De Coq à ML : l'extraction de programmes

Pierre Letouzey

Equipe πr^2 (PPS, Paris 7 / INRIA)

Coq 8.3 : avis aux courageux ...

Meme s'il n'y a pas encore d'annonce officielle ...

Le processus menant à Coq 8.3 est en route !
(une branche svn existe :-)

Durant cette phase de debug, tout retour d'experience est le bienvenu

Plan

I. **Introduction à l'extraction en Coq**
   1. Qu'est-ce que l'extraction ?
   2. Un exemple : la division euclidienne
   3. Difficultés

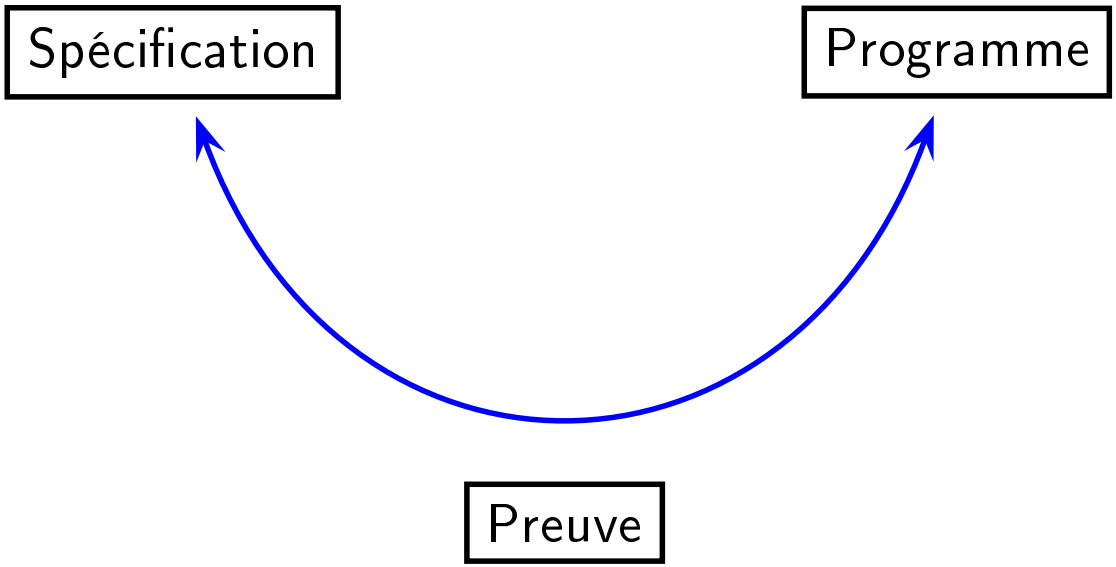
II. **Un peu de théorie ...**
   1. Définition formelle
   2. Preuves de correction
   3. Typage des termes extraits
   4. Modules

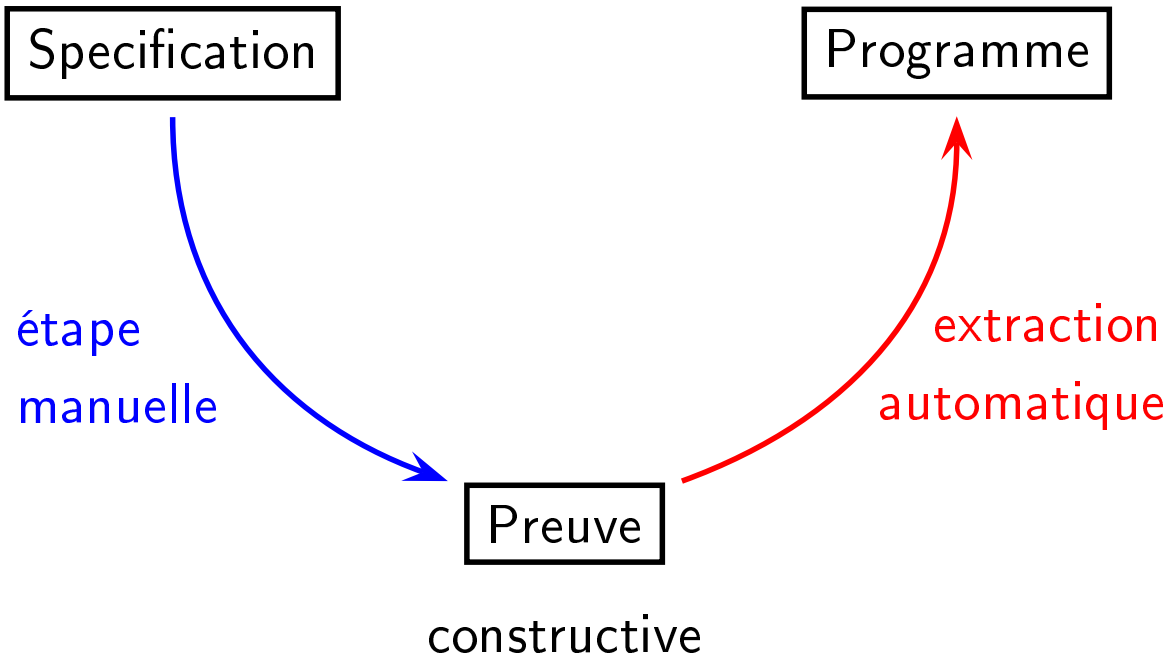
III. **Études de cas**

A priori demain...

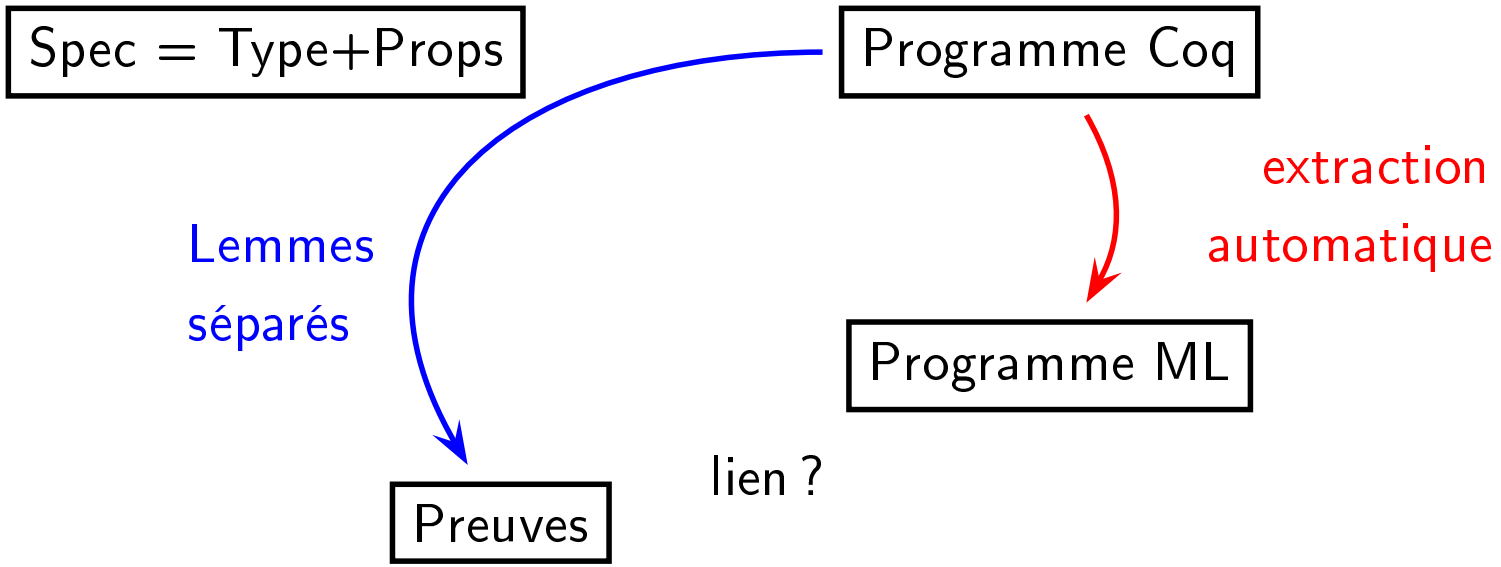
I

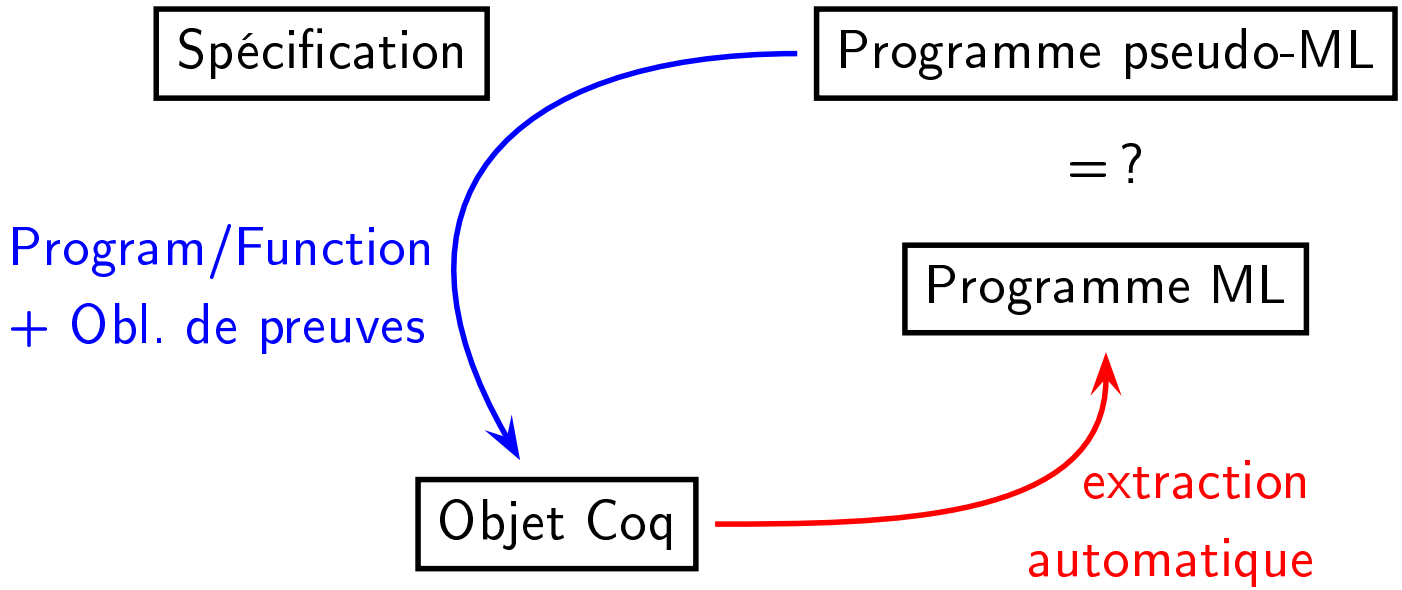
Introduction à l'extraction en Coq


Spécification
Programme
Preuve



fonctionnel :
Ocaml,
Haskell,...


Spec = Type+Props
Programme Coq
extraction
automatique
Lemmes
séparés
Programme ML
lien ?
Preuves


Spécification
Programme pseudo-ML
= ?
Program/Function
+ Obl. de preuves
Programme ML
Objet Coq
extraction
automatique

- PX (Hayashi, 1988)
- Nuprl (Constable & alii, 1985)
- Minlog (Schwichtenberg & alii, 1990)
- Isabelle (Berghofer, 2000)
- Phox (Raffalli, 2001 ?)
- Coq (Paulin & Werner, 1989)
- Coq (Letouzey, 2000-2004)
- Coq, extraction classique (Miquel, 2009)
- ...

- Spécification :

  $\Rightarrow$ Via son type

  $$\text{div} : \texttt{nat} \to \texttt{nat} \to \texttt{nat}$$

  $\Rightarrow$ Via une assertion : si `div a b = q`, alors :

  $$\texttt{q*b} \leq \texttt{a} < \texttt{(q+1)*b}$$

Exemple : la division euclidienne

- Spécification :

  $\Rightarrow$ Via son type dépendant

  $$\forall a{:}\mathtt{nat}, \forall b{:}\mathtt{nat}, b{\neq}0 \rightarrow \exists\, q{:}\mathtt{nat}, q{*}b \leq a < (q{+}1){*}b$$

  $\Rightarrow$ Véritable notation Coq :

  $$\forall a{:}\mathtt{nat}, \forall b{:}\mathtt{nat}, b{\neq}0 \rightarrow \{\, q{:}\mathtt{nat} \mid q{*}b \leq a < (q{+}1){*}b \,\}$$

- Preuve :

$$\forall \texttt{a:nat}, \forall \texttt{b:nat}, \texttt{b} \neq 0 \rightarrow \{\, \texttt{q:nat} \mid \texttt{q*b} \leq \texttt{a} < \texttt{(q+1)*b} \,\}$$

  Un script Coq de 10 lignes ...

  ... qui engendre un objet interne de 400 lignes

- Programme :

```
let rec div a b =
  match le_lt_dec b a with
    | true → S (div (minus a b) b)
    | false → O
```

On retrouve le type `nat → nat → nat`

1. Fournir du code fiable :
   - Obtenir de bonne propriétés de correction
2. Fournir du code rapide et concis :
   - Identifier les parties utiles (calculatoires)
   - Supprimer les parties superflues (logiques)
3. Fournir du code facilement (ré-)utilisable :
   - Acceptation par les compilateurs usuels $\Rightarrow$ typage
   - Intégration dans des développements plus larges $\Rightarrow$ interface

- Paulin / Werner (1989) :
  - D'abord de CC vers F_ω, étendu ensuite aux inductifs
  - Distinction `Prop`/`Set`
  - Techniques à base de réalisabilité
- Berardi / Boerio / Prost (1996) :
  - Analyse de code-mort
  - Le code inutile est remplacé par une constante dédiée
  - Techniques de simulation
- Pottier (2000) : Encapsulation du λ-calcul pur dans Ocaml

$$\forall a{:}\mathtt{nat}, \forall b{:}\mathtt{nat}, b{\neq}0 \rightarrow \{\, q{:}\mathtt{nat} \mid q{*}b \leq a < (q{+}1){*}b \,\}$$

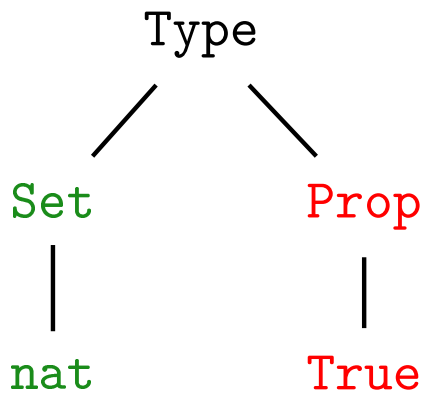
La séparation calculatoire/logique est assurée par le typage :

- `nat` est un type calculatoire, car `nat : Set`

  `Inductive nat : Set := O : nat | S : nat → nat`

- `b=0` est une proposition logique, car `b=0 : Prop`
- la séparation se propage aux cas non-primitifs.
  `b≠0` est en fait `b=0 → False`, de type `Prop`
- certaines construction combinent calculatoire/logique :
  $\{\, q : \mathtt{nat} \mid q{*}b \leq a < (q{+}1){*}b \,\}$ de type `Set`
- Pour tout `t` bien typé, `t : T : s` avec `s=Set`, `Prop`, ou `Type`

1. Support de l'ensemble de la logique de Coq ?
   - Mélange de sortes en passant par `Type`
   - Elimination forte : des types dependants de valeurs
   - Prop dans les calculs : `false_rec`, `eq_rec`, `Acc_rec`
2. Typabilité en ML ?
3. Exécution correcte ?
   - Eviter les "runtime error"
   - Eviter de boucler



Un terme Coq hybride :

```
if b then 0 else True
     : if b then nat else Prop
     : Type
```

```
Definition dp : (∀X:Set, X→X) → nat*bool :=
 fun f ⇒ (f nat 0, f bool true).
```

Extraction naturelle : `let dp = fun f → (f 0, f true)`

Mais ceci n'est pas typable en ML

Autre exemple :

```
Inductive any : Set := Any : ∀A:Set, A → any.
```

Une traduction infidèle : `type 'a any = Any of 'a`

- Reprenons `div : ∀a:nat, ∀b:nat, b≠0 → { q:nat | ...}`

  En Coq, on peut former `(div 0 0) : 0≠0 → { q:nat | ...}`

  L'extraction naïve est `(div 0 0)`, qui boucle.

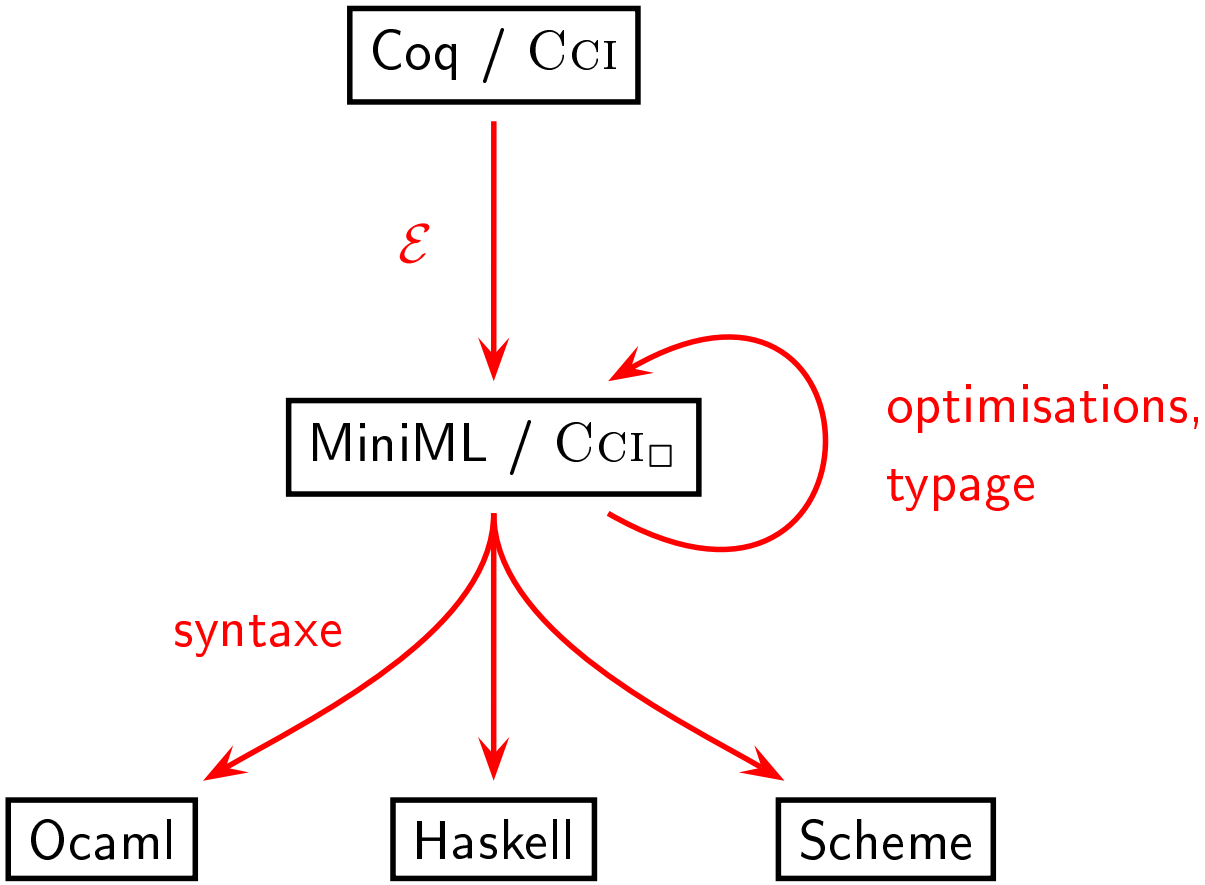
- Plus généralement, l'extraction naïve peut obliger à réduire des portions absurdes, inaccessibles en Coq :
  ⇒ levée d'exceptions, et même erreurs comme (0 0).

L'ancienne théorie de C. Paulin mais ne couvrait pas ces cas,
mais l'ancienne implantations si (oups).

II

Un peu de théorie...

- Compatibilité autant que possible : `Prop`/`Set`
- Une première passe pouvant laisser des traces logiques : $\Box$
- Des optimisations ultérieures qui en retirent beaucoup
- Un vérificateur de types ML embarqué dans l'extraction


Coq / CCI
$\mathcal{E}$
MiniML / CCI$_\Box$
optimisations,
typage
syntaxe
Ocaml
Haskell
Scheme

Deux cas d'élagage :

- Si t admet `Prop` comme sorte, alors $\mathcal{E}(t) = \square$
- Si t est un schéma de types, alors $\mathcal{E}(t) = \square$

Sinon, on discrimine selon la structure de t :

- $\mathcal{E}(\texttt{fun}\ x : T \Rightarrow t) = \texttt{fun}\ x \Rightarrow \mathcal{E}(t)$
- $\mathcal{E}(u\ v) = (\mathcal{E}(u)\ \mathcal{E}(v))$
- ...

Modifications de la reduction

Un redex au niveau Coq reste-t'il un redex au niveau extrait ?

$(\texttt{fun}\ x : T \Rightarrow t)\ u$ devient normalement soit $(\texttt{fun}\ x \Rightarrow \mathcal{E}(t))\ \mathcal{E}(u)$ soit $\square$

Mais apres quelques reductions, un $\square$ peut se retrouver appliqué[a].

Au niveau théorique, une réduction $(\square\ x) \rightarrow \square$. Au niveau pratique, une implantation récursive de $\square$.

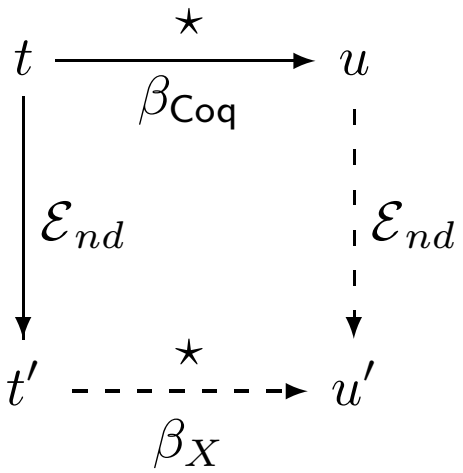
Pour les inductifs, des `match` $\square$ `with`... peuvent exister. Deux cas seulement :

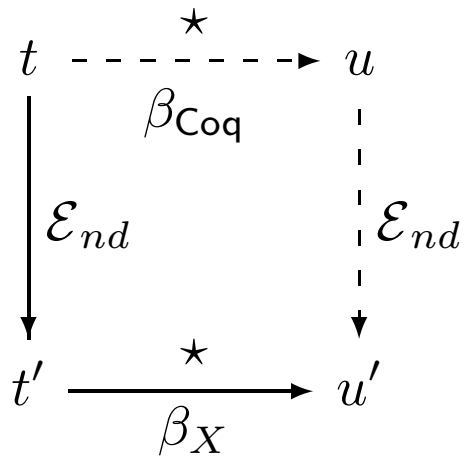
- zero branche (analyse à la `False_rec`) : pas de réduction, traduction finale en exception
- une branche (analyse à la `Eq_rec`) : on réduit directement vers cette branche.

Les points-fixes à arguments logiques (`Acc_rec`) sont aussi à traiter à part

[a] cf. `test_extraction.v` ligne 427

$\Rightarrow$ Preuve opérationnelle de bonne réduction





Condition : t clos

Deux cas pour la réduction X :

- réduction faible + stratégie paresseuse/CBN/Haskell
- réduction faible + stratégie stricte/CBV/Ocaml

En réalisabilité, usuellement ...

$$\begin{array}{lll}
\tau(A) & = \texttt{unit} & A \text{ sans contenu} \\
\tau(A \Rightarrow B) & = \tau(A) \Rightarrow \tau(B) & \\
\tau(\forall y{:}\sigma.\, B) & = \sigma \Rightarrow \tau(B) & \\
\tau(\exists y{:}\sigma.B) & = \sigma \times \tau(B) &
\end{array}$$

$$\begin{array}{lll}
x \ \mathbf{r} \ A & = A & A \text{ sans contenu} \\
f \ \mathbf{r} \ A \Rightarrow B & = \forall x{:}\tau(A).\ x \ \mathbf{r} \ A \ \Rightarrow \ f(x) \ \mathbf{r} \ B & \\
f \ \mathbf{r} \ \forall x{:}\sigma.\, B & = \forall x{:}\sigma.\ f(x) \ \mathbf{r} \ B & \\
x \ \mathbf{r} \ \exists y{:}\sigma.\, B & = \texttt{snd(x)} \ \mathbf{r} \ \texttt{B[y} := \texttt{fst(x)]} &
\end{array}$$

$\Rightarrow$ Preuve sémantique des propriétés logiques du code extrait

Par réalisabilité : si `t:T`, on prouve $\mathcal{E}$`(t) r T`

En fait ici : si `t:T`, on prouve `t` $\sim_{\texttt{T}}$ $\mathcal{E}$`(t)`

Par exemple pour `T=nat`, on définit $\sim_{\texttt{nat}}$ à partir de
`O` $\sim_{\texttt{nat}}$ $\texttt{O}_{\mathcal{E}}$ et `n` $\sim_{\texttt{nat}}$ `n'` $\rightarrow$ `(S n)` $\sim_{\texttt{nat}}$ $(\texttt{S}_{\mathcal{E}}$ `n')`

Et la propriété que l'on obtient pour `div` $\sim$ `div'` est :
$\forall$`a a',` `a` $\sim_{\texttt{nat}}$ `a'` $\rightarrow$
$\forall$`b b',` `b` $\sim_{\texttt{nat}}$ `b'` $\rightarrow$
$\forall$`h:b`$\neq$`O,` $\forall$`h',`
$\exists$`q q', (div a b h) = (q,_)` $\wedge$ `(div' a' b' h') = (q',_)`
$\wedge$ `q` $\sim_{\texttt{nat}}$ `q'` $\wedge$ `q*b`$\leq$`a<(q+1)*b`

- Les termes extraits vivent dans $\Lambda \neq \mathrm{CCI}$ (ex : `Inductive` Λ)
- On ne parle pas ici de terminaison, $\mathtt{p} \in \Lambda$ est vu modulo β
- Le système CCI exact est ingérable : besoin de marquer l'usage de la cumulativité (si T : `Prop` alors $T^{\dagger}$: `Type`)
- Usage de pairs dépendantes : $\sim_T = [\![T]\!]_2$

```
Record Set+ : Type := mk_Set
 { type_Set :> Set;
   pred_Set : type_Set → Λ → Prop }.

Record Prop+ : Type := mk_Prop
 { type_Prop :> Prop;
   pred_Prop : type_Prop → Λ → Prop := fun _ _ ⇒ True
```

- $[\![s]\!] = \texttt{mk\_Type}\ \ s^+\ \ \lambda\_,\ \lambda\_,\ \texttt{True}$
- $[\![t^\dagger]\!] = \texttt{Prop}^+\texttt{\_Type}^+\ [\![t]\!]$
- $[\![x]\!] = x$
- $[\![t\ t']\!] = [\![t]\!]\ [\![t']\!]$
- $[\![\lambda x : T,\ t]\!] = \lambda x : [\![T]\!]_{\mathbb{1}},\ [\![t]\!]$
- $[\![\forall x : T,\ T']\!] =$

  $\texttt{mk\_}s$

  $\forall x : [\![T]\!]_{\mathbb{1}},\ [\![T']\!]_{\mathbb{1}}$

  $\lambda t,\ \lambda p,\ \forall x : [\![T]\!]_{\mathbb{1}},\ \forall x' : \Lambda,\ [\![T]\!]_{\mathbb{2}}\ x\ x' \to [\![T']\!]_{\mathbb{2}}\ (t\ x)\ (p\ x')$
- ...

Travaux de thèse de Stéphane Glondu, a la fois par bisimulation et par réalisibilité.

- Des programmes extraits parfois mal typés, mais corrects
- `Obj.magic` : contournement local du typage, non-sûr

  `(Obj.magic t) : 'a`

  `(Obj.magic t)` $\rightarrow_\beta$ `t`
- But : en utiliser le moins possible
- But : des types proches de ceux de Coq, et prédictibles

$\Rightarrow$ Pour tout type `T` de Coq, une approximation Ocaml $\widehat{\mathcal{E}}(\mathtt{T})$, utilisant éventuellement un type inconnu $\mathbb{T}$

$\Rightarrow$ Pour `t:T`, un vérificateur de types impose $\mathcal{E}(\mathtt{t}) : \widehat{\mathcal{E}}(\mathtt{T})$ en plaçant des `Obj.magic` dans $\mathcal{E}(\mathtt{t})$.

Algorithme de typage $\mathcal{M}$ (Caml Light / Lee & Yi 1998) :

- Variante de $\mathcal{W}$, mais top-down et non bottom-up.
- Intérêt pour l'extraction : vérifie plutôt qu'infère un type.
- On commence donc avec le type espéré $\widehat{\mathcal{E}}(T)$.
- Seule modification : une unification qui ne rate jamais !
  Échec $\Rightarrow$ insertion d'un `Obj.magic`.

- Des termes extraits qui typent toujours, même avec l'extraction des modules.
- Mieux, on connaît leur type à l'avance : production de fichiers interface `.mli`.
- Un critère de simplicité : pas d'`Obj.magic` s'ils sont superflus pour que le terme extrait ait le type attendu.
- Mais parfois plus d'`Obj.magic` qu'il semble nécessaire.

Coq propose un système de modules/foncteurs/signatures

Similaire au système d'Ocaml, avec en plus les aspects Coq : types dépendants, parties logiques, ...

$\Rightarrow$ L'extraction traduit de l'un vers l'autre

Application : la certification de bibliothèques

Un exemple significatif : FSets

III

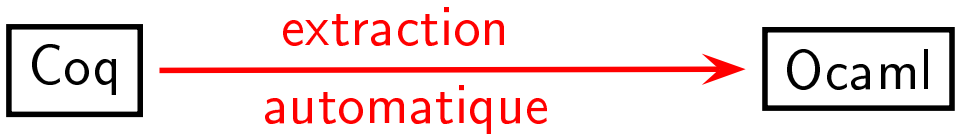
Études de cas

À l'origine, la bibliothèque Set de Ocaml :

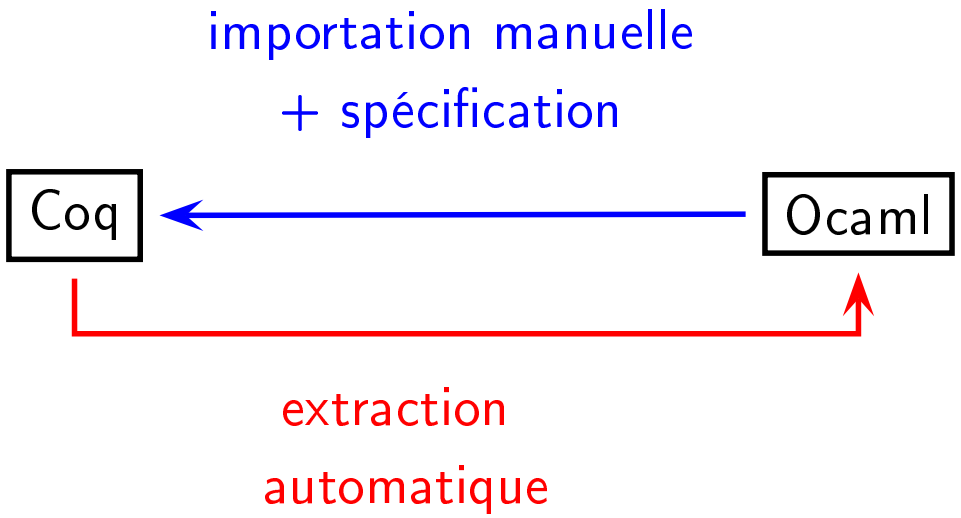
```
module type OrderedType = sig
  type t
  val compare : t → t → int
end
module Make (X:OrderedType) : sig

  type t
  val empty : t
  val add : X.t → t → t
  val union : t → t → t
  ⋮
end
```

- Extraction usuelle :



- Ici, on s'appuie sur du code Ocaml existant :



```
Module Type OrderedType.
  Parameter t : Set.
  Parameter eq lt : t → t → Prop.
  Parameter compare : ∀x y,{lt x y}+{eq x y}+{lt y x}.
  Axiom eq_refl : ∀x, eq x x.

  Axiom eq_sym : ∀x y, eq x y → eq y x.
  Axiom eq_trans : ∀x y z, eq x y → eq y z → eq x z.
  Axiom lt_trans : ∀x y z, lt x y → lt y z → lt x z.
  Axiom lt_not_eq : ∀x y, lt x y → ¬ eq x y.
End OrderedType.
```

```
Module Type S.
 Declare Module E : OrderedType.
 Definition elt := E.t.
 Parameter t : Set.
 ⋮
 (* add x s returns a set containing all elements of s,
 plus x. If x was already in s, s is returned unchanged.*)
 Parameter add : elt → t → t.
 Parameter add_spec : ∀x y:elt, ∀s:t,
   In y (add x s) ↔ E.eq y x ∨ In y s .
 ⋮
```

- Trois implantations certifiées :
  listes triées, arbres rouges-noirs, arbres AVL
- 7000 lignes de développement Coq
- Une erreur d'équilibrage trouvée dans le code d'origine !
- 700 lignes du code extrait, très similaire à l'original, presque aussi efficace (sauf l'arithmétique sur les hauteurs)
- Validation grandeur nature du coupe modules+extraction

Compcert : un compilateur C certifié

Travail impressionnant de Xavier Leroy et alii.

Autour de 10000 lignes de code extrait.

FingerTree

Bibliothèque d'arbres par Matthieu Sozeau.

Usage intense de types dépendents.

«Tout polynôme sur $\mathbb{C}$ non constant a au moins une racine»

Un développement impressionnant (40 000 lignes) à Nimègue

Preuve constructive $\Rightarrow$ méthode de recherche d'une racine

Mais l'extraction n'a pas été envisagée initialement

$\Rightarrow$ Beaucoup d'efforts pour obtenir un programme

Taille imposante $\Rightarrow$ besoin d'une partition fine `Prop`/`Set`.

- À l'origine, tout dans `Prop` : 0 Ko extrait
- Puis tout dans `Set` : 15 Mo extrait
- Étude détaillée : 250 Ko extrait

De nombreux `Obj.magic` (presque 400)

Problèmes d'efficacité des représentations (unaires/binaires) et des algorithmes.

Dilemme au niveau de la modularité : interface riche ou non ?

- Un programme extrait de taille «raisonnable», qui compile
- La recherche de racines : inutilisable
- Des résultats intermédiaires encourageants : séries
- Encore beaucoup à explorer (ex : réduire des fractions)

Quelques perspectives

- Extraction modulaire plus maligne
- Extraction pseudo-modulaire en Haskell
- `Eval extraction in ...`
- Programmation défensive
- Code extrait “sur-mesure” :
  - `Extract Inductive` vers des types non-inductifs
  - implicites / variables n.c.
  - traduction monadique vers exn / code impératif
  - fixpoints sans compteurs
  - ...