Preuves et Calculs

LOGique pour l'informatique

Mathieu Jaume

Université Paris 6 - LIP6

LOG
2007/2008

Logique et Informatique (1)

Programmation (en) logique (Prolog, ...) Exécuter un programme c'est construire une preuve et en extraire de l'information.

programme	$\equiv$	**théorie du premier ordre**
exécution d'un programme	$\equiv$	**recherche d'une preuve**

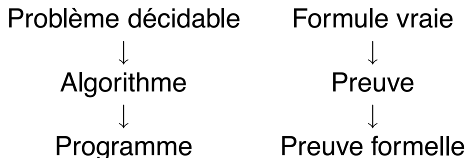
Programme P	Requête R
$\{\forall \overrightarrow{x}((A_1 \wedge \cdots \wedge A_n) \Rightarrow A)\}$	$\exists \overrightarrow{x}\ (B_1 \wedge \cdots \wedge B_k)$

Exécution : soumission d'une requête R étant donné un programme P ... construction d'une preuve de $P \vdash \exists \overrightarrow{x}\ (B_1 \wedge \cdots \wedge B_k)$ en utilisant la résolution, et extraction de cette preuve des valeurs associées à $\overrightarrow{x}$.

Exemple. Addition de deux entiers :

$$\left\{ \begin{array}{l} \forall x\ add(0, x, x) \\ \forall x \forall y \forall z\ add(x, y, z) \Rightarrow add(x+1, y, z+1) \end{array} \right\} \vdash \exists x\ add(2, 3, x) \qquad ...x = 5$$

Logique et Informatique (2.1)



L'isomorphisme de Curry-Howard établit un lien fort, d'une part, entre preuves et programmes, et d'autre part, entre propositions et types :

$$\begin{array}{rcccl} \textbf{preuve} & \equiv & \textbf{programme} & \equiv & \lambda\textbf{-terme} \\ \textbf{proposition} & \equiv & \textbf{spécification} & \equiv & \textbf{type} \end{array}$$

Logique et Informatique (2.2)

Programmation fonctionnelle (OCAML, ...)

preuve	$\equiv$	**programme**
proposition	$\equiv$	**type (spécification)**

Exécuter un programme c'est normaliser (une preuve, un terme).
Exemple. Programme d'addition de deux entiers = preuve (constructive) de la formule $\forall x \forall y \exists z \; add(x, y, z)$.
Remarque. Il existe des systèmes dans lesquels on peut prouver formellement la formule :

$$\forall x \; P(x) \Rightarrow \exists y \; Q(x, y)$$

Le système extrait alors automatiquement de la preuve un programme "certifié" qui, pour tout x vérifiant $P(x)$ calcule un élément y tel que $Q(x, y)$.

... écrire une preuve c'est écrire un programme !

Preuves et Calculs

- Systèmes d'inférence - Induction
- λ-calcul non typé
- λ-calcul typé
- Logique minimale
- Isomorphisme de Curry-Howard
- Système Coq
- Extensions – Logique intuitionniste propositionnelle

Systèmes d'inférence (1)

Permet la *définition* d'ensembles, de relations, ...

$\mathcal{J}$: ensemble de *jugements*

Exemples.

$$\begin{array}{rcl}\mathcal{J}_1 & = & \{\text{suite finie de symboles } \in \{0, S, (,)\}\} \\ & = & \{0, S(0, S(S(0)), (S, 000S, (S)),), \cdots\} \\ \mathcal{J}_2 & = & \{n_1 \leq n_2 \mid n_1, n_2 \in \mathbb{N}\} \\ & = & \{0 \leq S(0), S(S(0)) \leq S(S(S(S(S(0))))), S(S(0)) \leq 0, \cdots\}\end{array}$$

$\Phi[\mathcal{J}]$: *système d'inférence* = ensemble de *règles* portant sur des jugements de $\mathcal{J}$:

$$(R)\frac{j_1 \quad j_2 \quad \cdots \quad j_n}{j}$$

$\{j_1, \cdots, j_n\}$: *prémisses* qui doivent être “satisfaites” pour que la *conclusion* j le soit.

axiome ($n = 0$) : $(R)\frac{}{j}$ Le jugement j est toujours satisfait.

Systèmes d'inférence (2)

Exemples :

- Définition de l'ensemble $\mathbb{N}$:

$$\Phi_1[\mathcal{J}_1] = \left\{ (N_1)\frac{}{0}, (N_2)\frac{n}{S(n)} \right\}$$

- Définition de la relation $\leq$:

$$\Phi_2[\mathcal{J}_2] = \left\{ (L_1)\frac{}{n \leq n}, (L_2)\frac{n \leq m}{n \leq S(m)} \right\}$$

- Règles du calcul des séquents

$$(Hyp)\frac{}{\Gamma, A \vdash A} \quad (E_\Rightarrow)\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \quad (I_\Rightarrow)\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$$

Arbres d'inférence (de preuve, de dérivation) – Théorèmes

Un arbre d'inférence d'un jugement $j \in \mathcal{J}$ pour un système d'inférence $\Phi[\mathcal{J}]$ est un arbre fini dont la racine est j et tel que pour chaque noeud j_k dont les fils sont j_{k_1}, j_{k_2}, ... , j_{k_n}, il existe une règle de $\Phi[\mathcal{J}]$:

$$(r)\frac{j_{k_1} \quad j_{k_2} \quad \cdots \quad j_{k_n}}{j_k}$$

Les feuilles ... ne peuvent être obtenues qu'à partir des axiomes.
$\Phi[\mathcal{J}]$ caractérise un ensemble de théorèmes $\mathrm{Th}(\Phi[\mathcal{J}]) \subseteq \mathcal{J}$ contenant les jugements qui admettent un arbre d'inférence.

$j \in \mathcal{J}$ est un théorème s'il existe dans $\Phi[\mathcal{J}]$ une règle : $(R)\frac{}{j}$ ou :

$$(R)\frac{j_1 \quad j_2 \quad \cdots \quad j_n}{j}$$

telle que chaque j_i $(1 \leq i \leq n)$ soit un théorème.

Arbres d'inférence : Exemples

- $S(S(0)) \in \mathrm{Th}(\Phi_1[\mathcal{J}_1]) = \mathbb{N}$ mais $0S() \notin \mathrm{Th}(\Phi_1[\mathcal{J}_1])$

- $S(0) \leq S(S(S(0))) \in \mathrm{Th}(\Phi_2[\mathcal{J}_2])$ mais $S(S(0)) \leq 0 \notin \mathrm{Th}(\Phi_2[\mathcal{J}_2])$

Induction

Prouver par induction une propriété P sur tous les éléments de l'ensemble $\mathrm{Th}(\Phi[\mathcal{J}])$,
c'est prouver que pour toute règle de $\Phi[\mathcal{J}]$, si chacune des prémisses satisfait P (hypothèse d'induction), alors la conclusion satisfait aussi P.

Théorème

$$\text{Si } \left(\forall \frac{j_1 \quad \cdots \quad j_n}{j} \in \Phi[\mathcal{J}] \; (\forall k \in \{j_1, \cdots, j_n\} \; P(k)) \text{ implique } P(j)\right)$$
$$\text{alors } \forall x \in \mathrm{Th}(\Phi[\mathcal{J}]) \; P(x).$$

Exemples/Exercices.

• Induction sur $\mathbb{N}$: Si $P(0)$ et si pour tout $n \in \mathbb{N}$, $P(n) \Rightarrow P(S(n))$, alors $\forall n \in \mathbb{N} \; P(n)$.

$\forall n \in \mathbb{N} \quad 0 \leq n$

• Induction sur la dérivation de $n \leq m$: Si pour tout $n \in \mathbb{N}$, $P(n \leq n)$ et si pour tout $n, m \in \mathbb{N}$, $P(n \leq m) \Rightarrow P(n \leq S(m))$, alors pour tout $n, m \in \mathbb{N}$, $P(n \leq m)$.

$\forall n, m \in \mathbb{N} \quad n \leq m \Rightarrow S(n) \leq S(m)$

λ-calcul non typé

Fonctions $f : A \rightarrow B$

• Représentation en extension : sous-ensemble de $A \times B$

$$\{(x, f(x)) \mid x \in A\} \subseteq A \times B$$

Cas particulier d'une relation.

Différence en relation et fonction ? pour tout $x \in A$, il existe un unique $y = f(x) \in B$.

Représentation inadaptée à l'informatique

• Représentation en intention : règles de calcul $x \mapsto f(x)$

Le λ-calcul, introduit dans les années 1930 par A. CHURCH, permet une définition calculatoire de la notion de fonction

λ-termes

Termes du λ-calcul non typé

Ensemble Λ défini inductivement :

- tout symbole de variable est un terme
- si t_1 et t_2 sont des termes, alors $(t_1\ t_2)$ est un terme
  application de t_1 à l'argument t_2
- si x est un symbole de variable et t un terme, alors $\lambda x.t$ est un terme
  abstraction (fonction qui à x associe t)

V : ensemble de symboles de variable

$$(\mathrm{Var})\frac{}{x}(x \in V) \qquad (\mathrm{App})\frac{t_1 \quad t_2}{(t_1\ t_2)} \qquad (\mathrm{Abs})\frac{t}{\lambda x.t}(x \in V)$$

Exemples de λ-termes

- $I \triangleq \lambda x.x$: fonction identité
- $((\lambda x.x)\ (\lambda x.x))$: application de la fonction identité à elle-même
- $\lambda f.\lambda g.\lambda x.(f\ (g\ x))$: composition de fonctions
- $\Delta \triangleq \lambda x.(x\ x)$
- $K \triangleq \lambda x.\lambda y.x$
- $S \triangleq \lambda x.\lambda y.\lambda z.((x\ z)\ (y\ z))$

Notations

- l'application est associative à gauche
  $t_1\ t_2\ t_3 \cdots t_n \triangleq (((t_1\ t_2)\ t_3) \cdots t_n)$
- $\lambda x_1, x_2, \cdots, x_n.t \triangleq \lambda x_1.\lambda x_2. \cdots .\lambda x_n.t$

Arbre de syntaxe abstraite de S

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x}\,(\text{Var}) \quad \dfrac{}{z}\,(\text{Var})}{(x\ z)}\,(\text{App}) \quad \dfrac{\dfrac{}{y}\,(\text{Var}) \quad \dfrac{}{z}\,(\text{Var})}{(y\ z)}\,(\text{App})}{((x\ z)\ (y\ z))}\,(\text{App})}{\lambda z.((x\ z)\ (y\ z))}\,(\text{Abs})}{\lambda y.\lambda z.((x\ z)\ (y\ z))}\,(\text{Abs})}{\lambda x.\lambda y.\lambda z.((x\ z)\ (y\ z))}\,(\text{Abs})
$$

Variables libres – Variables liées (Rappels)

Occurrence de variable liée :

- dans une formule logique : x est liée dans $\forall x\, p(x, y)$
- dans un programme OCaml : x est liée dans `function x -> x + y`
- dans une "formule mathématique" : x est liée dans $\int_0^1 (x + y)dx$
- dans un programme OCaml : $\texttt{let}\ x = \underbrace{x}_{\text{occurrence libre}} + 1\ \texttt{in}\ \underbrace{x}_{\text{occurrence liée}} + 2$
- dans une formule logique : $\forall y((\exists x p(\underbrace{x}_{\text{occurrence liée}}, y)) \vee q(\underbrace{x}_{\text{occurrence libre}}, y))$

Variables libres

Tout comme $\forall$ et $\exists$ pour les formules logiques ...λ est un lieur (*binder*) de variable pour les λ-termes.

$$\mathcal{F} : \Lambda \to \wp(V)$$

$$\mathcal{F}(t) \stackrel{\triangle}{=} \begin{cases} \{x\} & \text{si } t \stackrel{\triangle}{=} x \\ \mathcal{F}(t_1) \cup \mathcal{F}(t_2) & \text{si } t \stackrel{\triangle}{=} (t_1\ t_2) \\ \mathcal{F}(t_0) \setminus \{x\} & \text{si } t \stackrel{\triangle}{=} \lambda x.t_0 \end{cases}$$

Exemple

$$\lambda y.(\lambda x.(y \underbrace{x}_{\text{liée}}) \underbrace{x}_{\text{libre}})$$

Les occurrences libres de x dans t sont liées dans $\lambda x.t$.

Variables libres : Exemple

$$\mathcal{F}(\lambda x.\lambda y.((\lambda z.y)\ (z\ x)))$$
$$=$$

Substitutions

$t_1[x := t_2]$: remplacer x par t_2 dans t_1
Exemple : $(\lambda x.(y\ x))[y := \lambda z.z] = \lambda x.((\lambda z.z)\ x)$
Remplacer toutes les occurrences de x dans t_1 ?
Non.
$\lambda x.(y\ x)$ et $\lambda w.(y\ w)$ représentent la même fonction, on souhaite donc que $(\lambda x.(y\ x))[x := z]$ et $(\lambda w.(y\ w))[x := z]$ représentent aussi la même fonction :

$$(\lambda x.(y\ x))[x := z] = \lambda x.(y\ x) \quad (\lambda w.(y\ w))[x := z] = \lambda w.(y\ w)$$
$$x \notin \mathcal{F}(\lambda x.(y\ x))$$

$t_1[x := t_2]$: remplacer toutes les **occurrences libres** de x dans t_1 par t_2

$$(\lambda x.t_1)[x := t_2] = \lambda x.t_1$$

... de la même manière $(\forall x\ P(x))[x := t] = \forall x\ P(x)$

Substitution d'une variable (rappels)

Substituer toutes les occurrences de x par t dans la formule logique φ ? Non ... uniquement les occurrences libres de x ... toutefois, ceci n'est correct que si les variables apparaissant dans t ne sont pas liées dans φ : phénomène de capture de variable.

Exemple 1. Si $f(x) = \int_0^1 (x+y)dy$, calculer $f(y)$, ce n'est pas calculer $\int_0^1 (y+y)dy$ mais c'est calculer $\int_0^1 (y+z)dz$ ... on renomme l'occurrence liée de y par une variable fraîche z.

Exemple 2. Substituer y par le terme $f(x)$ dans $\forall x p(x,y)$.

Un simple remplacement de y par $f(x)$ conduit $\forall x p(x, f(x))$ ce qui incorrect puisque l'occurrence de x dans le terme $f(x)$ devient liée.

Il faut donc, avant d'effectuer la substitution, renommer les variables liées de φ qui apparaissent dans t.

Sur l'exemple, ce renommage conduit à $\forall z p(z,y)$ (qui est logiquement équivalente à la formule $\forall x p(x,y)$) sur laquelle on peut effectuer la substitution de y par $f(x)$ pour obtenir $\forall z p(z, f(x))$.

Substitutions : Capture de variable

$\lambda x.(z\ x)$ et $\lambda y.(z\ y)$ représentent la même fonction, on souhaite donc que $(\lambda x.(z\ x))[z := x]$ et $(\lambda y.(z\ y))[z := x]$ représentent aussi la même fonction : $\lambda y.(x\ y)$.
Pour pouvoir appliquer la substitution $[x := t_2]$ au λ-terme t_1, il faut renommer les variables liées de t_1 "en dehors" des variables libres de t_2 ... inutile si x n'est pas libre dans t_1.

$(\lambda x.t_1)[y := t_2] = \lambda x.t_1[y := t_2]$	si $x \notin \mathcal{F}(t_2) \vee y \notin \mathcal{F}(t_1)$
$(\lambda x.t_1)[y := t_2] = \lambda z.t_1[x := z][y := t_2]$	si $x \in \mathcal{F}(t_2) \wedge y \in \mathcal{F}(t_1)$
renommage de x en z	z variable "fraîche"

Application d'une substitution

$x[x := t] = t$	
$y[x := t] = y$	si $x \neq y$
$(t_1\ t_2)[x := t] = (t_1[x := t]\ t_2[x := t])$	
$(\lambda x.t_1)[x := t_2] = \lambda x.t_1$	
$(\lambda x.t_1)[y := t_2] = \lambda x.t_1[y := t_2]$	si $x \notin \mathcal{F}(t_2) \vee y \notin \mathcal{F}(t_1)$
$(\lambda x.t_1)[y := t_2] = \lambda z.t_1[x := z][y := t_2]$	si $x \in \mathcal{F}(t_2) \wedge y \in \mathcal{F}(t_1)$
renommage de x en z	z variable “fraîche”

Application d'une substitution : Exemple

$$((\lambda x.((x\ y)\ z) \quad \lambda y.((x\ y)\ z)) \quad \lambda z.((x\ y)\ z))[x := y]$$

$$=$$

α-équivalence

Tout comme $\forall x \; P(x)$ et $\forall y \; P(y)$ sont des formules logiques “équivalentes”, $\lambda x.x$ et $\lambda y.y$ sont des λ-termes α-équivalents.

$$(R_{\alpha_1})\frac{}{t =_\alpha t} \qquad (R_{\alpha_2})\frac{y \notin \mathcal{F}(t)}{\lambda x.t =_\alpha \lambda y.t[x := y]}$$

$$(R_{\alpha_3})\frac{t =_\alpha t'}{\lambda x.t =_\alpha \lambda x.t'} \qquad (R_{\alpha_4})\frac{t =_\alpha t'}{(t \; t'') =_\alpha (t' \; t'')}$$

$$(R_{\alpha_5})\frac{t =_\alpha t'}{(t'' \; t) =_\alpha (t'' \; t')} \qquad (R_{\alpha_6})\frac{t =_\alpha t'}{t' =_\alpha t}$$

$$(R_{\alpha_7})\frac{t =_\alpha t' \quad t' =_\alpha t''}{t =_\alpha t''}$$

Ex : $\lambda x.(x + y) \neq_\alpha \lambda y.(y + y)$

Proposition : $=_\alpha$ est une relation d’équivalence.

α-équivalence : Exemple

$(\lambda z.z) \quad (\lambda x.(y\ x)) =_\alpha (\lambda w.w) \quad (\lambda z.(y\ z))$?

Calcul : β-réduction

Calculer (le résultat de) l'application de la fonction $\lambda x.t$ à l'argument t', c'est calculer le terme t dans lequel on a substitué x par t'.

$$(C_{Rad})\frac{}{(\lambda x.t)\ t' \rightarrow_\beta t[x := t']}$$

$$(C_\lambda)\frac{t \rightarrow_\beta t'}{\lambda x.t \rightarrow_\beta \lambda x.t'} \qquad (C_{App_1})\frac{t \rightarrow_\beta t'}{(t\ t'') \rightarrow_\beta (t'\ t'')} \qquad (C_{App_2})\frac{t \rightarrow_\beta t'}{(t''\ t) \rightarrow_\beta (t''\ t')}$$

C_λ, C_{App_1} et C_{App_2} indiquent comment se propagent les réductions à l'intérieur des termes : passage au contexte

β-réduction : Exemple

$$
\begin{array}{lll}
 & & (\boldsymbol{S}\ \boldsymbol{K})\ \boldsymbol{K} \\
 & \triangleq & (\lambda x.\lambda y.\lambda z.((x\ z)\ (y\ z))\quad \boldsymbol{K})\quad \boldsymbol{K} \\
C_{Rad}, C_{App_1} & \rightarrow_\beta & (\lambda y.\lambda z.((\boldsymbol{K}\ z)\ (y\ z)))\quad \boldsymbol{K} \\
C_{Rad} & \rightarrow_\beta & \lambda z.((\boldsymbol{K}\ z)\ (\boldsymbol{K}\ z)) \\
 & \triangleq & \lambda z.((\lambda x.\lambda y.x\ z)\ (\boldsymbol{K}\ z)) \\
C_{Rad}, C_\lambda, C_{App_1} & \rightarrow_\beta & \lambda z.(\lambda y.z\ (\boldsymbol{K}\ z)) \\
C_{Rad}, C_\lambda & \rightarrow_\beta & \lambda z.z \\
 & \triangleq & \boldsymbol{I}
\end{array}
$$

β-réductions

S'il existe une suite finie de transitions :

$$t_0 \rightarrow_\beta t_1 \rightarrow_\beta t_2 \rightarrow_\beta \cdots \rightarrow_\beta t_n$$

alors $t_0 \rightarrow_\beta^\star t_n$.
$\rightarrow_\beta^\star$ est la fermeture réflexive transitive de $\rightarrow_\beta$.

$$(C^\star)\frac{t \rightarrow_\beta t'}{t \rightarrow_\beta^\star t'} \quad (C^\star_{Refl})\frac{}{t \rightarrow_\beta^\star t} \quad (C^\star_{Trans})\frac{t \rightarrow_\beta^\star t' \quad t' \rightarrow_\beta^\star t''}{t \rightarrow_\beta^\star t''}$$

t et t' sont β-équivalent s'ils se réduisent en un même terme.
$=_\beta$ est la fermeture réflexive symétrique transitive de $\rightarrow_\beta$.

$$(R^\beta)\frac{t \rightarrow_\beta t'}{t =_\beta t'} \quad (R^\beta_{Refl})\frac{}{t =_\beta t} \quad (R^\beta_{Sym})\frac{t =_\beta t'}{t' =_\beta t} \quad (R^\beta_{Trans})\frac{t =_\beta t' \quad t' =_\beta t''}{t =_\beta t''}$$

Formes normales

• Un radical (β-redex) est un λ-terme de la forme $((\lambda x.t)\ t')$.
• Un λ-terme est en forme normale si aucun de ses sous-termes ne contient de radical (on ne peut plus le réduire ...).
• Mettre en forme normale – normaliser – un λ-terme t, c'est trouver un un λ-terme t' en forme normale tel que $t \rightarrow_\beta^\star t'$.

Exemples

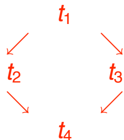
$\boldsymbol{I} \triangleq \lambda x.x$ est une forme normale de $(\boldsymbol{S}\ \boldsymbol{K})\ \boldsymbol{K}$.

$\Omega \triangleq (\Delta\ \Delta)$ n'admet pas de forme normale :

$$(\lambda x.(x\ x)\quad \lambda x.(x\ x)) \rightarrow_\beta (\lambda x.(x\ x)\quad \lambda x.(x\ x)) \rightarrow_\beta \cdots$$

Confluence

Théorème [CHURCH-ROSSER, 1936] Etant donnés trois termes t_1, t_2, et t_3, si $t_1 \rightarrow_\beta^\star t_2$ et $t_1 \rightarrow_\beta^\star t_3$, alors il existe un terme t_4 tel que $t_2 \rightarrow_\beta^\star t_4$ et $t_3 \rightarrow_\beta^\star t_4$.



Corollaire Si t est normalisable, alors sa forme normale est unique.
PREUVE S'il existe 2 formes normales distinctes t_1 et t_2 d'un terme t, alors, par confluence, il existe un terme t' tel que $t_1 \rightarrow_\beta^\star t'$ et $t_2 \rightarrow_\beta^\star t'$. Or, t_1 et t_2 étant en forme normale, il vient $t_1 = t'$ et $t_2 = t'$ et donc $t_1 = t_2$ ce qui est contradictoire.
Remarque Les formes normales disjonctives et conjonctives d'une formule logique ne sont pas uniques.

Programmer en λ-calcul (1)

$$\mathrm{T} \triangleq \boldsymbol{K} \triangleq \lambda x.\lambda y.x$$
$$\mathrm{F} \triangleq \lambda x.\lambda y.y$$
$$\mathrm{and} \triangleq \lambda x.\lambda y.((x\ y)\ \mathrm{F})$$
$$\mathrm{or} \triangleq \lambda x.\lambda y.((x\ \mathrm{T})\ y)$$
$$\mathrm{not} \triangleq \lambda z.\lambda x.\lambda y.((z\ y)\ x)$$
$$\mathrm{if} \triangleq \lambda z.\lambda x.\lambda y.((z\ x)\ y)$$
$$\mathrm{nil} \triangleq \lambda x.\mathrm{T}$$
$$\mathrm{cons} \triangleq \lambda x.\lambda y.\lambda z.((z\ x)\ y)$$
$$\mathrm{car} \triangleq \lambda x.(x\ \mathrm{T})$$
$$\mathrm{cdr} \triangleq \lambda x.(x\ \mathrm{F})$$
$$\mathrm{empty} \triangleq \lambda x.(x\ (\lambda y.(\lambda z.\mathrm{F})))$$

Programmer en λ-calcul (2)

$$\begin{array}{l}
\text{and T } y \rightarrow^{\star}_{\beta} y \\
\text{and F } y \rightarrow^{\star}_{\beta} \text{F} \\
\text{or T } y \rightarrow^{\star}_{\beta} \text{T} \\
\text{or F } y \rightarrow^{\star}_{\beta} y \\
(\text{car } ((\text{cons } a)\ b)) \rightarrow^{\star}_{\beta} a \\
(\text{empty } (\text{cdr } ((\text{cons } a)\ \text{nil}))) \rightarrow^{\star}_{\beta} \text{T}
\end{array}$$

Stratégies de réduction : Appel par nom

Appel par nom : réduction du radical le plus à gauche et n'étant pas sous la portée d'un λ dans un terme. L'argument d'une fonction n'est pas évalué avant d'être "transmis".

$$(\lambda x.((y\ x)\ (z\ x)))\quad(\lambda z.z\ w) \rightarrow_\beta ((y\ (\lambda z.z\ w))\quad(z\ (\lambda z.z\ w))) \rightarrow_\beta (y\ w)\ (z\ w)$$

$$\lambda x.y\quad(\lambda z.z\ w) \rightarrow_\beta y$$

- Avantages
  - Si le corps de la fonction n'utilise pas l'argument, alors cet argument n'est pas calculé.
  - **Si un terme admet une forme normale, alors la réduction en appel par nom termine.**
- Inconvénient : Si l'argument est utilisé plusieurs fois dans le corps de la fonction, alors il est calculé plusieurs fois.

Stratégies de réduction : Appel par valeur

Appel par valeur : réduction du radical $(\lambda x.t\ t')$ uniquement si t' est en forme normale.

$$\frac{(\lambda x.((y\ x)\ (z\ x)))\ (\lambda z.z\ w) \to_\beta (\lambda x.((y\ x)\ (z\ x)))\ w \to_\beta (y\ w)\ (z\ w)}{\lambda x.y \quad (\lambda z.z\ w) \to_\beta \lambda x.y \quad w \to_\beta y}$$

- Avantage : Si l'argument est utilisé plusieurs fois dans le corps de la fonction, alors il n'est calculé qu'une seule fois.
- Inconvénient : Si le corps de la fonction n'utilise pas l'argument, alors cet argument est tout de même calculé ... et si ce calcul ne termine pas, la forme normale du radical, si elle existe, ne peut pas être calculée non plus.

$$\lambda x.y \quad (\lambda x.(x\ x) \quad \lambda x.(x\ x)) \to_\beta \lambda x.y \quad (\lambda x.(x\ x) \quad \lambda x.(x\ x)) \to_\beta \cdots$$

Or la forme normale de $\lambda x.y \quad (\lambda x.(x\ x) \quad \lambda x.(x\ x))$ existe, c'est y !

λ-calcul typé

• Quelle fonction représente le λ-terme $\Delta \triangleq \lambda x.(x\ x)$? Quel est son domaine ? Quel est son codomaine ?
L'interprétation ensembliste de Δ n'est pas claire ...
Le λ-calcul typé permet de se rapprocher de la définition ensembliste d'une fonction : attribuer un type à certains λ-termes (les λ-termes typables) et rejeter les autres.
• Les termes du λ-calcul non typé ne sont pas tous normalisables ($(\Delta\ \Delta)$ par exemple) ... les λ-termes typables seront tous normalisables.

Typage

Typer un λ-terme – lui ajouter une information de type – c'est déjà lui donner une certaine sémantique.

Le typage permet de garantir certaines propriétés : si un terme (i.e., un programme) est typable, alors sa réduction (i.e., son exécution) ne provoquera pas d'erreur de typage.

cf. module APS (Analyse de Programmes et Sémantique)
M1 - deuxième semestre

Le typage est une interprétation abstraite des λ-termes : un type est une valeur abstraite représentant tous les λ-termes de ce type (par exemple la valeur abstraite $\mathsf{nat} \rightarrow \mathsf{nat}$ désigne toutes les fonctions des entiers dans les entiers).

cf. module InAbs (Interprétation abstraite et Analyse Statique)
M2 – premier semestre

$\lambda_{\rightarrow}$-calcul

Langage de types $\mathcal{T}$: Définition inductive des expressions de type

- Les types de base (types constants) $\top \in T_C$ sont des types.
- Si τ_1 et τ_2 sont des types alors $\tau_1 \rightarrow \tau_2$ est un type.

$$(T_1)\frac{}{\top}(\top \in T_C) \quad (T_2)\frac{\tau_1 \quad \tau_2}{\tau_1 \rightarrow \tau_2}$$

Il s'agit de l'ensemble $T_{T_C \cup \{\rightarrow\}}[\emptyset]$ de termes.

Notation : $\tau_1 \rightarrow \tau_2 \rightarrow \cdots \rightarrow \tau_{n-1} \rightarrow \tau_n \stackrel{\triangle}{=} (\tau_1 \rightarrow (\tau_2 \rightarrow \cdots \rightarrow (\tau_{n-1} \rightarrow \tau_n)))$
Type de $\lambda x.x$? non unique ! $\alpha \rightarrow \alpha$ où α est une variable de type désignant n'importe quel type.

```
# function x -> x;;
- : 'a -> 'a = <fun>
```

Contexte de typage

Un *contexte de typage* Γ est un ensemble fini de couples de la forme (x, τ) où $x \in V$ est une variable et $\tau \in \mathcal{T}$ est le type associé à cette variable.

$$\Gamma = \lfloor (x_1, \tau_1), (x_2, \tau_2), \cdots, (x_n, \tau_n) \rfloor = (x_1, \tau_1) \oplus \lfloor (x_2, \tau_2), \cdots, (x_n, \tau_n) \rfloor$$

Γ peut être vu comme une fonction partielle de V dans $\mathcal{T}$

$$\Gamma = (x_1, \tau) \oplus \Gamma'$$
$$\Gamma(x) = \begin{cases} \tau & \text{si } x = x_1 \\ \Gamma'(x) & \text{sinon} \end{cases}$$

Règles de typage

Jugements de typage : $\Gamma \vdash t : \tau$
Dans le contexte Γ, le λ-terme t a pour type τ.

$$(\textsc{Var})\frac{}{(x,\tau),\Gamma \vdash x : \tau} \qquad (\textsc{App})\frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash (t_1\ t_2) : \tau_2}$$

$$(\textsc{Abs})\frac{(x,\tau_1),\Gamma \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \rightarrow \tau_2}$$

Typage : Exemple (1)

Typage de $\boldsymbol{K} \triangleq \lambda x.\lambda y.x$

$$
(\textsc{Abs})\dfrac{(\textsc{Abs})\dfrac{(\textsc{Var})\dfrac{}{(y,\tau_2),(x,\tau_1) \vdash x : \tau_1}}{(x,\tau_1) \vdash \lambda y.x : \tau_2 \rightarrow \tau_1}}{\vdash \lambda x.\lambda y.x : \tau_1 \rightarrow (\tau_2 \rightarrow \tau_1)}
$$

```
# function x -> function y -> x;;
- : 'a -> 'b -> 'a = <fun>
```

Typage : Exemple (2)

Typage de $\boldsymbol{S} \triangleq \lambda x.\lambda y.\lambda z.((x\, z)\, (y\, z))$

```
# function x -> function y -> function z -> ((x z) (y z));;
- : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```

Typage : Exemple (3)

Typage de $\Delta \triangleq \lambda x.(x\ x)$
Si l'on suppose que τ_1 est le type de x et τ_2 est le type de $(x\ x)$, alors il faut que $\tau_1 = \tau_1 \to \tau_2$ puisque l'on applique x à x.
Or $\tau_1 = \tau_1 \to \tau_2$ n'admet pas de solution : ces deux types ne sont pas unifiables.
Δ n'est pas typable.

```
# function x -> (x x);;
This expression has type 'a -> 'b but is here used with type 'a
```

Remarque : c'est le "test d'occurrence" de l'algorithme d'unification
$x = f(x)$ n'admet pas de solution dans l'ensemble des termes finis ... mais si l'on autorise les termes infinis alors $f^\omega \triangleq f(f(f(\cdots$ est solution de $x = f(x)$.

Propriétés du $\lambda_{\rightarrow}$-calcul

- Tout λ-terme typable admet une forme normale unique.
- (*Subject Reduction*) Si $\Gamma \vdash t : \tau$ et $t \rightarrow_{\beta} t'$, alors $\Gamma \vdash t' : \tau$
- (Corollaire) Si t est de type τ, alors sa forme normale est aussi de type τ.

Logique minimale propositionnelle

Formules de la logique minimale propositionnelle :

- tout symbole propositionnel est une formule
- si φ_1 et φ_2 sont des formules, alors $\varphi_1 \Rightarrow \varphi_2$ est une formule.

$$(L_1)\frac{}{p}(p \in \mathcal{P}) \quad (L_2)\frac{\varphi_1 \quad \varphi_2}{\varphi_1 \Rightarrow \varphi_2}$$

Logique minimale propositionnelle : Règles de déduction

Jugements : $\Gamma \vdash \varphi$
La formule φ peut être déduite des hypothèses du contexte Γ.

$$(Hyp)\frac{}{A, \Gamma \vdash A} \qquad (I_{\Rightarrow})\frac{A, \Gamma \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad (E_{\Rightarrow})\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$(I_{\Rightarrow})$: règle d'introduction du connecteur $\Rightarrow$.
$(E_{\Rightarrow})$: règle d'élimination du connecteur $\Rightarrow$.

Déduction : Exemple (1)

$\vdash A \Rightarrow (B \Rightarrow A)$

$$
(I_{\Rightarrow})\dfrac{(I_{\Rightarrow})\dfrac{(Hyp)\dfrac{}{A, B \vdash A}}{A \vdash B \Rightarrow A}}{\vdash A \Rightarrow (B \Rightarrow A)}
$$

Déduction : Exemple (2)

$$\vdash (A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$$

Coupure

Dans un arbre de déduction, une coupure est l'application successive d'une règle d'introduction d'un connecteur logique et d'une règle d'élimination portant sur le connecteur introduit.

$$
(E_{\Rightarrow})\dfrac{(I_{\Rightarrow})\dfrac{(_)\dfrac{\vdots}{A, \Gamma \vdash B}}{\Gamma \vdash A \Rightarrow B} \quad (_)\dfrac{\vdots}{\Gamma \vdash A}}{\Gamma \vdash B}
$$

Elimination des coupures

Obtenir une déduction sans coupure à partir de :

$$(E_{\Rightarrow})\dfrac{(I_{\Rightarrow})\dfrac{(_)\dfrac{\vdots}{A, \Gamma \vdash B}}{\Gamma \vdash A \Rightarrow B} \quad (_)\dfrac{\vdots}{\Gamma \vdash A}}{\Gamma \vdash B}$$

- Suppression des occurrences de A dans tous les contextes d'hypothèse de la déduction : $(_)\dfrac{\vdots}{A, \Gamma \vdash B}$
- Remplacement dans cette déduction de toutes les applications de la règle (Hyp) sur A : $(Hyp)\dfrac{}{A, \Gamma, \Delta \vdash A}$ par la déduction : $(_)\dfrac{\vdots}{\Gamma, \Delta \vdash A}$ obtenue en ajoutant les hypothèses de Δ dans tous les contextes de la déduction : $(_)\dfrac{\vdots}{\Gamma \vdash A}$

Elimination des coupures : Exemple

$\Gamma = \{D \Rightarrow A, D, A \Rightarrow (D \Rightarrow B)\}$

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{}{A, \Gamma \vdash A \Rightarrow (D \Rightarrow B)}(Hyp)
        \quad
        \dfrac{}{A, \Gamma \vdash A}(Hyp)
      }{A, \Gamma \vdash D \Rightarrow B}(E_{\Rightarrow})
      \quad
      \dfrac{}{A, \Gamma \vdash D}(Hyp)
    }{A, \Gamma \vdash B}(E_{\Rightarrow})
  }{\Gamma \vdash A \Rightarrow B}(I_{\Rightarrow})
  \quad
  \dfrac{
    \dfrac{}{\Gamma \vdash D \Rightarrow A}(Hyp)
    \quad
    \dfrac{}{\Gamma \vdash D}(Hyp)
  }{\Gamma \vdash A}(E_{\Rightarrow})
}{\Gamma \vdash B}(E_{\Rightarrow})
$$

$$
\dfrac{
  \dfrac{
    \dfrac{}{\Gamma \vdash A \Rightarrow (D \Rightarrow B)}(Hyp)
    \quad
    \dfrac{
      \dfrac{}{\Gamma \vdash D \Rightarrow A}(Hyp)
      \quad
      \dfrac{}{\Gamma \vdash D}(Hyp)
    }{\Gamma \vdash A}(E_{\Rightarrow})
  }{\Gamma \vdash D \Rightarrow B}(E_{\Rightarrow})
  \quad
  \dfrac{}{\Gamma \vdash D}(Hyp)
}{\Gamma \vdash B}(E_{\Rightarrow})
$$

Déduction et Typage

Les règles de typage (VAR), (ABS) et (APP) correspondent exactement aux règles de déduction de la logique minimale (*Hyp*), ($I_{\Rightarrow}$) et ($E_{\Rightarrow}$) "décorées" par des λ-termes.

$$(Hyp)\frac{}{A, \Gamma \vdash A} \qquad (\text{VAR})\frac{}{(x, \tau), \Gamma \vdash x : \tau}$$

$$(I_{\Rightarrow})\frac{A, \Gamma \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad (\text{ABS})\frac{(x, \tau_1), \Gamma \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \rightarrow \tau_2}$$

$$(E_{\Rightarrow})\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \qquad (\text{APP})\frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash (t_1\ t_2) : \tau_2}$$

Déduction et Typage : Exemple

En “décorant” l’arbre de déduction de $\vdash A \Rightarrow (B \Rightarrow A)$, on obtient l’arbre de typage de $\boldsymbol{K}$:

$$\vdash \lambda x.\lambda y.x : \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$$

En “décorant” l’arbre de déduction de
$\vdash (A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$, on obtient l’arbre de typage de $\boldsymbol{S}$:

$$\vdash \lambda x.\lambda y.\lambda z.((x\,z)\,(y\,z)) : (\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)) \rightarrow ((\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_3))$$

C’est l’isomorphisme de Curry-Howard.

Isomorphisme de Curry-Howard

- Il existe une bijection $\mathbb{P}$ de l'ensemble des formules vers l'ensemble des types.
- Il existe une bijection $\mathbb{T}$ de l'ensemble des preuves vers l'ensemble des λ-termes typés.

preuve	$\equiv$	**λ-terme**	$\equiv$	**programme**
proposition	$\equiv$	**type**	$\equiv$	**spécification**

Preuves et algorithmes admettent une représentation uniforme : *"programmer = prouver"*

- Ecrire un programme, c'est prouver, de manière constructive, une proposition.
- Déterminer le type d'une expression, c'est trouver de quelle proposition cette expression est une preuve.

Sémantique de Brouwer-Heyting-Kolmogorov

Une formule logique n'est plus associée à une valeur de vérité ... mais au type de ses preuves.

• Introduction d'un type de base $\mathbb{P}(p)$ pour chaque symbole propositionnel $p \in \mathcal{P}$: $\mathbb{P}(p)$ est le type des preuves de p.

• Une preuve de $\varphi \Rightarrow \psi$ est une fonction qui associe une preuve de ψ à toute preuve de φ :

$$\mathbb{P}(\varphi \Rightarrow \psi) = \mathbb{P}(\varphi) \rightarrow \mathbb{P}(\psi)$$

$\mathbb{P}$ définit une bijection entre l'ensemble des formules et l'ensemble des types.

Construction d'un λ-terme de preuve

Proposition Si $A_1, A_2, \cdots, A_n \vdash A$ admet une preuve, alors il existe un λ-terme t tel que $(x_1, \mathbb{P}(A_1)), (x_2, \mathbb{P}(A_2)), \cdots, (x_n, \mathbb{P}(A_n)) \vdash t : \mathbb{P}(A)$.

PREUVE Par induction sur $A_1, A_2, \cdots, A_n \vdash A$.

Cette preuve est constructive : elle définit un procédé de construction d'un λ-terme $t = \mathbb{T}(\pi)$ de type $\mathbb{P}(A)$ à partir d'une preuve π de $A_1, A_2, \cdots, A_n \vdash A$.

Construction d'une déduction

Proposition Si t est un λ-terme t tel que $(x_1, \tau_1), (x_2, \tau_2), \cdots, (x_n, \tau_n) \vdash t : \tau$, alors $\mathbb{P}^{-1}(\tau_1), \mathbb{P}^{-1}(\tau_2), \cdots, \mathbb{P}^{-1}(\tau_n) \vdash \mathbb{P}^{-1}(\tau)$ admet une preuve.
PREUVE Par induction sur $(x_1, \tau_1), (x_2, \tau_2), \cdots, (x_n, \tau_n) \vdash t : \tau$

Normalisation $\equiv$ Elimination des coupures

Soit π la déduction : $$(E_{\Rightarrow})\dfrac{(I_{\Rightarrow})\dfrac{(_)\dfrac{\vdots}{A,\Gamma \vdash B}}{\Gamma \vdash A \Rightarrow B} \quad (_)\dfrac{\vdots}{\Gamma \vdash A}}{\Gamma \vdash B}$$

$$t_B = \mathbb{T}\left((_)\dfrac{\vdots}{A,\Gamma \vdash B}\right) \quad t_A = \mathbb{T}\left((_)\dfrac{\vdots}{\Gamma \vdash A}\right) \quad t = \mathbb{T}(\pi)$$

$$t = (\lambda x.t_B \quad t_A) \rightarrow_\beta t_B[x := t_A]$$

Or $t_B[x := t_A]$ correspond à une preuve sans coupure ... la propriété de *Subject reduction* garantit qu'il s'agit encore d'une déduction de B.
Puisque pour tout λ-terme typable on sait calculer en temps fini sa forme normale, on sait éliminer les coupures en temps fini dans toute déduction.

Normalisation $\equiv$ Elimination des coupures : Exemple (1)

$\Gamma = \{x : D \Rightarrow A, y : D, z : A \Rightarrow (D \Rightarrow B)\}$

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{}{A, \Gamma \vdash z : A \Rightarrow (D \Rightarrow B)}(Hyp)
        \quad
        \dfrac{}{A, \Gamma \vdash w : A}(Hyp)
      }{A, \Gamma \vdash (z\ w) : D \Rightarrow B}(E_{\Rightarrow})
      \quad
      \dfrac{}{A, \Gamma \vdash y : D}(Hyp)
    }{A, \Gamma \vdash ((z\ w)\ y) : B}(E_{\Rightarrow})
  }{\Gamma \vdash \lambda w.((z\ w)\ y) : A \Rightarrow B}(I_{\Rightarrow})
  \quad
  \dfrac{
    \dfrac{}{\Gamma \vdash x : D \Rightarrow A}(Hyp)
    \quad
    \dfrac{}{\Gamma \vdash y : D}(Hyp)
  }{\Gamma \vdash (x\ y) : A}(E_{\Rightarrow})
}{\Gamma \vdash ((\lambda w.((z\ w)\ y))\ (x\ y)) : B}(E_{\Rightarrow})
$$

$$((\lambda w.((z\ w)\ y))\ (x\ y))$$

Normalisation $\equiv$ Elimination des coupures : Exemple (2)

$$((\lambda w.((z\ w)\ y))\ (x\ y)) \rightarrow_\beta ((z\ w)\ y)[w := (x\ y)] = ((z\ (x\ y))\ y)$$

$((z\ (x\ y))\ y)$ correspond à la déduction de $\Gamma \vdash B$ sans coupure :
$\Gamma = \{x : D \Rightarrow A, y : D, z : A \Rightarrow (D \Rightarrow B)\}$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash z : A \Rightarrow (D \Rightarrow B)}(Hyp) \quad \dfrac{\dfrac{}{\Gamma \vdash x : D \Rightarrow A}(Hyp) \quad \dfrac{}{\Gamma \vdash y : D}(Hyp)}{\Gamma \vdash (x\ y) : A}(E_\Rightarrow)}{\Gamma \vdash (z\ (x\ y)) : D \Rightarrow B}(E_\Rightarrow) \quad \dfrac{}{\Gamma \vdash y : D}(Hyp)}{\Gamma \vdash ((z\ (x\ y))\ y) : B}(E_\Rightarrow)$$

Remarque : Si $((z\ w)\ y)$ contenait plusieurs occurrences de w, il y aurait eu duplication des déductions de $\Gamma \vdash A$ ... la substitution porte sur toutes les occurrences de w.

Logiciels d’aide à la preuve

• Le raisonnement n’est pas réductible au calcul ... la vérification de la validité du raisonnement l’est : la proposition “*p est une preuve de la proposition P*” est vérifiable par le calcul. Muni d’un formalisme adéquat, on peut donc obtenir des programmes de vérification automatique de preuves.
• C’est la décidabilité du typage des λ-termes typés représentant des preuves qui permet la vérification (automatique) de la validité d’une preuve.
• L’écriture des λ-termes peut être assistée : construction de la preuve en fonction d’indications fournissant le raisonnement à mener pour établir le résultat à prouver.
• Utilisation de procédures de démonstration automatique permettant d’établir certaines parties (faciles) d’une preuve.

COQ, ISABELLE, LEGO, HOL, LCF, ALF, PVS, ...

Système Coq

Logiciel développé à l'INRIA : `http://coq.inria.fr`
Coq permet de construire des preuves formelles :
le λ-terme de preuve est construit, de manière incrémentale et interactive, en partant du but initial et en le décomposant suivant des règles logiques à l'aide d'un ensemble de tactiques prédéfinies.
Une tactique est une fonction qui construit une preuve d'un but donné à partir de preuves "élémentaires" de sous-buts : l'application d'une tactique engendre donc les sous-buts qu'il reste à prouver :

$$\frac{\text{contexte } 0}{\text{But}} \xrightarrow{\text{Application d'une tactique}} \frac{\text{contexte } 1}{\text{Sous-but } 1}, \ldots, \frac{\text{contexte } n}{\text{Sous-but } n}$$

Exemple : $\frac{H}{B} \xrightarrow{\texttt{Cut}\ A} \frac{H}{A \rightarrow B}, \frac{H}{A}$
Coq permet l'extraction d'un programme certifié calculant pour tout x vérifiant $P(x)$ un élément $f(x)$ tel que $Q(x, f(x))$ à partir de la preuve de $\forall x\ P(x) \Rightarrow \exists y\ Q(x, y)$.

Termes du $\lambda_{\rightarrow}$-calcul en Coq et en OCaml

λ-calcul	Coq	OCaml
x	x	x
$\lambda x.t : \tau_1 \rightarrow \tau_2$	`fun (`x `:`τ_1`) =>` t	`function` x `->` t
	`fun` x `=>` t	
$(t_1\ t_2)$	`(`t_1 t_2`)`	`(`t_1 t_2`)`

Introduction de paramètres, définition de fonctions

Introduction de paramètres :
`Parameters` $(id_{11} \cdots id_{1n_1} : t_1) \cdots (id_{k1} \cdots id_{kn_k} : t_k)$.

```
Coq < Parameters (E1 E2 : Set).
E1 is assumed
E2 is assumed
```

Introduction d'une definition : `Definition` id := terme.

```
Coq < Definition f12 := fun (f:E1->E2)(x:E1) => (f x).
f12 is defined
Coq < Definition fe := fun (f:E1->E1)(x:E1) => (f x).
fe is defined
Coq < Definition id1 := fun (x:E1) => x.
id1 is defined
```

δ-réduction : remplacer id par terme.

Type et Définition

`Check` terme – `Print` ident

```
Coq < Check E1.
E1: Set
Coq < Check id1.
id1: E1 -> E1
Coq < Check fe.
fe: (E1 -> E1) -> E1 -> E1
Coq < Check (fe id1).
fe id1: E1 -> E1

Coq < Print f12.
f12 = fun (f : E1 -> E2) (x : E1) => f x
       : (E1 -> E2) -> E1 -> E2
```

Une première preuve (1)

$\vdash (A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$

Introduction de 3 variables :

```
Coq < Parameter (A B C: Prop).
A is assumed
B is assumed
C is assumed
```

Rappel. Prouver une proposition P c'est construire un terme de type P ...
Coq vérifie le typage du terme construit.

```
# function x -> function y -> function z -> ((x z) (y z));;
- : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```

```
Coq < Definition S :=
  fun (H1 : A -> (B -> C)) (H2 : A -> B) (H3 : A)
      => ((H1 H3) (H2 H3)).
Coq < S is defined
Coq < Check S.
S : (A -> B -> C) -> (A -> B) -> A -> C
```

Une première preuve (2)

Mode interactif.

```
Coq < Lemma S : (A -> (B -> C)) -> ((A -> B) -> (A -> C)).
1 subgoal
  ============================
   (A -> B -> C) -> (A -> B) -> A -> C

S < exact (fun (H1:A->(B->C))(H2:A->B)(H3:A)
      => ((H1 H3) (H2 H3))).
Proof completed.
S < Save.
exact (fun (H1:A -> B -> C) (H2:A -> B) (H3:A) => H1 H3 (H2 H3)).
S is defined
Coq < Check S.
S : (A -> B -> C) -> (A -> B) -> A -> C
Coq < Print S.
S = fun (H1 : A -> B -> C) (H2 : A -> B) (H3 : A) => H1 H3 (H2 H3
     : (A -> B -> C) -> (A -> B) -> A -> C
```

Une première preuve (3)

Mode interactif.

```
Coq < Lemma S : (A -> (B -> C)) -> ((A -> B) -> (A -> C)).
1 subgoal
  ============================
   (A -> B -> C) -> (A -> B) -> A -> C
S < intro H1.                 (* introduction d'une hypothese *)
1 subgoal
  H1 : A -> B -> C
  ============================
   (A -> B) -> A -> C
S < intros H2 H3.             (* introduction de plusieurs hypothese
1 subgoal
  H1 : A -> B -> C
  H2 : A -> B
  H3 : A
  ============================
   C
```

Une première preuve (4)

```
1 subgoal
  H1 : A -> B -> C   H2 : A -> B   H3 : A
  ============================
   C
S < apply H1.
2 subgoals
  H1 : A -> B -> C   H2 : A -> B   H3 : A
  ============================
   A
subgoal 2 is:
 B
S < assumption.
1 subgoal
  H1 : A -> B -> C   H2 : A -> B   H3 : A
  ============================
   B
```

Une première preuve (5)

```
S < apply H2.
1 subgoal
  H1 : A -> B -> C  H2 : A -> B  H3 : A
  ============================
   A
S < assumption.
Proof completed.
S < Save.
intros H1 H2 H3.
apply H1.
 assumption.
 apply H2.
   assumption.
S is defined
Coq < Print S.
S =
fun (H1 : A -> B -> C) (H2 : A -> B) (H3 : A) => H1 H3 (H2 H3)
     : (A -> B -> C) -> (A -> B) -> A -> C
```

Une première preuve (6)

Le terme Coq `S` correspond au λ-terme :

$$\mathbf{S} \triangleq \lambda x.\lambda y.\lambda z.((x\,z)\,(y\,z))$$

de type :

$$(\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)) \rightarrow ((\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_3))$$

```
Coq < Print S.
S =
fun (H1 : A -> B -> C) (H2 : A -> B) (H3 : A) => H1 H3 (H2 H3)
     : (A -> B -> C) -> (A -> B) -> A -> C
```

Normalisation et Elimination des coupures : Exemple (1)

$\Gamma = \{x : D \Rightarrow A, y : D, z : A \Rightarrow (D \Rightarrow B)\}$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{\dfrac{}{A, \Gamma \vdash z : A \Rightarrow (D \Rightarrow B)}(Hyp) \quad \dfrac{}{A, \Gamma \vdash w : A}(Hyp)}{A, \Gamma \vdash (z\ w) : D \Rightarrow B}(E_{\Rightarrow}) \quad \dfrac{}{A, \Gamma \vdash y : D}(Hyp)
}{A, \Gamma \vdash ((z\ w)\ y) : B}(E_{\Rightarrow})
}{\Gamma \vdash \lambda w.((z\ w)\ y) : A \Rightarrow B}(I_{\Rightarrow})
\quad
\dfrac{\dfrac{}{\Gamma \vdash x : D \Rightarrow A}(Hyp) \quad \dfrac{}{\Gamma \vdash y : D}(Hyp)}{\Gamma \vdash (x\ y) : A}(E_{\Rightarrow})
}{\Gamma \vdash ((\lambda w.((z\ w)\ y))\ (x\ y)) : B}(E_{\Rightarrow})
$$

$$((\lambda w.((z\ w)\ y))\ (x\ y))$$

Normalisation & Elimination des coupures : Ex (2)

```
Coq < Lemma ec : forall A B D : Prop,
     (D->A) -> D -> (A -> (D->B))
     -> B.

Coq < 1 subgoal
  ============================
   forall A B D : Prop, (D -> A) -> D -> (A -> D -> B) -> B

ec < intros A B D x y z.

1 subgoal
  A:Prop  B:Prop  D:Prop
  x:D->A  y:D     z:A->D->B
  ============================
   B

ec < exact ((fun w => ((z w) y)) (x y)).
Proof completed.
```

Normalisation & Elimination des coupures : Ex (3)

... ou de manière plus interactive :

```
ec < intros A B D x y z.
1 subgoal
  A:Prop  B:Prop  D:Prop
  x:D->A  y:D     z:A->D->B
  ============================
   B

ec < cut (A -> B).
2 subgoals
  A:Prop  B:Prop  D:Prop
  x:D->A  y:D     z:A->D->B
  ============================
   (A -> B) -> B
subgoal 2 is:
 A -> B

ec < intro f.
```

Normalisation & Elimination des coupures : Ex (4)

```
2 subgoals
  A:Prop   B:Prop   D:Prop
  x:D->A   y:D      z:A->D->B
  f:A->B
  ============================
   B                                             subgoal 2 is:
                                                 A -> B

ec < apply f.
2 subgoals
  A : Prop   B : Prop   D : Prop   x : D -> A   y : D
  z : A -> D -> B   f : A -> B
  ============================
   A                                             subgoal 2 is:
                                                 A -> B

ec < apply x.
```

Normalisation & Elimination des coupures : Ex (5)

```
2 subgoals
  A : Prop    B : Prop    D : Prop    x : D -> A    y : D
  z : A -> D -> B    f : A -> B
  ============================
   D                                                    subgoal 2 is:
                                                        A -> B
ec < exact y.
  A : Prop    B : Prop    D : Prop    x : D -> A
  y : D    z : A -> D -> B
  ============================
   A -> B
ec < intro w.
  A : Prop    B : Prop    D : Prop    x : D -> A
  y : D    z : A -> D -> B    w : A
  ============================
   B

ec < apply (z w).
```

Normalisation & Elimination des coupures : Ex (6)

```
  A : Prop    B : Prop           D : Prop    x : D -> A
  y : D       z : A -> D -> B    w : A
  ============================
   D

ec < exact y.
Proof completed.

ec < Defined.
...

Coq < Print ec.
ec =
fun (A B D : Prop) (x : D -> A) (y : D) (z : A -> D -> B) =>
let H := fun w : A => z w y in H (x y)
     : forall A B D : Prop, (D -> A) -> D -> (A -> D -> B) -> B
```

Normalisation & Elimination des coupures : Ex (7)

$$((\lambda w.((z\ w)\ y))\ (x\ y)) \rightarrow_\beta ((z\ w)\ y)[w := (x\ y)] = ((z\ (x\ y))\ y)$$

$((z\ (x\ y))\ y)$ correspond à la déduction de $\Gamma \vdash B$ sans coupure : $\Gamma = \{x : D \Rightarrow A, y : D, z : A \Rightarrow (D \Rightarrow B)\}$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash z : A \Rightarrow (D \Rightarrow B)}(Hyp) \quad \dfrac{\dfrac{}{\Gamma \vdash x : D \Rightarrow A}(Hyp) \quad \dfrac{}{\Gamma \vdash y : D}(Hyp)}{\Gamma \vdash (x\ y) : A}(E_\Rightarrow)}{\Gamma \vdash (z\ (x\ y)) : D \Rightarrow B}(E_\Rightarrow) \quad \dfrac{}{\Gamma \vdash y : D}(Hyp)}{\Gamma \vdash ((z\ (x\ y))\ y) : B}(E_\Rightarrow)$$

```
Coq < Eval compute in ec.
 =fun (A B D :Prop) (x:D -> A) (y:D) (z:A -> D -> B) => z (x y) y
 : forall A B D : Prop, (D -> A) -> D -> (A -> D -> B) -> B
```

Terme opaque / Terme transparent

Introduction d'un terme (de preuve) dans l'environnement.
`Defined` : terme transparent (on dispose du type et de la définition, réduction possible)
`Save`, `Qed` : terme opaque (on dispose uniquement du type, réduction impossible)

```
ec < Proof completed.

ec < Save.
...

Coq < Eval compute in ec.
     = ec
     : forall A B D : Prop, (D -> A) -> D -> (A -> D -> B) -> B
```

Conjonction / Produit de types

Ajout du connecteur $\wedge$: aux règles définissant l'ensemble des formules logiques (transparent 43), on ajoute

$$(L_3)\frac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2}$$

Une peuve de $A \wedge B$ est un couple constitué d'une preuve de A et d'une preuve de B.
Ajout d'un constructeur de couple et des opérateurs de projections : aux règles définissant l'ensemble Λ des λ-termes (transparent 12), on ajoute

$$(\text{Pair})\frac{t_1 \quad t_2}{(t_1, t_2)} \qquad (\text{Proj}_1)\frac{t}{\text{fst}(t)} \qquad (\text{Proj}_2)\frac{t}{\text{snd}(t)}$$

Le type du couple (t_1, t_2) est $\tau_1 \times \tau_2$ où τ_1 (resp. τ_2) est le type de t_1 (resp. t_2).
Ajout d'un constructeur de type : aux règles définissant l'ensemble $\mathcal{T}$ des types (transparent 12), on ajoute

$$(T_3)\frac{\tau_1 \quad \tau_2}{\tau_1 \times \tau_2}$$

λ-calcul avec des couples (1)

• Variables libres d'un couple : on complète la définition de $\mathcal{F}(t)$ (transparent 16) avec

$$\begin{array}{l} \mathcal{F}((t_1, t_2)) = \mathcal{F}(t_1) \cup \mathcal{F}(t_2) \\ \mathcal{F}(\mathsf{fst}(t)) = \mathcal{F}(t) \\ \mathcal{F}(\mathsf{snd}(t)) = \mathcal{F}(t) \end{array}$$

• Application d'une substitution sur un couple : on complète la définition de $t[x := t']$ (transparent 21) avec

$$\begin{array}{l} (t_1, t_2)[x := t] = (t_1[x := t], t_2[x := t]) \\ \mathsf{fst}(t)[x := t'] = \mathsf{fst}(t[x := t']) \\ \mathsf{snd}(t)[x := t'] = \mathsf{snd}(t[x := t']) \end{array}$$

λ-calcul avec des couples (2)

- α-équivalence : aux règles définissant $=_\alpha$ (transparent 23), on ajoute

$$(R_{\alpha_8})\frac{t_1 =_\alpha t_1'}{(t_1, t_2) =_\alpha (t_1', t_2)} \qquad (R_{\alpha_9})\frac{t_2 =_\alpha t_2'}{(t_1, t_2) =_\alpha (t_1, t_2')}$$

$$(R_{\alpha_{10}})\frac{t =_\alpha t'}{\mathsf{fst}(t) =_\alpha \mathsf{fst}(t')} \qquad (R_{\alpha_{10}})\frac{t =_\alpha t'}{\mathsf{snd}(t) =_\alpha \mathsf{snd}(t')}$$

- β-réduction : aux règles définissant $\rightarrow_\beta$ (transparent 25), on ajoute

$$(C_{Pair_1})\frac{t_1 \rightarrow_\beta t_1'}{(t_1, t_2) \rightarrow_\beta (t_1', t_2)} \qquad (C_{Pair_2})\frac{t_2 \rightarrow_\beta t_2'}{(t_1, t_2) \rightarrow_\beta (t_1, t_2')}$$

$$(C_{Proj_1})\frac{}{\mathsf{fst}((t_1, t_2)) \rightarrow_\beta t_1} \qquad (C_{Proj_1})\frac{}{\mathsf{snd}((t_1, t_2)) \rightarrow_\beta t_2}$$

$$(C_{\mathsf{fst}})\frac{t \rightarrow_\beta t'}{\mathsf{fst}(t) \rightarrow_\beta \mathsf{fst}(t')} \qquad (C_{\mathsf{snd}})\frac{t \rightarrow_\beta t'}{\mathsf{snd}(t) \rightarrow_\beta \mathsf{snd}(t')}$$

Déduction et Typage

Ajout aux règles de déduction et de typage de :

$$(I_\wedge)\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad (\textsc{Pair})\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 \times \tau_2}$$

$$(E^l_\wedge)\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad (\textsc{Fst})\frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \mathrm{fst}(t) : \tau_1}$$

$$(E^r_\wedge)\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \qquad (\textsc{Snd})\frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \mathrm{snd}(t) : \tau_2}$$

```
# fst;;
- : 'a * 'b -> 'a = <fun>
# snd;;
- : 'a * 'b -> 'b = <fun>
```

Isomorphisme de Curry-Howard (1)

Sémantique de Brouwer-Heyting-Kolmogorov : une peuve de $A \wedge B$ est un couple constitué d'une preuve de A et d'une preuve de B.

$$\mathbb{P}(\varphi \wedge \psi) = \mathbb{P}(\varphi) \times \mathbb{P}(\psi)$$

On complète la preuve de la proposition (transparent 54) : Si $A_1, A_2, \cdots, A_n \vdash A$ admet une preuve, alors il existe un λ-terme t tel que $(x_1, \mathbb{P}(A_1)), (x_2, \mathbb{P}(A_2)), \cdots, (x_n, \mathbb{P}(A_n)) \vdash t : \mathbb{P}(A)$.

PREUVE

- Si $A_1, A_2, \cdots, A_n \vdash A$ a été obtenu à partir de la règle $(I_\wedge)$, alors ...
- Si $A_1, A_2, \cdots, A_n \vdash A$ a été obtenu à partir de la règle $(E^l_\wedge)$, alors ...
- Raisonnement similaire pour $(E^r_\wedge)$.

Isomorphisme de Curry-Howard (2)

On complète la preuve de la proposition (transparent 55) : Si t est un λ-terme t tel que $(x_1, \tau_1), (x_2, \tau_2), \cdots, (x_n, \tau_n) \vdash t : \tau$, alors $\mathbb{P}^{-1}(\tau_1), \mathbb{P}^{-1}(\tau_2), \cdots, \mathbb{P}^{-1}(\tau_n) \vdash \mathbb{P}^{-1}(\tau)$ admet une preuve.

- Si $(x_1, \tau_1), (x_2, \tau_2), \cdots, (x_n, \tau_n) \vdash t : \tau$ a été obtenu à partir de la règle (PAIR), alors ...
- Si $(x_1, \tau_1), (x_2, \tau_2), \cdots, (x_n, \tau_n) \vdash t : \tau$ a été obtenu à partir de la règle (FST), alors ...
- Raisonnement similaire pour (SND).

Déduction et Typage : Exemple

$$
\dfrac{
  \dfrac{
    \dfrac{}{A \wedge B \vdash x : A \wedge B}\,(\textsc{Var}, Hyp)
  }{A \wedge B \vdash \mathrm{snd}(x) : B}\,(\textsc{Snd}, E^r_\wedge)
  \quad
  \dfrac{
    \dfrac{}{A \wedge B \vdash x : A \wedge B}\,(\textsc{Var}, Hyp)
  }{A \wedge B \vdash \mathrm{fst}(x) : A}\,(\textsc{Fst}, E^l_\wedge)
}{
  \dfrac{A \wedge B \vdash (\mathrm{snd}(x), \mathrm{fst}(x)) : B \wedge A}{\vdash \lambda x.(\mathrm{snd}(x), \mathrm{fst}(x)) : A \wedge B \Rightarrow B \wedge A}\,(\textsc{Abs}, I_\Rightarrow)
}\,(\textsc{Pair}, I_\wedge)
$$

```
# function x -> ((snd x),(fst x));;
- : 'a * 'b -> 'b * 'a = <fun>
```

Elimination des coupures / Normalisation

- Elimination des coupures :

$$(E^l_\wedge)\dfrac{(I_\wedge)\dfrac{(_)\dfrac{\vdots}{\Gamma \vdash A} \quad (_)\dfrac{\vdots}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}}{\Gamma \vdash A} \rightsquigarrow (_)\dfrac{\vdots}{\Gamma \vdash A}$$

$$(E^r_\wedge)\dfrac{(I_\wedge)\dfrac{(_)\dfrac{\vdots}{\Gamma \vdash A} \quad (_)\dfrac{\vdots}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}}{\Gamma \vdash B} \rightsquigarrow (_)\dfrac{\vdots}{\Gamma \vdash B}$$

- Normalisation :

$$\mathrm{fst}((t_A, t_B)) \rightarrow_\beta t_A \qquad \mathrm{snd}((t_A, t_B)) \rightarrow_\beta t_B$$

Une preuve avec Coq : Exemple (1)

$\vdash A \wedge B \Rightarrow B \wedge A$

Introduction de 2 variables `A` et `B` de type `Prop`.
Lemme à prouver :

```
Coq < Lemma and_commut : A /\ B -> B /\ A.
1 subgoal

  ============================
   A/\B->B/\A
```

Introduction de l'hypothèse.

```
and_commut < intro H1.
1 subgoal

  H1 : A/\B
  ============================
   B/\A
```

Une preuve avec Coq : Exemple (2)

Elimination de `H1` (règles $(E^l_\wedge)$ et $(E^r_\wedge)$) :

```
and_commut < elim H1.
1 subgoal

  H1 : A/\B
  ============================
   A->B->B/\A
```

Introduction des hypothèses :

```
and_commut < intros H2 H3.
1 subgoal

  H1 : A/\B
  H2 : A
  H3 : B
  ============================
   B/\A
```

Une preuve avec Coq : Exemple (3)

Preuve de la conjonction (règle $(I_\wedge)$) :

```
and_commut < split.
2 subgoals

  H1 : A/\B
  H2 : A
  H3 : B
  ============================
   B

subgoal 2 is:
 A
```

Utilisation des hypothèses :

```
assumption.
assumption.
Subtree proved!
```

Une preuve avec Coq : Exemple (4)

Le connecteur $\wedge$ est défini inductivement dans Coq.

```
Coq < Print and.
Inductive and (A:Prop) (B:Prop) :Prop :=  conj:A -> B -> A /\ B

Coq < Check and_ind.
and_ind
     : forall A B P : Prop, (A -> B -> P) -> A /\ B -> P

Coq < Print and_commut.
and_commut =
fun H1: A /\ B => and_ind (fun (H2:A) (H3:B) => conj H3 H2) H1
     : A /\ B -> B /\ A
```

Disjonction / Type somme (1)

Union disjointe d'ensembles :

$$E_1 \biguplus E_2 = \{g_{E_1,E_2}(x) \mid x \in E_1\} \cup \{d_{E_1,E_2}(x) \mid x \in E_2\}$$

Injections canoniques :

$$g_{E_1,E_2} : E_1 \rightarrow E_1 \biguplus E_2 \qquad d_{E_1,E_2} : E_2 \rightarrow E_1 \biguplus E_2$$

$$x \in E_1 \biguplus E_2 \Leftrightarrow (\exists y \in E_1 \ x = g_{E_1,E_2}(y)) \ XOR \ (\exists y \in E_2 \ x = d_{E_1,E_2}(y))$$

Ajout du connecteur $\vee$: aux règles définissant l'ensemble des formules logiques (transparent 43), on ajoute

$$(L_4)\frac{\varphi_1 \quad \varphi_2}{\varphi_1 \vee \varphi_2}$$

Une peuve de $A \vee B$ est une preuve de A ou une preuve de B.
C'est donc un élément de l'union disjointe des ensembles des preuves de A et des preuves de B.

Disjonction / Type somme (2)

Ajout d'un constructeur de type : aux règles définissant l'ensemble $\mathcal{T}$ des types (transparent 12), on ajoute

$$(T_4)\frac{\tau_1 \quad \tau_2}{\tau_1 + \tau_2}$$

La valeur (i.e., la forme normale) associée à un λ-terme de type $\tau_1 + \tau_2$ s'écrit $\mathrm{inj}^l_{\tau_1,\tau_2}(t)$ (si t est de type τ_1) ou $\mathrm{inj}^r_{\tau_1,\tau_2}(t)$ (si t est de type τ_2).
On ajoute une construction case permettant le filtrage :
$\mathrm{case}_{\tau_1,\tau_2}\ t\ \mathrm{of}\ \mathrm{inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \mathrm{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2$.
Aux règles définissant l'ensemble Λ des λ-termes (transparent 12), on ajoute

$$(\mathrm{Inj}^l)\frac{t}{\mathrm{inj}^l_{\tau_1,\tau_2}(t)} \qquad (\mathrm{Inj}^r)\frac{t}{\mathrm{inj}^r_{\tau_1,\tau_2}(t)}$$

$$(\mathrm{Case})\frac{t \quad t_1 \quad t_2}{\mathrm{case}_{\tau_1,\tau_2}\ t\ \mathrm{of}\ \mathrm{inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \mathrm{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2}(x_1, x_2 \in V)$$

λ-calcul avec des types somme (1)

- Variables libres : on complète la définition de $\mathcal{F}(t)$ (transparent 16) avec

$$\begin{array}{l}\mathcal{F}(\mathsf{inj}^l_{\tau_1,\tau_2}(t)) = \mathcal{F}(t)\\ \mathcal{F}(\mathsf{inj}^r_{\tau_1,\tau_2}(t)) = \mathcal{F}(t)\\ \mathcal{F}(\mathsf{case}_{\tau_1,\tau_2}\ t \text{ of } \mathsf{inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \mathsf{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2)\\ = \mathcal{F}(t) \cup (\mathcal{F}(t_1) \backslash \{x_1\}) \cup (\mathcal{F}(t_2) \backslash \{x_2\})\end{array}$$

- Application d'une substitution : on complète la définition de $t[x := t']$ (transparent 21) avec

$$\begin{array}{l}\mathsf{inj}^l_{\tau_1,\tau_2}(t)[x := t'] = \mathsf{inj}^l_{\tau_1,\tau_2}(t[x := t'])\\ \mathsf{inj}^r_{\tau_1,\tau_2}(t)[x := t'] = \mathsf{inj}^r_{\tau_1,\tau_2}(t[x := t'])\\ (\mathsf{case}_{\tau_1,\tau_2}\ t \text{ of } \mathsf{inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \mathsf{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2)[x := t'] =\\ \mathsf{case}_{\tau_1,\tau_2}\ t[x := t'] \text{ of } \mathsf{inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1[x := t'] \mid \mathsf{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2[x := t']\\ \text{si } x_1, x_2 \notin \mathcal{F}(t') \vee x \notin (\mathcal{F}(t_1) \cup \mathcal{F}(t_2))\end{array}$$

λ-calcul avec des types somme (2)

α-équivalence : aux règles définissant $=_\alpha$ (transparent 23), on ajoute

$$(R_{\alpha_{11}})\frac{t =_\alpha t'}{\mathrm{inj}^l_{\tau_1,\tau_2}(t) =_\alpha \mathrm{inj}^l_{\tau_1,\tau_2}(t')} \quad (R_{\alpha_{12}})\frac{t =_\alpha t'}{\mathrm{inj}^r_{\tau_1,\tau_2}(t) =_\alpha \mathrm{inj}^r_{\tau_1,\tau_2}(t')}$$

$$(R_{\alpha_{13}})\frac{t =_\alpha t'}{\begin{array}{ll} & \mathrm{case}_{\tau_1,\tau_2}\ t \text{ of } \mathrm{inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \mathrm{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2 \\ =_\alpha & \mathrm{case}_{\tau_1,\tau_2}\ t' \text{ of } \mathrm{inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \mathrm{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2 \end{array}}$$

$$(R_{\alpha_{14}})\frac{t_1 =_\alpha t_1'}{\begin{array}{ll} & \mathrm{case}_{\tau_1,\tau_2}\ t \text{ of } \mathrm{inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \mathrm{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2 \\ =_\alpha & \mathrm{case}_{\tau_1,\tau_2}\ t \text{ of } \mathrm{inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1' \mid \mathrm{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2 \end{array}}$$

$$(R_{\alpha_{14}})\frac{t_2 =_\alpha t_2'}{\begin{array}{ll} & \mathrm{case}_{\tau_1,\tau_2}\ t \text{ of } \mathrm{inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \mathrm{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2 \\ =_\alpha & \mathrm{case}_{\tau_1,\tau_2}\ t \text{ of } \mathrm{inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \mathrm{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2' \end{array}}$$

λ-calcul avec des types somme (3)

β-réduction : aux règles définissant $\rightarrow_\beta$ (transparent 25), on ajoute

$$(C^l_{Inj})\ \frac{t \rightarrow_\beta t'}{\text{inj}^l_{\tau_1,\tau_2}(t) \rightarrow_\beta \text{inj}^l_{\tau_1,\tau_2}(t')} \qquad (C^r_{Inj})\ \frac{t \rightarrow_\beta t'}{\text{inj}^r_{\tau_1,\tau_2}(t) \rightarrow_\beta \text{inj}^r_{\tau_1,\tau_2}(t')}$$

$$(C_{Case_1})\ \frac{t \rightarrow_\beta t'}{\begin{array}{ll} & \text{case}_{\tau_1,\tau_2}\ t \text{ of inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \text{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2 \\ \rightarrow_\beta & \text{case}_{\tau_1,\tau_2}\ t' \text{ of inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \text{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2 \end{array}}$$

$$(C_{Case_2})\ \frac{t_1 \rightarrow_\beta t'_1}{\begin{array}{ll} & \text{case}_{\tau_1,\tau_2}\ t \text{ of inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \text{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2 \\ \rightarrow_\beta & \text{case}_{\tau_1,\tau_2}\ t \text{ of inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t'_1 \mid \text{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2 \end{array}}$$

$$(C_{Case_3})\ \frac{t_2 \rightarrow_\beta t'_2}{\begin{array}{ll} & \text{case}_{\tau_1,\tau_2}\ t \text{ of inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \text{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2 \\ \rightarrow_\beta & \text{case}_{\tau_1,\tau_2}\ t \text{ of inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \text{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t'_2 \end{array}}$$

λ-calcul avec des types somme (4)

β-réduction (suite) :

$$(C_{Case_l})\frac{}{\begin{array}{ll} & \mathrm{case}_{\tau_1,\tau_2}\ \mathrm{inj}^l_{\tau_1,\tau_2}(t)\ \mathrm{of}\ \mathrm{inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \mathrm{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2 \\ \to_\beta & t_1[x_1 := t] \end{array}}$$

$$(C_{Case_r})\frac{}{\begin{array}{ll} & \mathrm{case}_{\tau_1,\tau_2}\ \mathrm{inj}^r_{\tau_1,\tau_2}(t)\ \mathrm{of}\ \mathrm{inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \mathrm{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2 \\ \to_\beta & t_2[x_2 := t] \end{array}}$$

Déduction et Typage

Ajout aux règles de déduction et de typage de :

$$(I^l_\vee)\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \quad (\textsc{Inj}_l)\frac{\Gamma \vdash t : \tau_1}{\Gamma \vdash \mathrm{inj}^l_{\tau_1,\tau_2}(t) : \tau_1 + \tau_2}$$

$$(I^r_\vee)\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \quad (\textsc{Inj}_r)\frac{\Gamma \vdash t : \tau_2}{\Gamma \vdash \mathrm{inj}^r_{\tau_1,\tau_2}(t) : \tau_1 + \tau_2}$$

$$(E_\vee)\frac{\Gamma \vdash A \vee B \quad A, \Gamma \vdash C \quad B, \Gamma \vdash C}{\Gamma \vdash C}$$

$$(\textsc{Case})\frac{\Gamma \vdash t : \tau_1 + \tau_2 \quad (x_1 : \tau_1), \Gamma \vdash t_1 : \tau \quad (x_2 : \tau_2), \Gamma \vdash t_2 : \tau}{\Gamma \vdash \mathrm{case}_{\tau_1,\tau_2}\ t \text{ of } \mathrm{inj}^l_{\tau_1,\tau_2}(x_1) \mapsto t_1 \mid \mathrm{inj}^r_{\tau_1,\tau_2}(x_2) \mapsto t_2 : \tau}$$

Isomorphisme de Curry-Howard (1)

Sémantique de Brouwer-Heyting-Kolmogorov : une peuve de $A \vee B$ est soit une preuve de A soit d'une preuve de B.

$$\mathbb{P}(\varphi \vee \psi) = \mathbb{P}(\varphi) + \mathbb{P}(\psi)$$

On complète la preuve de la proposition (transparent 54) : Si $A_1, A_2, \cdots, A_n \vdash A$ admet une preuve, alors il existe un λ-terme t tel que $(x_1, \mathbb{P}(A_1)), (x_2, \mathbb{P}(A_2)), \cdots, (x_n, \mathbb{P}(A_n)) \vdash t : \mathbb{P}(A)$.

PREUVE

- Si $A_1, A_2, \cdots, A_n \vdash A$ a été obtenu à partir de la règle $(I_\vee^l)$, alors ...
- Raisonnement similaire pour $(I_\vee^r)$.
- Si $A_1, A_2, \cdots, A_n \vdash A$ a été obtenu à partir de la règle $(E_\vee)$, alors ...

Isomorphisme de Curry-Howard (2)

On complète la preuve de la proposition (transparent 55) : Si t est un λ-terme t tel que $(x_1, \tau_1), (x_2, \tau_2), \cdots, (x_n, \tau_n) \vdash t : \tau$, alors $\mathbb{P}^{-1}(\tau_1), \mathbb{P}^{-1}(\tau_2), \cdots, \mathbb{P}^{-1}(\tau_n) \vdash \mathbb{P}^{-1}(\tau)$ admet une preuve.

- Si $(x_1, \tau_1), (x_2, \tau_2), \cdots, (x_n, \tau_n) \vdash t : \tau$ a été obtenu à partir de la règle $(\textsc{Inj}_l)$, alors ...
- Raisonnement similaire pour $(\textsc{Inj}_r)$.
- Si $(x_1, \tau_1), (x_2, \tau_2), \cdots, (x_n, \tau_n) \vdash t : \tau$ a été obtenu à partir de la règle $(\textsc{Case})$, alors ...

Déduction et Typage : Exemple

$$
\dfrac{
  \dfrac{}{A \lor B \vdash z : A \lor B}\ (\textsc{Var}\ Hyp)
  \quad
  \dfrac{\dfrac{}{A, A \lor B \vdash x_1 : A}\ (\textsc{Var}\ Hyp)}{A, A \lor B \vdash \mathrm{inj}^r_{B,A}(x_1) : B \lor A}\ (\textsc{Inj}_r\ I^r_\lor)
  \quad
  \dfrac{\dfrac{}{B, A \lor B \vdash x_2 : B}\ (\textsc{Var}\ Hyp)}{B, A \lor B \vdash \mathrm{inj}^l_{B,A}(x_2) : B \lor A}\ (\textsc{Inj}_r\ I^l_\lor)
}{
  \dfrac{A \lor B \vdash \mathrm{case}_{A,B}\ z\ \mathrm{of}\ \mathrm{inj}^l_{A,B}(x_1) \mapsto \mathrm{inj}^r_{B,A}(x_1) \mid \mathrm{inj}^r_{A,B}(x_2) \mapsto \mathrm{inj}^l_{B,A}(x_2) : B \lor A}{\vdash \lambda z.\mathrm{case}_{A,B}\ z\ \mathrm{of}\ \mathrm{inj}^l_{A,B}(x_1) \mapsto \mathrm{inj}^r_{B,A}(x_1) \mid \mathrm{inj}^r_{A,B}(x_2) \mapsto \mathrm{inj}^l_{B,A}(x_2) : A \lor B \Rightarrow B \lor A}\ (\textsc{Abs}\ I_\Rightarrow)
}\ (\textsc{Case}\ E_\lor)
$$

Elimination des coupures

Obtenir une déduction sans coupure à partir de :

$$(E_\vee)\dfrac{(I^l_\vee)\dfrac{(_)\dfrac{\vdots}{\Gamma \vdash A}}{\Gamma \vdash A \vee B} \quad (_)\dfrac{\vdots}{A, \Gamma \vdash C} \quad (_)\dfrac{\vdots}{B, \Gamma \vdash C}}{\Gamma \vdash C}$$

- Suppression des occurrences de A dans tous les contextes d'hypothèse de la déduction : $(_)\dfrac{\vdots}{A, \Gamma \vdash C}$
- Remplacement dans cette déduction de toutes les applications de la règle (Hyp) sur A : $(Hyp)\dfrac{}{A, \Gamma, \Delta \vdash A}$ par la déduction : $(_)\dfrac{\vdots}{\Gamma, \Delta \vdash A}$ obtenue en ajoutant les hypothèses de Δ dans tous les contextes de la déduction : $(_)\dfrac{\vdots}{\Gamma \vdash A}$

Normalisation $\equiv$ Elimination des coupures

Soit π la déduction : $(E_\vee)\dfrac{(I^l_\vee)\dfrac{(_)\dfrac{\vdots}{\Gamma \vdash A}}{\Gamma \vdash A \vee B} \quad (_)\dfrac{\vdots}{A, \Gamma \vdash C} \quad (_)\dfrac{\vdots}{B, \Gamma \vdash C}}{\Gamma \vdash C}$

$$t^l_\vee = \mathbb{T}\left((_)\frac{\vdots}{A, \Gamma \vdash C}\right) \quad t^r_\vee = \mathbb{T}\left((_)\frac{\vdots}{B, \Gamma \vdash C}\right) \quad t_A = \mathbb{T}\left((_)\frac{\vdots}{\Gamma \vdash A}\right) \quad t = \mathbb{T}(\pi)$$

$$\begin{array}{rl} & t = \mathrm{case}_{A,B}\ \mathrm{inj}^l_{A,B}(t_A)\ \mathrm{of}\ \mathrm{inj}^l_{A,B}(x_1) \mapsto t^l_\vee \mid \mathrm{inj}^r_{A,B}(x_2) \mapsto t^r_\vee \\ \rightarrow_\beta & t^l_\vee[x_1 := t_A] \end{array}$$

Or $t^l_\vee[x_1 := t_A]$ correspond à une preuve sans coupure ... la propriété de *Subject reduction* garantit qu'il s'agit encore d'une déduction de A.
Remarque : Il y a d'autres coupures possibles pour le $\vee$.

Normalisation $\equiv$ Elimination des coupures : Exemple

$\Gamma = \{x : C \Rightarrow A, y : C\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : C \Rightarrow A}\,(\text{VAR}\ Hyp) \quad \dfrac{}{\Gamma \vdash y : C}\,(\text{VAR}\ Hyp)}{\Gamma \vdash (x\ y) : A}\,(\text{APP}\ E_{\Rightarrow})}{\Gamma \vdash \text{inj}^{l}_{A,B}((x\ y)) : A \vee B}\,(\text{INJ}_l\ I^{l}_{\vee}) \quad \dfrac{}{A, \Gamma \vdash x_1 : A}\,(\text{VAR}\ Hyp) \quad \dfrac{\dfrac{}{B, \Gamma \vdash x : C \Rightarrow A}\,(\text{VAR}\ Hyp) \quad \dfrac{}{B, \Gamma \vdash y : C}\,(\text{VAR}\ Hyp)}{B, \Gamma \vdash (x\ y) : A}\,(\text{APP}\ E_{\Rightarrow})}{\Gamma \vdash \text{case}_{A,B}\ \text{inj}^{l}_{A,B}((x\ y))\ \text{of}\ \text{inj}^{l}_{A,B}(x_1) \mapsto x_1 \mid \text{inj}^{r}_{A,B}(x_2) \mapsto (x\ y) : A}\,(\text{CASE}\ E_{\vee})$$

$$\begin{array}{ll} & \text{case}_{A,B}\ \text{inj}^{l}_{A,B}((x\ y))\ \text{of}\ \text{inj}^{l}_{A,B}(x_1) \mapsto x_1 \mid \text{inj}^{r}_{A,B}(x_2) \mapsto (x\ y) \\ \rightarrow_{\beta} & x_1[x_1 := (x\ y)] = (x\ y) \end{array}$$

et on obtient :

$$\dfrac{\dfrac{}{\Gamma \vdash x : C \Rightarrow A}\,(\text{VAR}\ Hyp) \quad \dfrac{}{\Gamma \vdash y : C}\,(\text{VAR}\ Hyp)}{\Gamma \vdash (x\ y) : A}\,(\text{APP}\ E_{\Rightarrow})$$

Une preuve avec Coq : Exemple (1)

$\vdash A \vee B \Rightarrow B \vee A$

Introduction de 2 variables `A` et `B` de type `Prop`.
Lemme à prouver :

```
Coq < Lemma or_commut : A \/ B -> B \/ A.
1 subgoal

  ============================
   A\/B->B\/A
```

Introduction de l'hypothèse.

```
or_commut < intro H1.
1 subgoal

  H1 : A\/B
  ============================
   B\/A
```

Une preuve avec Coq : Exemple (2)

Elimination de `H1` (règle $(E^{l}_{\vee})$) :

```
or_commut < elim H1.
2 subgoals

  H1 : A\/B
  ============================
   A->B\/A

subgoal 2 is:
 B->B\/A
```

Une preuve avec Coq : Exemple (3)

Introduction de l'hypothèse :

```
or_commut < intro H2.
2 subgoals

  H1 : A\/B
  H2 : A
  ============================
   B\/A

subgoal 2 is:
 B->B\/A
```

Une preuve avec Coq : Exemple (4)

Introduction du $\vee$ (règle $(I^r_\vee)$) :

```
or_commut < right.
2 subgoals

  H1 : A\/B
  H2 : A
  ============================
   A

subgoal 2 is:
 B->B\/A
```

Utilisation de l'hypothèse : `Assumption.`
Puis ... `intro H2 ;left ;assumption.`

Une preuve avec Coq : Exemple (5)

Le connecteur $\vee$ est défini inductivement dans Coq.

```
Coq < Print or.
Inductive or (A : Prop) (B : Prop) : Prop :=
    or_introl : A -> A \/ B | or_intror : B -> A \/ B

Coq < Print or_ind.
or_ind =
fun (A B P : Prop) (f : A -> P) (f0 : B -> P) (o : A \/ B) =>
match o return P with
| or_introl x => f x  | or_intror x => f0 x  end
     : forall A B P : Prop, (A -> P) -> (B -> P) -> A \/ B -> P

Coq < Print or_commut.
or_commut =
fun H1 : A \/ B =>
or_ind (fun H2:A=> or_intror B H2)(fun H2:B =>or_introl A H2) H1
     : A \/ B -> B \/ A
```

Négation

$\neg A$ est une abbréviation de $A \Rightarrow \bot$ où $\bot$ dénote “le faux”.
$\bot$ est associé à une règle d'élimination :

$$(E_\bot)\,\frac{\Gamma \vdash \bot}{\Gamma \vdash A}$$

... *ex falso quodlibet sequitur*
Exemple $\vdash A \Rightarrow \neg\neg A$

$$(I_\Rightarrow)\,\dfrac{(I_\Rightarrow)\,\dfrac{(E_\Rightarrow)\,\dfrac{(Hyp)\,\dfrac{}{A, A \Rightarrow \bot \vdash A \Rightarrow \bot} \quad (Hyp)\,\dfrac{}{A, A \Rightarrow \bot \vdash A}}{A, A \Rightarrow \bot \vdash \bot}}{A \vdash (A \Rightarrow \bot) \Rightarrow \bot}}{\vdash A \Rightarrow \underbrace{((A \Rightarrow \bot) \Rightarrow \bot)}_{\neg\neg A}}$$

Sans le tiers exclus ($A \vee \neg A$) de la logique classique, on ne peut pas prouver $\neg\neg A \Rightarrow A$

Logique propositionnelle intuitioniste

$$(Hyp)\frac{}{A, \Gamma \vdash A}$$

$$(I_{\Rightarrow})\frac{A, \Gamma \vdash B}{\Gamma \vdash A \Rightarrow B} \quad (E_{\Rightarrow})\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$(I_{\wedge})\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \quad (E^{l}_{\wedge})\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \quad (E^{r}_{\wedge})\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

$$(I^{l}_{\vee})\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \quad (I^{r}_{\vee})\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \quad (E_{\vee})\frac{\Gamma \vdash A \vee B \quad A, \Gamma \vdash C \quad B, \Gamma \vdash C}{\Gamma \vdash C}$$

$$(E_{\bot})\frac{\Gamma \vdash \bot}{\Gamma \vdash A}$$

Logique propositionnelle intuitioniste : Règles dérivées

$$(E_\neg)\frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot} \qquad (I_\neg)\frac{A, \Gamma \vdash \bot}{\Gamma \vdash \neg A}$$

$$(E_\Rightarrow)\frac{\Gamma \vdash \underbrace{A \Rightarrow \bot}_{\neg A} \qquad \Gamma \vdash A}{\Gamma \vdash \bot}$$

$$(I_\Rightarrow)\frac{A, \Gamma \vdash \bot}{\Gamma \vdash \underbrace{A \Rightarrow \bot}_{\neg A}}$$

Une idée de la suite ... Types dépendants et Constructions inductives

Formules du 1er ordre (quanficateurs dans les formules) ... introduction de dépendances dans les types (par exemple le type des listes de longueur *n*, ...) – Preuve par induction et termes récursifs.
Type inductif : défini par un ensemble de constructeurs (nom + type).
Une valeur de ce type est obtenue en appliquant un nombre de fois fini ces constructeurs (l'ensemble des termes clos de ce type est l'algèbre libre engendrée par ces constructeurs).
Exemple.

```
Coq < Print nat.
Inductive nat : Set :=  O : nat | S : nat -> nat
```

Constructions inductives

```
Coq < Print nat.
Inductive nat : Set :=  O : nat | S : nat -> nat
```

A chaque définition inductive est associée un schéma d'élimination ...
permettant de mettre en oeuvre un raisonnement par induction.

```
nat_ind  : forall P : nat -> Prop, ...
nat_rec  : forall P : nat -> Set,  ...
nat_rect : forall P : nat -> Type, ...

  P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Schémas d'élimination

Exemple.

```
Coq < Print nat_rect.
nat_rect =
fun (P:nat->Type) (f: P 0)(f0: forall n:nat, P n -> P (S n)) =>
(fix F (n : nat) : P n :=
   match n return P with
   | O => f
   | S n0 => f0 n0 (F n0)
   end)
     : forall P : nat -> Type,
       P 0 -> (forall n:nat, P n -> P (S n)) -> forall n:nat, P n
```

Définitions inductives : $\leq$

Définition de la relation $\leq$ sur $\mathbb{N}$.

```
Inductive le (n : nat) : nat -> Prop :=
    le_n : n <= n
  | le_S : forall m : nat, n <= m -> n <= S m

le_ind
     : forall (n : nat) (P : nat -> Prop),
       P n ->
       (forall m : nat, n <= m -> P m -> P (S m)) ->
       forall n0 : nat, n <= n0 -> P n0
```

Seul le schéma `le_ind` est défini.

Preuves par induction (1)

```
Lemma le_O : forall n : nat , O <= n.

  ============================
   forall n : nat, 0 <= n

le_O < induction n.
2 subgoals

  ============================
   0 <= 0

subgoal 2 is:
 0 <= S n

le_O < apply le_n.
```

Preuves par induction (2)

```
  n : nat
  IHn : 0 <= n
  ============================
   0 <= S n

le_O < apply le_S.
1 subgoal

  n : nat
  IHn : 0 <= n
  ============================
   0 <= n

le_O < assumption.
Proof completed.
```

Preuves par induction (3)

```
Coq < Print le_O.
le_O =
fun n : nat =>
nat_ind (fun n0 : nat => 0 <= n0) (le_n 0)
  (fun (n0 : nat) (IHn : 0 <= n0) => le_S 0 n0 IHn) n
     : forall n : nat, 0 <= n
```

Preuves par induction (4)

```
Lemma le_succ : forall n m : nat , n <= m -> (S n) <= (S m).

  ============================
   forall n m : nat, n <= m -> S n <= S m

le_succ < intros.
1 subgoal

  n : nat
  m : nat
  H : n <= m
  ============================
   S n <= S m

le_succ < elim H.
```

Preuves par induction (5)

```
2 subgoals

  n : nat
  m : nat
  H : n <= m
  ============================
   S n <= S n

subgoal 2 is:
 forall m0 : nat, n <= m0 -> S n <= S m0 -> S n <= S (S m0)

le_succ < apply le_n.
```

Preuves par induction (6)

```
  n : nat
  m : nat
  H : n <= m
  ============================
   forall m0 : nat, n <= m0 -> S n <= S m0 -> S n <= S (S m0)

le_succ < intros.
  n : nat
  m : nat
  H : n <= m
  m0 : nat
  H0 : n <= m0
  H1 : S n <= S m0
  ============================
   S n <= S (S m0)

le_succ < apply le_S;assumption.
Proof completed.
```

Logique, programmation et sémantique ....

Preuves et Calculs : on a vu qu'écrire une preuve c'était écrire un programme en utilisant un langage de programmation (le λ-calcul) : c'est l' isomorphisme de Curry-Howard.

$$\begin{array}{rcl} \textbf{preuve} & \equiv & \textbf{programme} \\ \textbf{proposition} & \equiv & \textbf{type (spécification)} \end{array}$$

Pour interpréter des programmes par des preuves on a fait de la sémantique : typage
Pour qu'un programme corresponde à une preuve il faut qu'il soit typable ... *sémantique statique : le typage*
Exécuter un programme c'est éliminer les coupures de la preuve auquel il correspond ... *sémantique dynamique : l'exécution des programmes*
Etude de la *sémantique* du λ-calcul (langage de programmation)

Etude d'un langage de programmation : le λ-calcul

- de la *syntaxe* ...
- de la *sémantique* ...
  - *sémantique statique* : le typage $\Gamma \vdash t : \tau$
  - *sémantique dynamique* : l'exécution des programmes $t \rightarrow_\beta^\star t'$
- des *propriétés* ...
  - sur les programmes ... ($t_1 =_\alpha t_2$)
  - sur le langage ...
    - ⋆ Confluence
    - ⋆ Si un terme admet une forme normale, alors la réduction en appel par nom termine.
    - ⋆ Tout λ-terme typable admet une forme normale unique.
    - ⋆ (*Subject Reduction*) Si $\Gamma \vdash t : \tau$ et $t \rightarrow_\beta t'$, alors $\Gamma \vdash t' : \tau$

Etude d'un langage de programmation : la logique ?

... *pas tout à fait*

Programmation (en) logique (PROLOG, ...) Exécuter un programme c'est construire une preuve et en extraire de l'information.

programme	$\equiv$	**ensemble de formules logiques**
exécution d'un programme	$\equiv$	**recherche d'une preuve**

Pour obtenir un langage de programmation (en) logique, on se restreint à certaines formules logiques : les clauses de Horn.

Des clauses ...

Atome	$p(t_1, \cdots, t_n)$ p : predicat t_i : termes	$pair(succ(x)))$
Clause	$\forall \vec{x}(\neg A_1 \vee \cdots \vee \neg A_n \vee B_1 \vee \cdots B_k)$ $\forall \vec{x}((A_1 \wedge \cdots \wedge A_n) \Rightarrow (B_1 \vee \cdots \vee B_k))$ A_i, B_j : atomes	
Clause de Horn $(k \leq 1)$	$\forall \vec{x}(\neg A_1 \vee \cdots \vee \neg A_n \vee B)$ $\forall \vec{x}((A_1 \wedge \cdots \wedge A_n) \Rightarrow B)$ $\forall \vec{x}(\neg A_1 \vee \cdots \vee \neg A_n)$ $\neg \exists \vec{x}(A_1 \wedge \cdots \wedge A_n)$	$(k = 1)$ $(k = 1)$ Clause définie $(k = 0)$ $(k = 0)$ Clause négative

Ecrire un programme avec des “clauses définies”

Exemple. Addition de deux entiers :

$$\forall x \; add(0, x, x)$$
$$\forall x \forall y \forall z \; add(x, y, z) \Rightarrow add(succ(x), y, succ(z))$$

Comment “utiliser” un programme (en) logique ? ... en soumettant une requête de la forme $\exists \vec{x}(B_1 \wedge \cdots \wedge B_k)$.
Exemple. Additionner 1 avec 2 ? ... existe-t-il un x tel que $add(succ(0), succ(succ(0)), x)$?

$$\exists \vec{x} add(succ(0), succ(succ(0)), x)$$

Une clause négative correspond à la négation d’une requête.

Des sémantiques pour les programmes (en) logique

Sémantiques d'un programme *P* (en) logique :

- qu'est ce qu'on peut calculer avec *P* (sémantique déclarative) ?
- comment peut-on calculer avec *P* (i.e. comment **exécuter** un programme en logique) (sémantique opérationnelle) ?

Sémantique déclarative pour les programmes (en) logique

Qu'est ce qu'on peut calculer avec P (sémantique déclarative) à partir d'une requête R ? une solution θ (i.e. une substitution ... une fonction qui donne une valeur aux variables de la requête) telle que

$$P \models \theta(R)$$

Tous les modèles de P sont des modèles de $\theta(R)$.
Exemple.

$$\begin{array}{ll} & \left\{ \begin{array}{l} \forall x \ add(0, x, x) \\ \forall x \forall y \forall z \ add(x, y, z) \Rightarrow add(succ(x), y, succ(z)) \end{array} \right\} \\ \models & \theta(add(succ(0), succ(succ(0)), x)) \end{array}$$

θ est la substitution qui associe $succ(succ(succ(0)))$ à x.
Pourquoi se restreindre aux clauses de Horn ? Tout ensemble fini de clause de Horn admet un plus petit modèle ...

Sémantique opérationnelle pour les programmes (en) logique

Programme P	Requête R
$\{\forall \overrightarrow{x}((A_1 \wedge \cdots \wedge A_n) \Rightarrow A)\}$	$\exists \overrightarrow{x}\ (B_1 \wedge \cdots \wedge B_k)$

Exécution : soumission d'une requête R étant donné un programme P

- construction d'une preuve de $P \vdash \exists \overrightarrow{x}\ (B_1 \wedge \cdots \wedge B_k)$ par réfutation
- c-a-d réfutation l'ensemble de clauses $P \cup \neg\exists \overrightarrow{x}\ (B_1 \wedge \cdots \wedge B_k)$ ... en utilisant la SLD-résolution ...
- et extraction de cette preuve des valeurs associées à $\overrightarrow{x}$ ...
- il s'agit d'une substitution θ appelée réponse de P à R.

Sémantique opérationnelle : SLD-résolution

Résolution :

$$\frac{\begin{array}{c}\neg A_1 \vee \cdots \vee \neg A_{n_1} \vee B_1 \vee \cdots \vee B_j \vee \cdots \vee B_{m_1} \\ \neg A'_1 \vee \cdots \vee \neg A'_i \vee \cdots \vee \neg A'_{n_2} \vee B'_1 \vee \cdots \vee B'_{m_2}\end{array}}{\theta\left(\begin{array}{cl} & \neg A_1 \vee \cdots \vee \neg A_{n_1} \vee \neg A'_1 \vee \cdots \vee \neg A'_{i-1} \vee \neg A'_{i+1} \vee \cdots \vee \neg A'_{n_2} \\ \vee & B_1 \vee \cdots \vee B_{j-1} \vee B_{j+1} \vee \cdots \vee B_{m_1} \vee B'_1 \vee \cdots \vee B'_{m_2}\end{array}\right)}$$

si θ est un **unificateur** des atomes A'_i et B_j (i.e. $\theta(A'_i) = \theta(B_j)$). On choisit l'unificateur le plus général ... (*mgu : most general unifier*)

SLD-résolution : *Selection Linear Definite Resolution*

$$\frac{\begin{array}{ll}\neg A_1 \vee \cdots \vee \neg A_{n_1} \vee B & \text{(clause définie du programme)} \\ \neg A'_1 \vee \cdots \vee \neg A'_i \vee \cdots \vee \neg A'_{n_2} & \text{(clause négative (requête))}\end{array}}{\underbrace{\theta(\neg A'_1 \vee \cdots \vee \neg A'_{i-1} \vee \neg A_1 \vee \cdots \vee \neg A_{n_1} \vee \neg A'_{i+1} \vee \cdots \vee \neg A'_{n_2})}_{\text{clause négative}}}$$

si $\theta = mgu(B, A'_i)$

Sémantique opérationnelle : SLD-dérivation

SLD-résolution : *Selection Linear Definite Resolution*

$$\frac{\begin{array}{ll} \neg A_1 \vee \cdots \vee \neg A_{n_1} \vee B & \text{(clause définie du programme)} \\ \neg A'_1 \vee \cdots \vee \neg A'_i \vee \cdots \vee \neg A'_{n_2} & \text{(clause négative (requête))} \end{array}}{\underbrace{\theta(\neg A'_1 \vee \cdots \vee \neg A'_{i-1} \vee \neg A_1 \vee \cdots \vee \neg A_{n_1} \vee \neg A'_{i+1} \vee \cdots \vee \neg A'_{n_2})}_{\text{clause négative}}}$$

si $\theta = mgu(B, A'_i)$

$$\begin{array}{ccccccc} C_0 & & C_1 & & & C_n & \\ \searrow & & \searrow & & & \searrow & \\ R_0 & \xrightarrow{\theta_0} & R_1 & \xrightarrow{\theta_1} & \cdots & R_n & \xrightarrow{\theta_n} \emptyset \end{array}$$

Réponse : composition $\theta = \theta_n \circ \cdots \circ \theta_1$

SLD-dérivation : Exemple

$$\neg add(x_1, y_1, z_1) \vee add(succ(x_1), y_1, succ(z_1))$$

$$\theta_0 = \left[\begin{array}{ccc} x_1 & y_1 & x \\ 0 & succ(succ(0)) & succ(z_1) \end{array} \right]$$

$$add(0, x_2, x_2)$$

$$\theta_1 = \left[\begin{array}{cc} x_2 & z_1 \\ succ(succ(0)) & succ(succ(0)) \end{array} \right]$$

$$\neg add(succ(0), succ(succ(0)), x)$$

$$\downarrow$$

$$\neg add(0, succ(succ(0)), z_1)$$

$$\downarrow$$

$$\emptyset$$

$$\theta_1 \circ \theta_0(x) = succ(succ(succ(0)))$$

Etude d'un langage de programmation : fragment Hornien de la logique

- de la *syntaxe* ... clauses
- de la *sémantique* ... solutions (modèles) et réponses (preuves)
- des *propriétés* ...
  - Soundness : Si θ est une réponse alors θ est une solution.
  - Completness : Toute solution est une “instance” d'une réponse.
  - Lifting lemma : Si $\theta(R)$ admet une réponse alors R admet une réponse.
  - ...

SLD-résolution : non-déterminismes

Non-déterminisme : plusieurs stratégies possibles lors de la réduction d'un λ-terme (appel par nom, appel par valeur, ...) ... mais des propriétés de confluence.
La SLD-résolution est à la programmation (en) logique ce qu'est la β-réduction au λ-calcul.

$$\frac{\begin{array}{ll} \neg A_1 \vee \cdots \vee \neg A_{n_1} \vee B & \text{(clause définie du programme)} \\ \neg A'_1 \vee \cdots \vee \neg A'_i \vee \cdots \vee \neg A'_{n_2} & \text{(clause négative (requête))} \end{array}}{\underbrace{\theta(\neg A'_1 \vee \cdots \vee \neg A'_{i-1} \vee \neg A_1 \vee \cdots \vee \neg A_{n_1} \vee \neg A'_{i+1} \vee \cdots \vee \neg A'_{n_2})}_{\text{clause négative}}}$$

si $\theta = mgu(B, A'_i)$

Non-déterminisme :

- Dans le choix de la clause
- Dans le choix de l'atome considéré dans la clause négative

Non-déterminisme

Choix de la clause : Non-déterminisme par ignorance (*Don't know*)
mécanisme de *backtrack*
Stratégie PROLOG : ordre d'apparition des clauses dans le programme.
Choix de l'atome : Non-déterminisme par indifférence (*Don't care*)
Switching lemma

$$\underbrace{(\cdots, L_1, \cdots, L_2, \cdots)}_{R_0}$$

$$C_1 \swarrow \quad \searrow C_2$$

$$\underbrace{\theta(\cdots, C_1^-, \cdots, L_2, \cdots)}_{R_1} \qquad \sigma(\cdots, L_1, \cdots, C_2^-, \cdots)$$

$$C_2 \downarrow \qquad\qquad \downarrow C_1$$

$$\eta\theta(\cdots, C_1^-, \cdots, C_2^-, \cdots) \approx \mu\sigma(\cdots, C_1^-, \cdots, C_2^-, \cdots)$$

Stratégie PROLOG : l'atome le plus à gauche.

Sémantique des langages de programmation

- Programmation fonctionnelle
- Programmation (en) logique

Et les programmes impératifs ?
sémantique opérationnelle : programme = “transformateur” d’états de la mémoire
Définition d’une relation de transition entre états décrivant les changements produits par l’exécution d’une instruction.
Le rôle d’un programme c est de modifier l’état courant σ_1 de la mémoire en un état σ_2 :

$$\langle c, \sigma_1 \rangle \rightarrow \sigma_2$$

L’exécution de l’instruction c dans l’état σ_1 conduit à l’état σ_2.

... UE APS (Analyse de Programmes et Sémantique)

Sémantique des langages de programmation

- Programmation fonctionnelle
- Programmation (en) logique
- Programmation impérative (UE APS – Analyse de Programmes et Sémantique)

Et la programmation orientée objet ? *– plutôt en M2 –*
Exemple : Environnement de développement FOCAL
Repose sur un langage :

- orienté objet (héritage, redéfinition, ...)
- permettant d'écrire à la fois des programmes, des propriétés et des preuves
- ... passer progressivement de la spécification à un programme qui satisfait cette spécification (programme prouvé)

– sujet PSTL –

Sémantique des langages de programmation : Pourquoi ?

Utiliser des langages avec une sémantique formelle (i.e. exprimée dans un formalisme mathématique) ... pourquoi ?
Pour raisonner formellement sur les propriétés des programmes écrits dans ce langage ...
Raisonner formellement ? raisonner en utilisant un logiciel (Coq, FOCAL, ...)
Raisonner formellement ? Pourquoi ?
Augmenter la confiance que l'on peut avoir dans un logiciel ...
Atteintre des hauts niveaux de certification définis par des normes internationales (Critères Communs, ...) ... exigés pour du logiciel critique.