Sémantiques mécanisées, huitième cours

Coq en Coq: Mécaniser la logique d'un assistant à la démonstration

Xavier Leroy

2020-02-13

Collège de France, chaire de sciences du logiciel

La démarche du cours ≪Sémantiques mécanisées≫

≪ Ce cours est une introduction aux sémantiques formelles des langages de programmation et à leur utilisation pour construire et valider des outils de programmation et de vérification :

- systèmes de types ;
- logiques de programmes ;
- analyses statiques ;
- compilateurs.

Toutes les définitions, propriétés, et démonstrations sont mécanisées avec l'assistant Coq. ≫

La démarche du cours ≪Sémantiques mécanisées≫

≪ Ce cours est une introduction aux sémantiques formelles des langages de programmation et à leur utilisation pour construire et valider des outils de programmation et de vérification :

- systèmes de types ;
- logiques de programmes ;
- analyses statiques ;
- compilateurs.

Toutes les définitions, propriétés, et démonstrations sont mécanisées avec l'assistant Coq. ≫

Coq : un outil comme les autres

Tout au long du cours, nous avons utilisé Coq comme outil de programmation et de vérification.

Quelle confiance accorder à cet outil ?

Quels formalismes permettraient de valider cet outil ?

Comment une démonstration mécanisée peut-être fausse ?

(R. Pollack, *How to Believe a Machine-Checked Proof*, 1997)

Inadéquation : ce qui est démontré n’est pas ce que vous croyez.

Comment une démonstration mécanisée peut-être fausse ?

(R. Pollack, *How to Believe a Machine-Checked Proof*, 1997)

Inadéquation : ce qui est démontré n'est pas ce que vous croyez.

(Chris Casinghino, 2009-04-01)

```
Require Import Arith.
(* BEGIN PROOF OF FERMAT"S LAST THEOREM *)
Theorem fermat : forall n x y z,
  n > 2 ->
  x > 0 -> y > 0 -> z > 0 ->
  x ^ n + y ^ n <> z ^ n.
Proof.
  intros n x y z. trivial.
Qed.
(* END PROOF OF FERMAT"S LAST THEOREM *)
```

Comment une démonstration mécanisée peut-être fausse ?

(R. Pollack, *How to Believe a Machine-Checked Proof*, 1997)

Inadéquation : ce qui est démontré n'est pas ce que vous croyez.

Démonstrations `Admitted`; axiomes faux ou incohérents.

Exemple : l'axiome du tiers exclu est incohérent avec l'option `-impredicative-set`.

Comment une démonstration mécanisée peut-être fausse ?

(R. Pollack, *How to Believe a Machine-Checked Proof*, 1997)

Inadéquation : ce qui est démontré n'est pas ce que vous croyez.

Démonstrations `Admitted`; axiomes faux ou incohérents.

Un bug dans une partie critique de l'implémentation Coq.

L'implémentation suit l'architecture de de Bruijn :

- un noyau qui revérifie les ≪termes de preuve≫ (critique);
- des tactiques qui construisent ces termes (non critiques).

Comment une démonstration mécanisée peut-être fausse ?

(R. Pollack, *How to Believe a Machine-Checked Proof*, 1997)

Inadéquation : ce qui est démontré n'est pas ce que vous croyez.

Démonstrations `Admitted`; axiomes faux ou incohérents.

Un bug dans une partie critique de l'implémentation Coq.

Une incohérence dans la logique implémentée par Coq.

Cohérence d’une logique

La cohérence d'une logique

Une logique est cohérente si elle ne peut pas déduire un paradoxe ou une absurdité évidente comme p.ex.

$P \wedge \neg P$	pour un certain paradoxe P (logique classique)
$\bot$	(noté `False` en Coq) (logique intuitionniste)
$0 = 1$	(arithmétique de Peano)
$\forall P.\ P$	(logique d'ordre supérieur)

De manière équivalente : une logique est cohérente s'il existe une proposition qui n'est pas déductible.

(Principe *ex falso quod libet* : à partir d'une absurdité on peut déduire toutes les propositions.)

Exemple : une logique intuitionniste

$$\Gamma_1, P, \Gamma_2 \vdash P \ (\text{Ax})$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} \ (\Rightarrow\text{I}) \qquad \frac{\Gamma \vdash P \Rightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q} \ (\Rightarrow\text{E, } \textit{modus ponens})$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \ (\wedge\text{I}) \qquad \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \ (\wedge\text{E}_1) \qquad \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q} \ (\wedge\text{E}_2)$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P} \ (\bot\text{E, } \textit{quod libet})$$

Cohérence = il existe un P tel qu'on ne peut pas dériver $\vdash P$.

Le second théorème d'incomplétude de Gödel

Théorème (Gödel, 1931)

Soit L une logique cohérente qui contient l'arithmétique de Peano. L'énoncé «L est cohérente» peut s'exprimer dans L mais ne peut pas se démontrer dans L.

Corollaire : une démonstration de cohérence d'une logique se fait forcément dans une logique «plus puissante».

PAT : propositions as types, proofs as programs

La correspondance de Curry-Howard met en relation un bon nombre de logiques (dont celle de Coq) avec un langage fonctionnel typé :

Langage typé	Logique
type	proposition
terme	démonstration, ≪construction≫
évaluation	élimination des coupures

(Voir mon cours 2018-2019.)

Propositions = types

Langage typé	Logique
fonctions $\sigma \to \tau$	$P \Rightarrow Q$ implication
produits $\sigma \times \tau$	$P \wedge Q$ conjonction
sommes $\sigma + \tau$	$P \vee Q$ disjonction
type à 1 constructeur `unit`	$\top$ la trivialité
type à 0 constructeurs `empty`	$\bot$ l'absurdité
polymorphisme $\forall\alpha.\ \tau$	$\forall X.P$ pour tout
type abstrait $\exists\alpha.\ \tau$	$\exists X..P$ il existe

Règles de déduction = règles de typage

Lambda-calcul simplement typé

$$\Gamma_1, x : A, \Gamma_2 \vdash x : A$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1\ M : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2\ M : B}$$

$$\frac{\Gamma \vdash M : \texttt{empty}}{\Gamma \vdash \texttt{match}\ M\ \texttt{with}\ \texttt{end} : A}$$

Règles de déduction = règles de typage

Logique intuitionniste

$$\overline{\Gamma_1, \quad A, \overline{\Gamma_2} \vdash \quad A}$$

$$\frac{\overline{\Gamma}, \quad A \vdash \quad B}{\overline{\Gamma} \vdash \quad A \Rightarrow B} \qquad \frac{\overline{\Gamma} \vdash \quad A \Rightarrow B \quad \overline{\Gamma} \vdash \quad A}{\overline{\Gamma} \vdash \quad B}$$

$$\frac{\overline{\Gamma} \vdash \quad A \quad \overline{\Gamma} \vdash \quad B}{\overline{\Gamma} \vdash \quad A \wedge B} \qquad \frac{\overline{\Gamma} \vdash \quad A \wedge B}{\overline{\Gamma} \vdash \quad A} \qquad \frac{\overline{\Gamma} \vdash \quad A \wedge B}{\overline{\Gamma} \vdash \quad B}$$

$$\frac{\overline{\Gamma} \vdash \quad \bot}{\overline{\Gamma} \vdash \quad A}$$

$\overline{\Gamma}$ est Γ sans les noms de variables, p.ex. $\overline{x : A, y : A} = A, A.$

Cohérence logique = type non habité

Langage typé	Logique
Type habité τ $\quad (\exists M.\ \emptyset \vdash M : \tau)$	Proposition démontrable P
Il existe un type non habité	La logique est cohérente

Une méthode pour montrer qu'un type n'est pas habité

(Étend la démonstration de sûreté du typage du 7e cours.)

Théorème (Formes canoniques)

Soit v une valeur. Si $\emptyset \vdash v : \sigma \to \tau$, alors v est de la forme $\lambda x.M$.
Si $\emptyset \vdash v : \sigma \times \tau$, alors v est de la forme (v_1, v_2).
Il est impossible que $\emptyset \vdash v : \texttt{empty}$.

Théorème (Préservation)

Si $\Gamma \vdash M : \tau$ et $M \to N$, alors $\Gamma \vdash N : \tau$.

Théorème (Progression)

Si $\emptyset \vdash M : \tau$, ou bien M est une valeur ou bien M se réduit.

Théorème (Normalisation)

Tout terme typable a une forme normale :
si $\Gamma \vdash M : \tau$, il existe N tel que $M \xrightarrow{*} N \not\to$

Une méthode pour montrer qu'un type n'est pas habité

Corollaire (Cohérence logique)

Le type `empty` *n'est pas habité.*

Démonstration.

Supposons qu'il existe M tel que $\emptyset \vdash M : \texttt{empty}$.

Par normalisation forte on a N tel que $M \xrightarrow{*} N \not\to$.

Par préservation on a $\emptyset \vdash N : \texttt{empty}$.

Par progression on a que N est une valeur.

Par formes canoniques, on a une contradiction. □

Divergence et incohérence

La plupart des mécanismes qui rendent les langages de programmation Turing-complets rendent les logiques incohérentes.

Exemple : la récursion générale

`let rec f x = f x in f ()` a le type τ pour tout τ.

Vu comme principe de preuve, c'est $(P \Rightarrow P) \Rightarrow P$ …

Exemple : les types algébriques à occurrence négatives

`Inductive t : Type := Lam: (t -> t) -> t`
encode le lambda-calcul pur, et donc la divergence.

`Induction P : Prop := Hyp: (P -> False) -> P`
est tel que `P <-> (P -> False)` et donc implique `False`.

Démontrer la normalisation

Démontrer la propriété de normalisation

Une approche due à Tait (1967) pour les types simples puis étendue à système *F* par Girard (1972). Un cas particulier de relation logique (Plotkin, 1973 ; Statman, 1985).

On définit les ensembles $RED(\tau)$ par récurrence sur le type τ :

$$RED(\iota) = \{M \mid M \text{ termine, c.à.d. } \exists N, M \xrightarrow{*} N \not\to\}$$
$$RED(\sigma \to \tau) = \{M \mid \forall N \in RED(\sigma),\ M\ N \in RED(\tau)\}$$

(On note ι tous les types de base, `bool`, `nat`, etc)

Normalisation des types simples

$$RED(\iota) = \{M \mid M \text{ termine, c.à.d. } \exists N, M \xrightarrow{*} N \not\to\}$$
$$RED(\sigma \to \tau) = \{M \mid \forall N \in RED(\sigma),\ M\ N \in RED(\tau)\}$$

On démontre alors :

1. Si $M \in RED(\tau)$ alors M termine.
2. Si $\emptyset \vdash M : \tau$ alors $M \in RED(\tau)$, ou, plus généralement :

   Si $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau$ et $M_i \in RED(\tau_i)$ pour tout i, alors $M\{x_1 \leftarrow M_1, \ldots, x_n \leftarrow M_n\} \in RED(\tau)$.

Extension aux types polymorphes

Dans un système prédicatif comme ML, ou la théorie des types de Martin Löf, ou Agda, on peut prendre

$$\mathit{RED}(\forall\alpha.\tau) = \{M \mid \forall\sigma, M[\sigma] \in \mathit{RED}(\tau\{\alpha \leftarrow \sigma\})\}$$

Cette définition reste bien fondée car α ne peut être instanciée que par des types σ «plus petits» que $\forall\alpha.\tau$.

Dans un système imprédicatif comme système F ou Coq, α peut être instanciée par tout type, y compris $\forall\alpha.\tau$. Exemple :

$$\text{si} \quad \mathit{id} : \forall\alpha.\,\alpha \to \alpha \quad \text{alors} \quad \mathit{id}\ [\forall\alpha.\,\alpha \to \alpha]\ \mathit{id} : \forall\alpha.\,\alpha \to \alpha$$

La définition de *RED* est donc incorrecte.

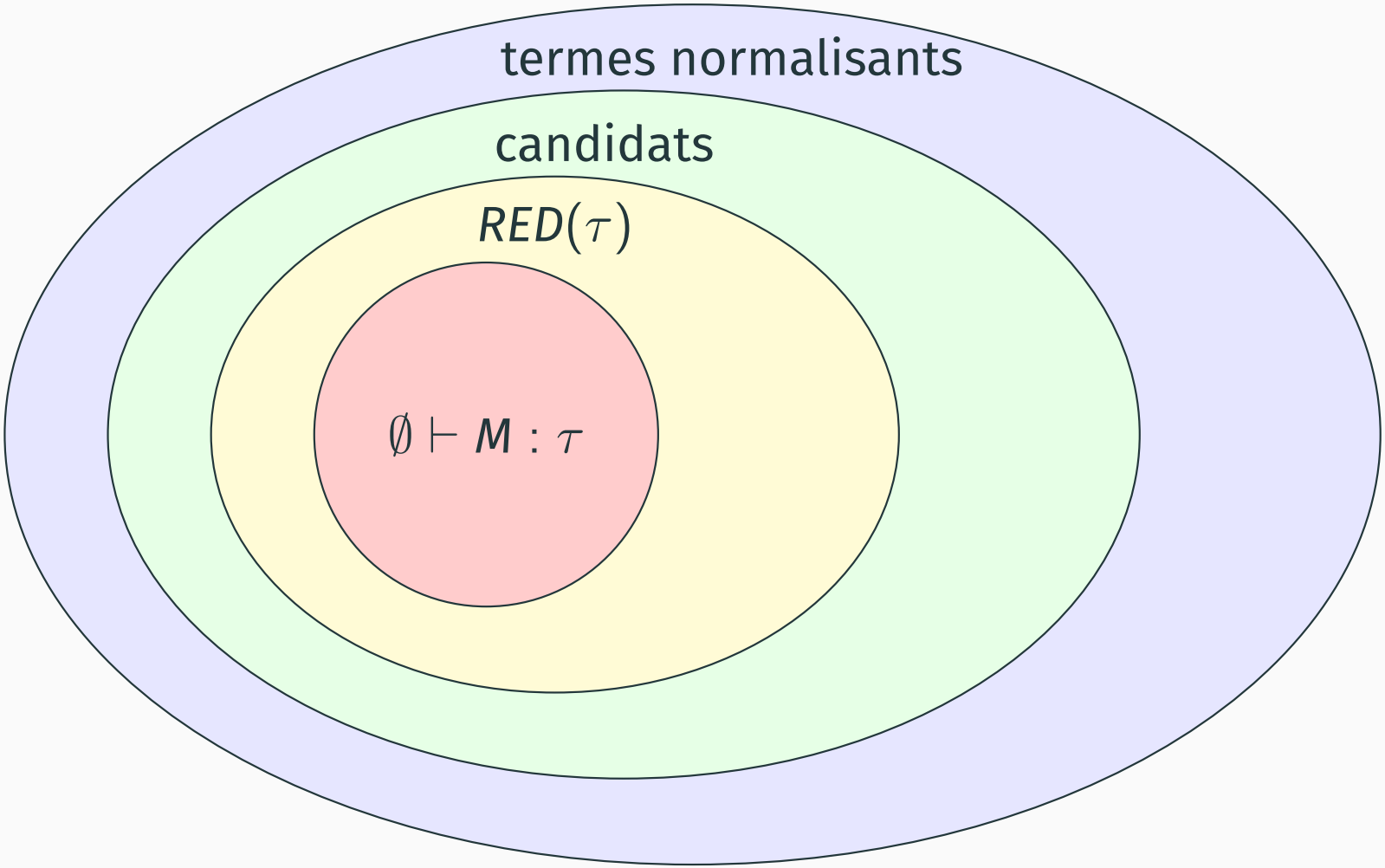
Les candidats de réductibilité

L'idée de Girard est d'interpréter les variables de type α pas seulement par les ensembles $RED(\sigma)$ pour un certain type σ, mais par une classe plus large d'ensembles :
les candidats de réductibilité.

Un ensemble U de termes de type τ est un candidat de réductibilité $U \in CAND(\tau)$ si

1. tout $M \in U$ termine ;
2. U est fermé par expansion : si $M \to M'$ et $M' \in U$ alors $M \in U$
3. U est fermé par certaines réductions.
   (Voir Girard, Lafont, Taylor, *Proofs and Types*, ch. 14)

Les candidats de réductibilité, visuellement



Normalisation du système F

Réductibilité : (Φ : variable de type $\to$ candidat)

$$
\begin{aligned}
\mathit{RED}(\iota, \Phi) &= \{M \mid M \text{ termine}\} \\
\mathit{RED}(\sigma \to \tau, \Phi) &= \{M \mid \forall N \in \mathit{RED}(\sigma, \Phi), M\ N \in \mathit{RED}(\tau, \Phi)\} \\
\mathit{RED}(\mathit{alpha}, \Phi) &= \Phi(\alpha) \\
\mathit{RED}(\forall \alpha.\, \tau, \Phi) &= \{M \mid \forall \sigma, \forall U \in \mathit{CAND}(\sigma), M[\sigma] \in \mathit{RED}(\tau,\ \Phi + \alpha \mapsto U)\}
\end{aligned}
$$

On démontre alors :

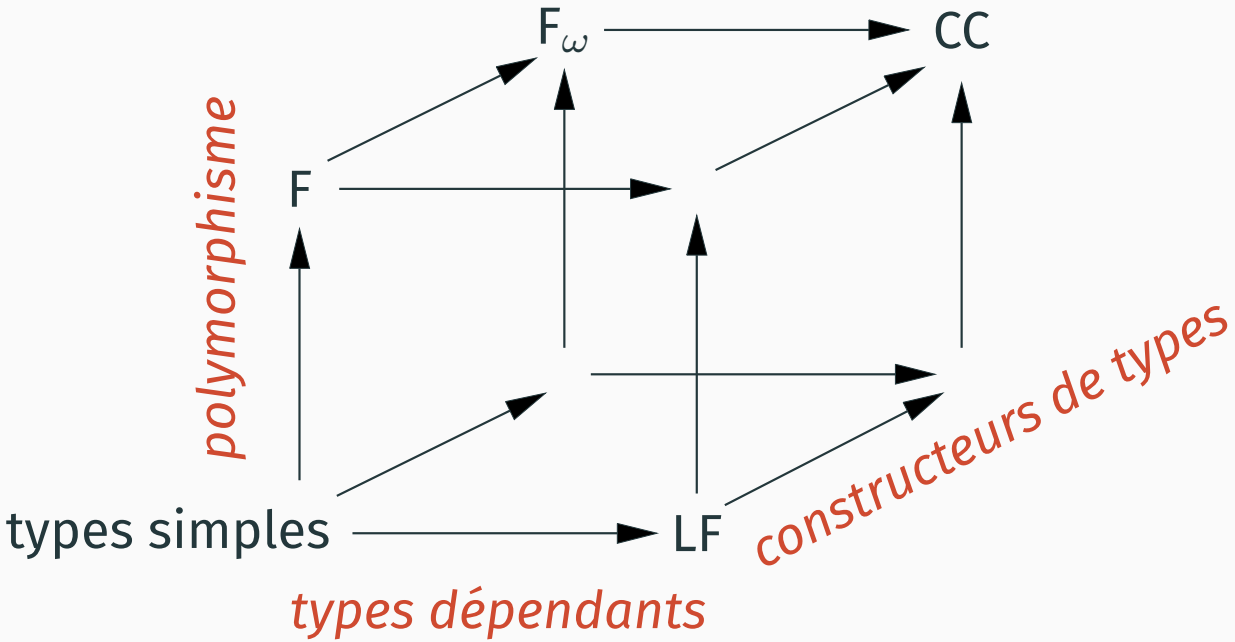
1. $\mathit{RED}(\tau, \Phi)$ est un candidat de réductibilité.
2. Si $\emptyset \vdash M : \tau$ alors $M \in \mathit{RED}(\tau, \Phi)$.

Formaliser et mécaniser Coq

Des types simples jusqu'au Calcul des Constructions

Types simples	$\texttt{neg} : \texttt{bool} \to \texttt{bool}$	terme $\mapsto$ terme
+ polymorphisme	$\texttt{id} : \forall\alpha.\ \alpha \to \alpha$	type $\mapsto$ terme
+ opérateurs de types	$\texttt{list} : \texttt{Type} \to \texttt{Type}$	type $\mapsto$ type
+ types dépendants	$\texttt{vec} : \texttt{nat} \to \texttt{Type}$	terme $\mapsto$ type

= Calcul des Constructions



Du Calcul des Constructions jusqu'à Coq

Calcul des Constructions

+ hiérarchie d'univers — $0 : \texttt{nat} : \texttt{Type}_0 : \texttt{Type}_1$
+ types inductifs — $\texttt{nat}, \texttt{list}, \wedge, \vee, \exists$
+ types coinductifs — $\texttt{stream}, \texttt{delay}$
+ cumulativité des univers
+ polymorphisme d'univers

$\approx$ Coq

Une formalisation du Calcul des Constructions avec univers

Dans le style des *Pure Type Systems* :

- Pas de distinction syntaxique entre termes et types.
- Un seul λ pour toutes les formes de fonctions
  ((terme $\mapsto$ terme, type $\mapsto$ terme, type $\mapsto$ type, etc)
- Un seul $\prod$ regroupant les types de fonction et les types $\forall$.
- Des univers pour stratifier en termes, types, sortes, etc.

Syntaxe abstraite

Univers :	U	$::= \texttt{Prop} \mid \texttt{Type}_i$	
Termes, types :	A, B	$::= x$	variables
		$\mid \lambda x : A.\, B$	abstractions
		$\mid A\ B$	applications
		$\mid U$	nom d'univers
		$\mid \Pi x : A.\, B$	type dépendant de fonction

Notation : $A \to B \stackrel{def}{=} \Pi x : A.\, B$ si x non libre dans B.

Règles de typage

$$\frac{(U, U') \in \mathcal{A}}{\emptyset \vdash U : U'}\ (\text{ax}) \qquad \frac{\Gamma \vdash A : U}{\Gamma, x : A \vdash x : A}\ (\text{var}) \qquad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : U}{\Gamma, x : C \vdash A : B}\ (\text{wk})$$

$$\frac{\Gamma \vdash A : U_1 \quad \Gamma, x : A \vdash B : U_2 \quad (U_1, U_2, U_3) \in \mathcal{R}}{\Gamma \vdash \Pi x : A.B : U_3}\ (\text{pi})$$

$$\frac{\Gamma, x : A \vdash B : C \quad \Gamma \vdash \Pi x : A.C : U}{\Gamma \vdash \lambda x : A.\, B : \Pi x : A.\, C}\ (\text{abstr})$$

$$\frac{\Gamma \vdash f : \Pi x : A.\, B \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B\{x \leftarrow a\}}\ (\text{app})$$

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : U \quad B \overset{*}{\rightarrow}\overset{*}{\leftarrow} B'}{\Gamma \vdash A : B'}\ (\text{conv})$$

La règle de conversion : typage modulo réductions

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : U \qquad B \overset{*}{\rightarrow}\overset{*}{\leftarrow} B'}{\Gamma \vdash A : B'} \text{ (conv)}$$

Les types sont identifiés à réductions (calculs) près.

Exemple 1 : le type `dtype (Fun Bool Bool)` contient les mêmes valeurs que le type `bool` $\rightarrow$ `bool`, car ces deux types sont égaux modulo calcul de la fonction `dtype`.

Exemple 2 : la preuve triviale de la proposition `4 = 4` est aussi une preuve de la proposition `2 + 2 = 4`, car ces deux propositions sont égales modulo calcul de la fonction +.

La règle de conversion : typage modulo réductions

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : U \qquad B \overset{*}{\rightarrow}\overset{*}{\leftarrow} B'}{\Gamma \vdash A : B'} \text{ (conv)}$$

Les types sont identifiés à réductions (calculs) près.

- Permet de nouvelles formes de programmation et de démonstration, p.ex. les démonstrations ≪par réflexion≫, où le calcul remplace la déduction logique.
- Un défi pour la métathéorie : le typage dépend du calcul.
- Un défi pour l'implémentation du vérificateur de types : besoin d'un évaluateur efficace pendant le typage.

La gestion des univers

$$\frac{(U, U') \in \mathcal{A}}{\emptyset \vdash U : U'}\ (\text{ax}) \qquad \frac{\Gamma \vdash A : U_1 \quad \Gamma, x : A \vdash B : U_2 \quad (U_1, U_2, U_3) \in \mathcal{R}}{\Gamma \vdash \Pi x : A.B : U_3}\ (\text{pi})$$

La relation $\mathcal{A}$ dit quel univers appartient à quel univers. En Coq :

$$\mathcal{A} = \{(\texttt{Prop}, \texttt{Type}_0), (\texttt{Type}_i, \texttt{Type}_{i+1})\}$$

La relation $\mathcal{R}$ détermine l'univers où «vit» $\Pi x : A.B$. En Coq :

$$\mathcal{R} = \{(U, \texttt{Prop}, \texttt{Prop}), (\texttt{Type}_i, \texttt{Type}_j, \texttt{Type}_{\max(i,j)})\}$$

Essentiel pour la cohérence logique ! P.ex. `Type : Type` ou le système U de Girard peuvent coder le paradoxe de Burali-Forti…

Vers la mécanisation en Coq de la logique de Coq

B. Barras, *Coq en Coq*, 1996.
Une formalisation complète de CC en Coq version 6.
Inclut normalisation, cohérence logique, et preuve + extraction d'un vérificateur de types.

B. Barras, *Auto-validation d'un système de preuves avec familles inductives*, thèse, 1999.
Extension à CC + types inductifs.
La normalisation n'est pas démontrée.

Vers la mécanisation en Coq de la logique de Coq

A. Charguéraud, *Locally nameless tutorial*, vers 2010.

`https://www.chargueraud.org/softs/ln/`

Une formalisation simple de CC + univers en Coq.
S'arrête au théorème de préservation.

M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, Th. Winterhalter, *Coq Coq Correct! Verification of type checking and erasure for Coq, in Coq*, 2020.

Une formalisation de PCUIC (Polymorphic Cumulative Calculus of Inductive Constructions). Admet la normalisation. Vérifie le reste de la métathéorie, un vérificateur de types efficace, et un algorithme d'extraction.

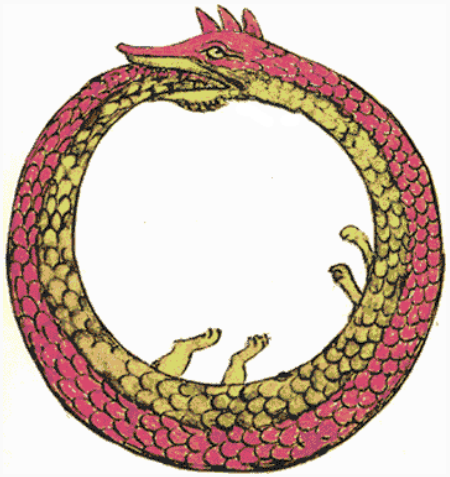
Vers la mécanisation en Agda de la logique d'Agda

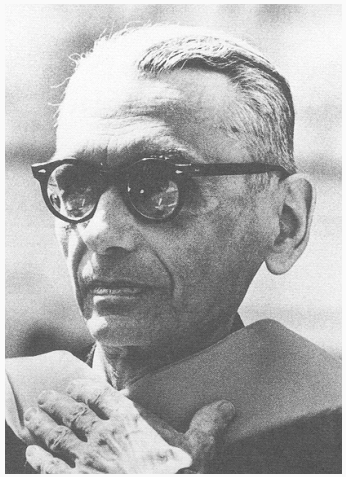
J. Chapman, *Type theory should eat itself*, 2008
Vers un algorithme de normalisation pour MLTT, en syntaxe intrinsèquement typée.

T. Altenkirch, A. Kaposi, *Type theory in type theory using quotient inductive types*, 2016
Une spécification de MLTT en syntaxe intrinsèquement typée, utilisant les types quotients de HoTT.

A. Abel, J. Öhman, A. Vezzosi, *Decidability of conversion for type theory in type theory*, 2018
Un algorithme de test de conversion pour types dépendants (1 univers) en Agda (MLTT + induction-récursion).

Proof assistants should eat themselves ?





Peut-on mécaniser un bon fragment de la logique d'un assistant à la démonstration dans un fragment à peine plus gros ?