Syntaxe, sémantique, calculabilité

Gérard Berry

Collège de France
Chaire Informatique et sciences numériques
Cours 1, 25 novembre 2009

Agenda

1. Notions de base sur le calcul
2. Calculabilité : machines, langages et fonctions
3. Les grands résultats
4. Appel par noms, appel par valeurs
5. Introduction au λ-calcul

Agenda

1. Notions de base sur le calcul
2. Calculabilité : machines, langages et fonctions
3. Les grands résultats
4. Appel par noms, appel par valeurs
5. Introduction au λ-calcul

Syntaxe et sémantique

- Domaine des concepts (ou du sens)
  nombres, fonctions, graphes, algorithmes, etc.
- Domaine des symboles (ou des signes)
  représentations décimales, chaînes de caractères, etc.

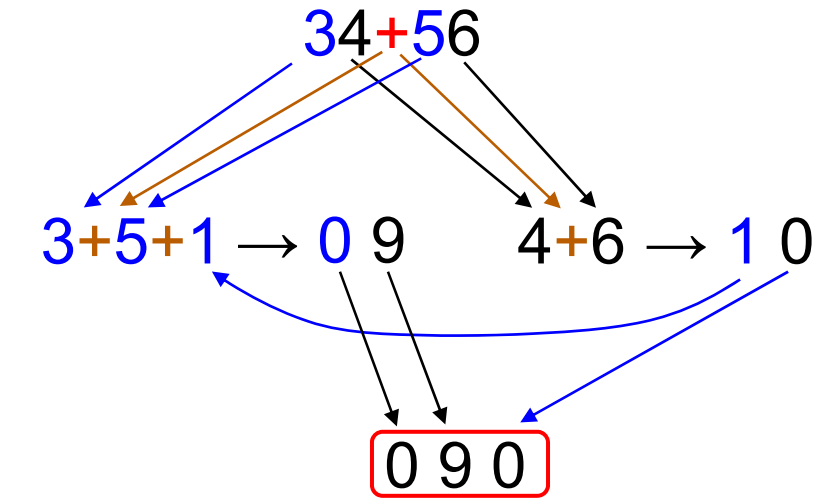
34+56

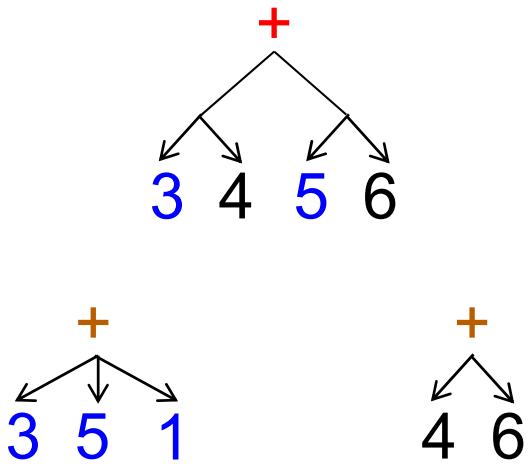
– ~~la somme du nombre 34 et du nombre 56~~
– des traits noirs
– une suite de symboles
– la représentation décimale de « trente-quatre »
  suivie du symbole de l'addition
  suivi de la représentation décimale
  de « cinquante-six »

Sémantique : relation entre concepts et symboles

Homme : la somme du nombre 34 et du nombre 56
est le nombre 90

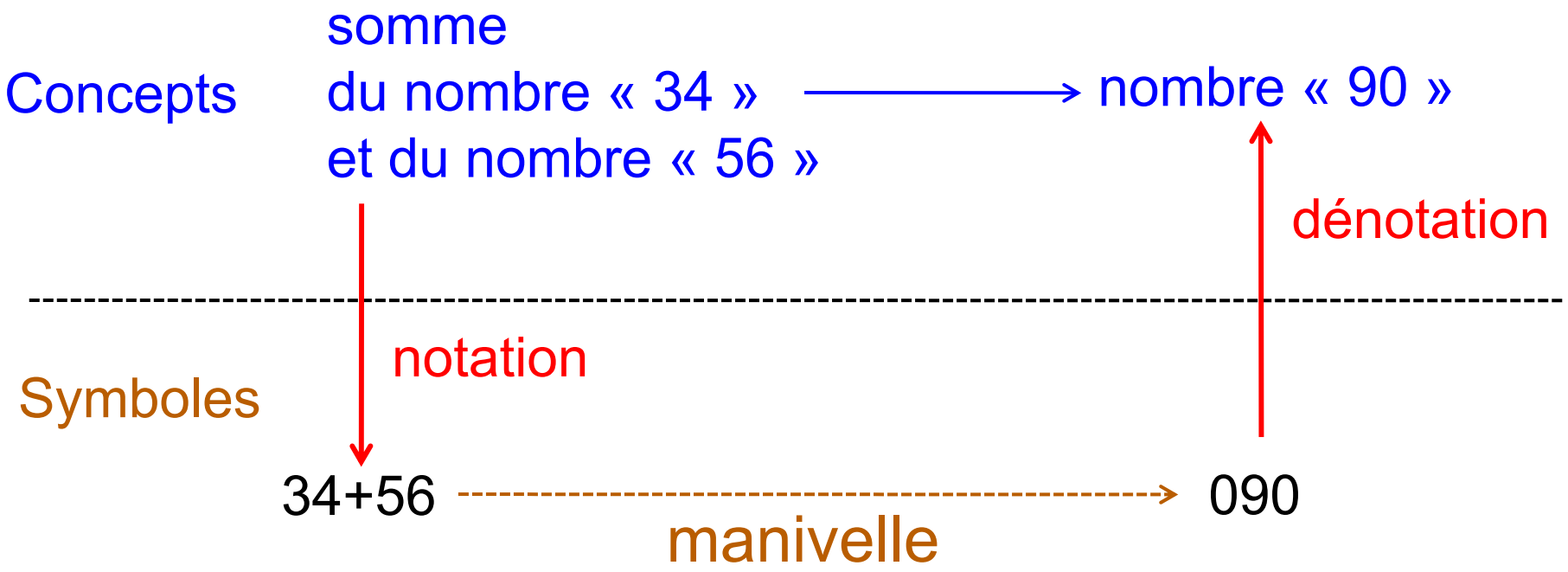
Machine : 34+56





Il n'y a pas de nombres dans un ordinateur
rien que des symboles, en nombre fini à chaque instant

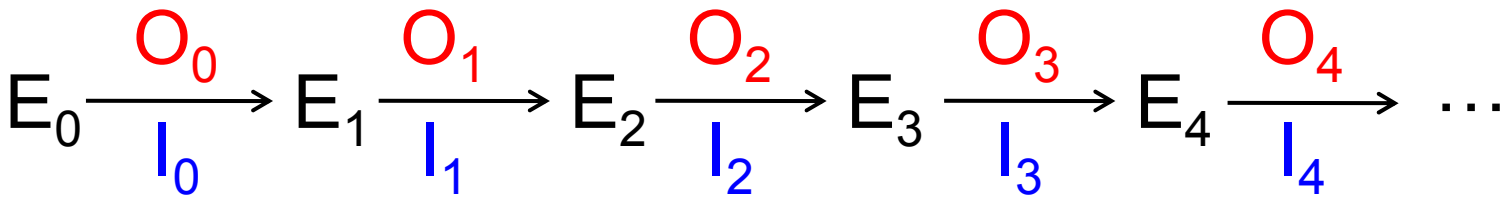
Le diagramme sémantique



Pour comprendre l'informatique, il faut arriver à
être aussi bête qu'un ordinateur
en utilisant toujours des ressources concrètes et finies

Trois sortes de calcul

- Transformationnels : données → résultats
- Interactifs et réactifs : interaction permanente



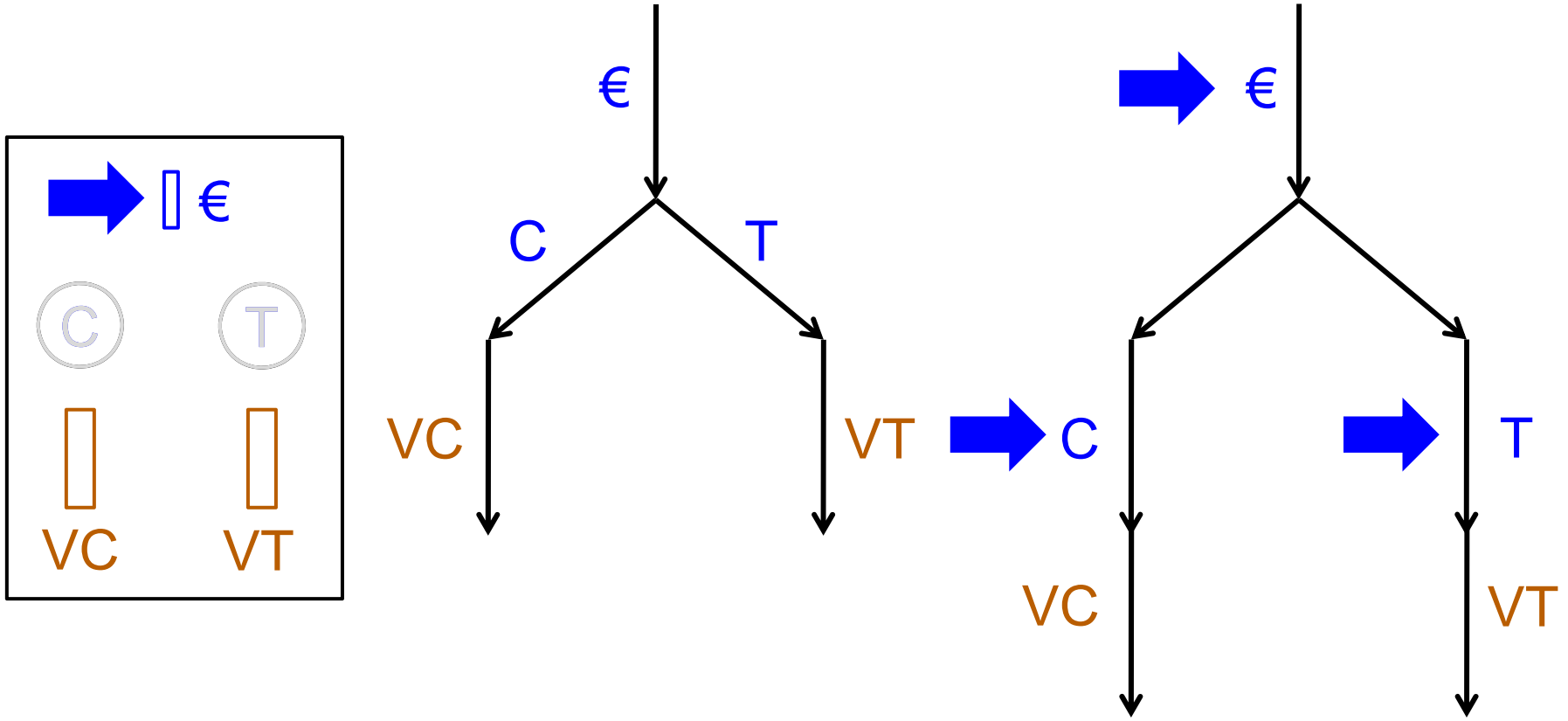
  - interactif : au rythme de la machine
    système d'exploitation, réseau, navigateur Web, etc.
  - réactif : au rythme de l'environnement
    pilotage d'avion, freinage de voiture, etc.
    souvent « temps réel »

Déterminisme / non-déterminisme

- Déterminisme : un seul résultat
  - calcul mathématique standard, Eratosthène, Darwin,
  - circuits électroniques
  - pilotage d'avions, conduite de voitures, etc.
  - déterminisme absolu : un seul calcul possible (Eratosthène)

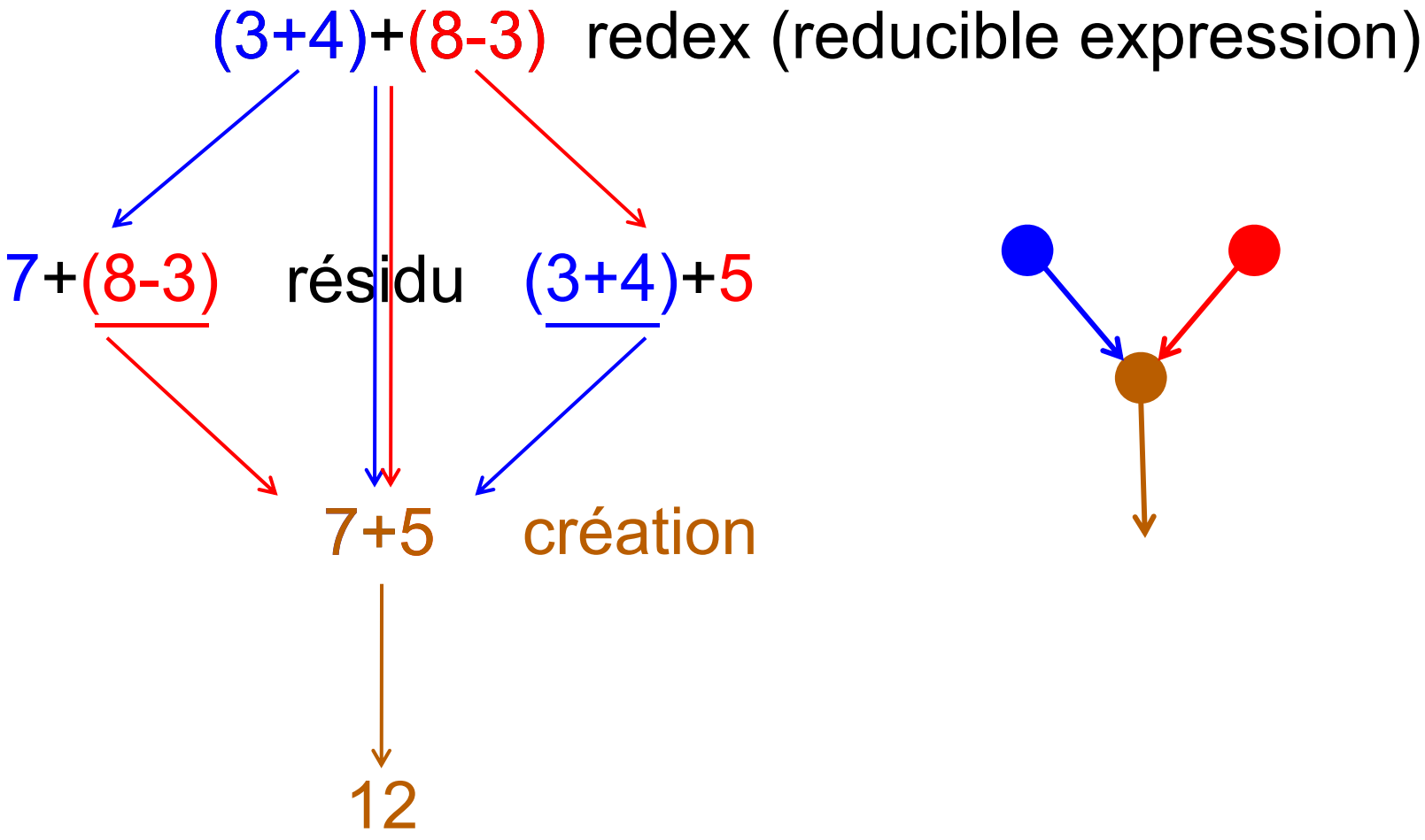
- Non-déterminisme : plusieurs résultats
  - navigateur Internet, moteur de recherches, etc.
  - incertitude : on ne peut pas connaître en même temps la position et le contenu d'une page Web
  - encore quelques voitures (hélas)

Deux machines à café / thé



Laquelle préférez vous ?

Causalité des calculs



La causalité détermine la liberté du calcul

La compositionnalité

- Add(m,n) = < code > : une routine d'addition
- P : ….Add(x,y)....Add(z,t)….Add(x,u)…..
- Pour montrer P correct, il faudra prouver que les trois appels à Add font bien une addition
- Solution: lemme reliant syntaxe et sémantique :
  - Si des expressions *e* et *e'* dénotent respectivement des nombres *n* et *n'*, alors, dans tout contexte, l'expression *Add*(*e*, *e'*) dénote le nombre *n+n'*

Compositionnalité : la valeur d'une expression ne dépend que des valeurs de ses sous-expressions, pas de détails de l'exécution

Exemple de non-compositionnalité : C

```
int x = 1;
int A() { x = x+1; return x; }
int B() { x = 2*x; return x; }
int F(x,y) { return x+y; }
main () { printf("%d", F(A(),B()); }
```

L'ordre d'évaluation des arguments est indéterminé

- A() exécuté avant B() => 2*(1+1) = 4
- B() exécuté avant A() => (2*1)+1 = 3

C n'est ni compositionnel ni déterministe

Agenda

1. Notions de base sur le calcul
2. Calculabilité : machines, langages et fonctions
3. Les grands résultats
4. Appel par noms, appel par valeurs
5. Introduction au λ-calcul

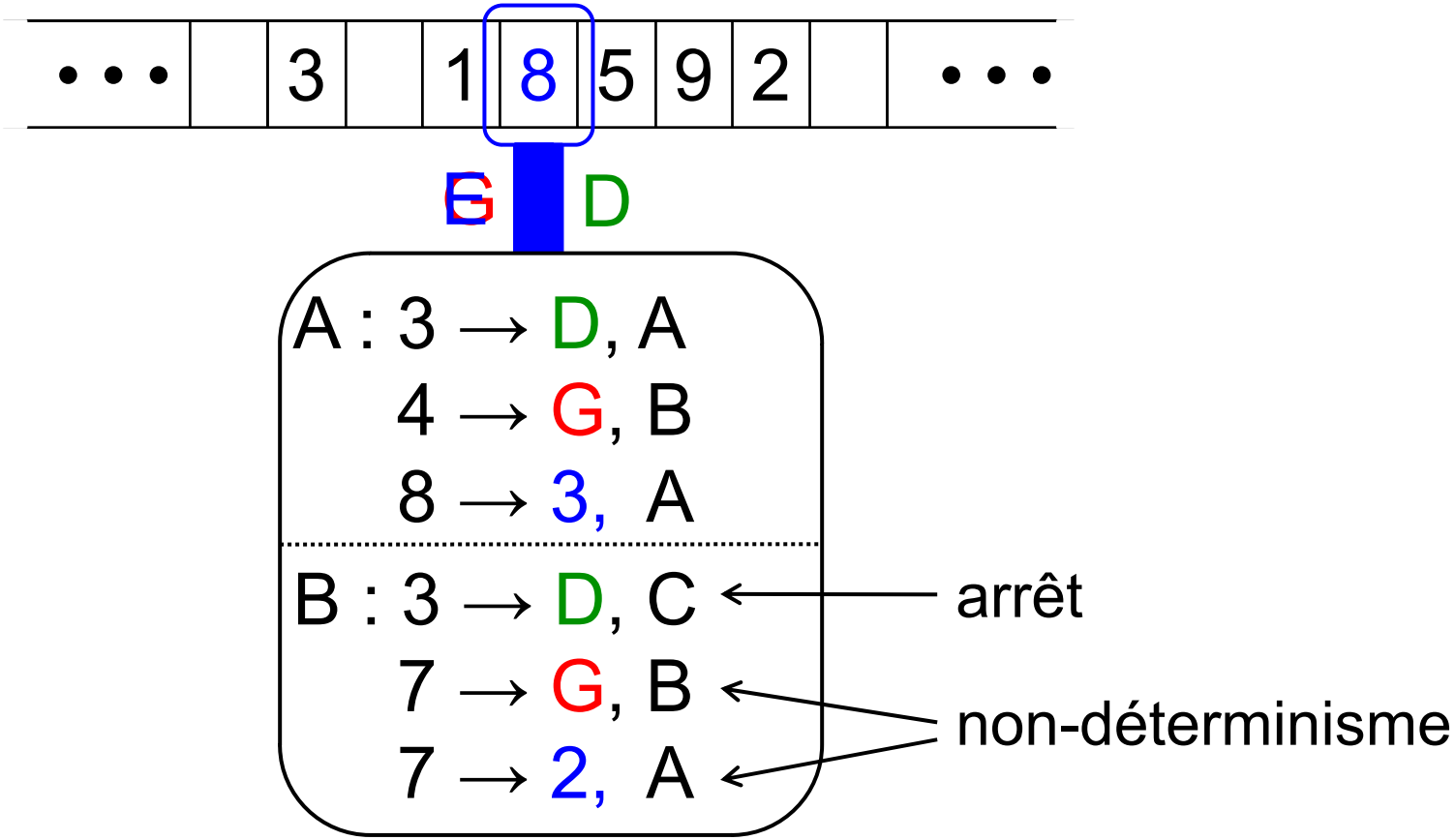
Comment définir la calculabilité ?

1. Par les machines
   machines de Turing, de von Neumann,
   automates cellulaires, pavages du plan, etc.
2. Par les langages et calculs algébriques
   définitions récursives de fonctions
   λ-calcul
3. Par les classes algébriques de fonctions
   fermetures par opérations appropriées

Toutes ces définitions sont équivalentes

Thèse de Church-Turing :
Toute nouvelle définition restera équivalente

La machine de Turing



La machine de Turing

- Symboles = lettres + blanc
- Etats = ensemble fini, dont un état initial
- Actions = Droite, Gauche, Ecrire(*s*)
- Table de transitions = { (symbole, état)→(action, état) }
  peut être non-déterministe : { (B,7)→(G,A), (B,7)→(E(2),A) }
- Sémantique opérationnelle
  - départ : état initial, bande initiale avec un ensemble finie de lettres, tête sur lettre la plus à gauche
  - appliquer les transitions une par une, avec choix arbitraire si plusieurs
  - arrêt : pas de transition pour (état courant, symbole courant)
- Sémantique dénotationnelle
  - bande = nombre en base suffisante
  - sens = fonction partielle $\mathrm{N} \rightarrow_p \mathrm{N}$, définie ssi arrêt

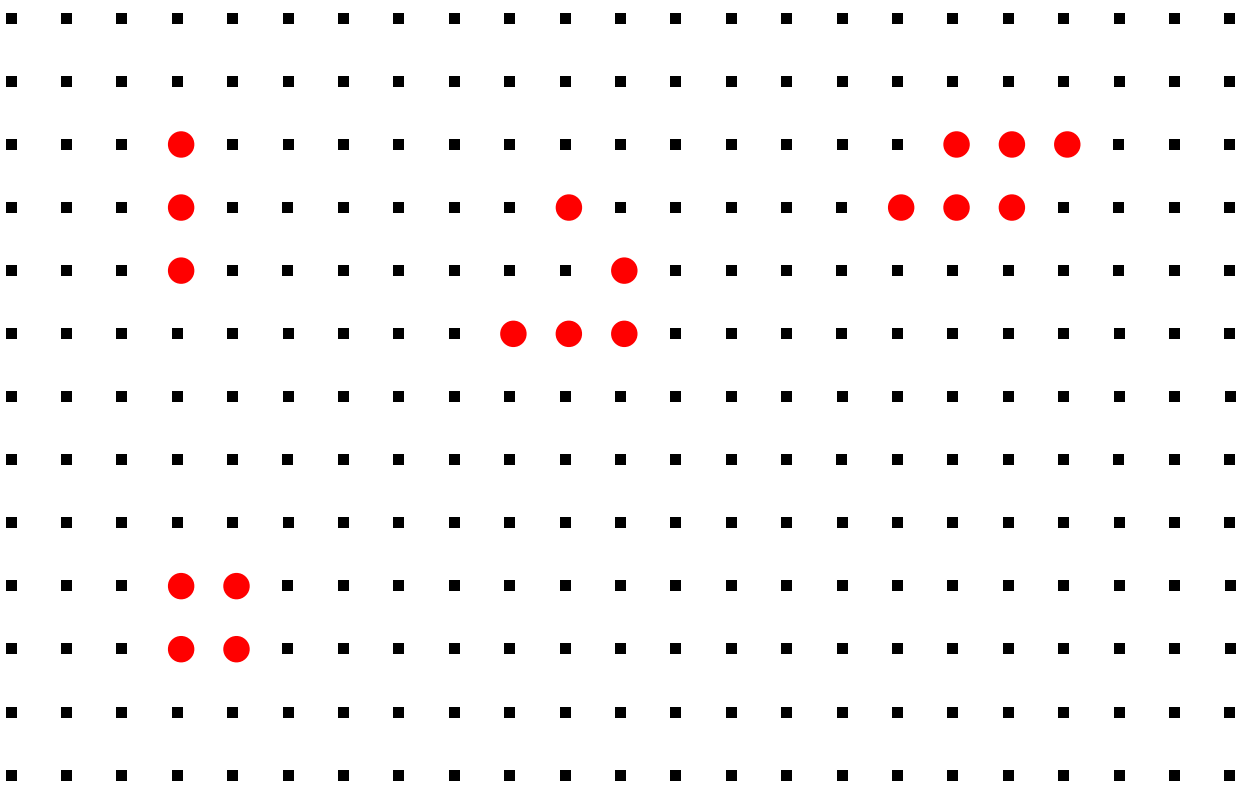
La machine de Turing est robuste

- En termes de fonctions calculables :
  - mettre plusieurs bandes ne change rien
  - déterminisme ou non déterminisme ne change rien
  - avoir deux lettres suffit, s'il y a assez d'états
  - avoir deux états suffit, s'il y a assez de lettres
- Mais les temps de calcul peuvent être très différents
- Pour étudier la plupart des propriétés, il suffit de s'intéresser à l'arrêt

Se prouve par simulations mutuelles
(techniques et ennuyeuses)

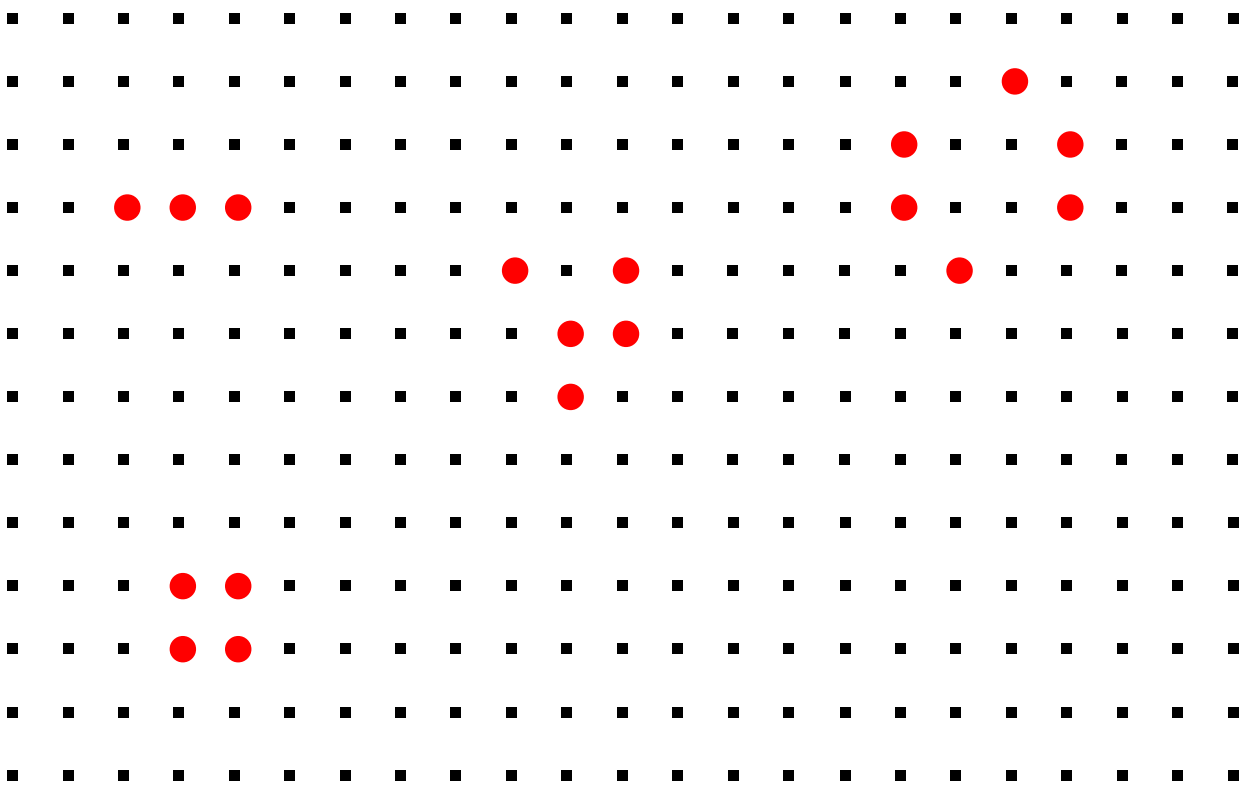
Automates cellulaires : le jeu de la vie

(J. H. Conway, 1970)

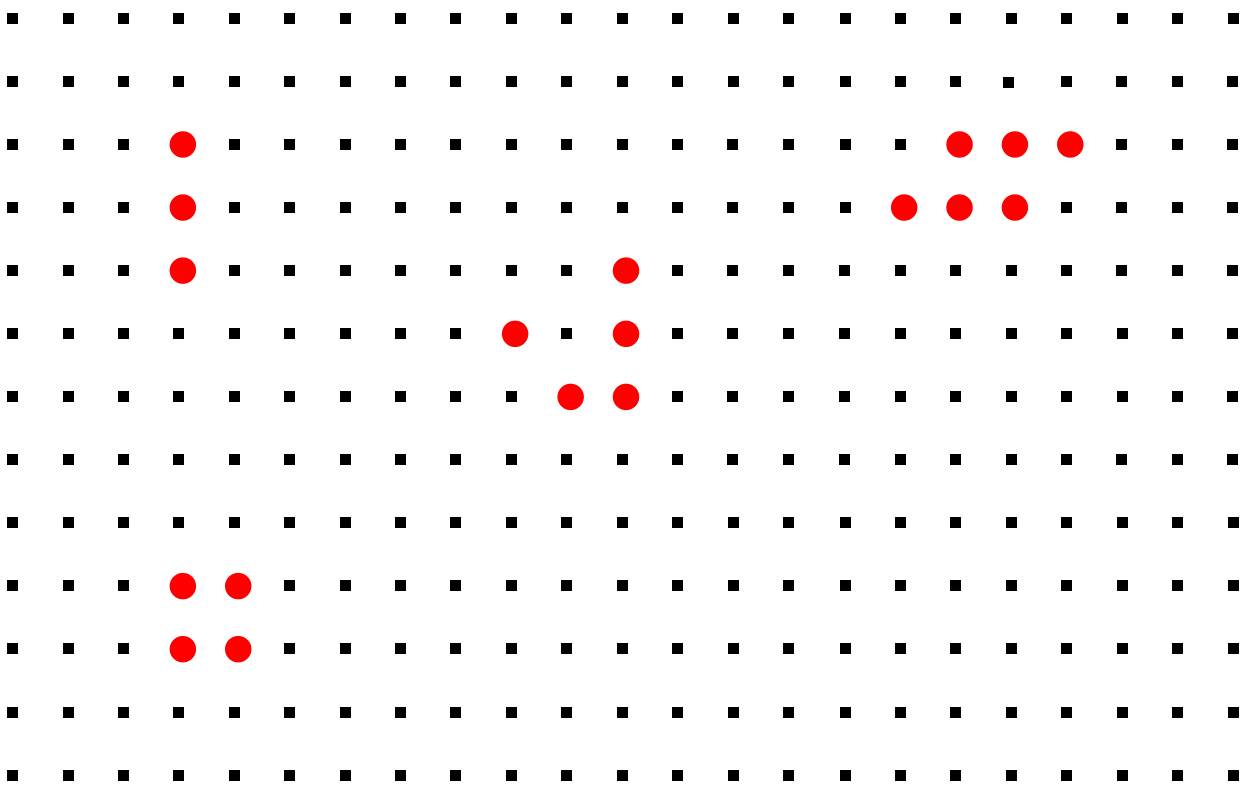


Un point devient vivant s'il avait 2 ou 3 voisins vivants
Un point vivant reste vivant s'il a exactement 2 voisins

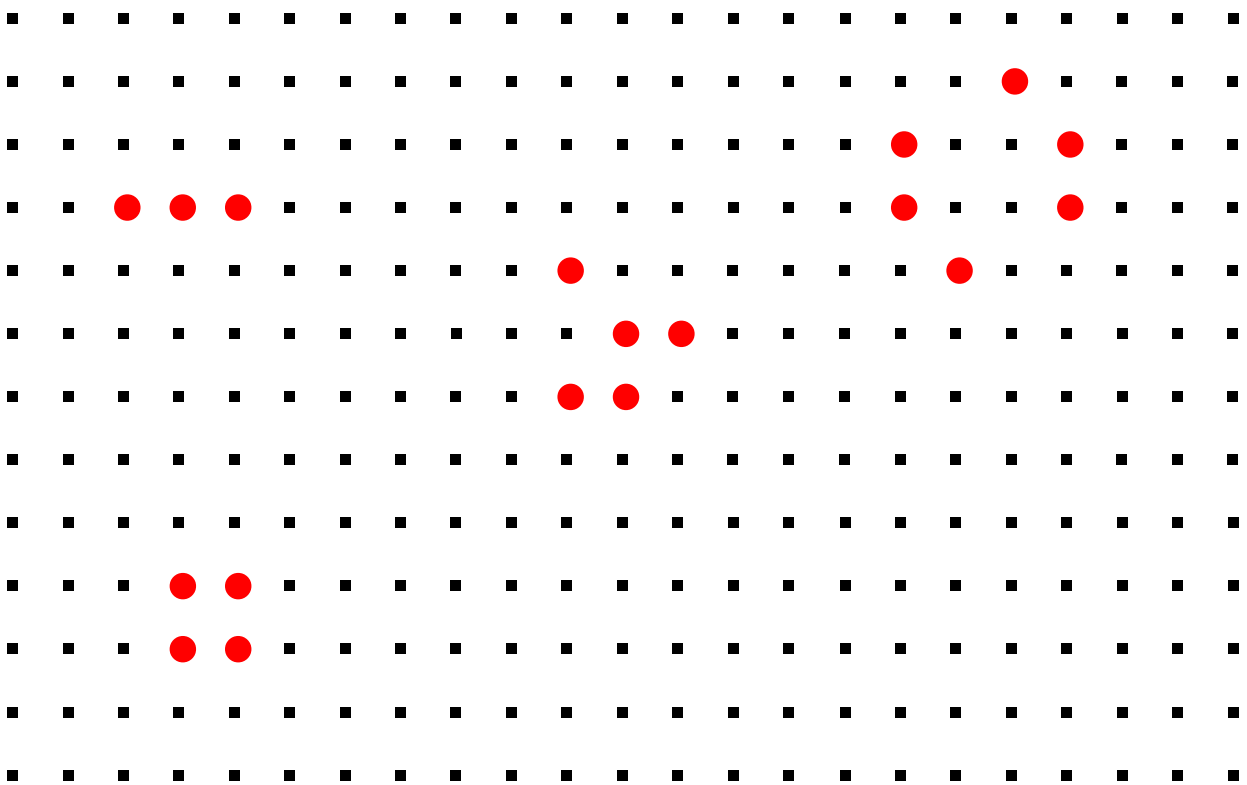
Automates cellulaires : le jeu de la vie



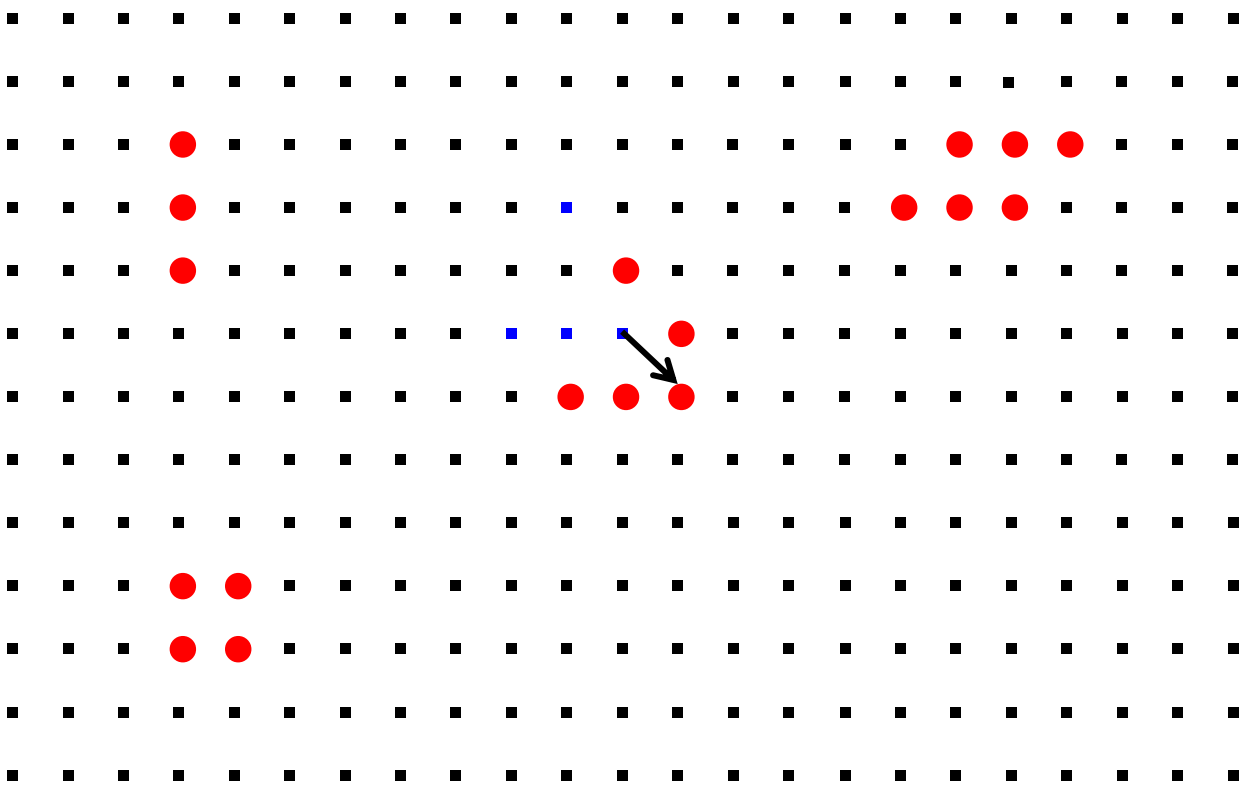
Automates cellulaires : le jeu de la vie



Automates cellulaires : le jeu de la vie



Automates cellulaires : le jeu de la vie



Même puissance que les machines de Turing !

Un langage : la récursion générale

- Add(m, n) = si m=0 alors n
  sinon si n=0 alors m
  sinon (Add(m–1, n–1) +1)+ 1
- Mult(m, n) = si m=0 alors 0
  sinon Add(Mult(m–1, n), n)
- Fact(m) = si m=1 alors 1
  sinon Mult(m, Fact(m–1))

- Attention, fonctions partielles !
  - Fact(0) provoque le calcul de 0-1, arrêt, résultat non défini
  - Pour F(m) = F(m), le calcul de F(m) boucle pour tout m

Fonction récursives

- Pour un prédicat $P : \mathrm{N} \rightarrow_p \{0,1\}$, soit $\mu m.P(m)$ le plus petit m tel que $P(m)$ est vrai s'il existe, ou l'indéfini sinon
- La classe R des fonctions récursives partielles est la plus petite classe de fonctions partielles $f : \mathrm{N}^m \rightarrow_p \mathrm{N}$ telle que
  - les fonctions +1, -1, et 0? sont dans R
  - les projections $p_i : (m_1, m_2, ..., m_n) \rightarrow m_i$ sont dans R
  - toute composition de fonctions de R est dans R
  - si $f : \mathrm{N}^2 \rightarrow_p \mathrm{N}$ est dans R alors la fonction g définie par $g(n) = \mu m.f(m,n)=0$ est aussi dans R

Ensembles récursifs et récursivement énumérables

- Une fonction *f* : $\mathrm{N}^m \rightarrow_p \mathrm{N}$ est récursive totale si elle est récursive et partout définie
- Un ensemble E $\subset \mathrm{N}$ est récursif si son prédicat d'appartenance *E* : $\mathrm{N} \rightarrow \{0,1\}$ est récursif total
- Un ensemble E $\subset \mathrm{N}$ est récursivement énumérable si son prédicat d'appartenance *E* : $\mathrm{N} \rightarrow_p \{0,1\}$ est récursif partiel

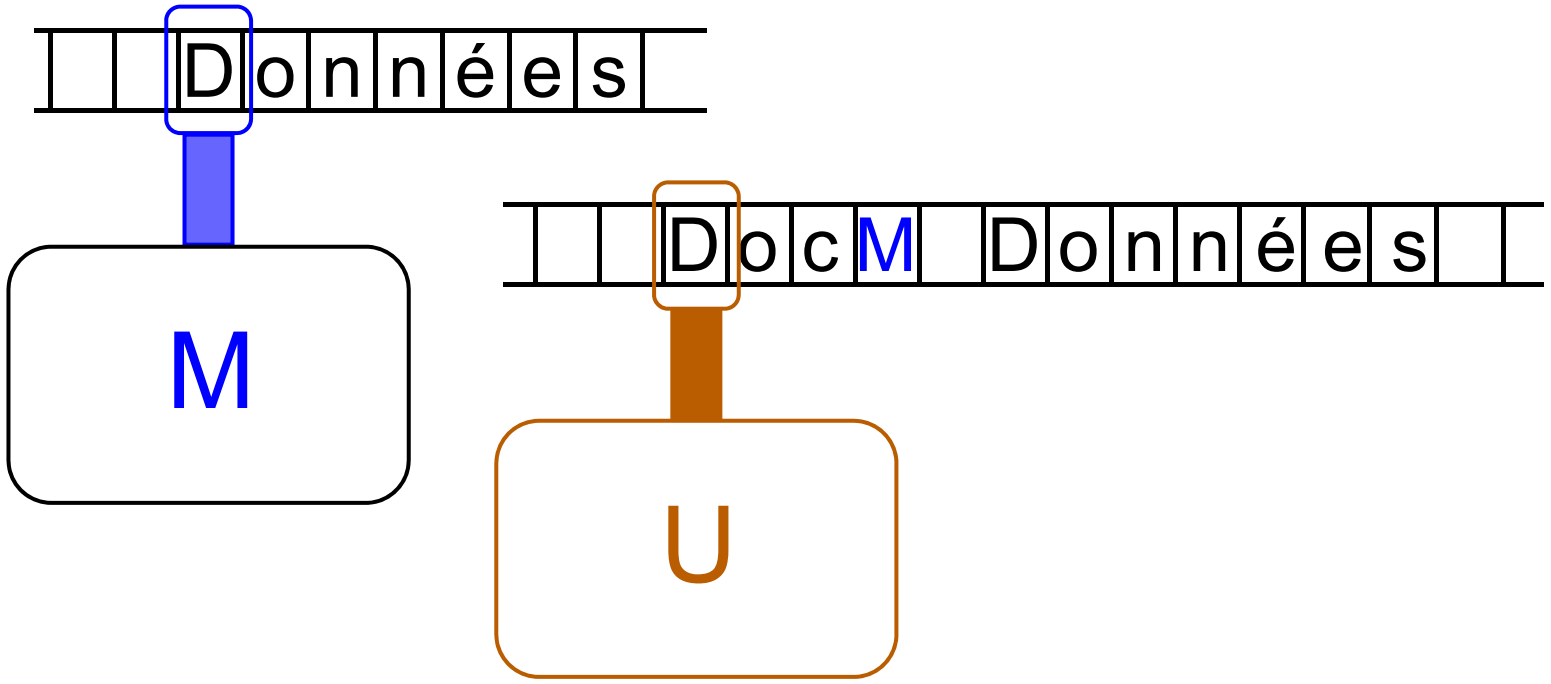
Théorème : E est récursivement énumérable s'il est l'image de N par une fonction récursive partielle ou totale.

Agenda

1. Notions de base sur le calcul
2. Calculabilité : machines, langages et fonctions
3. Les grands résultats
4. Appel par noms, appel par valeurs
5. Introduction au λ-calcul

Machine universelle

Théorème : il existe une machine universelle U qui permet de simuler toute machine sur toute donnée



Le programme enregistré est la clef de l'informatique

Fonction universelle

Théorème :

Il existe une fonction récursive partielle $\Phi : \mathrm{N}^2 \to \mathrm{N}$ telle que

$$\forall f : \mathrm{N} \to \mathrm{N}.\ \exists m \in \mathrm{N}.\ \forall n \in \mathrm{N}.\ \Phi(m, n) = \mathrm{f}(n)$$

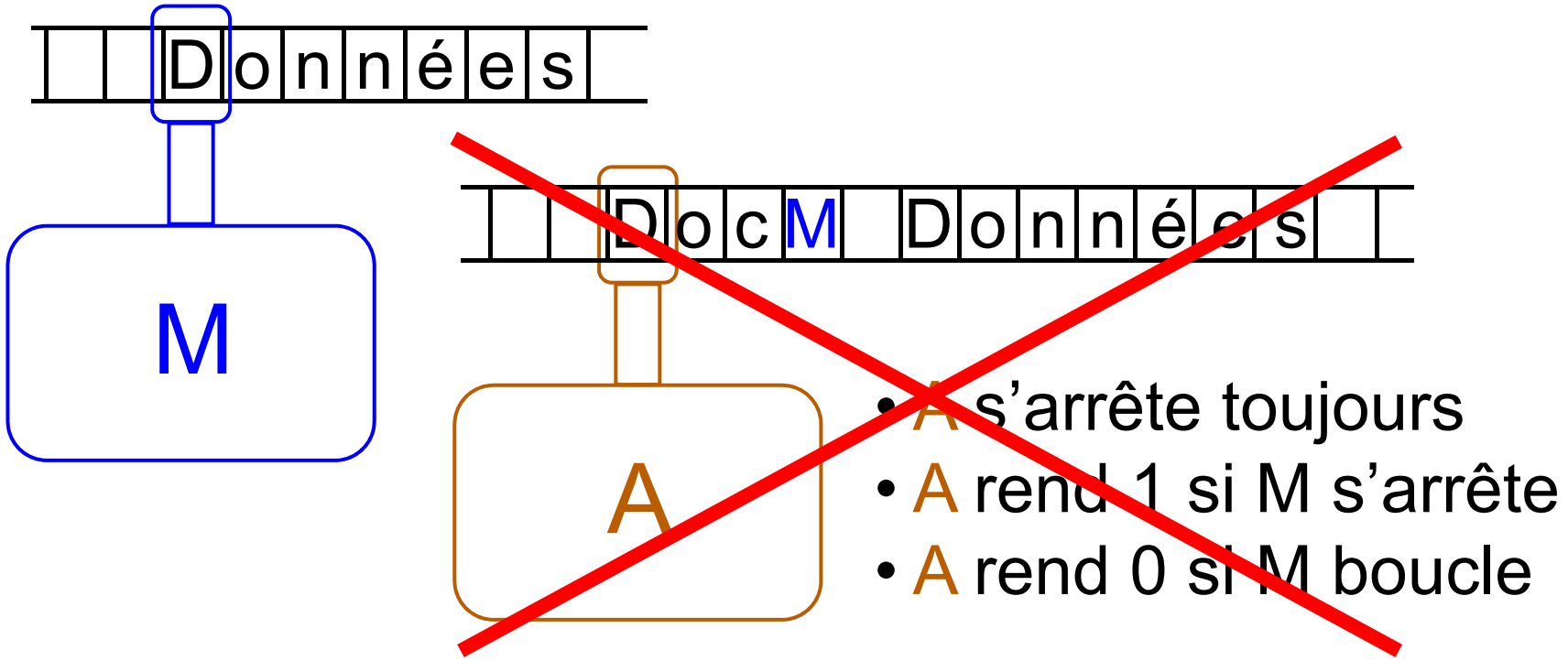
Alors Φ est appelée une fonction universelle,

m est appelé l'index de f pour Φ, et on note

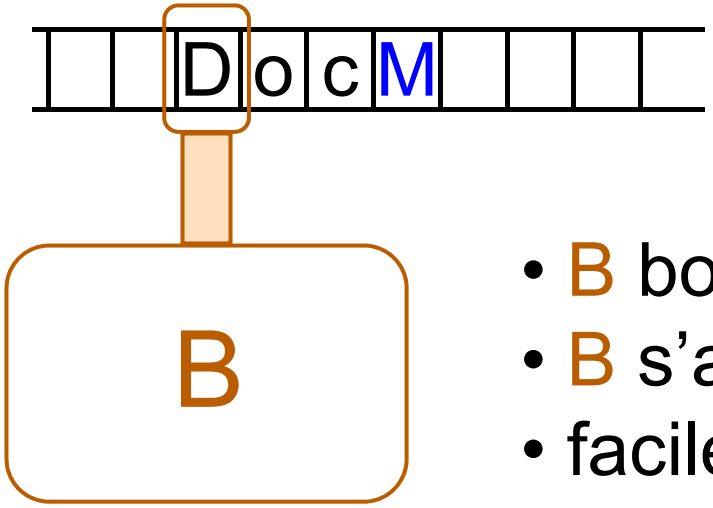
$$\Phi_m(n) = f(n)$$

Indécidabilité de l'arrêt

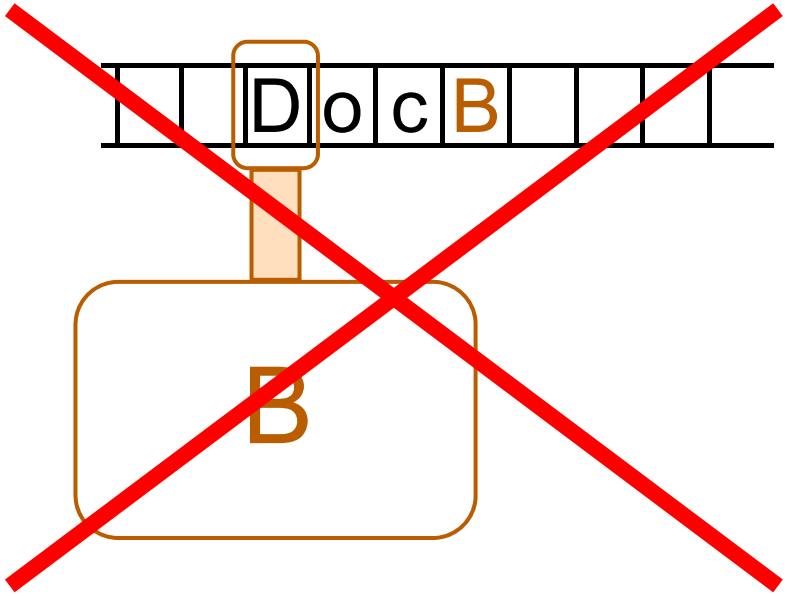
Théorème : il n'existe pas de machine testant si une machine M s'arrête sur une donnée D



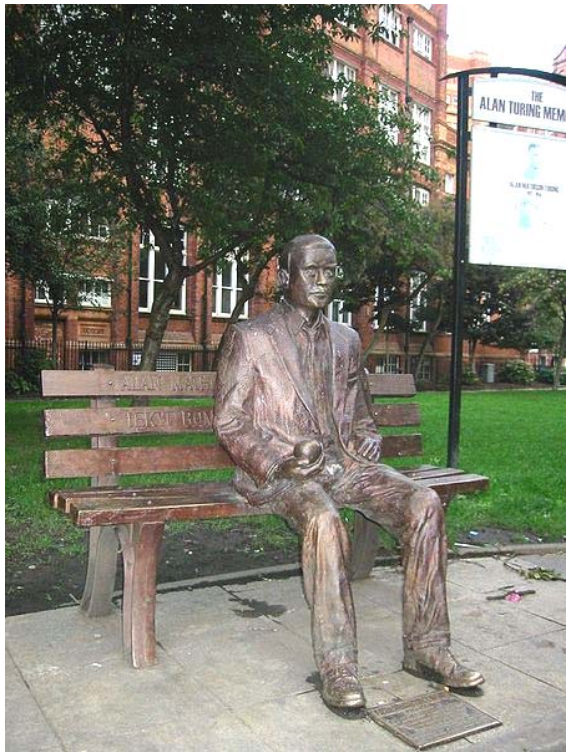
Pas de débogueur universel !



- B boucle si M s'arrête sur DocM
- B s'arrête si M boucle sur DocM
- facile en modifiant A !



- ~~B boucle sur DocB ssi B s'arrête sur DocB~~



Alan Turing (1912-1954)

1936 : grands théorèmes
1940 : casse le code allemand Enigma
1952 : condamné pour homosexualité
castré chimiquement
1954 : suicide (?)
2009 : excuses de Gordon Brown

A lui qui aimait tant l'argument diagonal, donnons le prix Turing à titre posthume

Version fonctions récursives

Théorème :

l'ensemble des *m* tels que Φ_m est une fonction totale n'est pas récursif

Théorème de Rice :
Toute propriété non-triviale de la classe des fonctions récursives est indécidable

Exemples :
arrêt de Φ_m , égalité $\Phi_m = \Phi_n$

Prouver la terminaison d'un programme peut être très dur, voire équivalent à un problème ouvert majeur

f(m) = si m=1 alors 1
sinon si m pair alors f(m/2)
sinon f(3*m+1)

Goldbach(m) =
si m impair ou ∃ p,q < m premiers t.q. m=p+q
alors 1 sinon Golbach(m)

Il n'existe pas de langage de programmation permettant de définir toutes les fonctions totales et rien que les fonctions totales

Récursion primitive

Add(0, n) = n

Add(m+1, n) = Add(m,n)+1

Mult(0, n) = 0

Mult(m+1, n) = Add(n, Mult (m, n))

Fact(0) = 1

Fact(m+1) = Mult(m+1, Fact(m))

Les fonctions définies par récursion primitive sont toujours totales

La hiérarchie primitive récursive

- Add(m, n) : 1 récursion depuis +1
- Mult(m, n): 1 récursion depuis Add
  → 2 récursions emboîtées
- Exp(m, n) = n^m : 1 récursion depuis Mult
  → 3 récursions emboîtées
- SupExp(m, n) = $n^{n^{n^{\cdot^{\cdot^{\cdot}}}}}$ (m fois)
  → 4 récursions emboîtées
- …

  → p récursions emboîtées

  plus d'expression algébrique !

Théorème : la hiérarchie primitive est infinie et stricte

Idée de preuve: à chaque niveau, les fonctions croissent strictement plus vite qu'au niveau précédent

- Ack(m, n) = si m=0 alors n+1
  sinon si m=1 alors Ack(m-1, 1)
  sinon Ack(m-1, Ack(m, n-1))
- $Ack(4,2) \cong 10^{19829}$

La fonction d'Ackermann n'est pas primitive récursive et croît plus vite que toutes les primitives récursives (m diagonalise le nombre de récursions)

Agenda

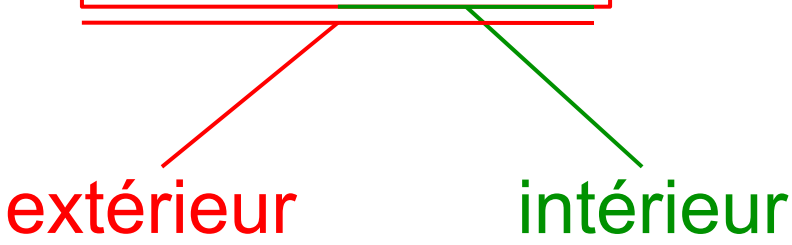
1. Notions de base sur le calcul
2. Calculabilité : machines, langages et fonctions
3. Les grands résultats
4. Appel par noms, appel par valeurs
5. Introduction au λ-calcul

Appel par nom, appel par valeur

Fact(3) = si 3=1 alors 1 sinon Mult(3, Fact(3–1))

= si faux alors 1 sinon Mult(3, Fact(2))

= Mult(3, Fact(2))



Deux redex (*reducible expression*)

Appel par valeur : l'intérieur d'abord

= Mult(3, Fact(2))

= Mult(3, si 2=1 alors 1 sinon Mult(2, Fact(2–1)))

= Mult(3, Mult(2, Fact(1)))

= Mult(3, Mult(2, si 1=1 alors 1 sinon Mult(1, Fact(1–1)))

= Mult(3, Mult(2, 1))

= …

= 6

Appel par nom : l'extérieur d'abord

= Mult(3, Fact(2))

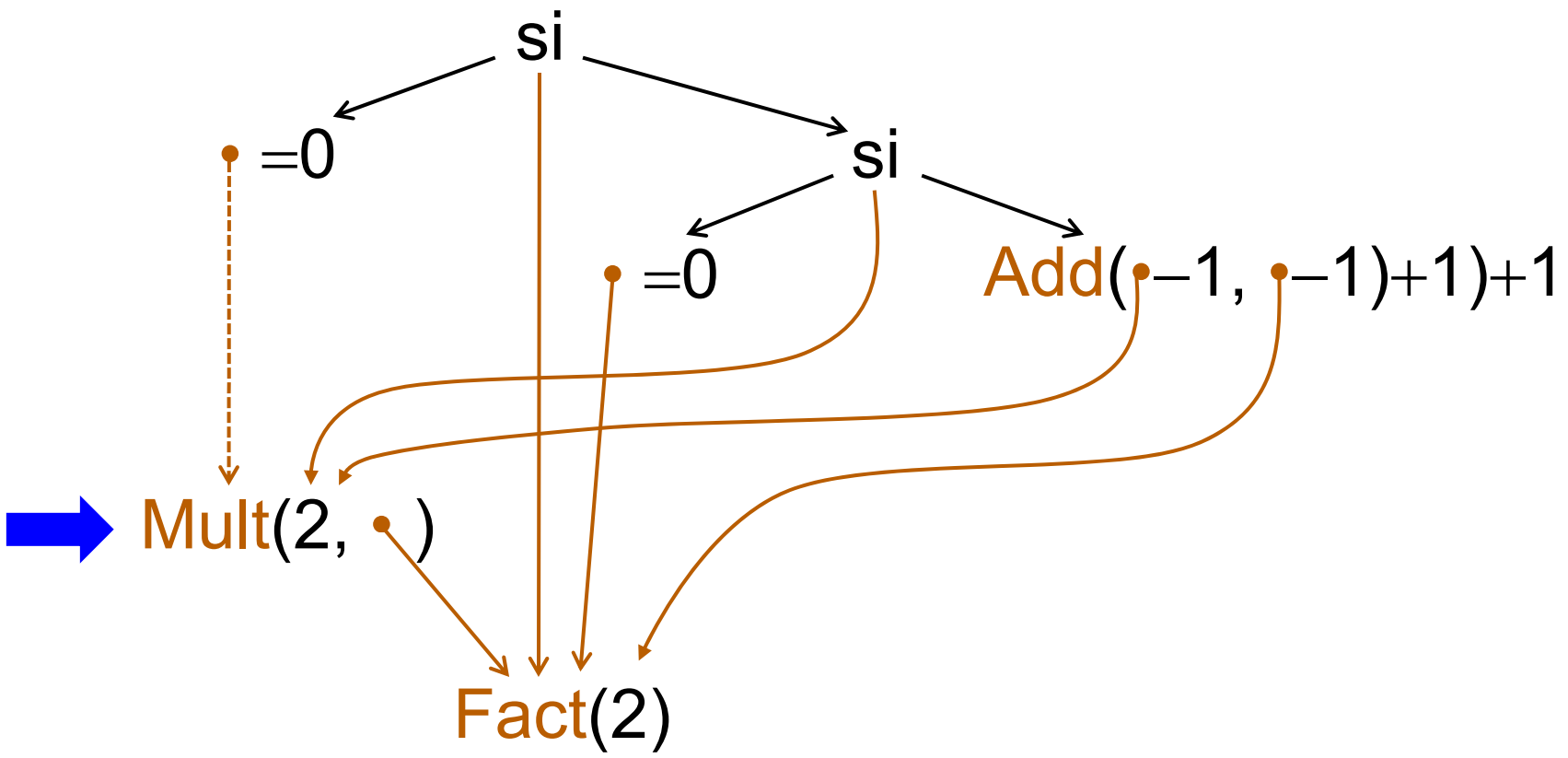
= si 3=0 alors 0
sinon si 3=1 alors Fact(2)
sinon Add(Mult(3–1, Fact(2)), Fact(2))

= Add(Mult(2, Fact(2)), Fact(2))

= si Mult(2, Fact(2))=0 alors Fact(2)
sinon si Fact(2)=0 alors Mult(m–1, Fact(2))
sinon (Add(Mult(2, Fact(2))–1, Fact(2)–1)+1)+1

Il y a de quoi grimper aux arbres !

si Mult(2, Fact(2))=0 alors Fact(2)
sinon si Fact(2)=0 alors Mult(2, Fact(2))
sinon (Add(Mult(2, Fact(2))–1, Fact(2)–1)+1)+1



Calculer sur les arbres
en partageant les sous-arbres communs

Quel appel est le meilleur ?

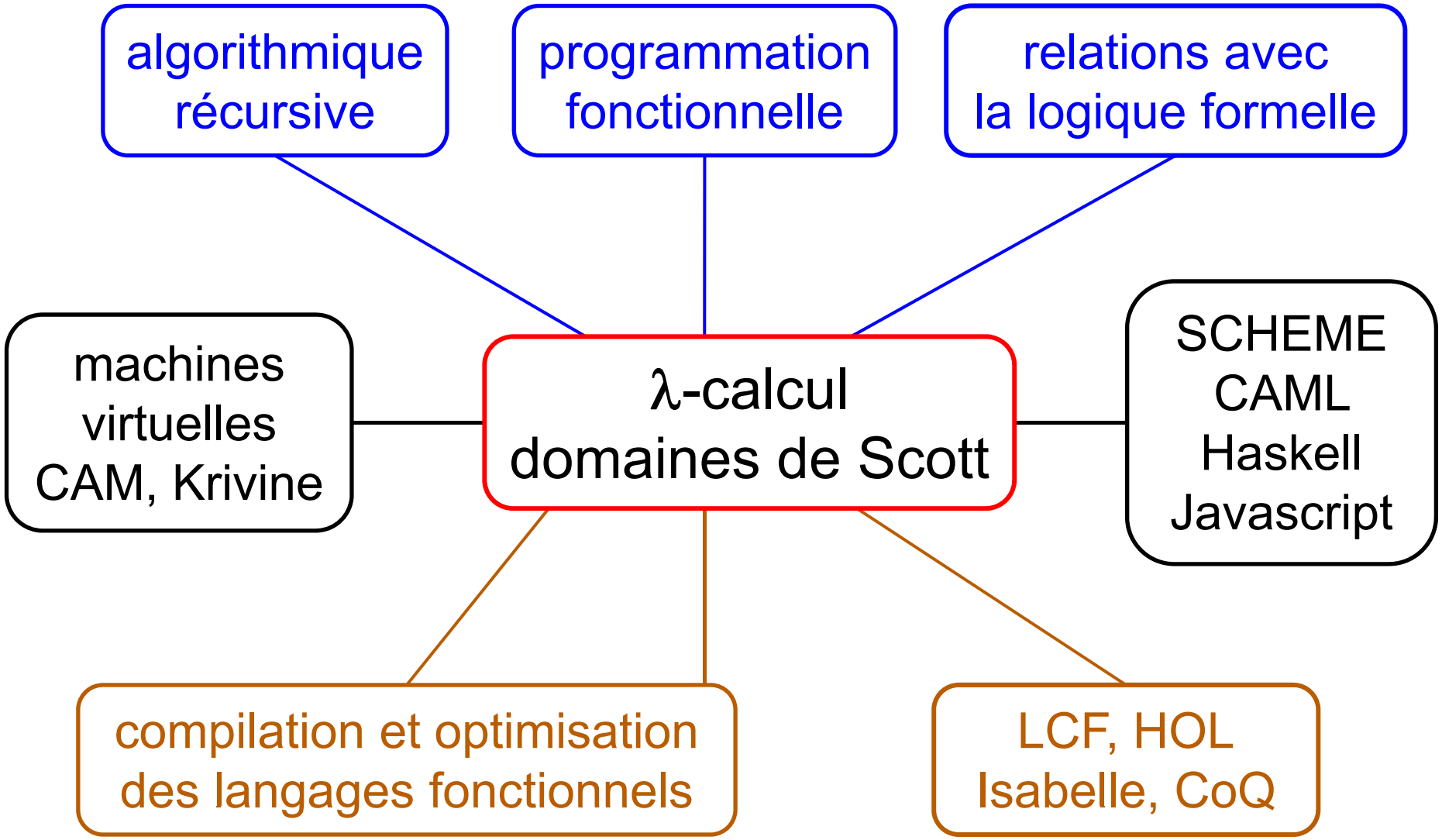
1. Si les deux donnent un résultat, c'est le même
2. L'appel par valeur est généralement plus efficace
3. Mais il ne donne pas toujours le résultat !
   F(m, n) = si m=0 alors 0 sinon F(m-1, F(m,n))
   ⇒ F(1, 0) donne 0 par nom, mais boucle par valeur
4. Les stratégies mixtes sont possibles, mais pas utilisées en pratique

Appel par valeur: Pascal, C, Ada, LISP, ML
Appel par nom : Haskell

Agenda

1. Notions de base sur le calcul
2. Calculabilité : machines, langages et fonctions
3. Les grands résultats
4. Appel par noms, appel par valeurs
5. Introduction au λ-calcul

La carte du λ-calcul



Le λ-calcul, joyau des modèles (A. Church, 1936)

- Un calcul hyper-simple des fonctions
- Equivalent à la machine de Turing
- Réunissant tous les avantages
  - sémantiques opérationnelle / dénotationnelle riches
  - grande robustesse aux changements
  - applicabilité directe à la programmation (CAML)
  - forte relation avec la déduction logique (Automath, SystemF, CoQ)
    ⇒ utilisation directe en vérification formelle (maths, programmes, circuits)
- … mais avec un gros inconvénient
  - intrinsèquement limité au calcul séquentiel

Idée de base

Donner un vrai statut à la notation des fonctions

- Notation classique des fonctions

$$f = [\, x \rightarrow x+1 \,]$$

$$g = [\, y \rightarrow y^2 \,]$$

$$g \circ f = [\, z \rightarrow g(f(z)) \,] = [\, z \rightarrow (z+1)^2 \,]$$

- λ-notation

$$f = \lambda x.\, x+1 \qquad f(2) = (\lambda x.\, x+1)\,(2) \rightarrow 2+1 \rightarrow 3$$

$$g = \lambda y.\, y^2 \qquad g(3) = (\lambda y.\, y^2)\,(3) \rightarrow 3^2 \rightarrow 9$$

$f = \lambda x.\, x+1$

$g = \lambda y.\, y^2$

$\circ = \lambda u.\, \lambda v.\, \lambda z.\, u(v(z))$

$\circ g\, f = (\lambda u.\, \lambda v.\, \lambda z.\, u(v(z)))\ (\lambda y.\, y^2)\ (\lambda x.\, x+1)$

$= (\lambda u.\, \lambda v.\, \lambda z.\, u(v(z)))\ (\lambda y.\, y^2)\ (\lambda x.\, x+1)$

$= \lambda z.\, (\lambda y.\, y^2)\ ((\lambda x.\, x+1)\ (z))$

$= \lambda z.\, (\lambda y.\, y^2)\ ((\lambda x.\, x+1)\ (z))$

$= \lambda z.\, (\lambda y.\, y^2)\ (z+1)$

$= \lambda z.\, (\lambda y.\, y^2)\ (z+1)$

$= \lambda z.\, (z+1)^2$

$g \circ f : [\ z \rightarrow (z+1)^2\ ]$

Le λ-calcul pur

- Variables x, y, z, +,…
- λ-termes M, N, P, …
  - variable x
  - abstraction λx.M : fonction de x définie par le corps M

    une abstraction lie sa variable
  - application (MN)

    association à gauche : MNPQ abrège (((MN)P)Q)

    Au lieu de 1+2, on écrit ((+ 1) 2), abrégé en + 1 2
- variable liée, variable libre
  - le nom d'une variable liée est indifférent, pas celui d'une libre

$$\lambda x.xy = \lambda z.zy \neq \lambda x.xz \neq \lambda x.xx$$

$$\int_0^y f(x,y)\,dx = \int_0^y f(z,y)\,dz \neq \int_0^z f(x,z)\,dx \neq \int_0^y f(y,y)\,dy$$

Rendre précises des notions floues

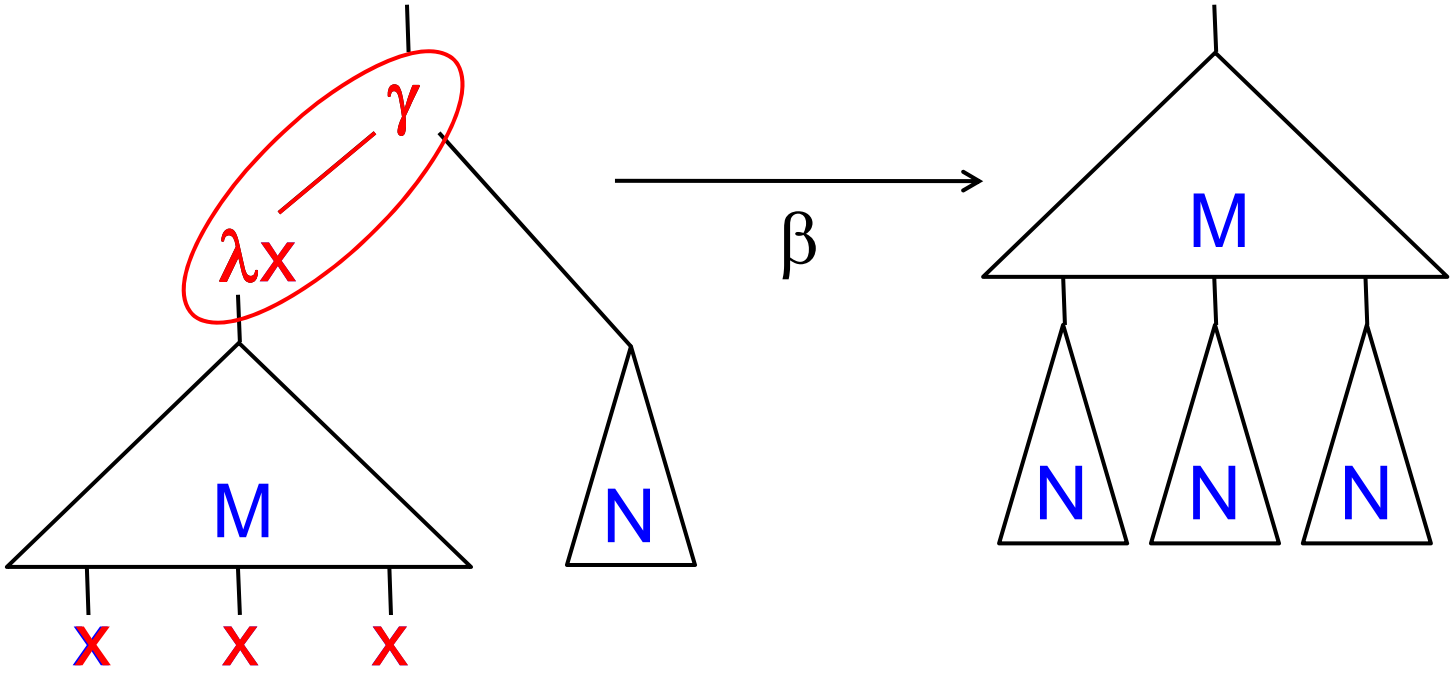
- La fonction x + y
  λx.λy. + x y
  λy.λx. + x y
- La fonction x + y « à y constant »
  λx. + x y (en laissant y libre)
- « Posons y=1 dans x + y »
  (λy. + x y) 1
- « Posons y=1 dans f(x) où f est l'addition à y constant »
  let y=1 in let f =(λz.+ z y)) in f x (ML / CAML)
  (λy. (λf. f x) (λz.+ z y)) 1

Niveau de précision indispensable
pour tout calcul formel

La β-conversion

- Redex : sous terme de la forme (λx.M) N
  application d'une fonction à son argument
- β-conversion : substitution de x par N dans M

$$(\lambda x.M)\ N \rightarrow M\,[N/x]$$

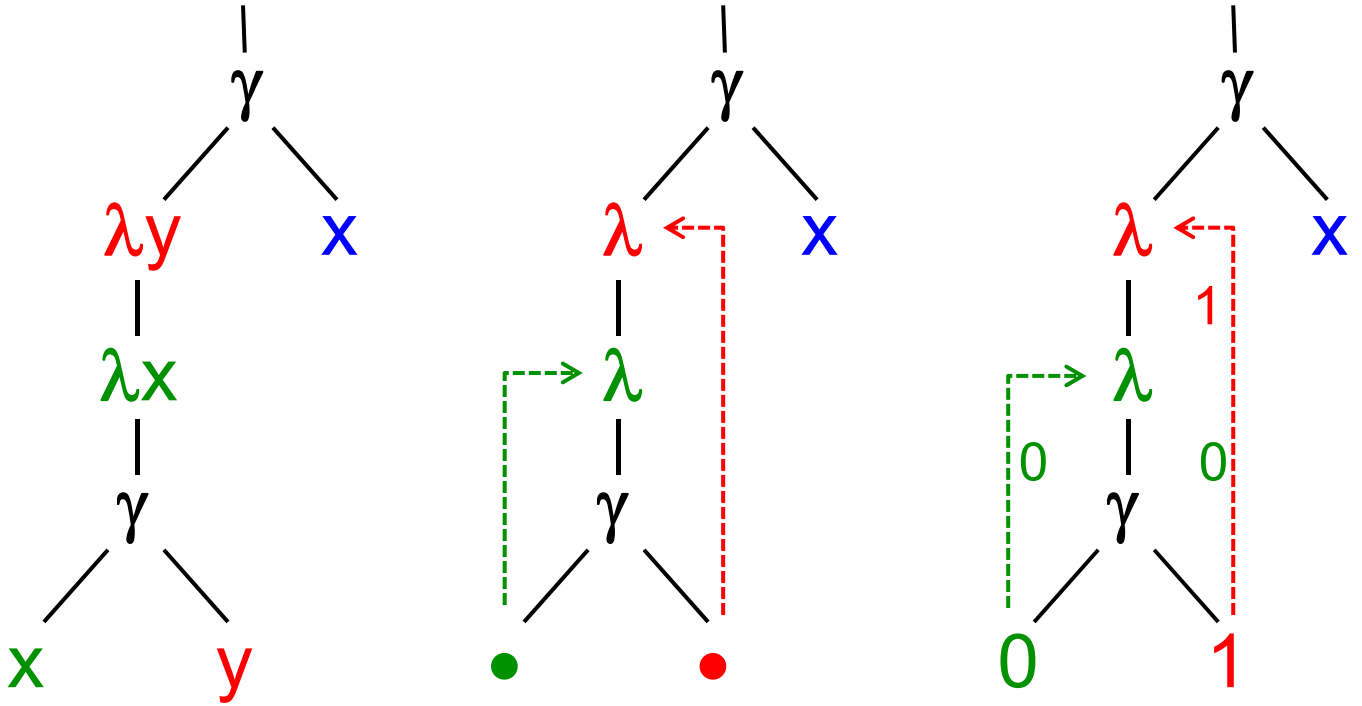


α-conversion

- Jamais de capture d'une variable libre par une liée !

~~$(\lambda y.(\lambda x.xy))x \rightarrow (\lambda x.xx)$~~

$(\lambda y.(\lambda x.xy))x \xrightarrow{\alpha} (\lambda y.(\lambda z.zy))x \rightarrow (\lambda z.zx))$



Bourbaki de Bruijn

Curryfication des fonctions n-aires

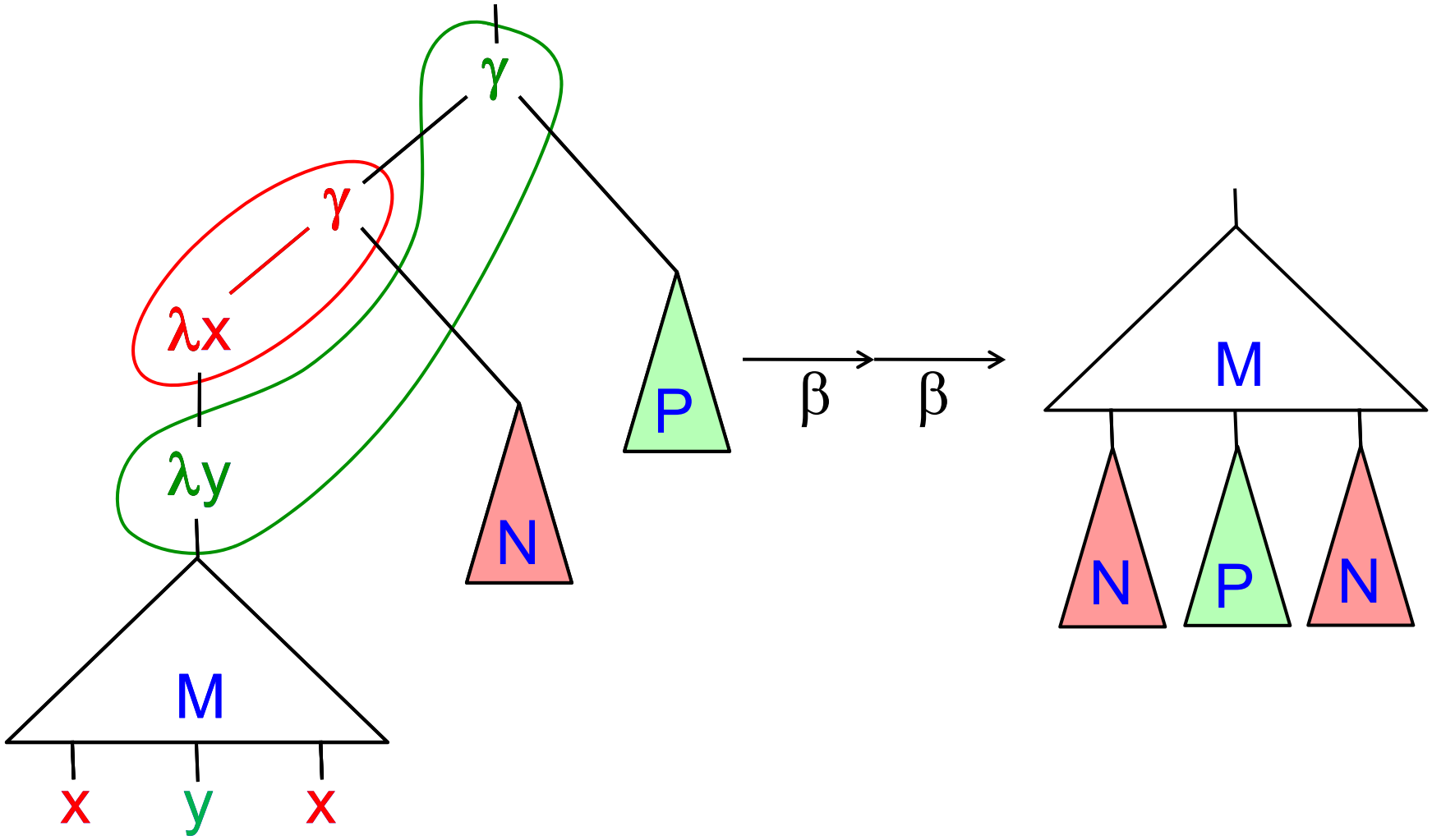
- On passe les arguments un par un

$$(\lambda x.\lambda y.M)\,N\,P \rightarrow (\lambda y.M\,[N/x])\,P$$
$$\rightarrow M\,[N/x]\,[P/y]$$
$$(\lambda x.\lambda y.M)\,N\,P \rightarrow M\,[N/x]\,[P/y]$$

(en supposant $x \neq y$, avec α-conversion si nécessaire)

- L'analogue ensembliste est l'isomorphisme

$$((D \times E) \rightarrow F) \Leftrightarrow (D \rightarrow (E \rightarrow F))$$



Pas d' interférence entre N/x et P/y

Premiers exemples

- $I = \lambda x.x$ identité : $I\,M \rightarrow M$
- $\Delta = \lambda x.xx$ diagonale : $\Delta\,M \rightarrow M\,M$

  bouclage : $\Delta\Delta = (\lambda x.xx)\,(\lambda x.xx) \rightarrow \Delta\Delta \rightarrow \Delta\Delta \rightarrow \Delta\Delta \rightarrow \ldots$
- $\underline{\text{vrai}} = \lambda x.\,\lambda y.\,x$

  $\underline{\text{faux}} = \lambda x.\,\lambda y.\,y$

  $\underline{\text{cond}} = \lambda c.\,\lambda x.\,\lambda y.\,c\,x\,y$
- $\underline{\text{cond}}\ \underline{\text{vrai}}\ u\ v = (\lambda c.\,\lambda x.\,\lambda y.\,c\,x\,y)\,(\lambda x.\,\lambda y.\,x)\,u\,v$

  $= (\lambda c.\,\lambda x.\,\lambda y.\,c\,x\,y)\,(\lambda x.\lambda y.\,x)\,u\,v$

  $= (\lambda x.\lambda y.x)\,u\,v$

  $= (\lambda x.\lambda y.x)\,u\,v$

  $= u$

Le codage des entiers

Mais ce calcul est vide ! Où sont donc les entiers, la récursion, etc ?

$$\underline{0} = \lambda f.\, \lambda x.\, x$$

$$\underline{1} = \lambda f.\, \lambda x.\, fx$$

$$\underline{2} = \lambda f.\, \lambda x.\, f(fx)$$

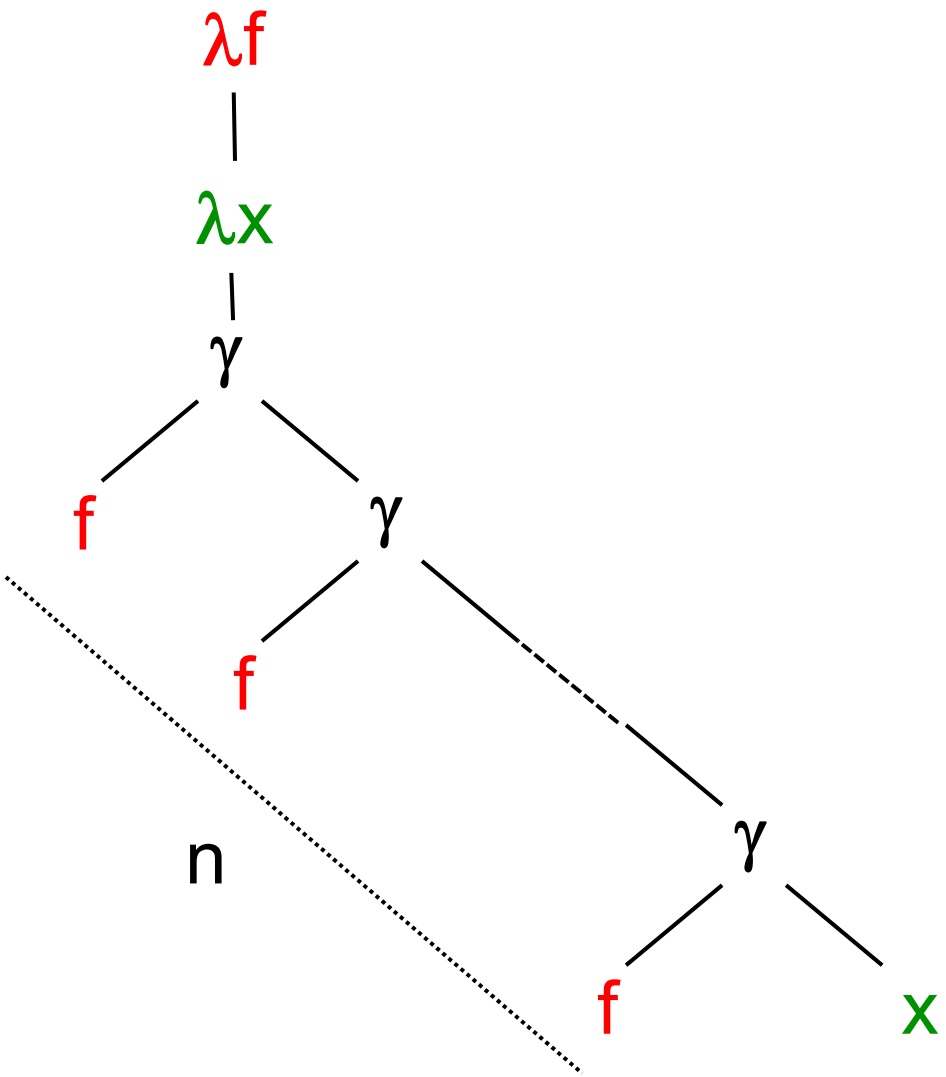
...

$$\underline{n} = \lambda f.\, \lambda x.\, f^n(x) = f(f(\ldots f(x)\ldots))$$

...

Le nombre n devient l'algorithme
qui applique n fois une fonction f à son argument x

$$\underline{0}\ f\ x \rightarrow x \qquad \underline{n+1}\ f\ x \rightarrow f\ (\underline{n}\ f\ x)$$


λf
λx
γ
f
γ
f
n
γ
f
x

Les fonctions entières de base

$\underline{+1} = \lambda n.\, \lambda f.\, \lambda x.f(nfx)$

$\underline{-1} = \lambda n.\, \lambda f.\, \lambda x.\, n\,(\lambda g.\, \lambda h.h(fg))\,(\lambda u.x)\,(\lambda v.v)$

$\underline{0?} = \lambda n.\, n\,(\lambda x.\underline{vrai})\,\underline{faux}$

La fonction successeur

$$\underline{+1} = \lambda n.\lambda f.\lambda x.\ f\,(n\,f\,x)$$

$$\underline{+1}\ \underline{0} = (\lambda n.\lambda f.\lambda x.\ f\,(n\,f\,x))\ (\lambda f.\lambda x.x)$$

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

$$= \lambda f.\lambda x.\ f\,x$$

$$= \underline{1}$$

cet f disparaît, car
f est absent du corps
x de la fonction $(\lambda f.\lambda x.\ x)$

Pour n > 0

$$+1\ n = (\lambda n.\lambda f.\lambda x.\ f\ (n\ f\ x))\ (\lambda f.\lambda x.\ f^n(x))$$

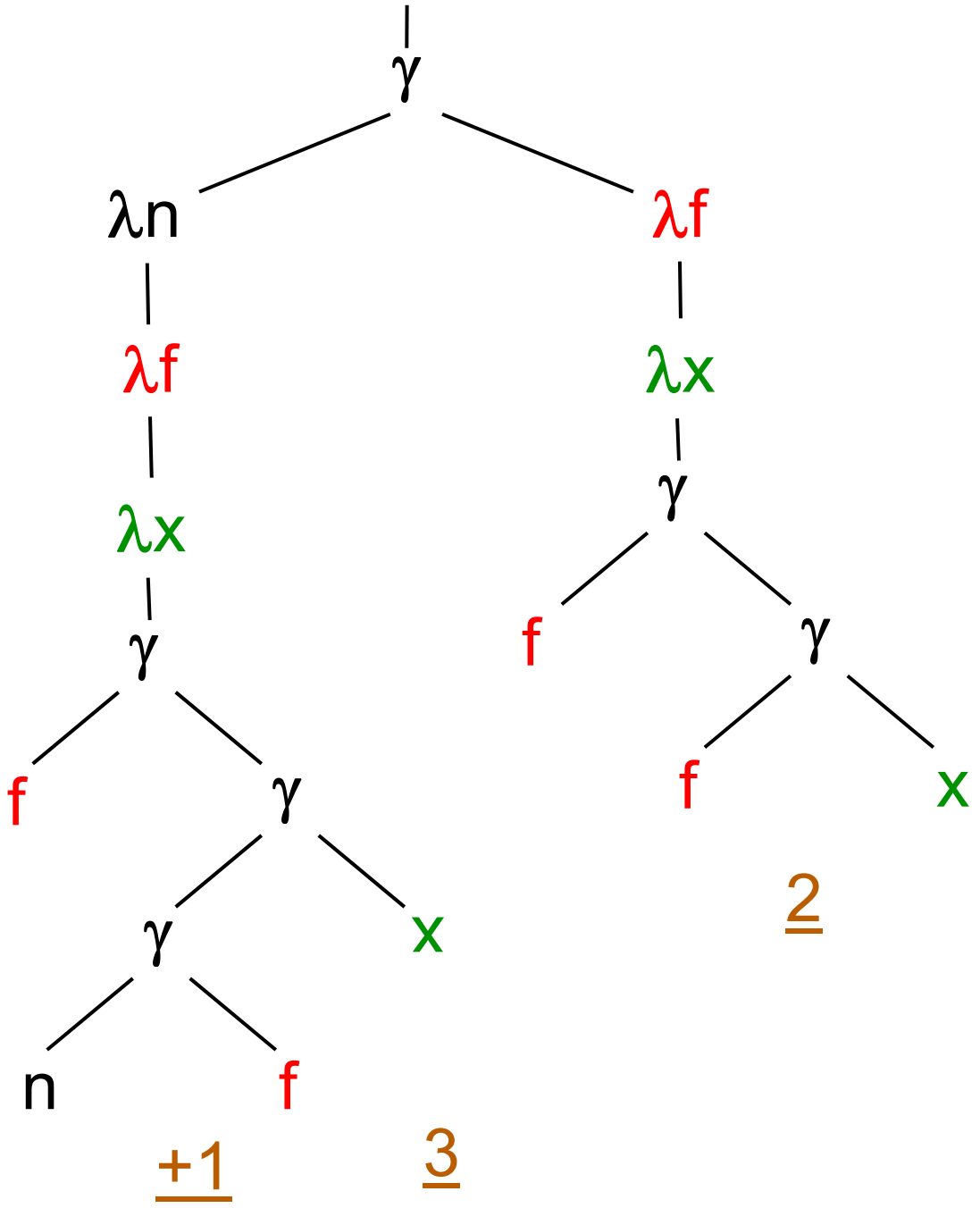
$$= \lambda f.\lambda x.\ f\ ((\lambda f.\lambda x.\ f^n(x))\ f\ x)$$

$$= \lambda f.\lambda x.\ f(\ f^n(x))$$

$$= \lambda f.\lambda x.\ f^{n+1}(x)$$

$$= n+1$$

Calcul de +1 2 → 3



Le test à zéro

$$0? = \lambda n.\ n\ (\lambda z.faux)\ vrai$$

$$0?\ 0 = (\lambda n.\ n\ (\lambda z.faux)\ vrai)\ (\lambda f.\lambda x.x)$$

$$= (\lambda f.\lambda x.x)\ (\lambda z.faux)\ vrai$$

$$= vrai$$

disparaît car f absent du corps x de $(\lambda f.\lambda x.x)$

$$0?\ n+1 = (\lambda n.\ n\ (\lambda z.faux)\ vrai)\ (\lambda f.\lambda x.f(f^n(x)))$$

$$= (\lambda f.\lambda x.f(f^n(x)))\ (\lambda z.faux)\ vrai$$

$$= (\lambda z.faux)\ ((\lambda z.faux)^n\ (vrai))$$

$$= faux$$

disparaît car z absent du corps faux de $\lambda z.faux$

$$\underline{-1} = \lambda n.\ \lambda f.\ \lambda x.\ n\ (\lambda g.\ \lambda h.h(fg))\ (\lambda u.x)\ (\lambda v.v)$$

- Posons $C = (\lambda g.\lambda h.\ h(g\ f))$

  $\underline{-1}\ \underline{n+1} = \lambda f.\lambda x.\ \underline{n+1}\ C\ (\lambda u.x)\ (\lambda v.v)$

  $\rightarrow \lambda f.\lambda x.\ C\ (\underline{n}\ C\ (\lambda u.\ x))\ (\lambda v.\ v)$

  $= \lambda f.\lambda x.\ (\lambda g.\lambda h.\ h(g\ f))\ (\underline{n}\ C\ (\lambda u.\ x))\ (\lambda v.\ v)$

  $\rightarrow \lambda f.\lambda x.\ (\lambda v.\ v)\ (\underline{n}\ C\ (\lambda u.\ x)\ f)$

- Posons $D_p = \underline{p}\ C\ (\lambda u.\ x)\ f$
- Par récurrence, $D_p \rightarrow f^p\ x$
  - $D_0 = (\lambda u.\ x)\ f \rightarrow x$
  - $D_{p+1} = \underline{p+1}\ C\ (\lambda u.\ x)\ f$

    $\rightarrow C(\ \underline{p}\ C\ (\lambda u.\ x))\ f$

    $= (\lambda g.\lambda h.\ h(g\ f))\ (\ \underline{p}\ C\ (\lambda u.\ x))\ f$

    $\rightarrow f\ (\underline{p}\ C\ (\lambda u.\ x)\ f)$

    $\rightarrow f(f^p\ x)$

    $= f^{p+1}\ x$
- Donc

  $\underline{-1}\ \underline{n+1} = \lambda f.\lambda x.\ \underline{n}\ C\ (\lambda u.\ x)\ f$

  $\rightarrow \lambda f.\lambda x.f^n\ x$

  $= \underline{n}$ CQFD

- A noter

  $\underline{-1}\ \underline{0} = \lambda f.\lambda x.\ \underline{0}\ C\ (\lambda u.\ x)\ f$

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

  $\rightarrow \lambda f.\lambda x.\ x$

  $= \underline{0}$

Récursion et point fixe

Fact = λm. si m=1 alors 1 sinon Mult (m, Fact (m–1))

FACT = λf. λm. si m=1 alors 1 sinon Mult (m, f (m–1))

Fact = FACT (Fact) - équation de point fixe

Fact = Y FACT avec Y = (λx.λy.y(xxy)) (λx.λy.y(xxy))

Posons A = (λx. λy.y(xxy)), donc Y = A A

Posons Fact = Y FACT

Fact = A A FACT = (λx. λy.y(xxy)) A FACT

→ FACT (A A FACT)

= FACT (Fact) CQFD

λ-calcul vs. CAML

```
let rec Fact  m = si m=1 alors 1 sinon Mult (m, Fact (m–1))
```

```
Fact = Y (FACT)
       avec Y = (λx. λy.y(xxy)) (λx. λy.y(xxy))
```

Pour comprendre comment un calcul évolue,
le mieux est de comprendre ce qu'il laisse fixe !

Conséquence du codage des entiers

- Le codage unaire est inefficace, mais il implique que le λ-calcul est Turing-complet
- Et donc que toutes les propriétés importantes du λ-calcul sont indécidables
  - la réduction d'un terme M peut-elle terminer ?
  - deux termes M et N sont-ils inter-réductibles?
  - etc.

Conclusion

- La théorie de la calculabilité a produit des avancées fondamentales dans l'étude du calcul
- Mais ses machines et langages restent théoriques : la machine de Turing se vend mal, et le calcul pur est trop sauvage pour être pratique
- En pratique, on utilise des λ- calculs typés, auxquels on ajoute les entiers, l'arithmétique de base et le point fixe Y / rec (ex. PCF / CAML, voir cours 3). Mais ceci ne supprime pas l'indécidabilité !

Références

- D. Harel
  Computers Ltd : What They Really Can't Do
  Oxford University Press, 2000.

- Hartley Rogers
  Theory of Recursive Functions and Effective Computability
  MIT Press

- Henk Barendregt
  The Lambda-Calculus, its Syntax and Semantics
  Elsevier Science Ltd. (1987)

- Jean-Louis Krivine
  Lambda-calcul, types et modèles. Masson (1990)

  Lambda-Calculus Types and Models
  http://www.pps.jussieu.fr/~krivine/articles/Lambda.pdf

... et bien d'autres belles choses