

Extraction de programmes à partir de preuves classiques en Coq

Alexandre Miquel

U. Paris 7 (PPS) & ENS Lyon (LIP/Plume)

16 avril 2009 – LIX/TypiCal

La question existentielle (en logique)

- Quelle est la signification du quantificateur existentiel?

(dans une logique/théorie donnée)

- 1 Qu'est ce qu'une preuve de $(\exists x : D) P(x)$?
- 2 Que peut on faire (i.e. calculer) avec ?
- 3 Dans quelle mesure peut-on en extraire un **témoin** ?

- Recouvre aussi le problème de la disjonction :

$$\begin{aligned} A \vee B &\Leftrightarrow (\exists b : \mathbb{B}) ((b = \text{tt} \Rightarrow A) \wedge (b = \text{ff} \Rightarrow B)) \\ &\Leftrightarrow (\exists b : \mathbb{B}) \text{ if } b \text{ then } A \text{ else } B \end{aligned}$$

Plan

- 1 Extraction de programmes en logique constructive
- 2 Typage vs. réalisabilité
- 3 Traductions négatives et extraction classique
- 4 Réalisabilité classique dans PA2 (Krivine)
- 5 Réalisabilité classique : de PA2 à pI_{CC} + EM
- 6 Le schéma d'extraction classique en Coq
- 7 Exemples

Plan

- 1 Extraction de programmes en logique constructive
- 2 Typage vs. réalisabilité
- 3 Traductions négatives et extraction classique
- 4 Réalisabilité classique dans PA2 (Krivine)
- 5 Réalisabilité classique : de PA2 à pI^{CC} + EM
- 6 Le schéma d'extraction classique en Coq
- 7 Exemples

L'extraction de programme en logique constructive

- Basée sur la propriété du témoin en **logique intuitionniste** :

$$\vdash \exists x \overset{\vdots}{A}(x) \quad \rightsquigarrow \quad t \quad + \quad \vdash \overset{\vdots}{A}(t)$$

- À travers Curry-Howard (intuitionniste) :

$$\begin{aligned} (\exists x : D) P(x) &\equiv \Sigma x : D . P(x) && \text{(sigma-type)} \\ p : \Sigma x : D . P(x) &\rightsquigarrow (t, p') && \text{(avec } t:D \text{ et } p':P(t)) \end{aligned}$$

- **Difficulté** : élimination du **code mort** (i.e. la justification p')
- Extraction de programme :

$$p : (\forall x : A) (\exists y : B) P(x, y) \quad \rightsquigarrow \quad f : A \rightarrow B$$

- Les principes :

- ▶ Extraction : $HA \rightsquigarrow$ Système T (Gödel)
- ▶ Élimination des formules de Harrop :

$$H ::= \perp \mid \top \mid t = u \mid H \wedge H \mid A \Rightarrow H \mid \forall x H$$

+ **réalisabilité modifiée**

- ▶ Implémenté dans l'assistant MINLOG à Munich (Schwichtenberg *et al.*)
- ▶ Limite : l'élimination des formules de Harrop ne passe pas au 2nd ordre

- Techniquement :

- ▶ Types : $\tau, \sigma ::= \mathbb{N} \mid \mathbb{B} \mid \tau \times \sigma \mid \tau \rightarrow \sigma \mid \mathbb{L}(\tau)$
- ▶ Formules : $A, B ::= \perp \mid t = u \mid A \wedge B \mid A \Rightarrow B \mid \forall^{\tau} x A \mid \exists^{\tau} x A$
- ▶ Type d'une formule A : $\text{ty}(A) \in \text{Types} \cup \{\epsilon\}$ ($\epsilon =$ formule de Harrop)

+ codages : $\neg A \equiv A \Rightarrow \perp, \quad A \vee B \equiv \exists^{\mathbb{B}} b ((b = \text{tt} \Rightarrow A) \wedge (b = \text{ff} \Rightarrow B))$

Type d'une formule : $ty(A) \in \text{Types} \cup \{\epsilon\}$

$$ty(A \wedge B) \equiv \begin{cases} \epsilon & \text{si } ty(A) \equiv ty(B) \equiv \epsilon \\ ty(B) & \text{si } ty(A) \equiv \epsilon, ty(B) \not\equiv \epsilon \\ ty(A) & \text{si } ty(A) \not\equiv \epsilon, ty(B) \equiv \epsilon \\ ty(A) \times ty(B) & \text{si } ty(A) \not\equiv \epsilon, ty(B) \not\equiv \epsilon \end{cases}$$

$$ty(\perp) \equiv \epsilon$$

$$ty(A \Rightarrow B) \equiv \begin{cases} \epsilon & \text{si } ty(B) \equiv \epsilon \\ ty(B) & \text{si } ty(A) \equiv \epsilon, ty(B) \not\equiv \epsilon \\ ty(A) \rightarrow ty(B) & \text{si } ty(A) \not\equiv \epsilon, ty(B) \not\equiv \epsilon \end{cases}$$

$$ty(t = u) \equiv \epsilon$$

$$ty(\forall x^\sigma A) \equiv \begin{cases} \epsilon & \text{si } ty(A) \equiv \epsilon \\ \sigma \rightarrow ty(A) & \text{si } ty(A) \not\equiv \epsilon \end{cases}$$

$$ty(\exists x^\sigma A) \equiv \begin{cases} \sigma & \text{si } ty(A) \equiv \epsilon \\ \sigma \times ty(A) & \text{si } ty(A) \not\equiv \epsilon \end{cases}$$

- Exemple : $ty(A \vee B) \equiv \mathbb{B} \times A \times B$ (si $ty(A) \not\equiv \epsilon, ty(B) \not\equiv \epsilon$)
- Réalisabilité modifiée : $p \mathbf{mr} A$ (avec $p : ty(A)$)

- **Le schéma d'extraction de Christine Paulin** (1989–2002)

- ▶ Fonction d'**effacement des dépendances** $CC \rightarrow F\omega$
- ▶ Schéma amélioré avec l'introduction de la distinction **Prop/Set**
- ▶ Le schéma devient peu à peu caduc avec l'extension de Coq (élimination forte, inductifs généralisés, univers flottants)

- **Le schéma d'extraction de Pierre Letouzey** (2002–)

- ▶ Extraction vers un langage **non typé**
CIC \rightarrow Haskell, SML, Caml... + **Obj.magic**
- ▶ Termes de preuve ($: T : \text{Prop}$) effondrés sur un unique objet \square
(calculatoirement équivalent au démon de la ludique)
- ▶ Types (familles de types, prédicats, ...) effondrés sur \square
- ▶ Autres constructions traduites par elles-mêmes (ou presque...)

Les propriétés de CIC qui fondent l'extraction constructive

1 Les types comme simples **décorations** du calcul :

- ▶ Dans CIC (comme dans les PTS), les types ne servent qu'à calculer d'autres types (pas d'analyse par cas sur les types)

2 **Proof-irrelevance** au niveau calculatoire :

- ▶ Restriction du schéma d'élimination dans Prop :

```
Inductive boolP : Prop := trueP : boolP | falseP : boolP.
```

```
Definition bool_of_boolP : boolP -> bool :=  
  fun bp => if bp then true else false.
```

Error:

*Incorrect elimination of "bp" in the inductive type "boolP":
the return type has sort "Set" while it should be "Prop".
Elimination of an inductive object of sort Prop is not allowed
on a predicate in sort Set because **proofs can be eliminated
only to build proofs.***

- ▶ Correspond à l'interprétation ensembliste (triviale) de Prop...

... indispensable pour conserver la compatibilité avec les maths classiques
(cf paradoxe de Chicli, Pottier & Simpson, 2002)

Contraintes liées à la stratification

- Nécessité de bien stratifier ses preuves en vue de l'extraction

Les trois formes de quantification existentielle en Coq :

- ▶ `exists x : T, P x` (dans `Prop`, avec `T : Type` et `P x : Prop`)
 \rightsquigarrow rien n'est extrait
- ▶ `{x : T | P x}` (dans `Type`, avec `T : Type` et `P x : Prop`)
 \rightsquigarrow seul le témoin est extrait
- ▶ `{x : T & P x}` (dans `Type`, avec `T : Type` et `P x : Type`)
 \rightsquigarrow le témoin et la justification sont extraits

- Exemple : la division euclidienne par 2

- ▶ `forall x : nat, exists y : nat, x = 2y \vee x = 2y + 1`
- ▶ `forall x : nat, { y : nat | x = 2y \vee x = 2y + 1 }`
- ▶ `forall x : nat, { y : nat & {x = 2y} + {x = 2y + 1} }`

Plan

- 1 Extraction de programmes en logique constructive
- 2 Typage vs. réalisabilité
- 3 Traductions négatives et extraction classique
- 4 Réalisabilité classique dans PA2 (Krivine)
- 5 Réalisabilité classique : de PA2 à pI_{CC} + EM
- 6 Le schéma d'extraction classique en Coq
- 7 Exemples

Typage et réalisabilité

Pourquoi le terme $\lambda x . x$ a pour type $\text{nat} \rightarrow \text{nat}$?

Typage

$$\frac{x : \text{nat} \vdash x : \text{nat}}{\vdash \lambda x . x : \text{nat} \rightarrow \text{nat}}$$

- Analyse syntaxique des termes
- Au minimum semi-décidable
- Justification simple : dérivation

Réalisabilité

pour tout $n \in \text{nat}$

$$(\lambda x . x) n \succ n \in \text{nat}$$

- Analyse calculatoire des termes
- Fortement indécidable
- Justification externe (preuve)

Adéquation :

Correction pour le typage \Rightarrow Correction pour la réalisabilité

Mais la réalisabilité a été introduite en **logique** (Kleene'45)

comme une relation $t \Vdash A$ entre des **programmes** et des **formules**

Syntaxe

Types	$A, B ::= \text{nat} \mid A \rightarrow B$
Termes	$M, N ::= x \mid \lambda x. M \mid MN$ $\mid 0 \mid s \mid \text{rec}$

Règles de réduction

$$\begin{aligned}(\lambda x. M)N &\succ M\{x := N\} \\ \text{rec } M_0 M_1 0 &\succ M_0 \\ \text{rec } M_0 M_1 (s N) &\succ M_1 N (\text{rec } M_0 M_1 N)\end{aligned}$$

Le système T : typage (1/2)

$$\frac{}{\Gamma \vdash x : A} \quad (x:A) \in \Gamma \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \rightarrow B}$$
$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$
$$\frac{}{\Gamma \vdash 0 : \text{nat}} \qquad \frac{}{\Gamma \vdash s : \text{nat} \rightarrow \text{nat}}$$
$$\frac{}{\Gamma \vdash \text{rec} : A \rightarrow (\text{nat} \rightarrow A \rightarrow A) \rightarrow \text{nat} \rightarrow A}$$

- Définition purement **syntaxique** (termes)
 - ▶ Induction implicite sur la structure du terme
 - ▶ Porte sur des termes *ouverts* \rightsquigarrow **contextes de typage**
 - ▶ Justification simple : **dérivation de typage**
 - ▶ Inférence/vérification de type décidables (dirigées par la syntaxe)
- Mais la réduction n'est jamais mentionnée...
... **quelle garantie vis-à-vis du calcul?**

Le système T : typage (2/2)

3 lemmes garantissent la correction vis-à-vis du calcul :

1. Subject reduction

Si $\Gamma \vdash M : A$ et $M \succ M'$, alors $\Gamma \vdash M' : A$

2. Valeurs de type nat

Si $\Gamma \vdash M : \text{nat}$ et si M est une valeur, alors $M = s^n 0$

(valeur = forme normale close)

3. Normalisation (forte)

Si $\Gamma \vdash M : A$, alors M est (fortement) normalisable

- $1 + 2 + 3 \Rightarrow$ Tout programme clos : nat se réduit sur un naturel

Le système T : réalisabilité

Relation binaire $M \Vdash A$ (M terme clos)

Définition de la réalisabilité

- 1 $M \Vdash \text{nat}$ si $M \succ^* s^n 0$
- 2 $M \Vdash A \rightarrow B$ si $N \Vdash A$ entraîne $MN \Vdash B$ (pour tout N)

- Définition purement **calculatoire** (syntaxe = boîte noire)
 - ▶ Induction sur la structure du type
 - ▶ Termes clos \rightsquigarrow pas de contexte
 - ▶ Pas de justification élémentaire (telle qu'une dérivation)
 - \rightsquigarrow recours à une **justification externe** : preuve
 - ▶ Relation $M \Vdash A$ **indécidable**, non récursivement énumérable
- Pas de correction à démontrer : tout est dans la définition !
- Point de vue beaucoup plus riche que le typage (cf slide suivant)

Lemme d'adéquation

Si $x_1 : A_1, \dots, x_k : A_k \vdash M : B$, alors pour tous N_1, \dots, N_k
 $N_1 \Vdash A_1, \dots, N_k \Vdash A_k$ entraîne $M\{x_1 := N_1; \dots; x_k := N_k\} \Vdash B$

Cas particulier (contexte vide) : $\vdash M : A$ implique $M \Vdash A$

- Typage + adéquation \Rightarrow

Tout programme clos de type nat se réduit vers un naturel

... sans passer par $1 + 2 + 3$

- En fait, 3 (SN) se prouve... par réalisabilité (trafiquée)
- **En pratique** : on ne construit que de petits réalisateurs à la main...
... pour le reste, on passe par typage + lemme d'adéquation

Réalisabilité et typage (métaphore)



Plan

- 1 Extraction de programmes en logique constructive
- 2 Typage vs. réalisabilité
- 3 Traductions négatives et extraction classique**
- 4 Réalisabilité classique dans PA2 (Krivine)
- 5 Réalisabilité classique : de PA2 à pI^{CC} + EM
- 6 Le schéma d'extraction classique en Coq
- 7 Exemples

- **Disjonction sans alternative :**

- ▶ $A \vee \neg A$ ($A \equiv$ conj. de Riemann, hyp. du continu, etc.)
- ▶ $e + \pi$ transcendant $\vee e \times \pi$ transcendant

- **Existentiel sans témoin :**

- ▶ $(\exists x : \mathbb{N}) ((A \wedge x = 1) \vee (\neg A \wedge x = 0))$ (mêmes exemples pour A)

- **Fonctions non calculables (oracles) :**

- ▶ Fonction d'arrêt des machines de Turing :

$$(\forall x : \mathbb{N}) (\exists y : \mathbb{N}) ((\text{Halt}(x) \wedge y = 1) \vee (\neg \text{Halt}(x) \wedge y = 0))$$

- ▶ Principe du minimum ($X \neq \emptyset$) :

$$(\forall f : X \rightarrow \mathbb{N}) (\exists x : X) (\forall y : X) f(x) \leq f(y)$$

Traductions négatives

- **Problème** : Comment **rétracter** une théorie classique \mathcal{T}_c (PA, Z, ZF, ...) sur son (ou un) fragment intuitionniste $\mathcal{T}_i \subset \mathcal{T}_c$? (HA, IZ, IZF, ...)

Traduction négative = traduction $A \mapsto A^*$ telle que

$$\mathcal{T}_c \vdash_{(\text{LK})} A \quad \text{entraîne} \quad \mathcal{T}_i \vdash_{(\text{LJ})} A^*$$

- Si de plus $\mathcal{T}_i \vdash_{(\text{LJ})} \perp^* \Rightarrow \perp$ alors \mathcal{T}_c et \mathcal{T}_i sont **équiconsistantes**
- **Exemple** : les **$\neg\neg$ -traductions**
 - ▶ Basées sur l'insertion de $\neg\neg$ à des points critiques de A (\forall, \exists, \dots)
 - ▶ Utiles pour montrer l'équiconsistance de \mathcal{T}_c et \mathcal{T}_i
 - ▶ Les $\neg\neg$ -traductions n'ont aucun intérêt pour l'**extraction de programme** !
... car les formules $\neg\neg$ -traduites sont des formules de Harrop

- R -traduction = généralisation de la $\neg\neg$ -traduction où l'on remplace la négation ordinaire $\neg A \equiv A \Rightarrow \perp$ par une **négation relative**

$$\overset{R}{\neg} A \equiv A \Rightarrow R$$

où R est un paramètre de la traduction (donc : $\neg\neg$ -traduction = cas où $R \equiv \perp$)

Exemple de R -traduction au 1er ordre

$$\begin{array}{ll} \perp^* & \equiv R \\ p(\vec{t})^* & \equiv \overset{R}{\neg} \overset{R}{\neg} p(\vec{t}) \\ (A \vee B)^* & \equiv \overset{R}{\neg} (\overset{R}{\neg} A^* \wedge \overset{R}{\neg} B^*) \\ (\exists x A)^* & \equiv \overset{R}{\neg} \forall x \overset{R}{\neg} A^* \end{array} \quad \begin{array}{ll} (A \Rightarrow B)^* & \equiv A^* \Rightarrow B^* \\ (A \wedge B)^* & \equiv A^* \wedge B^* \\ (\forall x A)^* & \equiv \forall x A^* \end{array}$$

- En pratique, l'existence d'une R -traduction de \mathcal{T}_c dans \mathcal{T}_i implique :
 - ▶ L'équiconsistance de \mathcal{T}_c et \mathcal{T}_i (cas où $R \equiv \perp$)
 - ▶ La **conservativité** de \mathcal{T}_c sur \mathcal{T}_i pour les formules Σ_1^0, Π_2^0 (astuce de Friedman)
 - ▶ L'équ**il**-consistance de \mathcal{T}_c et \mathcal{T}_i

R-traduction et extraction de programmes

- R-traduction + Curry-Howard (ou réalis.) intuitionniste
 \rightsquigarrow **extraction de programme** pour les formules Σ_1^0 et Π_2^0

Transformation de preuve d'une formule Σ_1^0 (de \mathcal{T}_c dans \mathcal{T}_i)

- 1 π : $\exists x f(x) = 0$ (dans \mathcal{T}_c)
- 2 π^* : $\overset{R}{\neg} \forall x \overset{R}{\neg} f(x) = 0$ (dans \mathcal{T}_i , $\overset{R}{\neg} A \equiv A \Rightarrow R$)
 : $\forall x (f(x) = 0 \Rightarrow R) \Rightarrow R$
- 3 Astuce de Friedman : $R \equiv \exists y f(y) = 0$
- 4 π^* : $\underbrace{\forall x (f(x) = 0 \Rightarrow \exists y f(y) = 0)}_{\exists\text{-intro}} \Rightarrow \exists y f(y) = 0$
- 5 π^* (\exists -intro) : $\exists y f(y) = 0$ (dans \mathcal{T}_i)

- Berardi, Berger, Coquand, Kohlenbach, Schwichtenberg

Exemples et contre-exemples

- Exemples de $(\mathcal{T}_c, \mathcal{T}_i)$ pour lesquels une R -traduction existe :

- ▶ Les arithmétiques : $(\text{PA}_n, \text{HA}_n)$ $(1 \leq n \leq \omega)$
- ▶ Théories des ensembles : $(\text{Z}, \text{IZ}), (\text{ZF}, \text{IZF}_C)$ [Friedman]
- ▶ PTS logiques non dépendants [Coquand-Herbelin]

- Contre-exemple : $(\text{MLTT} + \text{EM}, \text{MLTT})$
où MLTT = Théorie des types de Martin-Löf (et ses variantes)

- Statut inconnu : $(\text{pCIC} + \text{EM}, \text{pCIC})$ (conjecture : oui)

- Pour chacun des exemples où une R -traduction existe :

- ▶ \mathcal{T}_c et \mathcal{T}_i sont **équiconsistantes**
- ▶ \mathcal{T}_c est conservative sur \mathcal{T}_i pour les formules Σ_1^0, Π_2^0
- ▶ \mathcal{T}_c et \mathcal{T}_i sont **équi-1-consistantes**

• La découverte fondamentale

[Felleisen & Griffin 90]

- ▶ L'opérateur de contrôle `call/cc` [Felleisen] a pour type :

$$((A \rightarrow B) \rightarrow A) \rightarrow A$$

- ▶ Côté logique [Griffin 90], il s'agit de la **loi de Peirce...**
... qui entraîne (constructivement) le tiers-exclu
- ▶ Logique classique = programmation par **continuations**
= méthode **essai/erreur**
- ▶ Relié aux **traductions négatives** par les **transformations CPS**

• Exemple : comment prouver

$$A \vee (A \Rightarrow \perp) \quad ?$$

Réponse : `call-cc λk . Right (λx . k (Left(x)))`

Plan

- 1 Extraction de programmes en logique constructive
- 2 Typage vs. réalisabilité
- 3 Traductions négatives et extraction classique
- 4 Réalisabilité classique dans PA2 (Krivine)
- 5 Réalisabilité classique : de PA2 à pI^{CC} + EM
- 6 Le schéma d'extraction classique en Coq
- 7 Exemples


La réalisabilité classique (Krivine)

- **Réalisabilité classique** = réalisabilité à la Kleene à travers une R -traduction¹, le tout reformulé en style direct :

$$\text{Krivine} = (\text{CPS})^{-1} \circ \text{Kleene} \circ (\text{R-traduction})$$

- Un langage de réalisateurs : $\lambda_c = \lambda + \text{call-cc} + \dots$ (extensible)
+ évaluation en **appel par nom** (exit les problèmes de confluence!)

- La réalisabilité classique est définie :

- ▶ dans PA2 + DC (DC = choix dépendant)
- ▶ dans ZF (+ DC)
- ▶ dans CC^ω (calcul des constructions avec univers)
- ▶ dans pCIC  (pCIC = CIC – Set imprédicatif)

¹R-traduction de Lafont-Reus-Streicher-Oliva

Le λ_c -calcul (Krivine)

Syntaxe

Termes $t, u ::= x \mid \lambda x . t \mid \underbrace{tu \mid \overbrace{\alpha \mid \dots}^{\text{instructions}} \mid k_\pi}_{\text{quasi-preuves}}$

Piles $\pi ::= \alpha \mid u \cdot \pi \quad (u, \pi \text{ clos})$

Processus $p, q ::= t \star \pi \quad (t, \pi \text{ clos})$

$k_\pi =$ (constante de) continuation associée à π , $\alpha =$ (constante de) fond de pile

Règles d'évaluation : $p \succ p'$

Push $tu \star \pi \succ t \star u \cdot \pi$

Grab $\lambda x . t \star u \cdot \pi \succ t\{x := u\} \star \pi$

Call/cc $\alpha \star t \cdot \pi \succ t \star k_\pi \cdot \pi$

Resume $k_\pi \star t \cdot \pi' \succ t \star \pi$

...

...

Principes de la réalisabilité classique

- **Intuitions :**

- ▶ terme = “**preuve**” / pile = “**contre-preuve**”
- ▶ processus = “**contradiction**” (ne jamais faire confiance à un réalisateur !)

- À chaque formule A on associe :

- ▶ Un ensemble de piles $\|A\|$ (**valeur de fausseté**)
- ▶ Un ensemble de termes $|A|$ (**valeur de vérité**)

Exemple : $\|A \Rightarrow B\| = |A| \cdot \|B\| = \{t \cdot \pi : t \in |A|, \pi \in \|B\|\}$

- Définition de la valeur de vérité $|A|$ par **orthogonalité** :

$$|A| = A^\perp = \{t \in \Lambda : \forall \pi \in \|A\| \ t \star \pi \in \perp\}$$

où \perp est un ensemble de processus, paramètre de la construction ($\perp \approx R$)

Théorème d'adéquation (dans PA2, ZF, pCIC, ...)

À toute preuve classique de A on sait associer un réalisateur $t \in |A|$

Structure du quantificateur existentiel

En réalisabilité intuitionniste et classique, un réalisateur de $\exists^{\mathbb{N}}x A(x)$ contient un couple **témoin/justification** (n, p) , avec $n \in \mathbb{N}$ et $p \Vdash A(n)$

- En réalisabilité intuitionniste

- ▶ On a la garantie que n vérifie $A(n)$
- ▶ La justification $p \Vdash A(n)$ est donc superflue (code mort)

- En réalisabilité classique

- ▶ Aucune garantie que n vérifie $A(n)$! (possibilité de **faux-témoin**)
- ▶ La justification $p : A(n)$ n'offre aucune garantie... (**parapreuve**)
... mais contient le matériel nécessaire pour backtracker

- Solution pour obtenir des témoins fiables :

- ▶ Interroger le témoin (procédure de test)
- ▶ Répudier un faux-témoignage $p : A(n)$...
... en lui opposant une **réfutation** $p' \Vdash \neg A(n)$

Recherche de témoin par essai/erreur

- Les ingrédients :

- ▶ Une preuve (classique) $p : (\exists x : \mathbb{N}) A(x)$
- ▶ Une fonction de décision $d : \mathbb{N} \rightarrow \mathbb{B}$ du prédicat A
- ▶ Une fonction de réfutation $r : r\ n \Vdash \neg A(n)$ dès que $\neg A(n)$

- L'algorithme :

- 1 Extraire $n : \mathbb{N}$ et $p_0 : A(n)$ de la preuve p
- 2 Avec la fonction d , tester si $A(n)$. Si oui, sortir.
- 3 Sinon, construire une réfutation $r_0 := r\ n : \neg A(n)$ et appliquer $r_0 : \neg A(n)$ à $p_0 : A(n)$ pour déduire \perp (**backtrack**)

- L'implémentation : $p\ (M\ (\lambda xy . d\ x\ (\text{stop}\ x)\ (r\ x\ y)))$

Proposition

- 1 Cette procédure converge en temps fini [Krivine, Raffalli, Guillermo]
- 2 À une CPS près : même algorithme que Friedman [TLCA'09]

Plan

- 1 Extraction de programmes en logique constructive
- 2 Typage vs. réalisabilité
- 3 Traductions négatives et extraction classique
- 4 Réalisabilité classique dans PA2 (Krivine)
- 5 Réalisabilité classique : de PA2 à pICC + EM**
- 6 Le schéma d'extraction classique en Coq
- 7 Exemples

Réalisabilité classique : de PA2 à pCIC

- En réalisabilité classique (dans PA2, ZF), une **formule** A est interprétée par un ensemble de piles $\|A\| \subseteq \Pi$ (**valeur de fausseté**)
 - Dans pCIC, un type T peut représenter
 - ▶ Une proposition (si $T : \text{Prop}$)
 - ▶ Un type de données informatif (si $T : \text{Set/Type}$)
 - ▶ Un type de types (si T est une sorte)
- \rightsquigarrow Structure plus complexe que dans PA2!

Définition (Π -ensemble)

[CSL'07]

Un **Π -ensemble** est un couple $X = (|X|, \perp_X)$ formé par

- Un ensemble $|X|$ (**support**)
- Une relation $(\perp_X) \subset |X| \times \Pi$ (**relation de réfutation locale**)

- Réalisabilité : $t \Vdash v \in X \equiv \forall \pi (v \perp_X \pi \Rightarrow t \star \pi \in \perp)$

Éléments d'interprétation

- Fonction d'interprétation : $M \mapsto \llbracket M \rrbracket$ (pCIC $\rightarrow \mathcal{M}$)

Correction

Si $M : T$, alors : $\llbracket T \rrbracket$ est un Π -ensemble et $\llbracket M \rrbracket \in \llbracket T \rrbracket$

- Fonction d'**extraction** : $M \mapsto M^*$ (pCIC $\rightarrow \lambda_c$)

Adéquation

Si $M : T$, alors $M^* \Vdash \llbracket M \rrbracket \in \llbracket T \rrbracket$

- Terminaison assurée par un choix judicieux de \perp (**générique**)

Lien entre les deux modèles de réalisabilité

Sur le fragment commun (PA2), le modèle de réalisabilité de pCIC coïncide avec le modèle de Krivine. En particulier :

$$t \Vdash_{\text{pCIC}} \Pi x : \text{nat}. A(x) \iff t \Vdash_{\text{PA2}} \forall x (\text{nat}(x) \rightarrow A(x))$$

Disjonction et somme disjointe

- La **disjonction** $A \vee B$ (Prop) et la somme directe $A + B$ (Type) ne sont pas du tout interprétées de la même manière dans le modèle :
 - ▶ Disjonction : $\llbracket A \vee B \rrbracket = \{\bullet\}$
 - ★ Pas d'information de latéralité (i.e. preuve gauche / preuve droite)
 - \rightsquigarrow disjonction **classique**
 - \rightsquigarrow essentiel pour réaliser $A \vee \neg A$
 - ▶ Somme directe : $\llbracket A + B \rrbracket = \llbracket A \rrbracket + \llbracket B \rrbracket$
 - ★ Information de latéralité dans chaque dénotation de $M : A + B$
 - \rightsquigarrow disjonction **intuitionniste**
 - \rightsquigarrow interdit de réaliser $A + (A \rightarrow \perp)$ (même avec call-cc)
 - ▶ $\{A\} + \{B\}$ ("sumbool") se comporte comme $A + B$
- **Moralité** : Le modèle de réalisabilité classique de pCIC devient intuitionniste dans les univers supérieurs (Set/Type)

Plan

- 1 Extraction de programmes en logique constructive
- 2 Typage vs. réalisabilité
- 3 Traductions négatives et extraction classique
- 4 Réalisabilité classique dans PA2 (Krivine)
- 5 Réalisabilité classique : de PA2 à pI_{CC} + EM
- 6 Le schéma d'extraction classique en Coq
- 7 Exemples

Extraction sur le PTS sous-jacent

- Une projection $M \mapsto M^*$ des termes de pLCC sur le λ -calcul, avec effondrement des types sur une instruction inerte :

$$\begin{aligned}x^* &\equiv x \\(\text{fun } x : T \Rightarrow M)^* &\equiv \lambda x . M^* \\(MN)^* &\equiv M^* N^* \\(\text{forall } x : T, U)^* &\equiv .\text{type} \\ \text{Prop}^* &\equiv .\text{type} \\ \text{Set}^* &\equiv .\text{type} \\ \text{Type}^* &\equiv .\text{type}\end{aligned}$$

En fait on pose $M^* \equiv .\text{type}$ dès que M est un type (ou un prédicat).

- **Remarque** : pas de distinction suivant la sorte du terme source $M : T : \text{Prop/Set/Type}$ (contrairement à l'extraction constructive)

Extraction des constantes

- L'environnement Coq introduit 4 formes de « constantes » :

- ▶ Types/prédicats (co)inductifs nat, list, eq
- ▶ Constructeurs, liés à un (co)inductif 0, cons, refl_equal
- ▶ Constantes définies (Definition) not
- ▶ Axiomes classic (tiers-exclu)

- Pour chaque **constante** de plCC, on introduit une **instruction** de même nom (long) en λ_c -calcul :

$$c^* \equiv \text{nom_long_de_}c$$

Exemple : S (constructeur de nat) devient : `Coq.Init.Datatypes.S`

- **Une exception :** Pas d'instruction pour les constantes de types/prédicats (co)inductifs. Suivant la politique de décimation des types, on pose :

$$l^* \equiv \text{.type}$$

- Le filtrage de Coq est traduit par :

$$\left(\begin{array}{l} \text{match } M \text{ with} \\ | c_1 x_1 \cdots x_{n_1} \Rightarrow N_1 \\ \quad \vdots \\ | c_k x_1 \cdots x_{n_k} \Rightarrow N_k \\ \text{end} \end{array} \right)^* \equiv \begin{array}{l} l\% \text{case } M^* (\lambda x_1 \cdots x_{n_1} . N_1^*) \\ \quad \vdots \\ (\lambda x_1 \cdots x_{n_k} . N_k^*), \end{array}$$

où

- ▶ l est le type/prédicat (co)inductif auquel appartient M
- ▶ c_1, \dots, c_k sont les constructeurs, d'arités réelles n_1, \dots, n_k
- ▶ $l\% \text{case}$ est un **combinateur de filtrage** (= instruction) associé à l

Extraction des Fixpoint/Cofixpoint

- Fixpoint et Cofixpoint sont traduits de la même manière

$$\left(\begin{array}{l} (\text{co})\text{fix } f_1 : T_1 := M_1 \\ \vdots \\ \text{with } f_n : T_n := M_n \text{ for } f_i \end{array} \right)^* \equiv \begin{array}{l} .\text{fix_n_i } (\lambda f_1 \cdots f_n . M_1^*) \\ \vdots \\ (\lambda f_1 \cdots f_n . M_n^*) \end{array}$$

où les instructions $.\text{fix_n_i}$ ($1 \leq i \leq n$) sont définies par

$$\begin{array}{l} .\text{fix_n_i } F_1 \cdots F_n \quad := \\ F_i \left(.\text{fix_n_1 } F_1 \cdots F_n \right) \cdots \left(.\text{fix_n_n } F_1 \cdots F_n \right) \end{array}$$

- Dans l'extracteur, les instructions $.\text{fix_n_i}$ sont engendrées à la demande

Extraction des structures de données (co)inductives

- La représentation des données d'un type/prédicat (co)inductif I dans λ_c (calcul cible) est déterminée par l'implémentation
 - ▶ des instructions c_1, \dots, c_k associées au constructeurs de I
 - ▶ du combinateur de filtrage $I\%case$ associé à I

Changer l'implémentation de ces instructions, c'est changer la représentation de toutes les données de type (famille) I dans tout le code extrait.

- Par défaut, on utilise les codages habituels en λ -calcul :

$$c_i := \lambda \underbrace{y_1 \cdots y_p}_{\text{paramètres (ignorés)}} \underbrace{x_1 \cdots x_{n_i}}_{\substack{\text{arguments} \\ \text{réels}}} \underbrace{e_1 \cdots e_k}_{\text{éliminateurs}} \cdot (e_i) x_1 \cdots x_{n_i}$$

$$I\%case := \lambda z. z$$

Exemple

- Les entiers naturels (codage par défaut) :

```
Coq.Init.Datatypes.0      :=  $\lambda e_1 e_2 . e_1$   
Coq.Init.Datatypes.S     :=  $\lambda x e_1 e_2 . e_2 x$   
Coq.Init.Datatypes.nat%case :=  $\lambda z . z$ 
```

- ▶ Représentation (encore) plus verbeuse que les entiers de Church
- ▶ Intuition : même codage que le type option

`nat = option nat, 0 = None, S = Some`

- ▶ Prédécesseur en temps constant (\neq entiers de Church)
- ▶ Récursion à coup de `.fix_n_i` (pas possible autrement !)

Le mécanisme d'*override*

- À chaque extraction d'une nouvelle constante, l'extracteur consulte un fichier de définitions "override.lco" pour voir si l'instruction correspondante y est définie.
 - ▶ Si oui, la définition correspondante est placée dans le code extrait
 - ▶ Sinon, l'extracteur produit une définition de l'instruction correspondant au mécanisme par défaut (décrit ci-avant)
- En pratique le mécanisme d'*override* sert à :
 - ▶ Changer la représentation de certains inductifs (`nat`)
 - ▶ Réaliser certains axiomes (qui par défaut sont réalisés par des constantes inertes) \rightsquigarrow **logique classique**
 - ▶ Attacher à certains objets de la librairie des **réalisateurs optimisés** (par rapport à ceux produits suivant le schéma par défaut)

Utilisation des entiers machine

- Redéfinition des entiers naturels dans “override.lc” :

```
nat n k := k n
```

```
Coq.Init.Datatypes.0 := nat 0
```

```
Coq.Init.Datatypes.S n := n (λn.int_succ n nat)
```

```
Coq.Init.Datatypes.nat%case z e0 e1 :=  
z (λn.int_null n e0 (int_pred n (λp.e1 (nat p))))
```

- Dans le langage cible (λ_c étendu avec entiers primitifs) :

- ▶ 42 = entier-donnée = instruction sans règle d'évaluation
- ▶ $\text{nat } 42$ ($=_{\beta} \lambda k.k \ 42$) = entier-programme = **entier paresseux**
- ▶ Primitives sur les entiers-données :

$$\text{int_succ } n \ k \quad \succ \quad k \ (n + 1)$$
$$\text{int_pred } n \ k \quad \succ \quad k \ (n - 1)$$
$$\text{int_null } n \ k_0 \ k_1 \quad \succ \quad k_{0|1} \quad (\text{suivant que } n \text{ est nul ou non})$$

Autres *overrides* possibles

- Le changement de représentation des naturels induit un changement de **complexité spatiale**, pas de **complexité temporelle**...
... car les fonctions écrites en Coq continuent à compter sur les doigts

- Extension de l'*override* aux fonctions usuelles :

```
Coq.Init.Datatypes.plus n m :=  
  n ( $\lambda n. m$  ( $\lambda m. \text{int\_plus } n m \text{ nat}$ ))
```

```
Coq.Init.Datatypes.mult n m :=  
  n ( $\lambda n. m$  ( $\lambda m. \text{int\_mult } n m \text{ nat}$ ))
```

```
Coq.Init.Datatypes.pred n :=  
  n ( $\lambda n. \text{int\_null } n (\text{nat } 0)$  ( $\text{int\_pred } n \text{ nat}$ ))
```

- Redéfinition de certains réalisateurs extraits de la librairie standard : par ex. la loi de Peirce (prouvée en Coq à l'aide du tiers-exclu)
- Redéfinition des ordres \leq , $<$, \geq , $>$ et de leurs propriétés

```

Coq.Init.Datatypes.nat_rect =
  \P\f\f0 .fix_1_1 (\F\n Coq.Init.Datatypes.nat%case n f (\n f0 n (F n)))

Coq.Init.Datatypes.nat_ind =
  \P Coq.Init.Datatypes.nat_rect P

Coq.Init.Peano.plus_n_0 =
  \n
  Coq.Init.Datatypes.nat_ind
  .type (Coq.Init.Logic.refl_equal .type (nat 0))
  (\n\IHn
    Coq.Init.Logic.f_equal
    .type .type Coq.Init.Datatypes.S n (Coq.Init.Peano.plus n (nat 0))
    IHn) n

Coq.Init.Peano.plus_n_Sm =
  \n\m
  Coq.Init.Datatypes.nat_ind
  .type (Coq.Init.Logic.refl_equal .type (Coq.Init.Datatypes.S m))
  (\n\IHn
    Coq.Init.Logic.f_equal
    .type .type Coq.Init.Datatypes.S
    (Coq.Init.Datatypes.S (Coq.Init.Peano.plus n m))
    (Coq.Init.Peano.plus n (Coq.Init.Datatypes.S m)) IHn) n

Coq.Arith.Plus.plus_comm =
  \n\m
  Coq.Init.Datatypes.nat_ind
  .type (Coq.Init.Peano.plus_n_0 m)
  (\y\H
    Coq.Init.Logic.eq_ind
    .type (Coq.Init.Datatypes.S (Coq.Init.Peano.plus m y)) .type
    (Coq.Init.Logic.f_equal
      .type .type Coq.Init.Datatypes.S (Coq.Init.Peano.plus y m)
      (Coq.Init.Peano.plus m y) H)
    (Coq.Init.Peano.plus m (Coq.Init.Datatypes.S y))
    (Coq.Init.Peano.plus_n_Sm m y)) n

```

Exemple : commutativité de $+_{\text{nat}}$ (réalisateur optimisé)

```
Coq.Arith.Plus.plus_comm =  
  \n\m\z z
```

Les commandes d'extraction classique (1/2)

L'extracteur produit incrémentalement une **liste de λ_c -définitions**,
de la forme $instruction\ ident_1 \cdots ident_n = \lambda_c-term$

- Classical Extraction Reset.

Efface la liste courante de λ_c -définitions

- Classical Extraction Save "*file.lco*".

Sauve la liste courante de λ_c -définitions dans "*file.lco*"
(*.lco* = *lambda-calculus object*)

- Classical Extract *qualid*₁ ... *qualid*_n.

Ajoute à la liste courante les λ_c -définitions nécessaires pour
réaliser les objets de l'environnement *qualid*₁ ... *qualid*_n,
ainsi que tous les objets dont ceux-ci dépendent

- Classical Extract Witness *ex_proof* using *dec_proof*.

Extrait une fonction de calcul de témoins existentiels à partir d'une preuve d'existence et d'une preuve de décidabilité

$$ex_proof \quad : \quad \text{forall } \vec{x} : \vec{T}, \text{ exists } y : U, P \vec{x} y$$
$$dec_proof \quad : \quad \text{forall } \vec{x} : \vec{T}, \text{ forall } y : U, \{P \vec{x} y\} + \{\sim P \vec{x} y\}$$

sous la forme d'une instruction `ex_proof%witness`

Plan

- 1 Extraction de programmes en logique constructive
- 2 Typage vs. réalisabilité
- 3 Traductions négatives et extraction classique
- 4 Réalisabilité classique dans PA2 (Krivine)
- 5 Réalisabilité classique : de PA2 à pI_{CC} + EM
- 6 Le schéma d'extraction classique en Coq
- 7 Exemples

Le principe du minimum et ses conséquences

Principe du minimum

Si X est un type de données habit  :

$$(\forall f : X \rightarrow \mathbb{N}) (\exists x : X) (\forall y : X) f(x) \leq f(y)$$

Preuve. Par l'absurde, avec une r currence forte.

- **Remarque :** Pas de preuve intuitionniste (oracle)

Corollaire

$$(\forall f : \mathbb{N} \rightarrow \mathbb{N}) (\exists x : \mathbb{N}) f(x) \leq f(2x + 1)$$

Plus g n ralement : $(\forall f, g : \mathbb{N} \rightarrow \mathbb{N}) (\exists x : \mathbb{N}) f(x) \leq f(g(x))$

Preuve. On prend le point x donn  par le principe du minimum.

- **Remarque :** Il y a aussi une preuve intuitionniste (car Π_2^0)

Démo