Calculabilité II

(Re)Découverte du λ-calcul

Fabien Givors

d'après les cours de Jeff Foster, University of Maryland

Université Nice — Sophia-Antipolis

UE: SMZIFO12

Année universitaire 2014-2015

Déroulement de l'UE

Planning Cours, TD et Examen

- 4 séances de 3h30
  - 1h30 de cours
  - 2h00 de TD
- Puis examen écrit la 5e séance (sous forme de TD noté)

Programme de l'UE

- Découverte d'autres systèmes permettant le calcul (SAP)
- En particulier :
  - λ-calcul
  - Automates cellulaires
  - Pavages

1. vers le λ-calcul

Motivations et éléments

Historique

- Système mathématique formel
- Fondements de la logique mathématique
  - Déterminer et comprendre ces fondements
  - Étudier les fonctions, la récursivité
- Modéliser la notion de calcul
- Référence pour comprendre les langages fonctionnels

Points importants :

- Inventé en 1936 par Alonzo Church (1903—1995)
- Thèse de Church-Turing :
  Identification des fonctions calculables comme une classe mathématique « concrète ».
- **Théorème de Church** *Le calcul des prédicats égalitaires du premier ordre dans le langage de l'arithmétique est algorithmiquement indécidable.*

Le paradigme fonctionnel

Tout est fonction.

- Passer un paramètre équivaut à passer le code d'une fonction
- Un nombre est une fonction
- Un booléen est une fonction
- … (encodage)
- Tous les objets du langage sont des fonctions.

Tout est fonction.

Un mot sur la logique du premier ordre

Forme Normale de Kleene

Il existe une fonction élémentaire F et un prédicat récursif primitif T tels que:

$$\forall e, x, \varphi_u\left(\langle e, x\rangle\right) \cong \varphi_e(x) \cong F(\mu y.T(e, x, y))$$

Unique schéma μ
Indice de fonction
Flot d'exécution
Entrée
Vérificateur

Un mot sur la logique du premier ordre

Forme Normale de Kleene

Il existe une fonction élémentaire F et un prédicat récursif primitif T tels que:

$$\forall e, x, \varphi_u\left(\langle e, x\rangle\right) \cong \varphi_e(x) \cong F(\mu y.T(e, x, y))$$

En terme de logique du premier ordre

Il existe une fonction élémentaire F et un prédicat récursif primitif T tels que:

$$\forall e, x, \exists y, T(e, x, y) \wedge \forall y' < y, \neg T(e, x, y')$$

Et s'il existe un tel y,

$$\varphi_e(x) = F(y)$$

Unique schéma μ
Indice de fonction
Flot d'exécution
Entrée
Vérificateur

Un mot sur la logique du premier ordre

Forme Normale de Kleene

Il existe une fonction élémentaire F et un prédicat récursif primitif T tels que:

$$\forall e, x, \varphi_u\left(\langle e, x\rangle\right) \cong \varphi_e(x) \cong F(\mu y.T(e, x, y))$$

En terme de logique du premier ordre

Il existe une fonction élémentaire F et un prédicat récursif primitif T tels que:

$$\forall e, x, \exists y, T(e, x, y) \wedge \forall y' < y, \neg T(e, x, y')$$

Et s'il existe un tel y,

$$\varphi_e(x) = F(y)$$

Unique schéma μ
Indice de fonction
Flot d'exécution
Entrée
Vérificateur

Théorème de Church, arrêt et indécidabilité

- Le problème de l'arrêt est indécidable.

Théorème de Church, arrêt et indécidabilité

- Le problème de l'arrêt est indécidable.
- Une fonction calculable correspond à une formule du premier ordre, et inversement.

Théorème de Church, arrêt et indécidabilité

- Le problème de l'arrêt est indécidable.
- Une fonction calculable correspond à une formule du premier ordre, et inversement.
- Une formule du premier ordre peut encoder l'arrêt d'une machine.

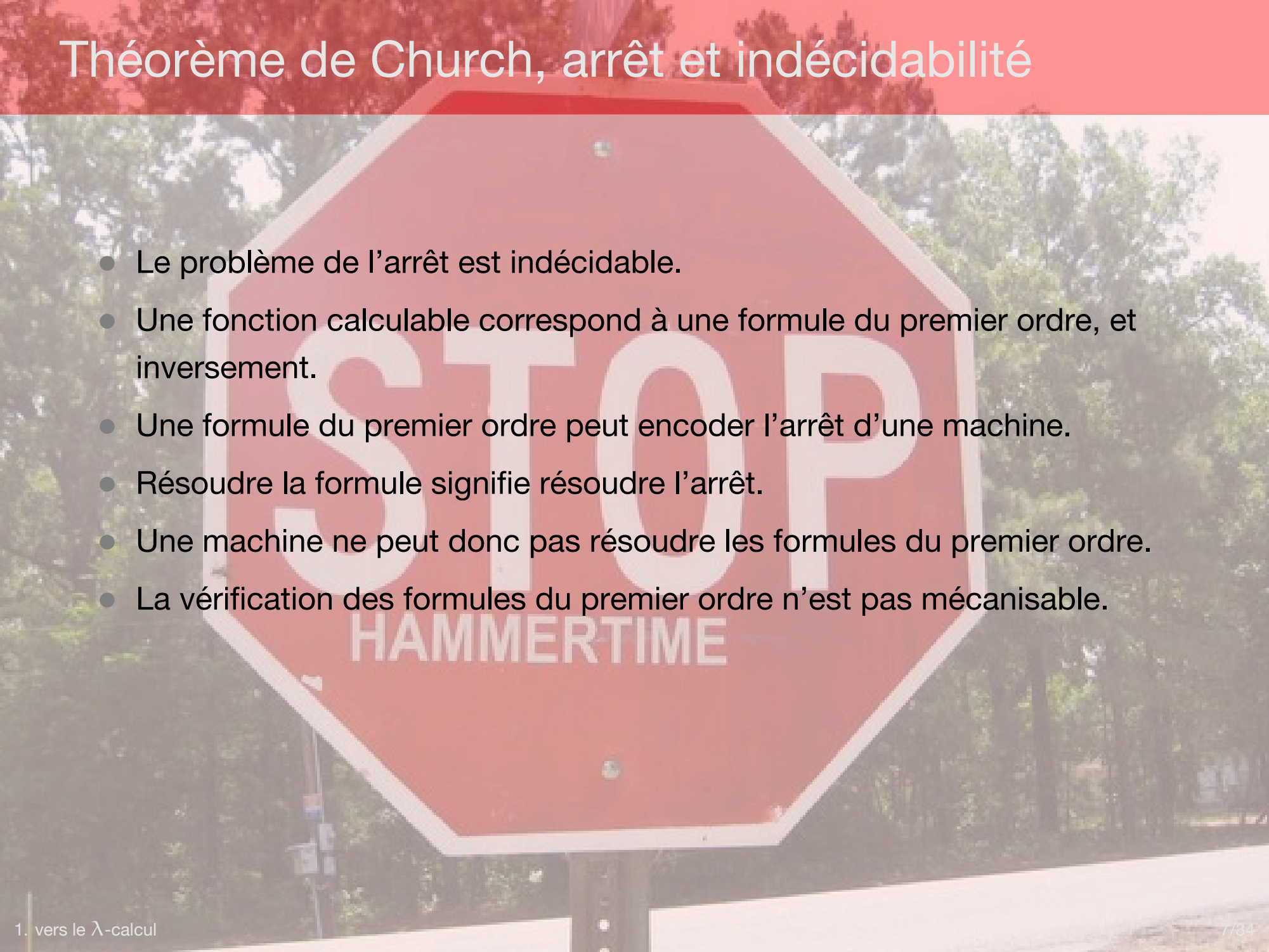
Théorème de Church, arrêt et indécidabilité

- Le problème de l'arrêt est indécidable.
- Une fonction calculable correspond à une formule du premier ordre, et inversement.
- Une formule du premier ordre peut encoder l'arrêt d'une machine.
- Résoudre la formule signifie résoudre l'arrêt.

Théorème de Church, arrêt et indécidabilité

- Le problème de l'arrêt est indécidable.
- Une fonction calculable correspond à une formule du premier ordre, et inversement.
- Une formule du premier ordre peut encoder l'arrêt d'une machine.
- Résoudre la formule signifie résoudre l'arrêt.
- Une machine ne peut donc pas résoudre les formules du premier ordre.

Théorème de Church, arrêt et indécidabilité



- Le problème de l'arrêt est indécidable.
- Une fonction calculable correspond à une formule du premier ordre, et inversement.
- Une formule du premier ordre peut encoder l'arrêt d'une machine.
- Résoudre la formule signifie résoudre l'arrêt.
- Une machine ne peut donc pas résoudre les formules du premier ordre.
- La vérification des formules du premier ordre n'est pas mécanisable.

Théorème de Church, arrêt et indécidabilité

- Le problème de l'arrêt est indécidable.
- Une fonction calculable correspond à une formule du premier ordre, et inversement.
- Une formule du premier ordre peut encoder l'arrêt d'une machine.
- Résoudre la formule signifie résoudre l'arrêt.
- Une machine ne peut donc pas résoudre les formules du premier ordre.
- La vérification des formules du premier ordre n'est pas mécanisable.

La logique du premier ordre est un modèle de calcul de même puissance que les machines de Turing ou les fonctions récursives.

2. introduction au λ-calcul

Définitions

La syntaxe la plus simple du monde

- Grammaire pour un terme e :

$$
\begin{array}{rcll}
e & ::= & x & \text{Variables} \\
 & | & \lambda x.\, e & \text{Abstraction} \\
 & | & e\, e & \text{Application}
\end{array}
$$

La syntaxe la plus simple du monde

- Grammaire pour un terme e :

$$\begin{array}{rcll} e & ::= & x & \text{Variables} \\ & | & \lambda x.\, e & \text{Abstraction} \\ & | & e\, e & \text{Application} \end{array}$$

- Tous les termes e sont des fonctions.

La syntaxe la plus simple du monde

- Grammaire pour un terme e :

$$\begin{array}{rcll} e & ::= & x & \text{Variables} \\ & | & \lambda x.\, e & \text{Abstraction} \\ & | & e\, e & \text{Application} \end{array}$$

- Tous les termes e sont des fonctions.
- Les arguments sont des fonctions.

La syntaxe la plus simple du monde

- Grammaire pour un terme e :

$$\begin{array}{rcll} e & ::= & x & \text{Variables} \\ & | & \lambda x.\, e & \text{Abstraction} \\ & | & e\, e & \text{Application} \end{array}$$

- Tous les termes e sont des fonctions.
- Les arguments sont des fonctions.
- Les valeurs retournées sont des fonctions.

La syntaxe la plus simple du monde

- Grammaire pour un terme e :

$$\begin{array}{rcll} e & ::= & x & \text{Variables} \\ & | & \lambda x.\, e & \text{Abstraction} \\ & | & e\, e & \text{Application} \end{array}$$

- Tous les termes e sont des fonctions.
- Les arguments sont des fonctions.
- Les valeurs retournées sont des fonctions.
- Les variables représentent des fonctions.

La syntaxe la plus simple du monde

- Grammaire pour un terme e :

$$\begin{array}{rcll} e & ::= & x & \text{Variables} \\ & | & \lambda x.\, e & \text{Abstraction} \\ & | & e\, e & \text{Application} \end{array}$$

- Tous les termes e sont des fonctions.
- Les arguments sont des fonctions.
- Les valeurs retournées sont des fonctions.
- Les variables représentent des fonctions.
- Les abstractions de termes sont des fonctions.

Les λ-abstractions

- Pour représenter la fonction identité :

$$i : n \mapsto n$$

On écrira :

$$I = \lambda x.x$$

Les λ-abstractions

- Pour représenter la fonction identité :

$$i : n \mapsto n$$

  On écrira :

$$I = \lambda x.\, x$$

- Pour représenter l'application du second argument au premier argument :

$$app : f, n \mapsto f(n)$$

  On écrira :

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

Réécriture, substitution

- En λ-calcul, le ruban, l'espace de travail, c'est le terme syntaxique.

Réécriture, substitution

- En λ-calcul, le ruban, l'espace de travail, c'est le terme syntaxique.
- Réécriture de l'expression à chaque étape

Réécriture, substitution

- En λ-calcul, le ruban, l'espace de travail, c'est le terme syntaxique.
- Réécriture de l'expression à chaque étape
- Opération fondamentale: la substitution

Réécriture, substitution

- En λ-calcul, le ruban, l'espace de travail, c'est le terme syntaxique.
- Réécriture de l'expression à chaque étape
- Opération fondamentale: la substitution

$$e_1[x \longleftarrow e_2]$$

- Signification : «On remplace les occurrences *(libres)* de x dans e_1 par e_2.»

Réécriture, substitution

- En λ-calcul, le ruban, l'espace de travail, c'est le terme syntaxique.
- Réécriture de l'expression à chaque étape
- Opération fondamentale: la substitution

$$e_1[x \longleftarrow e_2]$$

- Signification : «On remplace les occurrences *(libres)* de x dans e_1 par e_2.»
- Exemple:

$$(x\,y\,x\,z)[x \longleftarrow \lambda a \lambda b \lambda c.\, b]$$

Réécriture, substitution

- En λ-calcul, le ruban, l'espace de travail, c'est le terme syntaxique.
- Réécriture de l'expression à chaque étape
- Opération fondamentale: la substitution

$$e_1[x \leftarrow e_2]$$

- Signification : «On remplace les occurrences *(libres)* de x dans e_1 par e_2.»
- Exemple:

$$(x\,y\,x\,z)[x \leftarrow \lambda a\lambda b\lambda c.\,b]$$

- Donne…

$$(\lambda a\lambda b\lambda c.\,b)\,y\,(\lambda a\lambda b\lambda c.\,b)\,z$$

Application

- L'application remplace la variable abstraite (par l'abstraction λ)

Application

- L'application remplace la variable abstraite (par l'abstraction λ)
- La remplace par le premier argument

Application

- L'application remplace la variable abstraite (par l'abstraction λ)
- La remplace par le premier argument
- Si plusieurs abstractions, on procède une par une

Application

- L'application remplace la variable abstraite (par l'abstraction λ)
- La remplace par le premier argument
- Si plusieurs abstractions, on procède une par une
- Exemple :

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

Application

- L'application remplace la variable abstraite (par l'abstraction λ)
- La remplace par le premier argument
- Si plusieurs abstractions, on procède une par une
- Exemple :

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

- Vu comme :

$$(\lambda x.\, e_1)\, e_2$$

Application

- L'application remplace la variable abstraite (par l'abstraction λ)
- La remplace par le premier argument
- Si plusieurs abstractions, on procède une par une
- Exemple :

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

- Vu comme :

$$(\lambda x.\, e_1)\, e_2$$

- Suivant le schéma :

$$e_1[x \leftarrow e_2]$$

Application

- L'application remplace la variable abstraite (par l'abstraction λ)
- La remplace par le premier argument
- Si plusieurs abstractions, on procède une par une
- Exemple :

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

- Vu comme :

$$(\lambda x.\, e_1)\, e_2$$

- Suivant le schéma :

$$e_1[x \leftarrow e_2]$$

- Donne :

$$x[x \leftarrow z]$$

Application

- L'application remplace la variable abstraite (par l'abstraction λ)
- La remplace par le premier argument
- Si plusieurs abstractions, on procède une par une
- Exemple :

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

- Vu comme :

$$(\lambda x.\, e_1)\, e_2$$

- Suivant le schéma :

$$e_1[x \leftarrow e_2]$$

- Donne :

$$x[x \leftarrow z]$$

- Soit :

$$z$$

Exemple de tout à l'heure

- Le terme :

$$(\lambda x.\, x\, y\, x\, z)(\lambda a \lambda b \lambda c.\, b)$$

Exemple de tout à l'heure

- Le terme :

$$(\lambda x.\, x\, y\, x\, z)(\lambda a \lambda b \lambda c.\, b)$$

- Donne…

$$(x\, y\, x\, z)[x \leftarrow \lambda a \lambda b \lambda c.\, b]$$

Exemple de tout à l'heure

- Le terme :

$$(\lambda x.\, x\, y\, x\, z)(\lambda a \lambda b \lambda c.\, b)$$

- Donne…

$$(x\, y\, x\, z)[x \leftarrow \lambda a \lambda b \lambda c.\, b]$$

- Puis…

$$(\lambda a \lambda b \lambda c.\, b)\, y\, (\lambda a \lambda b \lambda c.\, b)\, z$$

Exemple de tout à l'heure

- Le terme :

$$(\lambda x.\, x\, y\, x\, z)(\lambda a \lambda b \lambda c.\, b)$$

- Donne…

$$(x\, y\, x\, z)[x \leftarrow \lambda a \lambda b \lambda c.\, b]$$

- Puis…

$$(\lambda a \lambda b \lambda c.\, b)\, y\, (\lambda a \lambda b \lambda c.\, b)\, z$$

- Donc…

$$(\lambda b \lambda c.\, b)\, (\lambda a \lambda b \lambda c.\, b)\, z$$

Exemple de tout à l'heure

- Le terme :

$$(\lambda x.\, x\, y\, x\, z)(\lambda a \lambda b \lambda c.\, b)$$

- Donne…

$$(x\, y\, x\, z)[x \leftarrow \lambda a \lambda b \lambda c.\, b]$$

- Puis…

$$(\lambda a \lambda b \lambda c.\, b)\, y\, (\lambda a \lambda b \lambda c.\, b)\, z$$

- Donc…

$$(\lambda b \lambda c.\, b)\, (\lambda a \lambda b \lambda c.\, b)\, z$$

- Soit…

$$(\lambda b.\, b)\, (\lambda a \lambda b \lambda c.\, b)$$

Exemple de tout à l'heure

- Le terme :

$$(\lambda x.\, x\, y\, x\, z)(\lambda a \lambda b \lambda c.\, b)$$

- Donne…

$$(x\, y\, x\, z)[x \leftarrow \lambda a \lambda b \lambda c.\, b]$$

- Puis…

$$(\lambda a \lambda b \lambda c.\, b)\, y\, (\lambda a \lambda b \lambda c.\, b)\, z$$

- Donc…

$$(\lambda b \lambda c.\, b)\, (\lambda a \lambda b \lambda c.\, b)\, z$$

- Soit…

$$(\lambda b.\, b)\, (\lambda a \lambda b \lambda c.\, b)$$

- Et finalement :

$$\lambda a \lambda b \lambda c.\, b$$

Variables libres, variables liées

- Dans le terme :

$$e = \lambda x.\, x\, y\, x\, z$$

Variables libres, variables liées

- Dans le terme :

$$e = \lambda x.\, x\, y\, x\, z$$

- On distingue deux types de variables :

Variables libres, variables liées

- Dans le terme :

$$e = \lambda x.\, x\, y\, x\, z$$

- On distingue deux types de variables :
  - x est une variable qui est abstraite par un λ

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

Variables libres, variables liées

- Dans le terme :

$$e = \lambda x.\, x\, y\, x\, z$$

- On distingue deux types de variables :
  - x est une variable qui est abstraite par un λ

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

  - Si on applique à e un terme f, alors toutes les occurrences de x actuelles vont disparaître.

Variables libres, variables liées

- Dans le terme :

$$e = \lambda x.\, x\, y\, x\, z$$

- On distingue deux types de variables :
  - x est une variable qui est abstraite par un λ

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

  - Si on applique à e un terme f, alors toutes les occurrences de x actuelles vont disparaître.
  - On dit que les occurrences de x sont *liées*.

Variables libres, variables liées

- Dans le terme :

$$e = \lambda x.\, x\, y\, x\, z$$

- On distingue deux types de variables :
  - y est une variable qui n'est abstraite par aucun λ

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

Variables libres, variables liées

- Dans le terme :

$$e = \lambda x.\, x\, y\, x\, z$$

- On distingue deux types de variables :
  - y est une variable qui n'est abstraite par aucun λ

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

  - Si on applique à e un terme f, alors les occurrences de y actuelles seront inchangées.

Variables libres, variables liées

- Dans le terme :

$$e = \lambda x.\, x\, y\, x\, z$$

- On distingue deux types de variables :
  - y est une variable qui n'est abstraite par aucun λ

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

  - Si on applique à e un terme f, alors les occurrences de y actuelles seront inchangées.
  - On dit que les occurrences de y sont *libres*.

Variables libres, variables liées

- Dans le terme :

$$e = \lambda x.\, x\, y\, x\, z$$

- On distingue deux types de variables :
  - y est une variable qui n'est abstraite par aucun λ

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

  - Si on applique à e un terme f, alors les occurrences de y actuelles seront inchangées.
  - On dit que les occurrences de y sont *libres*.
  - Par extemsion, on dit que la variable y est une variable libre.

Variables libres, variables liées (2)

- Dans le terme :

$$e = (\lambda a \lambda y.\, a\, x)(\lambda x.\, x\, y\, x\, z)$$

Variables libres, variables liées (2)

- Dans le terme :

$$e = (\lambda a \lambda y.\, a\, x)(\lambda x.\, x\, y\, x\, z)$$

- x apparaît libre :

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

Variables libres, variables liées (2)

- Dans le terme :

$$e = (\lambda a \lambda y.\, a\, x)(\lambda x.\, x\, y\, x\, z)$$

- x apparaît libre :

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

- et liée :

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

Variables libres, variables liées (2)

- Dans le terme :

$$e = (\lambda a \lambda y.\, a\, x)(\lambda x.\, x\, y\, x\, z)$$

- x apparaît libre :

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

- et liée :

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

- x est donc à la fois libre et liée dans e

Variables libres, variables liées (2)

- Dans le terme :

$$e = (\lambda a \lambda y.\, a\, x)(\lambda x.\, x\, y\, x\, z)$$

- x apparaît libre :

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

- et liée :

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

- x est donc à la fois libre et liée dans e
- En revanche, les occurrences sont soit libres, soit liées.

Variables libres, variables liées (2)

- Dans le terme :

$$e = (\lambda a \lambda y.\, a\, x)(\lambda x.\, x\, y\, x\, z)$$

- x apparaît libre :

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

- et liée :

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

- x est donc à la fois libre et liée dans e
- En revanche, les occurrences sont soit libres, soit liées.
- Après application :

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

Variables libres, variables liées (2)

- Dans le terme :
$$e = (\lambda a \lambda y.\, a\, x)(\lambda x.\, x\, y\, x\, z)$$
- x apparaît libre :
$$(\lambda a \lambda y.\, a\, x)(\lambda x.\, x\, y\, x\, z)$$
- et liée :
$$(\lambda a \lambda y.\, a\, x)(\lambda x.\, x\, y\, x\, z)$$
- x est donc à la fois libre et liée dans e
- En revanche, les occurrences sont soit libres, soit liées.
- Après application :
$$\lambda y \lambda x.\, x\, y\, x\, z\, x$$
- y qui semblait libre a été capturée par le λy

Variables libres, variables liées (2)

- Dans le terme :
$$e = (\lambda a \lambda y.\, a\, x)(\lambda x.\, x\, y\, x\, z)$$
- x apparaît libre :
$$(\lambda a \lambda y.\, a\, x)(\lambda x.\, x\, y\, x\, z)$$
- et liée :
$$(\lambda a \lambda y.\, a\, x)(\lambda x.\, x\, y\, x\, z)$$
- x est donc à la fois libre et liée dans e
- En revanche, les occurrences sont soit libres, soit liées.
- Après application :
$$\lambda y \lambda x.\, x\, y\, x\, z\, x$$
- y qui semblait libre a été capturée par le λy
- D'où la restriction de la substitution aux variables libres, et pour ça…

3. réductions

Enfin du calcul

α-conversion

- Dans le terme :

$$e = \lambda x.\, x\, y\, x\, z$$

α-conversion

- Dans le terme :

$$e = \lambda x.\, x\, y\, x\, z$$

- Toutes les occurrences de x sont liées.

α-conversion

- Dans le terme :

$$e = \lambda x.\, x\, y\, x\, z$$

- Toutes les occurrences de x sont liées.
- x est une variable « muette »

α-conversion

- Dans le terme :

$$e = \lambda x.\, x\, y\, x\, z$$

- Toutes les occurrences de x sont liées.
- x est une variable « muette »
- En fait, on peut remplacer x par n'importe quelle variable non utilisée :

$$e = \lambda x.\, x\, y\, x\, z \cong \lambda w.\, w\, y\, w\, z$$

α-conversion

- Dans le terme :

$$e = \lambda x.\, x\, y\, x\, z$$

- Toutes les occurrences de x sont liées.
- x est une variable « muette »
- En fait, on peut remplacer x par n'importe quelle variable non utilisée :

$$e = \lambda x.\, x\, y\, x\, z \cong \lambda w.\, w\, y\, w\, z$$

- Ce procédé s'appelle l'α-conversion.

α-conversion

- Dans le terme :

$$e = \lambda x.\, x\, y\, x\, z$$

- Toutes les occurrences de x sont liées.
- x est une variable « muette »
- En fait, on peut remplacer x par n'importe quelle variable non utilisée :

$$e = \lambda x.\, x\, y\, x\, z \cong \lambda w.\, w\, y\, w\, z$$

- Ce procédé s'appelle l'α-conversion.
- On note cette réduction (qui ne change rien) $\longrightarrow_\alpha$

$$\lambda x.\, x\, y\, x\, z \longrightarrow_\alpha \lambda w.\, w\, y\, w\, z$$

β-reduction

- Dans le terme :

$$e = (\lambda x.e_1)\, e_2$$

β-reduction

- Dans le terme :

$$e = (\lambda x.e_1)\, e_2$$

- L'application peut être effectuée. On obtient alors :

$$e_1[x \leftarrow e_2]$$

β-reduction

- Dans le terme :

$$e = (\lambda x.e_1)\, e_2$$

- L'application peut être effectuée. On obtient alors :

$$e_1[x \leftarrow e_2]$$

- Ce procédé s'appelle la β-reduction.

β-reduction

- Dans le terme :

$$e = (\lambda x.e_1)\, e_2$$

- L'application peut être effectuée. On obtient alors :

$$e_1[x \leftarrow e_2]$$

- Ce procédé s'appelle la β-reduction.
- On note cette réduction (qui n'est pas réversible) $\rightarrow_\beta$

$$(\lambda x.e_1)\, e_2 \rightarrow_\beta e_1[x \leftarrow e_2]$$

β-reduction

- Dans le terme :

$$e = (\lambda x.e_1)\, e_2$$

- L'application peut être effectuée. On obtient alors :

$$e_1[x \leftarrow e_2]$$

- Ce procédé s'appelle la β-reduction.
- On note cette réduction (qui n'est pas réversible) $\rightarrow_\beta$

$$(\lambda x.e_1)\, e_2 \rightarrow_\beta e_1[x \leftarrow e_2]$$

- Un terme pouvant se réduire via β est appelé un *redex.*

β-reduction

- Dans le terme :

$$e = (\lambda x.e_1)\, e_2$$

- L'application peut être effectuée. On obtient alors :

$$e_1[x \leftarrow e_2]$$

- Ce procédé s'appelle la β-reduction.
- On note cette réduction (qui n'est pas réversible) $\rightarrow_\beta$

$$(\lambda x.e_1)\, e_2 \rightarrow_\beta e_1[x \leftarrow e_2]$$

- Un terme pouvant se réduire via β est appelé un *redex.*
- NB: seules les occurrences libres de x de e_1 sont remplacées par e_2

β^*-reduction

- On note β^* la cloture réflexo-transitive de la réduction β

β^*-reduction

- On note β^* la cloture réflexo-transitive de la réduction β
- i.e. $a \to_{\beta^*} c$ ssi $\exists b_0, \ldots, b_{n-1}, \begin{cases} a \to_\beta b_0 \\ \forall i < n, b_{i-1} \to_\beta b_i \quad \text{ou } a \cong c \\ b_{n-1} \to_\beta c \end{cases}$

Notion de convergence

- Que dire du terme suivant :

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

Notion de convergence

- Que dire du terme suivant :

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

- C'est un *regex*

Notion de convergence

- Que dire du terme suivant :

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

- C'est un *regex*
- Il se réduit en…

Notion de convergence

- Que dire du terme suivant :

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

- C'est un *regex*
- Il se réduit en…
- Lui-même…

Notion de convergence

- Que dire du terme suivant :

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

- C'est un *regex*
- Il se réduit en…
- Lui-même…
- Que dire du terme suivant :

$$\Omega_3 = (\lambda x.\, x\, x\, x)(\lambda x.\, x\, x\, x)$$

Notion de convergence

- Que dire du terme suivant :

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

- C'est un *regex*
- Il se réduit en…
- Lui-même…
- Que dire du terme suivant :

$$\Omega_3 = (\lambda x.\, x\, x\, x)(\lambda x.\, x\, x\, x)$$

- C'est un *regex*

Notion de convergence

- Que dire du terme suivant :

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

- C'est un *regex*
- Il se réduit en…
- Lui-même…
- Que dire du terme suivant :

$$\Omega_3 = (\lambda x.\, x\, x\, x)(\lambda x.\, x\, x\, x)$$

- C'est un *regex*
- Il se réduit en…

Notion de convergence

- Que dire du terme suivant :

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

- C'est un *regex*
- Il se réduit en…
- Lui-même…
- Que dire du terme suivant :

$$\Omega_3 = (\lambda x.\, x\, x\, x)(\lambda x.\, x\, x\, x)$$

- C'est un *regex*
- Il se réduit en…
- Oulah…

Notion de confluence

- Que faire lorsque plusieurs réductions sont possibles en même temps ?

Notion de confluence

- Que faire lorsque plusieurs réductions sont possibles en même temps ?
- Dans quelle ordre les faire ?

Notion de confluence

- Que faire lorsque plusieurs réductions sont possibles en même temps ?
- Dans quelle ordre les faire ?
- Exemple :

$$(\lambda x.e_1)\,(\lambda x \lambda y.\, e_2\, e_3)\,((\lambda z.\, z\, z)t)$$

Notion de confluence

- Que faire lorsque plusieurs réductions sont possibles en même temps ?
- Dans quelle ordre les faire ?
- Exemple :

$$(\lambda x.e_1)\,(\lambda x \lambda y.\, e_2\, e_3)\,((\lambda z.\, z\, z)t)$$

Théorème (propriété du diamant)

Si $a \longrightarrow_\beta b_1$ et $a \longrightarrow_\beta b_2$

Alors il existe d tel que $b_1 \longrightarrow_{\beta^*} d$ et $b_2 \longrightarrow_{\beta^*} d$.

Notion de confluence

- Que faire lorsque plusieurs réductions sont possibles en même temps ?
- Dans quelle ordre les faire ?
- Exemple :

$$(\lambda x.e_1)\,(\lambda x \lambda y.\, e_2\, e_3)\,((\lambda z.\, z\, z)t)$$

Théorème (propriété du diamant)

Si $a \longrightarrow_\beta b_1$ et $a \longrightarrow_\beta b_2$

Alors il existe d tel que $b_1 \longrightarrow_{\beta^*} d$ et $b_2 \longrightarrow_{\beta^*} d$.

Théorème (propriété de confluence)

Si $a \longrightarrow_{\beta^*} b_1$ et $a \longrightarrow_{\beta^*} b_2$

Alors il existe d tel que $b_1 \longrightarrow_{\beta^*} d$ et $b_2 \longrightarrow_{\beta^*} d$.

Notion forme normale

- Un terme est sous forme normale lorsqu'il ne peut pllus être réduit

Notion forme normale

- Un terme est sous forme normale lorsqu'il ne peut pllus être réduit
- Tous les termes n'ont pas de forme normale

Notion forme normale

- Un terme est sous forme normale lorsqu'il ne peut pllus être réduit
- Tous les termes n'ont pas de forme normale

Théorème (unicité de la forme normale)

Un terme ne peut pas avoir plus d'une forme normale

Notion forme normale

- Un terme est sous forme normale lorsqu'il ne peut pllus être réduit
- Tous les termes n'ont pas de forme normale

Théorème (unicité de la forme normale)

Un terme ne peut pas avoir plus d'une forme normale

Théorème (propriété de confluence)

Si $a \longrightarrow_{\beta*} b_1$ et $a \longrightarrow_{\beta*} b_2$

Alors il existe d tel que $b_1 \longrightarrow_{\beta*} d$ et $b_2 \longrightarrow_{\beta*} d$.

4. point fixe et récursion

Définitions

Récursion sur f ?

- Auto-application

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

Récursion sur f ?

- Auto-application

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

- Un appel récursif ?

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

Récursion sur f ?

- Auto-application

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

- Un appel récursif ?

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

- n appels récursifs ? (récursion primitive)

$$\underline{n}\, f\, x$$

Récursion sur f ?

- Auto-application

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

- Un appel récursif ?

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

- n appels récursifs ? (récursion primitive)

$$\underline{n}\, f\, x$$

- Point fixe ?

Récursion ouverte

En OCaml :

```
let f g n =
  if n=0
  then 1
  else n * (g g (n-1))

let fact = f f
```

En Scheme :

```
(define (f g n)
  (if (= n 0)
    1
    (* n (g g (- n 1)))
  )
)
(define (fact n) (f f n))
```

Récursion ouverte

En OCaml :

```
let f g n =
  if n=0
  then 1
  else n * (g g (n-1))

let fact = f f
```

En Scheme :

```
(define (f g n)
  (if (= n 0)
    1
    (* n (g g (- n 1)))
  )
)
(define (fact n) (f f n))
```

En λ-calcul :

$$(\lambda f.\,(\lambda g.\,f\,g\,g)\,(\lambda n.\,f\,n\,n))$$
$$(\lambda f \lambda n.\,\text{si } n = 0,\, 1 \text{ sinon } n * f(n-1))$$

f (Récursion sur f)

Notons $Y = (\lambda f.\,(\lambda g.\,f\,(g\,g))(\lambda n.f\,(n\,n)))$:

- $$Y\,F$$

f (Récursion sur f)

Notons $Y = (\lambda f.\,(\lambda g.\,f\,(g\,g))(\lambda n.f\,(n\,n)))$:

- $$Y\,F$$
- $$((\lambda n.\,F\,(n\,n))\,(\lambda n.\,F\,(n\,n)))$$

f (Récursion sur f)

Notons $Y = (\lambda f.\,(\lambda g.\,f\,(g\,g))(\lambda n.f\,(n\,n)))$:

- $$Y\,F$$
- $$((\lambda n.\,F\,(n\,n))\,(\lambda n.\,F\,(n\,n)))$$
- $$F\,((\lambda n.\,F\,(n\,n))\,(\lambda n.\,F\,(n\,n)))$$

f (Récursion sur f)

Notons $Y = (\lambda f.\,(\lambda g.\,f\,(g\,g))(\lambda n.f\,(n\,n)))$:

- $$Y\,F$$
- $$((\lambda n.\,F\,(n\,n))\,(\lambda n.\,F\,(n\,n)))$$
- $$F\,((\lambda n.\,F\,(n\,n))\,(\lambda n.\,F\,(n\,n)))$$
- $$F\,(Y\,F)$$

f (Récursion sur f)

Notons $Y = (\lambda f.\,(\lambda g.\,f\,(g\,g))(\lambda n.f\,(n\,n)))$:

- $$Y\,F$$
- $$((\lambda n.\,F\,(n\,n))\,(\lambda n.\,F\,(n\,n)))$$
- $$F\,((\lambda n.\,F\,(n\,n))\,(\lambda n.\,F\,(n\,n)))$$
- $$F\,(Y\,F)$$

- Forme récursion ouverte (ou programmation par continuation)

$$(\lambda f.\,(\lambda g.\,f\,((\lambda n.\,f\,(n\,n))\,(\lambda n.\,f\,(n\,n)))\,F$$

f (f (Récursion sur f))

$$\text{Notons } G = (\lambda f \lambda n.\, \text{si } n = 0,\, 1 \text{ sinon } n * f(n-1)) :$$

- *In extenso*:

$$(\lambda f.\, (\lambda g.\, f\,(g\,g))\,(\lambda n.\, f\,(n\,n)))\,(\lambda f \lambda n.\, \text{si } n = 0 \text{ alors } 1 \text{ sinon } n * f(n-1))\,42$$

f (f (Récursion sur f))

Notons $G = (\lambda f \lambda n.\, \text{si } n = 0,\, 1 \text{ sinon } n * f(n-1))$:

- *In extenso*:

$$(\lambda f.\, (\lambda g.\, f\,(g\,g))\,(\lambda n.\, f\,(n\,n)))\,(\lambda f \lambda n.\, \text{si } n = 0 \text{ alors } 1 \text{ sinon } n * f(n-1))\,42$$

- En utilisant la forme *réduite* $Y\,F = F\,(Y\,F)$

$$(\lambda f \lambda n.\, \text{si } n = 0 \text{ alors } 1 \text{ sinon } n * f(n-1))\,(Y\,G)\,42$$

f (f (Récursion sur f))

$$\text{Notons } G = (\lambda f \lambda n.\, \text{si } n = 0,\, 1 \text{ sinon } n * f(n-1)) :$$

- *In extenso*:

$$(\lambda f.\, (\lambda g.\, f\,(g\,g))\,(\lambda n.\, f\,(n\,n)))\,(\lambda f \lambda n.\, \text{si } n = 0 \text{ alors } 1 \text{ sinon } n * f(n-1))\,42$$

- En utilisant la forme *réduite* $Y\,F = F\,(Y\,F)$

$$(\lambda f \lambda n.\, \text{si } n = 0 \text{ alors } 1 \text{ sinon } n * f(n-1))\,(Y\,G)\,42$$

- Et donc…

$$\text{si } 42 = 0 \text{ alors } 1 \text{ sinon } 42 * ((Y\,G)\,41)$$

$$42 * ((Y\,G)\,41) = 42 * 41 * \ldots * ((Y\,G)\,0) = !42$$

f (f (f (Récursion sur f)))

- Un terme Y tel que

$$Y\,f\,n = f\,(Y\,f)\,n$$

f (f (f (Récursion sur f)))

- Un terme Y tel que

$$Y\,f\,n = f\,(Y\,f)\,n$$

- Combinateur de point fixe :

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

f (f (f (Récursion sur f)))

- Un terme Y tel que

$$Y\,f\,n = f\,(Y\,f)\,n$$

- Combinateur de point fixe :

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

- Attention, CBN vs. CBV

f (f (f (Récursion sur f)))

- Un terme Y tel que

$$Y\,f\,n = f\,(Y\,f)\,n$$

- Combinateur de point fixe :

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

- Attention, CBN vs. CBV
- Différentes manières de construire Y…

f (f (f (Récursion sur f)))

- Un terme Y tel que

$$Y\,f\,n = f\,(Y\,f)\,n$$

- Combinateur de point fixe :

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

- Attention, CBN vs. CBV
- Différentes manières de construire Y…
- Exemple (combinateun CBV)

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

5. λ-calcul (simplement) typé

Définitions

La syntaxe la plus simple du monde, et son typage

- Grammaire pour un terme e :

$$
\begin{array}{rcll}
e & ::= & x & \text{Variables} \\
 & | & \lambda x : \sigma . e & \text{Abstraction} \\
 & | & e\, e & \text{Application}
\end{array}
$$

La syntaxe la plus simple du monde, et son typage

- Grammaire pour un terme e :

$$\begin{array}{rcll} e & ::= & x & \text{Variables} \\ & | & \lambda x : \sigma.\, e & \text{Abstraction} \\ & | & e\, e & \text{Application} \end{array}$$

- Définition inductive des types :

$$\begin{array}{rcll} \sigma & ::= & k & \text{Types de base} \\ & | & \sigma_1 \to \sigma_2 & \text{Type fonction} \end{array}$$

Règles de typage

- Contexte (termes déjà typés) :

$$(e, \sigma) \in \Gamma \Leftrightarrow \Gamma \vdash e : \sigma$$

Règles de typage

- Contexte (termes déjà typés) :

$$(e, \sigma) \in \Gamma \Leftrightarrow \Gamma \vdash e : \sigma$$

- Typage abstraction :

$$\Gamma \cup \{(x, \sigma_1)\} \vdash e_2 : \sigma_2 \Rightarrow \Gamma \vdash (\lambda x : \sigma_1 . e_2) : \sigma_1 \rightarrow \sigma_2$$

Règles de typage

- Contexte (termes déjà typés) :

$$(e, \sigma) \in \Gamma \Leftrightarrow \Gamma \vdash e : \sigma$$

- Typage abstraction :

$$\Gamma \cup \{(x, \sigma_1)\} \vdash e_2 : \sigma_2 \Rightarrow \Gamma \vdash (\lambda x : \sigma_1 . e_2) : \sigma_1 \rightarrow \sigma_2$$

- Typage application :

$$\Gamma \vdash e_1 : \sigma_1 \wedge \Gamma \vdash e : \sigma_1 \rightarrow \sigma_2 \Rightarrow \Gamma \vdash e\, e_1 : \sigma_2$$

Exemples

- $e_1 = \lambda x : a . x, e_1 : ?$

Exemples

- $e_1 = \lambda x : a.\, x, e_1 : ?$
  - $\{(x, a)\} \vdash x : ?$

Exemples

- $e_1 = \lambda x : a.\, x, e_1 : ?$
  - $\{(x, a)\} \vdash x : ?$
  - $\{(x, a)\} \vdash x : a$

Exemples

- $e_1 = \lambda x : a.x, e_1 : ?$
  - $\{(x, a)\} \vdash x : ?$
  - $\{(x, a)\} \vdash x : a$
  - $\emptyset \vdash \lambda x : a.x : a \to a$

Exemples

- $e_1 = \lambda x : a.x, e_1 : ?$
  - $\{(x, a)\} \vdash x : ?$
  - $\{(x, a)\} \vdash x : a$
  - $\emptyset \vdash \lambda x : a.x : a \to a$
  - $\vdash e_1 : a \to a$

Exemples

- $e_1 = \lambda x : a . x, e_1 : ?$
  - $\{(x, a)\} \vdash x : ?$
  - $\{(x, a)\} \vdash x : a$
  - $\emptyset \vdash \lambda x : a . x : a \to a$
  - $\vdash e_1 : a \to a$
- $e_2 = \lambda f : a \to a \lambda x : a . f x, e_2 : ?$

Exemples

- $e_1 = \lambda x : a.\, x,\ e_1 : ?$
  - $\{(x, a)\} \vdash x : ?$
  - $\{(x, a)\} \vdash x : a$
  - $\emptyset \vdash \lambda x : a.\, x : a \to a$
  - $\vdash e_1 : a \to a$
- $e_2 = \lambda f : a \to a \lambda x : a.\, f x,\ e_2 : ?$
  - $\{(f, a \to a)\} \vdash \lambda x : a.\, f\, x : ?$

Exemples

- $e_1 = \lambda x : a.x, e_1 : ?$
  - $\{(x, a)\} \vdash x : ?$
  - $\{(x, a)\} \vdash x : a$
  - $\emptyset \vdash \lambda x : a.x : a \to a$
  - $\vdash e_1 : a \to a$
- $e_2 = \lambda f : a \to a \lambda x : a.fx, e_2 : ?$
  - $\{(f, a \to a)\} \vdash \lambda x : a.fx : ?$
  - $\{(f, a \to a), (x, a)\} \vdash fx : ?$

Exemples

- $e_1 = \lambda x : a.x, e_1 : ?$
  - $\{(x, a)\} \vdash x : ?$
  - $\{(x, a)\} \vdash x : a$
  - $\emptyset \vdash \lambda x : a.x : a \to a$
  - $\vdash e_1 : a \to a$
- $e_2 = \lambda f : a \to a \lambda x : a.fx, e_2 : ?$
  - $\{(f, a \to a)\} \vdash \lambda x : a.fx : ?$
  - $\{(f, a \to a), (x, a)\} \vdash fx : ?$
  - $\{(f, a \to a), (x, a)\} \vdash fx : a$

Exemples

- $e_1 = \lambda x : a.x, e_1 : ?$
  - $\{(x, a)\} \vdash x : ?$
  - $\{(x, a)\} \vdash x : a$
  - $\emptyset \vdash \lambda x : a.x : a \to a$
  - $\vdash e_1 : a \to a$
- $e_2 = \lambda f : a \to a \lambda x : a.fx, e_2 : ?$
  - $\{(f, a \to a)\} \vdash \lambda x : a.fx : ?$
  - $\{(f, a \to a), (x, a)\} \vdash fx : ?$
  - $\{(f, a \to a), (x, a)\} \vdash fx : a$
  - $\{(f, a \to a)\} \vdash \lambda x : a.fx : a \to a$

Exemples

- $e_1 = \lambda x : a.x, e_1 : ?$
  - $\{(x, a)\} \vdash x : ?$
  - $\{(x, a)\} \vdash x : a$
  - $\emptyset \vdash \lambda x : a.x : a \to a$
  - $\vdash e_1 : a \to a$
- $e_2 = \lambda f : a \to a \lambda x : a.fx, e_2 : ?$
  - $\{(f, a \to a)\} \vdash \lambda x : a.fx : ?$
  - $\{(f, a \to a), (x, a)\} \vdash fx : ?$
  - $\{(f, a \to a), (x, a)\} \vdash fx : a$
  - $\{(f, a \to a)\} \vdash \lambda x : a.fx : a \to a$
  - $\emptyset \vdash \lambda f : a \to a \lambda x : a.fx : (a \to a) \to a \to a$

Exemples

- $e_1 = \lambda x : a.x, e_1 : ?$
  - $\{(x, a)\} \vdash x : ?$
  - $\{(x, a)\} \vdash x : a$
  - $\emptyset \vdash \lambda x : a.x : a \to a$
  - $\vdash e_1 : a \to a$
- $e_2 = \lambda f : a \to a \lambda x : a.fx, e_2 : ?$
  - $\{(f, a \to a)\} \vdash \lambda x : a.fx : ?$
  - $\{(f, a \to a), (x, a)\} \vdash fx : ?$
  - $\{(f, a \to a), (x, a)\} \vdash fx : a$
  - $\{(f, a \to a)\} \vdash \lambda x : a.fx : a \to a$
  - $\emptyset \vdash \lambda f : a \to a \lambda x : a.fx : (a \to a) \to a \to a$
  - $\vdash e_2 : (a \to a) \to a \to a$

Logique intuitioniste (déduction naturelle)

- $A \Longrightarrow B$

Logique intuitioniste (déduction naturelle)

- $A \implies B$
- « Si j'ai une preuve de A, je peux obtenir une preuve de B. »

Logique intuitioniste (déduction naturelle)

- $A \implies B$
- « Si j'ai une preuve de A, je peux obtenir une preuve de B. »
- Constructiviste.

Logique intuitioniste (déduction naturelle)

- $A \implies B$
- « Si j'ai une preuve de A, je peux obtenir une preuve de B. »
- Constructiviste.
- Un théorème est un algorithme transformant un énoncé mathématique en un autre énoncé mathématique.

Logique intuitioniste (déduction naturelle)

- $A \implies B$
- « Si j'ai une preuve de A, je peux obtenir une preuve de B. »
- Constructiviste.
- Un théorème est un algorithme transformant un énoncé mathématique en un autre énoncé mathématique.
- Règles logiques déductives respectées = théorème vrai

Logique intuitioniste (déduction naturelle)

- $A \implies B$
- « Si j'ai une preuve de A, je peux obtenir une preuve de B. »
- Constructiviste.
- Un théorème est un algorithme transformant un énoncé mathématique en un autre énoncé mathématique.
- Règles logiques déductives respectées = théorème vrai
- De la même manière, types des λ-termes constructifs

Correspondance de Curry-Howard

- « *Les énoncés vrais en logique intuitioniste sont les types des termes du λ-calcul bien typé.* »

Correspondance de Curry-Howard

- « *Les énoncés vrais en logique intuitionniste sont les types des termes du λ-calcul bien typé.* »
- Axiomes :

$$(e, \sigma) \in \Gamma \vdash e : \sigma$$

correspond à

$$A \in \Gamma \vdash A$$

Correspondance de Curry-Howard

- « *Les énoncés vrais en logique intuitionniste sont les types des termes du λ-calcul bien typé.* »
- Axiomes :

$$(e, \sigma) \in \Gamma \vdash e : \sigma$$

  correspond à

$$A \in \Gamma \vdash A$$

- Implication :

$$\Gamma \cup \{(x, \sigma_1)\} \vdash e_2 : \sigma_2 \Rightarrow \Gamma \vdash (\lambda x : \sigma_1 . e_2) : \sigma_1 \rightarrow \sigma_2$$

  correspond à

$$\Gamma, A \vdash B \implies \Gamma \vdash A \rightarrow B$$

Correspondance de Curry-Howard

- « *Les énoncés vrais en logique intuitionniste sont les types des termes du λ-calcul bien typé.* »
- Axiomes :

$$(e, \sigma) \in \Gamma \vdash e : \sigma$$

  correspond à

$$A \in \Gamma \vdash A$$

- Implication :

$$\Gamma \cup \{(x, \sigma_1)\} \vdash e_2 : \sigma_2 \Rightarrow \Gamma \vdash (\lambda x : \sigma_1 . e_2) : \sigma_1 \rightarrow \sigma_2$$

  correspond à

$$\Gamma, A \vdash B \implies \Gamma \vdash A \rightarrow B$$

- Utilisation de l'application :

$$\Gamma \vdash e_1 : \sigma_1 \wedge \Gamma \vdash e : \sigma_1 \rightarrow \sigma_2 \Rightarrow \Gamma \vdash e\, e_1 : \sigma_2$$

  correspond à

$$\Gamma \vdash A \rightarrow B \wedge \Gamma \vdash A \implies \Gamma \vdash B$$