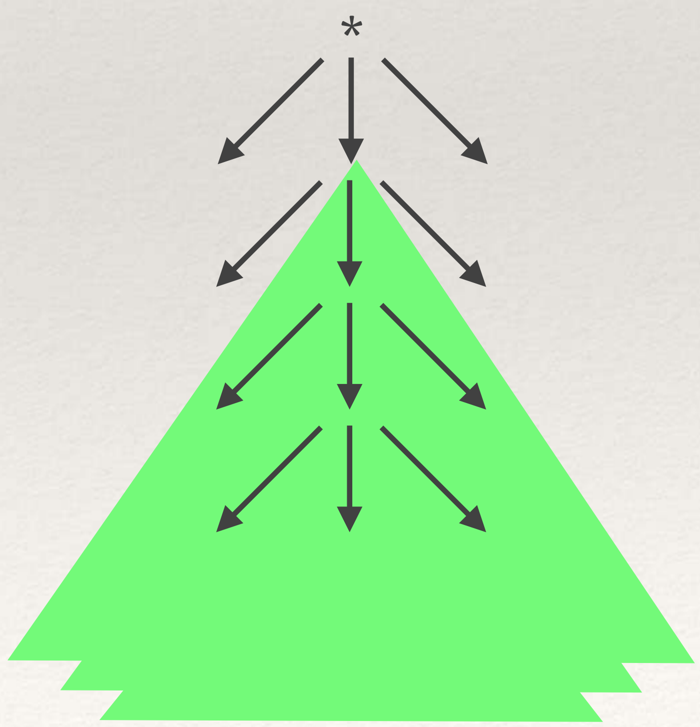
Le lemme de Newman

- **Lemme (Newman 1941).** Toute relation localement confluente
  et fortement normalisable
  est confluente.
- Deux preuves, dont l'une sous une hypothèse supplémentaire
  (mais plus simple que l'autre).
- Première preuve: on suppose que $\to$ est **à branchement fini**:
  pour tout u, $\{v \mid u \to v\}$ est fini
  (c'est clairement le cas en λ-calcul)
- Alors pour tout u, $\nu(u) \stackrel{\text{def}}{=}$ longueur max.
  d'une réduction partant de u existe:
  pourquoi?



Le lemme de Kőnig

- **Lemme (Kőnig).** Tout arbre à branchement fini
  et dont toutes les branches sont finies
  … est fini (n'a qu'un nombre fini de sommets).
- *Preuve.* Soit T un arbre **infini**, à branchement fini.
- * n'a qu'un #fini de successeurs
- **Tiroirs et chaussettes**: un de ceux-là est racine d'un sous-arbre **infini**
- … et l'on continue à l'infini, produisant une **branche infinie**.



Pourquoi $\nu(u)$ existe-t-il?

- Supposons $\to$ fortement normalisable et **à branchement fini**:
  pour tout u, $\{v \mid u \to v\}$ est fini
- $\forall\, u$, $\nu(u) \stackrel{\text{def}}{=}$ longueur max. d'une réduction partant de u existe:
  on forme l'arbre $T(u) \stackrel{\text{def}}{=} \{v \mid u \to^* v\}$
  - Il est à branchement fini par hypothèse
  - Ses branches sont finies car $\to$ fortement normalisable
- Par Kőnig, $T(u)$ est **fini**, et il n'y a en particulier
  qu'un nombre **fini** de réductions partant de u

Pourquoi $\nu(u)$ existe-t-il?

- Supposons $\to$ fortement normalisable et **à branchement fini**:
  pour tout u, $\{v \mid u \to v\}$ est fini
- $\forall\, u$, $\nu(u) \stackrel{\text{def}}{=}$ longueur max. d'une réduction partant de u existe:
  on forme l'arbre $T(u) \stackrel{\text{def}}{=} \{v \mid u \to^* v\}$

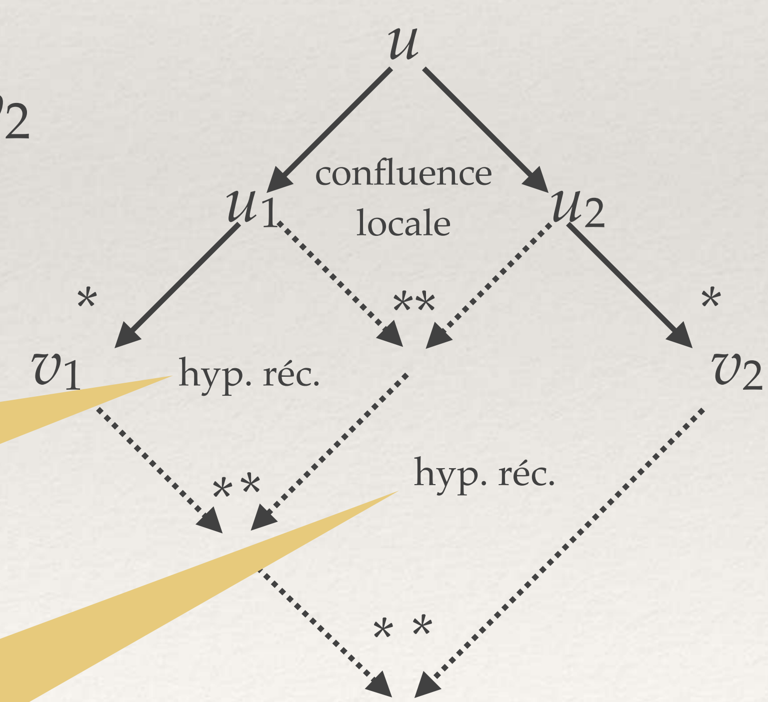
Note (facile mais importante).
Si $u \to v$ alors $\nu(u) > \nu(v)$.

- Par Kőnig, $T(u)$ est **fini**, et il n'y
  qu'un nombre **fini** de ré

Les gens rigoureux dans l'assistance auront remarqué que $T(u)$ est un graphe orienté, pas un arbre… deux solutions:
(1) montrer que Kőnig reste vrai pour tout **graphe** acyclique à branchement fini
(2) définir les sommets de $T(u)$ comme les réductions finies partant de u elles-mêmes, poser $^* \stackrel{\text{def}}{=} (u)$ et
$p(u \to u_1 \to \ldots \to u_{n-1} \to u_n) = (u \to u_1 \to \ldots \to u_{n-1})$

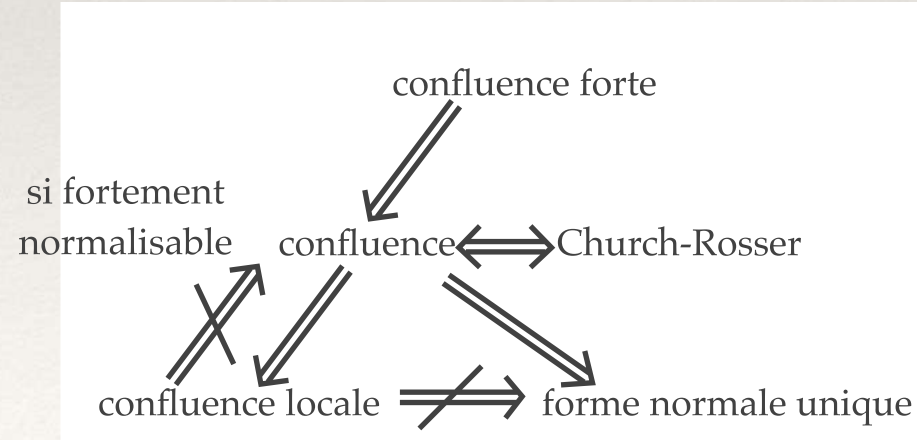
Le lemme de Newman: 1ère preuve

- **Lemme (Newman 1941).** Toute relation localement confluente
  et fortement normalisable et à branchement fini
  est confluente.
- On montre que si $v_1 \leftarrow^* u \rightarrow^* v_2$ alors v_1 et v_2 ont un réduit commun **par récurrence sur $\nu(u)$**
- Les cas $u{=}v_1$ et $u{=}v_2$ sont comme avant
- Sinon:



s'applique car $\nu(u) > \nu(u_1)$

s'applique car $\nu(u) > \nu(u_2)$



Le lemme de Newman: 2ème preuve

- **Lemme (Newman 1941).** Toute relation localement confluente **et** fortement normalisable ~~et à branchement fini~~ est confluente.
- Même preuve, mais on raisonne par **récurrence bien fondée sur *u*, directement**, ordonné strictement par $< \stackrel{\text{def}}{=} \leftarrow^+$
- **Principe de récurrence bien fondée:** voir transparent suivant

Récurrence bien fondée

- Soit $<$ un ordre strict. Les affirmations suivantes sont équivalentes:

1. $<$ est **bien fondée**: pas de chaîne ∞ décroissante $u_0 > u_1 > \ldots > u_n > \ldots$
2. pour toute prop. P, si $(\forall u, (\forall v < u, P(v))$ implique $P(u))$ alors $\forall u, P(u)$

- Preuve (non 2 implique non 1). Il y a une prop. P falsifiée par un u_0, et vérifiant $(\forall u, (\forall v < u, P(v))$ implique $P(u))$
- par contraposée, $\forall u_n$, si non $P(u_n)$ alors $\exists u_{n+1} < u_n$, non $P(u_{n+1})$
- (2 implique 1). Prendre $P(u) \stackrel{\text{def}}{=}$ « pas de chaîne ∞ décroissante partant de u »

Récurrence bien fondée

- Soit $<$ un ordre strict. Les affirmations suivantes sont équivalentes:

1. $<$ est **bien fondée**: pas de chaîne ∞ décroissante $u_0 > u_1 > \ldots > u_n > \ldots$
2. pour toute prop. P, si $(\forall u, (\forall v < u, P(v))$ implique $P(u))$ alors $\forall u, P(u)$

- Preuve (non 2 implique non 1). Il y a une prop. P falsifiée par un u_0, et vérifiant $(\forall u, (\forall v < u, P(v))$ implique $P(u))$

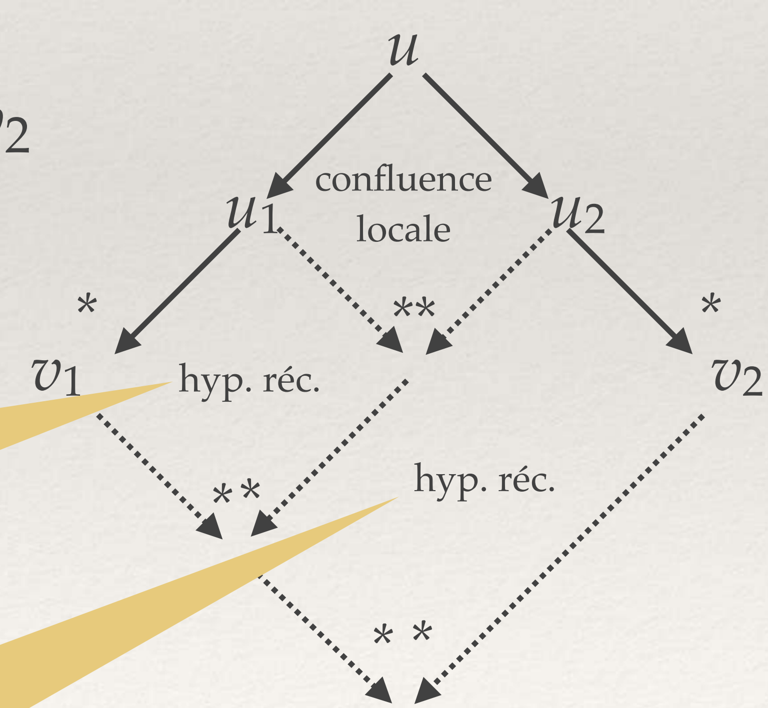
- u)

- e partant

Autrement dit, pour prouver $\forall u, P(u)$, il suffit de prendre un u quelconque, et de prouver $P(u)$ sous l'**hypothèse de récurrence**: $\forall v < u, P(v)$

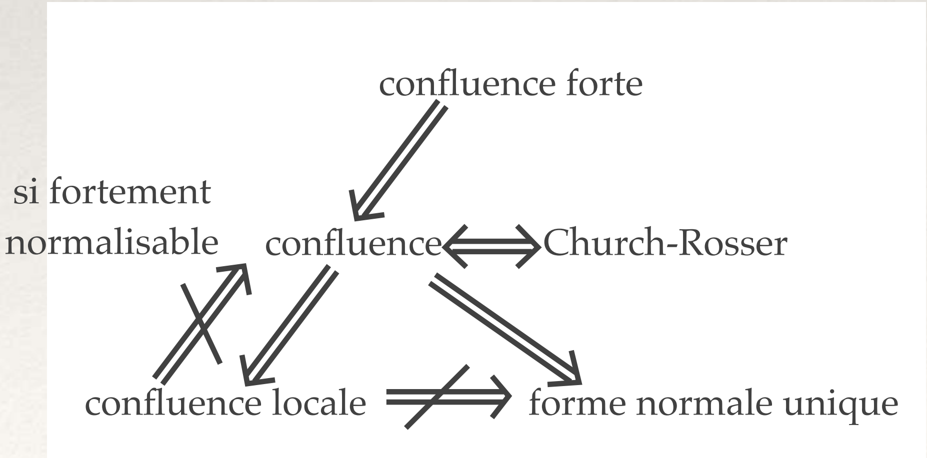
Le lemme de Newman: 2ème preuve

- **Lemme (Newman 1941).** Toute relation localement confluente
  et fortement normalisable ~~et à branchement fini~~
  est confluente.
- On montre que si $v_1 \leftarrow^* u \rightarrow^* v_2$ alors v_1 et v_2 ont un réduit commun **par récurrence bien fondée sur u strictement ordonné par** $< \stackrel{\text{def}}{=} \leftarrow^+$
- Les cas $u=v_1$ et $u=v_2$ sont comme avant
- Sinon:



s'applique car $u > u_1$

s'applique car $u > u_2$



Le λ-calcul est confluent

Réductions parallèles

- La preuve standard de confluence du λ-calcul
- On introduit une relation $\Rightarrow$ de **réduction parallèle**

  qui intuitivement autorise à réduire plusieurs rédexes, à condition qu'aucun n'en contienne un autre
- On montre que $\Rightarrow$ est **fortement confluente**

  donc confluente
- Enfin, $\rightarrow \subseteq \Rightarrow \subseteq \rightarrow^*$

  donc $\rightarrow$ sera confluente.

Réductions parallèles

0 réduction parallèle
(nouvelle règle)

$$\frac{}{u \Rightarrow u}\ (0)$$

$$\frac{u \Rightarrow u' \quad v \Rightarrow v'}{(\lambda x\ .\ u)\ v \Rightarrow u'[x{:=}v']}\ (\beta)$$

$$\frac{u \Rightarrow u'}{\lambda x\ .\ u \Rightarrow \lambda x\ .\ u'}\ (\lambda)$$

$$\frac{u \Rightarrow u' \quad v \Rightarrow v'}{uv \Rightarrow u'v'}\ (@)$$

On peut réduire
en parallèle
dans u et dans v

- Pour simplifier, raisonnons à α-équivalence près:
  les règles suivantes sont donc inutiles

$$\frac{u =_\alpha v \quad v \to v'}{u \to v'} \qquad \frac{u \to u' \quad u' =_\alpha v'}{u \to v'}$$

A comparer avec:

$$\frac{u \to u'}{uv \to u'v} \qquad \frac{v \to v'}{uv \to uv'}$$

$\rightarrow \subseteq \Rightarrow \subseteq \rightarrow^*$

$$\frac{}{u \Rightarrow u}\ (0)$$

$$\frac{u \Rightarrow u' \quad v \Rightarrow v'}{(\lambda x \,.\, u)\, v \Rightarrow u'[x{:=}v']}\ (\beta)$$

$$\frac{u \Rightarrow u'}{\lambda x \,.\, u \Rightarrow \lambda x \,.\, u'}\ (\lambda)$$

$$\frac{u \Rightarrow u' \quad v \Rightarrow v'}{uv \Rightarrow u'v'}\ (@)$$

- Commençons par les derniers résultats (les plus faciles)
- **Lemme** ($\rightarrow \subseteq \Rightarrow$). Si $u \rightarrow v$ alors $u \Rightarrow v$.
  (Récurrence sur la profondeur du rédex contracté dans u.)
- **Lemme** ($\Rightarrow \subseteq \rightarrow^*$). Si $u \Rightarrow v$ alors $u \rightarrow^* v$.
  (Récurrence sur la taille de la dérivation de $u \Rightarrow v$.)
- **Corl.** $\rightarrow^* = \Rightarrow^*$.

Le lemme de substitution pour $\Rightarrow$

$$\frac{}{u \Rightarrow u}\ (0) \qquad \frac{u \Rightarrow u' \quad v \Rightarrow v'}{(\lambda x\,.\,u)\,v \Rightarrow u'[x{:=}v']}\ (\beta) \qquad \frac{u \Rightarrow u'}{\lambda x\,.\,u \Rightarrow \lambda x\,.\,u'}\ (\lambda) \qquad \frac{u \Rightarrow u' \quad v \Rightarrow v'}{uv \Rightarrow u'v'}\ (@)$$

- **Lemme.** Si $u \Rightarrow u'$ et $w \Rightarrow w'$
  alors $u[z{:=}w] \Rightarrow u'[z{:=}w']$

- *Preuve.* Récurrence sur la **taille de la dérivation** donnée de $u \Rightarrow u'$.
  (Exercice. Vous aurez à prouver quelques lemmes auxiliaires.
  Attention à α-renommer dans le cas de la règle (β).)

- **Note**: avec la β-réduction (ordinaire), on a:
  — si $w \to w'$ alors $u[z{:=}w] \to^* u'[z{:=}w']$
  — si $u \to u'$ alors $u[z{:=}w] \to u'[z{:=}w]$

$\Rightarrow$ est fortement confluente

$$\frac{}{u \Rightarrow u}\ (0)$$

$$\frac{u \Rightarrow u' \quad v \Rightarrow v'}{(\lambda x\,.\,u)\,v \Rightarrow u'[x{:=}v']}\ (\beta)$$

$$\frac{u \Rightarrow u'}{\lambda x\,.\,u \Rightarrow \lambda x\,.\,u'}\ (\lambda)$$

$$\frac{u \Rightarrow u' \quad v \Rightarrow v'}{uv \Rightarrow u'v'}\ (@)$$

- Supposons $s \Rightarrow t_1$ et $s \Rightarrow t_2$
- On montre qu'il existe t_3 / $t_1, t_2 \Rightarrow t_3$ par récurrence sur **la somme des tailles des dérivations** de $s \Rightarrow t_1$ et $s \Rightarrow t_2$
- A symétrie près, 10 cas
- (0)/–: $s{=}t_1$ [$=u$], poser $t_3 \stackrel{\text{def}}{=} t_2$
- (β)/(λ), (λ)/(@): impossibles
- (λ)/(λ), (@)/(@): par hyp. réc.
- (β)/(β): voir transparent suivant

	(0)	(β)	(λ)	(@)
(0)	✔	✔	✔	✔
(β)	–		–	
(λ)	–	–	✔	–
(@)	–	–	–	✔

$\Rightarrow$ est fortement confluente

$$\frac{}{u \Rightarrow u}\ (0) \qquad \frac{u \Rightarrow u' \quad v \Rightarrow v'}{(\lambda x\,.\,u)\ v \Rightarrow u'[x{:=}v']}\ (\beta) \qquad \frac{u \Rightarrow u'}{\lambda x\,.\,u \Rightarrow \lambda x\,.\,u'}\ (\lambda) \qquad \frac{u \Rightarrow u' \quad v \Rightarrow v'}{uv \Rightarrow u'v'}\ (@)$$

- (β)/(β): on a

$$\frac{\begin{matrix}\vdots & \vdots\\ u \Rightarrow u_1 & v \Rightarrow v_1\end{matrix}}{\underbrace{(\lambda x\,.\,u)\ v}_{s} \Rightarrow \underbrace{u_1[x{:=}v_1]}_{t_1}} \qquad \frac{\begin{matrix}\vdots & \vdots\\ u \Rightarrow u_2 & v \Rightarrow v_2\end{matrix}}{\underbrace{(\lambda x\,.\,u)\ v}_{s} \Rightarrow \underbrace{u_2[x{:=}v_2]}_{t_2}}$$

  - Par h.r., on a $u_1, u_2 \Rightarrow u_3$ et $v_1, v_2 \Rightarrow v_3$
  - Par le **Lemme.** Si $u \Rightarrow u'$ et $v \Rightarrow v'$ alors $u[x{:=}v] \Rightarrow u'[x{:=}v']$
    $t_1 \Rightarrow u_3[x{:=}v_3]$
  - Par le **Lemme.** Si $u \Rightarrow u'$ et $v \Rightarrow v'$ alors $u[x{:=}v] \Rightarrow u'[x{:=}v']$
    $t_2 \Rightarrow u_3[x{:=}v_3]$

C'est le t_3 désiré

	(0)	(β)	(λ)	(@)
(0)	✔	✔	✔	✔
(β)	–	✔	–	
(λ)	–	–	✔	–
(@)	–	–	–	✔

$\Rrightarrow$ est fortement confluente

$$\frac{}{u \Rightarrow u}\;(0) \qquad \frac{u \Rightarrow u' \quad v \Rightarrow v'}{(\lambda x\,.\,u)\;v \Rightarrow u'[x{:=}v']}\;(\beta) \qquad \frac{u \Rightarrow u'}{\lambda x\,.\,u \Rightarrow \lambda x\,.\,u'}\;(\lambda) \qquad \frac{u \Rightarrow u' \quad v \Rightarrow v'}{uv \Rightarrow u'v'}\;(@)$$

- Dans le (dernier) cas (β)/(@), on a:

$$\frac{\begin{array}{cc}\vdots & \vdots \\ u \Rightarrow u_1 & v \Rightarrow v_1\end{array}}{\underbrace{(\lambda x\,.\,u)\;v}_{s} \Rightarrow \underbrace{u_1[x{:=}v_1]}_{t_1}} \qquad \frac{\begin{array}{cc}\vdots & \vdots \\ (\lambda x\,.\,u) \Rightarrow w & v \Rightarrow v_2\end{array}}{\underbrace{(\lambda x\,.\,u)v}_{s} \Rightarrow \underbrace{wv_2}_{t_2}}$$

- Mais $(\lambda x\,.\,u) \Rightarrow w$ ne peut être dérivé que par (0) ou (λ) … donc la situation est…

	(0)	(β)	(λ)	(@)
(0)	✔	✔	✔	✔
(β)	–	✔	–	
(λ)	–	–	✔	–
(@)	–	–	–	✔

$\Rightarrow$ est fortement confluente

$$\frac{}{u \Rightarrow u}\ (0) \qquad \frac{u \Rightarrow u' \quad v \Rightarrow v'}{(\lambda x\,.\,u)\,v \Rightarrow u'[x{:=}v']}\ (\beta) \qquad \frac{u \Rightarrow u'}{\lambda x\,.\,u \Rightarrow \lambda x\,.\,u'}\ (\lambda) \qquad \frac{u \Rightarrow u' \quad v \Rightarrow v'}{uv \Rightarrow u'v'}\ (@)$$

- Dans le (dernier) cas (β)/(@), on a:

$$\frac{\begin{matrix}\vdots \\ u \Rightarrow u_1\end{matrix} \quad \begin{matrix}\vdots \\ v \Rightarrow v_1\end{matrix}}{\underbrace{(\lambda x\,.\,u)\,v}_{s} \Rightarrow \underbrace{u_1[x{:=}v_1]}_{t_1}} \qquad \frac{\dfrac{\begin{matrix}\vdots \\ u \Rightarrow u_2\end{matrix}}{(\lambda x\,.\,u) \Rightarrow (\lambda x\,.\,u_2)} \quad \begin{matrix}\vdots \\ v \Rightarrow v_2\end{matrix}}{\underbrace{(\lambda x\,.\,u)v}_{s} \Rightarrow \underbrace{(\lambda x\,.\,u_2)\,v_2}_{t_2}}$$

  - Par h.r., on a $u_1, u_2 \Rightarrow u_3$
    et $v_1, v_2 \Rightarrow v_3$
  - Par le **Lemme.** Si $u \Rightarrow u'$ et $v \Rightarrow v'$ alors $u[x{:=}v] \Rightarrow u'[x{:=}v']$
    $t_1 \Rightarrow u_3[x{:=}v_3]$
  - Par (β),
    $t_2 \Rightarrow u_3[x{:=}v_3]$

C'est le t_3 désiré

	(0)	(β)	(λ)	(@)
(0)	✔	✔	✔	✔
(β)	–	✔	–	✔
(λ)	–	–	✔	–
(@)	–	–	–	✔

Le λ-calcul est confluent

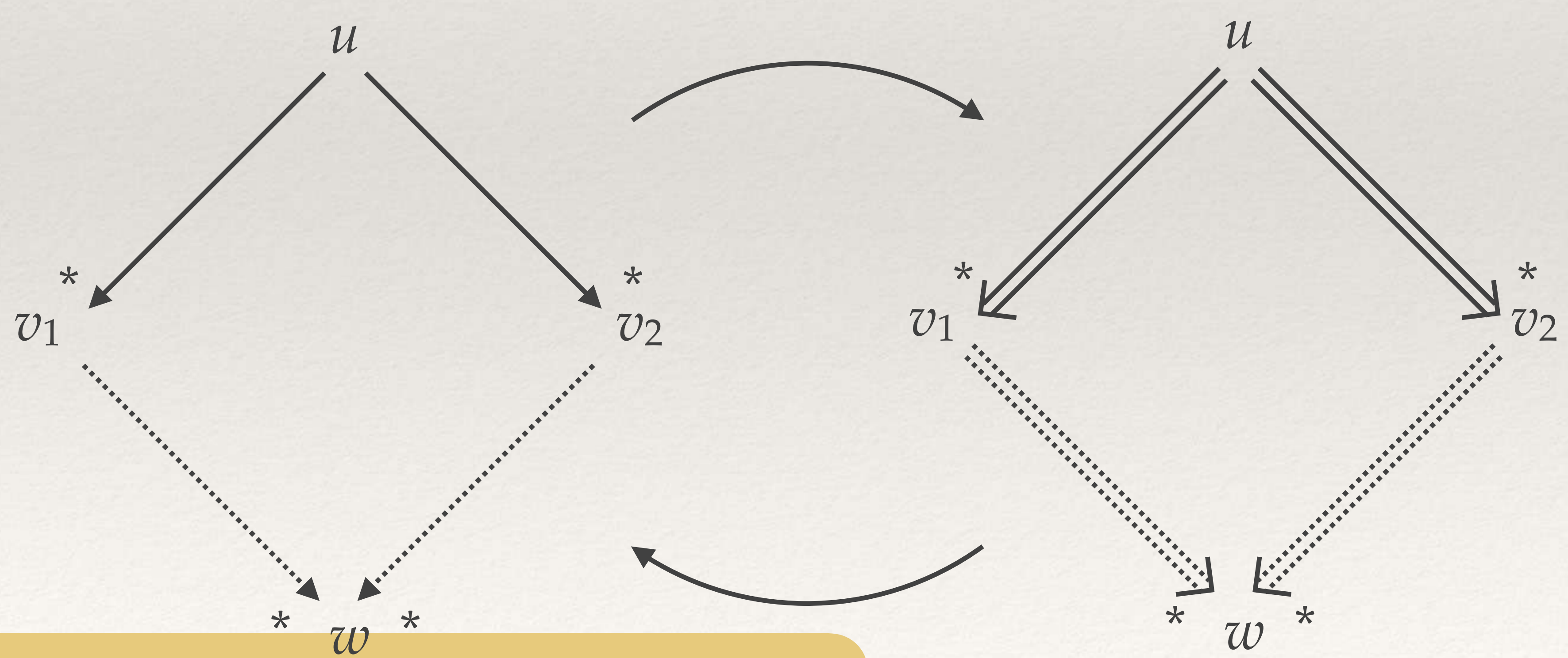
- Nous venons de démontrer:

  Prop. Si $s \Rightarrow t_1$ et $s \Rightarrow t_2$ alors il existe t_3 / $t_1, t_2 \Rightarrow t_3$.

- I.e., $\Rightarrow$ est **fortement confluente** … donc **confluente**.

- On rappelle aussi: $\rightarrow^* = \Rightarrow^*$.

- Si:



Donc le λ-calcul est confluent (pour la β-réduction). $\square$

Confluence: β n'entraîne pas η

- Vous vous souvenez de la règle suivante?
  (η) $\lambda x \,.\, ux \to u$ (si x pas libre dans u)
  et considérer la βη-réduction $\to_{\beta\eta}$
- Elle a l'air raisonnablement mathématiquement (pas informatiquement). Surtout, elle est **indépendante** de β:
- **Prop.** En général, $\lambda x \,.\, ux \neq_\beta u$ (même si x pas libre dans u).
- *Preuve.* On prend u = une variable $\neq x$:
  les deux côtés sont en forme β-normale est pas α-équivalents, donc pas β-équivalents
  (confluence implique forme normale **unique**). □

La prochaine fois

- Pouvoir expressif: fonctions récursives, combinateurs de point fixe
- Et plus tard: stratégies