Premier contact

Preuves en logique propositionnelle

- DÉMO `logique.v`
- `Prop` le *type* des propositions logiques
- `forall A B : Prop, ...`
  - `A` et `B` sont de type `Prop`
  - `A` et `B` sont des énoncés logiques

Programmation en Coq

- DÉMO `arbres.v`
- `Set` le "type des types" des programmes

  `nat:Set` `bool:Set` `nat->nat:Set`
- en Coq, toutes les fonctions
  - terminent
  - sont totales (pas de `failwith`, p.ex. pour `List.tl`)
  - ont des appels récursifs avec des arguments qui décroissent structurellement (sur l'*un* de leurs arguments (fixé))

    ↪ garantie de terminaison
- *rien* n'est primitif, ou plutôt le moins possible
  - booléens, entiers, ...
  - NB: `bool` ≠`Prop`

Mélanger preuves et programmes

- `Prop` et `Set` cohabitent en Coq
  - un module: un ensemble de types, de programmes, et de propriétés sur ces programmes
  - un enregistrement: programmes et propriétés
    - p.ex. un groupe c'est un ensemble support, un neutre, une loi de composition, et des preuves sur la loi de composition
- le parallèle preuves/programmes

`Set`	`Prop`
`nat->nat`	`forall n:nat, n+0 = n`
`double`	`n_zero`
`Definition double : nat->nat :=...`	`Lemma n_zero : forall n:nat, n+0 = n.` ⋮

- principe de l'extraction: on efface tout ce qui est `Prop`

  `forall k:nat, exists n:nat,` $n^2 \leq k \leq (n+1)^2$

  `nat:Set` `e≤e':Prop`

  DÉMO `extract.v`

Lever le voile

La correspondance de Curry-Howard

- Coq est un *Caml aux hormones*
  - *davantage de types, moins de programmes*
  - on veut davantage de contrôle
- on va le voir, les mathématiques qui sont "au-dessus" du langage (*meta*) en Caml sont ici accessibles
- tout s'articule autour du typage
  - les valeurs (structures de données, fonctions) ont un type
  - les types sont aussi des propriétés logiques
    - les valeurs sont alors des preuves

      (que l'on ne voit guère explicitement)
  - les types ont des types (`Prop`, `Set`, `Type`)
- voyons certains aspects du système de types de Coq
  - théorie des types (Russel, Martin-Löf)
  - Calcul des Constructions (Coquand, 1985)
  - Calcul des Constructions Inductives (Paulin-Mohring, Werner)

Typer les fonctions

- rappel: le λ-calcul simplement typé
  - termes $M ::= x \mid \lambda x.M \mid (M_1\,M_2)$
  - réduction (sémantique opérationnelle) β+autres règles
  - types simples $\tau ::= \iota \mid \tau_1 \to \tau_2$

  règles de typage

$$\frac{}{\Gamma, x:\tau \vdash x:\tau}\qquad \frac{}{\Gamma, x:\tau \vdash x:\tau}$$

$$\frac{\Gamma, x:\tau_1 \vdash M:\tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \to \tau_2}\qquad \frac{\Gamma, x:\tau_1 \vdash M:\tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash (M_1\,M_2) : \tau_2}\qquad \frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash (M_1\,M_2) : \tau_2}$$

- déduction naturelle en logique minimale

  Curry-Howard

Le rôle des tactiques

- après `Qed`, on peut regarder la preuve: c'est un λ-terme

  DÉMO `preuve.v`
- l'application d'une tactique (élémentaire) fait un petit bout de la construction du λ-terme de preuve
  - `intro` descendre sous un λ
  - `apply` appliquer le modus ponens, i.e., mettre un nœud "application"
  - `exact` le cas de la variable en λ-calcul
  - `assert` une β-expansion
- en principe, on pourrait tout le temps écrire

  `exact` ⟨gros machin⟩ ...bon courage
- tactiques comme `auto` : "intelligence artificelle"
- de nombreuses autres tactiques manipulent le contexte, mais ne "construisent" rien
- prouver, c'est construire un terme
  - DÉMO `double.v`
  - l'égalité en Coq est *intensionnelle*

    deux algorithmes extensionnellement égaux ne sont pas nécessairement convertibles

Types dépendants

- parmi les types, on peut construire des *produits dépendants*

$$\Pi(x : T).\ T' \quad \text{type dépendant}$$

  - un *type* peut dépendre d'un *terme*
- en logique
  - $\geq : \texttt{nat} \to \texttt{nat} \to \texttt{Prop}$
    en particulier, $5 \geq 2 : \texttt{Prop}$ est un *type* (de type Prop)
  - $\Pi(n : \texttt{nat})\, n * n \geq n$ représente $\forall n.\, n * n \geq n$
- en programmation
  - $\Pi(n : \texttt{nat}).\ \mathit{list}(n)$ type des listes à n éléments
  - à chaque n est associé un type donné
    - $\mathit{list}(19)$ et $\mathit{list}(27)$ sont des types différents
    - $\mathit{append} : \Pi(n).\ \mathit{list}(n) \to \Pi(k).\ \mathit{list}(k) \to \mathit{list}(n+k)$
- cas particulier: $\Pi(x : T).\ T'$
  lorsque x n'apparaît pas dans T', c'est $T \to T'$

Types dépendants, règles de typage

- règles légèrement simplifiées, pour l'abstraction et l'application

$$\frac{\Gamma, x : T \vdash M : T' \qquad \Gamma \vdash \Pi(x : T)\ T' : \texttt{Type}}{\Gamma \vdash \lambda x.\, M : \Pi(x : T).\ T'}$$

$$\frac{\Gamma \vdash M : \Pi(x : T).\ T' \qquad \Gamma \vdash M' : T}{\Gamma \vdash (M\, M') : T'[M'/x]}$$

- on type le type (et on vérifie la bonne formation du type dépendant)
  - dans les deux règles ci-dessus, T' *parle de* x
  - $T'[M'/x]$: on substitue dans le type
- règle de *conversion*

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T \equiv U \qquad \Gamma \vdash U : \texttt{Type}}{\Gamma \vdash t : U}$$

- enregistrements dépendants: dépendances entre les champs

Retour sur les connecteurs logiques

- logique classique, logique intuitionniste
  - A n'est pas équivalent à $\neg\neg A$
    DÉMO negneg.v
    - l'axiome du tiers exclu (`forall A, A\/~A`) n'est pas dans la théorie qui sous-tend Coq
    - des symétries manquent en logique intuitionniste
    - la loi de Peirce
      `forall A B, ((A->B)->A)->A`
      ne peut être prouvée en Coq
- les connecteurs logiques sont définis inductivement
  - DÉMO connecteurs.v vide.v
  - `split, right, left` tactiques spécialisées

La fonction `printf` en Cayenne

- Cayenne: extension de Haskell avec des types dépendants (L. Augustsson)
- le code:

```
PrintfType ::  String -> #  (* #:  le type de tous les types *)
PrintfType "" =                  String
PrintfType ('%':'d':cs) = Int    -> PrintfType cs
PrintfType ('%':'s':cs) = String -> PrintfType cs
PrintfType ('%':_:cs) =           PrintfType cs
PrintfType (cs) =                 PrintfType cs

printf ::  (fmt::String) -> PrintfType fmt
printf fmt = pr fmt ""

pr ::  (fmt::String) -> String -> PrintfType fmt
pr "" res = res
pr ('%':'d':cs) res = \  (i::Int) -> pr cs (res ++ show i)
pr ('%':'s':cs) res = \  (s::String) -> pr cs (res ++ s)
pr ('%':c:cs) res = \  (s::String) -> pr cs (res ++ [c])
pr ('%':c:cs) res = \  (s::String) -> pr cs (res ++ [c])
```

- `if b then 2008 else "bonne annee!"`
  - typable *(en principe)* avec les types dépendants
  - le type du résultat d'une fonction peut dépendre de la valeur de l'argument

Types inductifs

- définition inductive
  - plus petit ensemble clos par application des constructeurs
  - automatiquement: principes pour
    - faire des preuves sur les gens du nouveau type
      principe de récurrence
    - programmer avec les gens du nouveau type
      `match...with` de Caml
    - encore une fois, c'est pareil (Curry Howard)
  - tarte à la crème: les naturels DÉMO nats.v
- types inductifs dans `Set`: structures de données
- types inductifs dans `Prop`: prédicats définis "à la Prolog"
  DÉMO petit.v
- types inductifs: contraintes
  - la récursion est *structurelle*
  - le filtrage doit être *total*
  - utilisation dans les constructeurs du type que l'on est en train de définir soumise à des contraintes (positivité)

Maturité de Coq

- POPLMark challenge
  - *Principles Of Programming Languages*
    conférence majeure dans le domaine de la conception des langages de programmation
  - slogan: les articles qu'on soumet à POPL devraient être prouvés en Coq
    - en est-on là?
    - que manque-t-il?
    - Coq Isabelle Twelf PVS HOL ...?
- Compilateur certifié (Leroy et al.)
  - on vérifie le soft
  - on vérifie le hard
  - le chaînon manquant, plein de technologie complexe
- statut de la preuve en mathématiques?
  - preuve du théorème des quatre couleurs en Coq
    (Gonthier, Werner)
  - je recommande
    G. Dowek, *"Les Métamorphoses du calcul : Une étonnante histoire des mathématiques"* *(grand public)*