Master Parisien de Recherche en Informatique
cours 2-7-2 : assistants de preuve

Preuve de programmes fonctionnels (3)

Jean-Christophe Filliâtre

2005–2006

Extraction de programmes et réalisabilité

preuves intuitionnistes $\overset{?}{\leftrightarrow}$ fonctions récursives

Kleene 1952

Plan

1.1 Logique classique / logique intuitionniste

logique intuitionniste = pas de tiers exclu

on ne peut dériver $A \vee \neg A$ ou bien $\neg\neg A \Rightarrow A$ pour une formule A arbitraire

est-ce embêtant ?

Cadre
on se place dans l'arithmétique du premier ordre ou d'ordre supérieur
on note $\vdash_C A$ lorsque A est prouvable de manière classique et $\vdash_I A$ lorsque A est prouvable de manière intuitionniste

Résultats

- si $\Gamma \vdash_I A$ alors $\Gamma \vdash_C A$
- si $\Gamma \vdash_C A$ alors $\underbrace{(C_i \vee \neg C_i)_i}_{\text{schéma d'axiome du tiers exclu}}, \Gamma \vdash_I A$
- en logique classique

$$\vdash_C A \vee B \Leftrightarrow \neg(\neg A \wedge \neg B)$$

$$\vdash_C \exists n : \mathtt{nat}.\, P(n) \Leftrightarrow \neg\forall n : \mathtt{nat}.\, \neg P(n)$$

en logique intuitionniste, un seul sens peut être démontré

$$\vdash_I A \vee B \Rightarrow \neg(\neg A \wedge \neg B)$$

$$\vdash_I \exists n : \mathtt{nat}.\, P(n) \Rightarrow \neg\forall n : \mathtt{nat}.\, \neg P(n)$$

Résultats (suite)

- toute formule A peut être transformée en une formule $A^\star$ classiquement équivalente telle que

$$\vdash_C A \Leftrightarrow A^\star \quad \text{et si} \quad \vdash_C A \text{ alors } \vdash_I A^\star$$

- les formules $\forall n \exists m\, Q(n,m)$ avec Q sans quantificateur (formules dites Π^0_2) sont démontrables de manière intuitionniste si et seulement si elles le sont de manière classique

$$\vdash_C \forall n \exists m\, Q(n,m) \quad \text{si et seulement si} \quad \vdash_I \forall n \exists m\, Q(n,m)$$

Propriétés spécifiques des formules intuitionnistes

- **propriété de la disjonction**

$$\text{si} \quad \vdash_I A \vee B \quad \text{alors} \quad \vdash_I A \quad \text{ou} \quad \vdash_I B$$

- **propriété du témoin**

$$\text{si} \quad \vdash_I \exists x,\, P(x) \quad \text{alors il existe } t \text{ tel que} \quad \vdash_I P(t)$$

- les preuves intuitionnistes vérifient l'**axiome du choix**
  - si $\vdash_I \forall x,\, P(x) \vee \neg P(x)$ alors il existe f fonction **récursive** telle que $\vdash_I f(x) = \mathtt{true} \Leftrightarrow P(x)$ pour tout x
  - si $\vdash_I \forall x,\, \exists y,\, P(x,y)$ alors il existe f fonction **récursive** telle que $\vdash_I P(x, f(x))$ pour tout x

Propriétés spécifiques des preuves classiques

- résultats parfois non conformes à l'intuition de la vérité
  exemple : le théorème des buveurs
- toute preuve classique de $\vdash_C \exists n,\, P(n)$ avec $P(n)$ **atomique** s'évalue en une preuve intuitionniste $\vdash_I \exists n,\, P(n)$ qui dévoile un terme t tel que $\vdash P(t)$
- la réalisation de l'axiome du choix

$$(\forall x,\, \exists y,\, P(x,y)) \Rightarrow \exists f,\, \forall x,\, P(x, f(x))$$

pose problème : f ne peut être réalisée par une fonction calculable
considérer par exemple la preuve classique de

$$\forall x,\, \exists b,\, b = \mathtt{true} \Leftrightarrow P(x)$$

qui donnerait f telle que $\forall x,\, f(x) = \mathtt{true} \Leftrightarrow P(x)$

Preuves intuitionnistes et récursivité

la logique intuitionniste permet de parler de décidabilité de manière implicite sans faire appel à une théorie de la récursivité

pour montrer qu'un prédicat est décidable ou qu'une relation fonctionnelle est récursive il suffit de construire une preuve d'une formule disjonctive ou existentielle

remarque : on ne capture **pas toutes** les fonctions récursives

exemple : on code les termes du calcul dans des entiers (codage de Gödel) et on définit la relation de réduction sur les termes
la preuve de totalité de la fonction de normalisation permet de montrer la cohérence logique et ne peut donc être montrée dans le système lui-même (théorème d'incomplétude de Gödel)

Exemple 1

toute fonction sur les entiers admet un minimum

$$\forall f,\ \exists n,\ \forall m,\ f(m) \geq f(n)$$

vrai en logique classique mais pas de manière intuitionniste, même si on se limite aux fonctions récursives
sinon on pourrait décider pour toute fonction si elle prend la valeur nulle et donc on pourrait décider du **problème de l'arrêt**

Exemple 2

$$\vdash_C \exists x \notin \mathbb{Q}\ \exists y \notin \mathbb{Q}\ x^y \in \mathbb{Q}$$

- si $\sqrt{2}^{\sqrt{2}} \in \mathbb{Q}$
  alors on prend $x = y = \sqrt{2}$ car $\sqrt{2} \notin \mathbb{Q}$
- si $\sqrt{2}^{\sqrt{2}} \notin \mathbb{Q}$
  alors on prend $x = \sqrt{2}^{\sqrt{2}}$ et $y = \sqrt{2}$ car $\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = 2$

mais cette preuve ne permet pas d'exhiber une solution

Exemple 3

$A \vee \neg A$ n'est pas démontrable en général en logique intuitionniste

en effet, supposons-le : soit $T(n, m, p)$ le prédicat de Kleene signifiant « la fonction récursive de code n s'exécute sur l'entrée de code m pour effectuer un calcul de code p » ; considérons alors

$$P(n) \equiv \exists p,\ T(n, n, p)$$

si $P(n)$ est récursif alors $\neg P(n)$ aussi donc il existe une fonction récursive de code q qui converge exactement lorsque $\neg P(n)$ est vérifiée i.e.

$$\exists p,\ T(q, n, p) \Leftrightarrow \neg \exists p,\ T(n, n, p)$$

$n = q$ donne une contradiction
et si $P(x) \vee \neg P(x)$ est montrable alors de même $\forall x,\ P(x) \vee \neg P(x)$ d'où la décidabilité de P, contradiction

1.2 Constructivité du Calcul des Constructions Inductives

pour montrer le caractère constructif de la logique de Coq on s'appuie sur l'isomorphisme de Curry-Howard (preuves = λ-termes fortement normalisables)

Propriété

un terme **normal clos** de CCI dont le type est une instance d'une définition **inductive** est de la forme

$$C\ t_1\ \cdots\ t_n$$

avec c **constructeur** du type inductif et $t_1, \dots, t_n$ des termes clos normaux

Preuve

tout terme t s'écrit $c\ t_1\ \cdots\ t_n$ avec c pas une application, donc c = abstraction, variable, constructeur, sorte, produit, case, point fixe
par récurrence sur la structure du terme et par cas

- c pas abstraction car alors t normal $\Rightarrow n = 0$ et le type de t serait un produit
- c pas variable car t est clos
- c pas sorte ou produit car on aurait alors $n = 0$ et le type de t serait une sorte
- c pas case car l'argument principal serait un terme normal clos et par HR serait de la forme $(c'\ \cdots)$ avec c' constructeur et t ne serait donc pas normal

Preuve (suite)

- c pas point fixe : c aurait un type $\forall x_1 : A_1, \dots \forall x_p : A_p, B$ avec x_p argument de décroissance dont le type est inductif ; on aurait $n \geq p$ car sinon t aurait un type produit et t_p normal clos commencerait par un constructeur par HR $\Rightarrow t$ pas normal
- donc c constructeur de type $\forall x_1 : A_1, \dots \forall x_p : A_p, I$ avec I instance d'un type inductif et clairement $n = p$ car on a $n \leq p$ et si $n < p$ alors t a un type produit

$\square$

Justification de la constructivité

- si $A \vee B$ est prouvable **sans hypothèse** alors il existe une preuve de $A \vee B$ sous la forme d'un terme **clos** de type $A \vee B$ qui est

```
Inductive or (A B : Prop) : Prop :=
| or_introl : A → or A B
| or_intror : B → or A B.
```

en normalisant cette preuve on obtient
  - **soit** `or_introl` a avec a clos de type A donc A **prouvable**
  - **soit** `or_intror` b avec b clos de type B donc B **prouvable**

Justification de la constructivité

- de même $\exists x : A,\ P(x)$ est

```
Inductive ex (A : Set) (P : A → Prop) : Prop :=
  ex_intro : ∀(x : A), P x → ex A P.
```

une preuve normale close de $\exists x : A,\ P(x)$ est `ex_intro` t p avec t terme clos de type A et p preuve de $(P\ t)$

Justification de la constructivité

- si on a une preuve close de $\forall x,\ \exists y,\ P(x, y)$ alors il existe un terme F du type correspondant `forall (x:A), exists (y:B), P(x,y)`

  pour tout a terme clos de type A on peut appliquer F à a ce qui nous donne un terme clos de type $\exists y,\ P(a, y)$ que l'on peut normaliser, ce qui donne `ex_intro` t p avec t clos de type B et p preuve de $P(a, t)$

  il y a donc une fonction récursive f qui transforme a en t tel que $P(a, f(a))$ est montrable pour tout a

  Note
  - ne dit pas que f est représentable
  - ni que $\forall x,\ P(x, f(x))$ est démontrable

Justification de la constructivité

- de même, si on a une preuve de $\forall x,\ P(x) \vee \neg P(x)$ alors il existe un prédicat récursif p tel que $p(a) = \texttt{true}$ si et seulement si $P(a)$ démontrable et $P(a) = \texttt{false}$ si et seulement si $\neg P(a)$ démontrable.

1.3 Limites de l'isomorphisme de Curry-Howard

soit une preuve de

$$\forall a\, b,\ b > 0 \Rightarrow \exists q,\ \exists r,\ a = b \times q + r \wedge r < b$$

il faut fournir les entrées a et b **mais aussi** une justification de $b > 0$ et le programme calculera le quotient et le reste **mais aussi** une preuve de $a = b \times q + r \wedge r < b$

donc on fournit quelque chose d'**inutile pour le calcul** (mais nécessaire à la terminaison et à la correction) et on calcule quelque chose d'**inutile** (preuve de correction)

la méthode est **inefficacce**

Limites de l'isomorphisme de Curry-Howard

l'isomorphisme de Curry-Howard n'est pas satisfaisant lorsqu'il s'agit de mettre en évidence les fonctions récursives sous-jacentes aux preuves : il ne permet pas de traiter les preuves **sous axiomes** (les preuves ne sont plus **closes** $\Rightarrow$ le calcul peut dépendre de l'hypothèse)

preuve de

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

est-il possible de construire un programme f tel que

$$\forall x,\ P(x) \Rightarrow Q(x, f(x))$$

faux en général : la preuve de $P(x)$ peut transporter une information servant au calcul du témoin y

exemple

$$\forall n\, m,\ n \leq m \Rightarrow \exists p,\ n + p = m$$

par récurrence sur la preuve de $n \leq m$
alors le calcul de p dépend de cette preuve

2. Réalisabilité

Principes généraux
Kleene 1952
interprétation sémantique des propositions en logique intuitionniste

chaque proposition P est interprétée comme un ensemble de **réalisations** (des programmes), en général défini par récurrence sur la structure de P, intentionnellement par une propriété

$$x\ \mathsf{r}\ P$$

où x nouvelle variable libre représente une réalisation

si l'interprétation de P est non vide alors P est dite **réalisable**

Réalisabilité

idée de base
- l'**absurde** est interprété par l'ensemble **vide**
- $A \Rightarrow B$ est interprétée comme l'ensemble des réalisations représentant des **fonctions** des réalisations de A dans les réalisations de B
- $A \wedge B$ est interprétée comme l'ensemble des réalisations représentants des **couples** formés d'une réalisation de A et d'une réalisation de B
- $\exists x,\ P(x)$ est interprétée comme l'ensemble des réalisations représentant des **couples** formés d'un objet t et d'une réalisation de $P(t)$

Validité

validité
si A est prouvable alors A est réalisable (son interprétation est non vide) ; de plus il est possible de construire une réalisation particulière par récurrence sur la preuve

validité sous contexte
si $\Gamma \vdash A$ est prouvable et toute hypothèse de Γ est réalisable alors A est réalisable

Applications

- **cohérence de certains axiomes**
  si A est réalisable alors A est cohérente avec la théorie (en effet si $A \vdash \perp$ prouvable alors $A \vdash \perp$ réalisable mais alors $\perp$ réalisable, ce qui est absurde)
- **non prouvabilité**
  $$A \text{ non réalisable} \Rightarrow A \text{ non prouvable}$$
  exemples
  - le principe de Markov n'est pas prouvable dans HA_ω
    $$(\forall x : \texttt{nat},\ P(x) \vee \neg P(x)) \Rightarrow \neg\neg\exists x : \texttt{nat},\ P(x) \Rightarrow \exists x : \texttt{nat},\ P(x)$$
  - de même que le principe d'indépendance des prémisses
    $$(\neg A \Rightarrow \exists x,\ P(x)) \Rightarrow \exists x,\ \neg A \Rightarrow P(x)$$

Applications

- **pouvoir d'expression** (cf. *Proof and Types*)
  $$\begin{array}{ll} & \text{fonctions définissables dans le système T} \\ = & \text{fonctions prouvablement totales dans PA} \end{array}$$
  $$\begin{array}{ll} & \text{fonctions définissables dans le système F} \\ = & \text{fonctions prouvablement totales dans } PA_2\ (HA_2) \end{array}$$
  $$\begin{array}{ll} & \text{fonctions définissables dans CC} \\ = & \text{fonctions prouvablement totales dans } HA_\omega \end{array}$$
- **développement de programmes corrects**

2.2 Différentes notions de réalisabilité

- nature du langage de réalisation : entiers (codes de fonctions), λ-calcul pur, typé, etc.
- différents ingrédients : f r $A \Rightarrow B$ peut exiger que $f(a)$ termine, que B soit prouvable, etc.
- x r A peut être une formule du système logique lui-même

Réalisabilité récursive (Kleene 1952)

dans HA (arithmétique intuitionniste du premier ordre)
réalisations = fonctions récursives partielles

- e r P (P atomique) si $e = 0$ et $\vdash P$
- e r $A \wedge B$ si e code du couple (a, b) avec a r A et b r B
- e r $A \vee B$ si e code du couple (x, y) avec $x = 0$ et y r A ou $x = 1$ et y r B
- e r $A \Rightarrow B$ si e code de Gödel d'une fonction récursive ϕ telle que pour tout a tel que a r A, $\phi(a)$ termine et $\phi(a)$ r B
- e r $\exists x,\ A(x)$ si e code du couple (x, a) et si a r $A(x)$
- e r $\forall x,\ A(x)$ si e code de Gödel d'une fonction récursive ϕ telle que pour tout x $\phi(x)$ termine et $\phi(x)$ r $A(x)$

Réalisabilité modifiée (Kreisel)

idée : terminaison assurée par une condition de bon typage

termes typés dans un λ-calcul simple, avec un produit et des entiers

$$\begin{array}{ll|ll}
t(A) & = \texttt{nat} & x \text{ r } A & = x = 0 \wedge A \\
t(A \wedge B) & = t(A) \times t(B) & x \text{ r } A \wedge B & = \texttt{fst}(x) \text{ r } A \ \wedge\ \texttt{snd}(x) \text{ r } B \\
t(A \Rightarrow B) & = t(A) \rightarrow t(B) & f \text{ r } A \Rightarrow B & = \forall x{:}t(A).\ x \text{ r } A \ \Rightarrow\ f(x) \text{ r } B \\
t(\exists y{:}\sigma.B) & = \sigma \times t(B) & x \text{ r } \exists y{:}\sigma.\,B & = \texttt{snd}(x) \text{ r } B[y \leftarrow \texttt{fst}(x)] \\
t(\forall y{:}\sigma.\,B) & = \sigma \rightarrow t(B) & f \text{ r } \forall x{:}\sigma.\,B & = \forall x{:}\sigma.\ f(x) \text{ r } B
\end{array}$$

Extraction de programmes dans Coq

but

- obtenir des programmes plus efficaces qu'avec l'isomorphisme de Curry-Howard, en ne conservant que la partie de la preuve utile pour le calcul des témoins $\Rightarrow$ distinction `Prop` / `Set`
- produire du code Ocaml ou Haskell efficace et certifié correct (par une notion de réalisabilité)

3.1 Réalisabilité dans le Calcul des Constructions

les notes de cours présentent l'**ancienne** notion d'extraction / réalisabilité dans CCI

- oubli des types dépendants
- élimination des termes dans la sorte `Prop`

mais en pratique plusieurs problèmes :

- termes extraits dont l'évaluation échoue (sur une exception)
- termes extraits dont l'évaluation ne termine pas
- pas d'extraction des termes dans `Type`

une nouvelle notion d'extraction [Letouzey 2002] implantée dans Coq 8.0

3.2 Distinction Prop / Set : difficultés

une suppression brutale de tous les termes dans Prop conduit à des programmes erronés

```
Definition pred (n:nat) : n<>0 → nat :=
  match n return n<>0 → nat with
  | O ⇒ fun h ⇒ False_rec nat (h (refl_equal O))
  | S p ⇒ fun _ ⇒ p
  end.

Extraction pred.

(** val pred : nat → nat **)

let pred = function
  | O → assert false (* absurd case *)
  | S p → p
```


Mais alors quid de...

```
Definition pred0 := pred 0.   (* de type 0<>0 → nat *)

Extraction pred0.
```

le code extrait

```
let pred0 = pred O    (* de type nat *)
```

produirait une erreur à l'exécution

de même on pourrait construire

- d'autres types d'erreurs à l'aide d'eq_rec
- des évaluations qui ne terminent pas avec Acc_rec

Solution

conserver les termes de sorte Prop sous une **forme dégénérée**

```
Definition pred0 := pred 0.

Extraction pred0.

(** val pred0 : __ → nat **)

let pred0 _ = pred O
```


Complications

il existe des entorses à la règle « on ne peut construire quelque chose d'informatif à partir de quelque chose de logique »

- **inductif vide**
$$\frac{p : \mathtt{False} : \mathtt{Prop} \quad T : \mathtt{Set}}{\mathtt{case}(p, T, \emptyset) : T}$$
- **inductif singleton logique** (un seul constructeur ayant uniquement des arguments logiques)
$$\frac{p : x = y : \mathtt{Prop} \quad q : P\ x : \mathtt{Set}}{\mathtt{case}(p, P, q) : P\ y : Set}$$
- la **garde** d'un point fixe informatif peut être un **inductif logique**

  rappelez-vous Acc_rec

3.3 Extraction dans le CCI

$$\mathrm{CCI}_\square \qquad \begin{array}{rcl} t & ::= & \square \mid s \mid x \mid c \mid C \mid I \\ & \mid & \forall x : t,\ t \mid \lambda x : t,\ t \mid \mathtt{let}\ x := t\ \mathtt{in}\ t \mid t\ t \\ & \mid & \mathtt{case}(t, t, t, \dots, t) \\ & \mid & \mathtt{fix}\ x_i\ \{x_1/k_1 : t := t, \dots, x_n/k_n : t := t\} \end{array}$$

extraction vers un $\mathrm{CCI}_\square$ **non typé** augmenté d'une constante spéciale $\square$

arité : terme dont la forme normale est $\forall x_1 : X_1, \dots \forall x_n : X_n,\ s$ avec s une sorte
schéma de types : terme dont le type est une arité

Fonction d'extraction

fonction d'extraction $\mathcal{E}$

- $\mathcal{E}(t) = \square$ si t est un schéma de types ou de sorte Prop

sinon, par récurrence sur le terme

- $\mathcal{E}(x) = x$ si x variable, constante ou constructeur
- $\mathcal{E}(\lambda x : T,\ t) = \lambda x : \square,\ \mathcal{E}(t)$
- $\mathcal{E}(\mathtt{let}\ x := t\ \mathtt{in}\ u) = \mathtt{let}\ x := \mathcal{E}(t)\ \mathtt{in}\ \mathcal{E}(u)$
- $\mathcal{E}(u\ v) = \mathcal{E}(u)\ \mathcal{E}(v)$
- $\mathcal{E}(\mathtt{case}(e, P, f_1, \dots, f_n)) = \mathtt{case}(\mathcal{E}(e), \square, \mathcal{E}(f_1), \dots, \mathcal{E}(f_n))$
- $\mathcal{E}(\mathtt{fix}\ f_i\ \{f_1/k_1 : A_1 := t_1, \dots, f_n/k_n : A_n := t_n\})$
  $= \mathtt{fix}\ f_i\ \{f_1/k_1 : \square := \mathcal{E}(t_1), \dots, f_n/k_n : \square := \mathcal{E}(t_n)\}$

Réduction des termes extraits

dans $\mathrm{CCI}_\square$ on ajoute les réductions suivantes

- $\square\ u \to \square$
- $\mathtt{case}_n(\square, P, f) \to f\ \underbrace{\square \cdots \square}_{n}$

  lorsqu'il s'agissait d'un inductif singleton avec n arguments à son constructeur
- $\mathtt{fix}\ f_i\ \{F\}\ u_1 \dots u_{k_i} \to t_i[f_j \leftarrow \mathtt{fix}\ f_j\ \{F\}]_{\forall j}\ u_1 \dots u_{k_i}$
  lorsque $u_{k_i} = \square$

alors on a la **bisimulation**

$$\begin{tikzcd} \mathrm{CCI} & \mathrm{t} \arrow[r, "r_w^\star"] \arrow[d, "\mathcal{E}"] & \mathrm{u} \arrow[d, "\mathcal{E}"] \\ \mathrm{CCI}_\square & \mathrm{t'} \arrow[r, "r_w^\star"] & \mathrm{u'} \end{tikzcd}$$

Correction de l'extraction

on montre que l'extraction de $t : T$ est un terme qui « vérifie le type T » par une notion de réalisabilité, i.e.

$$\mathcal{E}(t)\ \mathrm{r}\ T$$

voir [Letouzey 2002] pour les détails

Vers Ocaml et Haskell

les stratégies de réductions d'Ocaml et Haskell sont **faibles** (pas de réduction sous les λ) ; restent à régler

- élimination sur un **inductif vide**
  c'est du code qui ne sera jamais atteint ⇒ remplacé par du code arbitraire (`assert false` en Ocaml, `error` en Haskell)
- les **singletons logiques** ; la règle

$$\mathtt{case}_n(\square, P, f) \rightarrow f\ \underbrace{\square \cdots \square}_{n}$$

  est intégrée à l'extraction

Vers Ocaml et Haskell

- les **arguments de □** (réduction $\square\ u \rightarrow u$)
  - Ocaml : point-fixe absorbant ses arguments `let rec f x = f`
  - Haskell : évaluation paresseuse ⇒ réduction jamais utilisée ⇒ □ peut être implanté par un terme arbitraire
- les **gardes logiques** de points fixes
  - Ocaml : aucun problème, la réduction se moque du fait que l'argument de cadre commence par un constructeur
  - Haskell : plus délicat, car le dépliage d'un point fixe se fait sans réduction préalable des arguments

3.4 Typage des termes extraits

les « machines » d'Ocaml et d'Haskell conviennent (λ-calcul + inductifs) mais on souhaite produire du **code source** (lisibilité, confiance, réutilisation, etc.)

problème : les termes extraits ne sont **pas toujours typables** car les systèmes de types d'Ocaml et d'Haskell sont **plus pauvres** que celui de Coq

remarque : on pourrait extraire vers des langages non typés comme Lisp ou Scheme (mais en pratique ce n'est pas fait)

Analyse des problèmes de typage

le système de types de Coq offre des possibilités sans contrepartie en Ocaml ou Haskell :

- filtrage au niveau des types
- points fixes au niveau des types
- polymorphisme non prénexe ou dans les types des constructeurs

Analyse des problèmes de typage : exemple

```
Definition P (b:bool) : Set := if b then nat else bool.

Definition p (b:bool) : P b :=
  match b return P b with
  | true ⇒ 0
  | false ⇒ true
  end.
```

le terme extrait

```
let p = function
  | True → 0
  | False → True
```

n'est typable ni en Ocaml ni en Haskell

Analyse des problèmes de typage : exemple

fonctions entières à n arguments

```
Fixpoint F (n:nat) : Set := match n with
  | O ⇒ nat
  | S n ⇒ nat → F n
  end.

Fixpoint f (n:nat) : F n := match n return F n with
  | O ⇒ O
  | S n ⇒ fun _ ⇒ f n
  end.
```

donne le terme extrait non typable

```
let rec f = function
  | O → O
  | S n → (fun _ → f n)
```


Analyse des problèmes de typage : exemple

types existentiels

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

la solution

```
type 'a any = Any of 'a
```

n'est pas satisfaisante (une variable de type s'échappe) ; pire encore

```
Inductive anyList : Type :=
  | AnyNil : anyList
  | AnyCons : ∀A:Type, A → anyList → anyList.

Definition l := AnyCons bool true (AnyCons nat 0 AnyNil).
```

est sans solution Ocaml / Haskell

Analyse des problèmes de typage : exemple

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

ici c'est le polymorphisme **prénexe** d'Ocaml / Haskell qui est une limite (il n'est là que pour rendre l'**inférence** de types décidable)

Correction des erreurs de typage

le typeur peut être « contourné » avec `Obj.magic` en Ocaml et `unsafeCoerce` en Haskell

problème : quand doit-on le faire ?

en pratique, **pas souvent** (jamais dans toute la bibliothèque standard, seulement 4 fois sur 73 contributions recensées des utilisateurs de Coq)

Des types pour les programmes extraits

1. définition d'un type attendu $\hat{\mathcal{E}}(T)$ pour l'extraction de $t : T$ tel que

$$\vdash_{CCI} t : T \quad \Rightarrow \quad \vdash_{ML} \mathcal{E}(t) : \hat{\mathcal{E}}(T)$$

2. on force $\mathcal{E}(t)$ à avoir le type $\hat{\mathcal{E}}(T)$ en insérant des `Obj.magic` / `unsafeCoerce`, mais le moins souvent possible

Extraction des types

- un type Ocaml pour les types Coq que l'on ne peut pas représenter

```
type __ = Obj.t
```

- une définition pour □

```
let __ = let rec f _ = Obj.repr f in Obj.repr f
```


Extraction des types

certains types peuvent être extraits à l'identique

- inductifs

```
Inductive list (A:Set) : Set :=
  nil : list A | cons : A → list A → list A
```

```
type 'a list = Nil | Cons of 'a * 'a list
```

- schémas de types

```
Definition sch : Set → Set := fun X:Set ⇒ X→X.
```

```
type 'x sch = 'x → 'x
```


Extraction des types

mais d'autres seront **approchés**

- inductifs

```
Inductive list2 : ∀A:Set,Type :=
  | nil2 : ∀A:Set, list2 A
  | cons2 : ∀A:Set, A → list2 A → list2 (A * A).
```

```
type 'a list2 = Nil2 | Cons2 of __ * __ list2
```

- schémas

```
Definition sch2 : (bool → Set) → Set :=
  fun (X:bool→Set) ⇒ X true → X false.
```

```
type 'x sch2 = 'x → 'x
```


L'insertion de `Obj.magic`

idée : utiliser un algorithme d'inférence / vérification de types et insérer des `Obj.magic` aux endroits où il aurait échouer

algorithme **W** de Damas-Milner : analyse **ascendante**

W : `env * expr → type * subst`

algorithme **M** de Lee-Yi : analyse **descendante**

M : `env * expr * type → subst`

L'algorithme M

$M(\Gamma, x, \tau) =$
  $\mathrm{mgu}(\tau, \mathrm{Inst}(\Gamma(x)))$

$M(\Gamma, c, \tau) =$
  $\mathrm{mgu}(\tau, \mathrm{Inst}(\mathrm{type}(c)))$

$M(\Gamma, \mathrm{fun}\ x \rightarrow a, \tau) =$
  let $\sigma = \mathrm{mgu}(\tau, \alpha \rightarrow \beta)$ in — α, β fraîches
  let $\phi = M(\Gamma + x : \sigma(\alpha), a, \sigma(\beta))$ in
  $\phi \circ \sigma$

$M(\Gamma, a_1\ a_2, \tau) =$
  let $\phi_1 = M(\Gamma, a_1, \alpha \rightarrow \tau)$ in — α fraîche
  let $\phi_2 = M(\phi_1(\Gamma), a_2, \phi_1(\alpha))$ in
  $\phi_2 \circ \phi_1$

$M(\Gamma, \mathrm{let}\ x = a_1\ \mathrm{in}\ a_2, \tau) =$
  let $\phi_1 = M(\Gamma, a_1, \alpha)$ in — α fraîche
  let $\phi_2 = M(\phi_1(\Gamma) + x : Gen(\phi_1(\alpha), \phi_1(\Gamma)), a_2, \phi_1(\tau))$ in
  $\phi_2 \circ \phi_1$

L'algorithme M modifé

$M(\Gamma, x, \tau) =$
  let $\sigma = \mathrm{mgu}(\tau, \mathrm{Inst}(\Gamma(x)))$ in
  if σ = error then (id, Obj.magic x) else (σ, x)

$M(\Gamma, c, \tau) =$
  let $\sigma = \mathrm{mgu}(\tau, \mathrm{Inst}(\mathrm{type}(c)))$ in
  if σ = error then (id, Obj.magic c) else (σ, c)

$M(\Gamma, \mathrm{fun}\ x \rightarrow a, \tau) =$
  let $\sigma = \mathrm{mgu}(\tau, \alpha \rightarrow \beta)$ in — α, β fraîches
  if σ = error then
    let $(\phi, a') = M(\Gamma + x : \alpha, a, \beta)$ in (ϕ, Obj.magic (fun $x \rightarrow a'$))
  else
    let $(\phi, a') = M(\Gamma + x : \sigma(\alpha), a, \sigma(\beta))$ in $(\phi \circ \sigma, \mathrm{fun}\ x \rightarrow a')$

$M(\Gamma, a_1\ a_2, \tau) =$
  ...inchangé...

$M(\Gamma, \mathrm{let}\ x = a_1\ \mathrm{in}\ a_2, \tau) =$
  ...inchangé...

3.5 L'extraction en pratique

- extraction des **modules**
  module (resp. signature) Coq → module (resp. signature) Ocaml
  quelques problèmes de typage rédhibitoires cependant
- extraction des **types co-inductifs**
  - Haskell : trivial grâce à l'évaluation paresseuse
  - Ocaml : utilisation de du module Lazy

```
CoInductive stream (A:Set) : Set :=
  Cons : A → stream A → stream A.
```

```
type 'a stream = 'a __stream Lazy.t
and 'a __stream = Cons of 'a * 'a stream
```

  + `Lazy.force` dans les `match` et `lazy` devant les constructeurs

Efficacité du code extrait

l'utilisation quasi systématique des **principes de récurrence** venant avec les types inductifs conduit à un code inefficace dans un langage strict comme Ocaml

comparer les programmes Ocaml

```
let rec exists p = function
  | [] → false
  | x :: l → p x || exists p l
```

et

```
let exists p l = List.fold_left (fun b x → p x || b) false
```

le premier est efficace, le second non (parcourt toujours toute la liste)

Efficacité du code extrait

pour y remédier, l'extraction de Coq **déplie** systématiquement

- tous les principes de récurrence
- les fonctions
  - dont le corps n'est pas trop gros
  - dont certains arguments sont potentiellement non utiles car utilisés sous une seule branche d'un filtrage

on contrôle ce déliage globalement avec la commande

```
Set/Unset Extraction AutoInline.
```

et au cas par cas avec la commande

```
Extraction Inline/NoInline id.
```


4. Autres méthodes d'analyse

la méthode d'extraction de Coq ne permet pas de supprimer du programme certains arguments informatifs inutiles

ainsi, le constructeur `cons` des listes de longueur n a pour type

```
cons : (n:nat) A → list n → list (S n)
```

ce qui donne dans le code extrait

```
cons : nat → A → list → list
```

i.e. l'entier représentant la longueur de la liste est conservé

Autres méthodes d'analyse

deux solutions possibles

- déplacer les marques des sortes (`Prop`/`Set`) vers les **quantificateurs** [Takayama, Hayashi, ...]
  on a alors
  $$\forall x : A, B \qquad \text{et} \qquad \forall_\circ x : A, B$$
  avec
  $$\mathcal{E}(\forall_\circ x : A, B) = \mathcal{E}(B) \qquad \text{et} \qquad f \mathrel{\mathrm{r}} \forall_\circ x : A, B = \forall x : A, f \mathrel{\mathrm{r}} B$$
- analyses de **code mort** [Berardi, Boerio, ...]