

Extraction de programmes à partir de preuves classiques en Coq

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

26 février 2009

La question existentielle (en logique)

- Dans une logique/théorie donnée :
 - 1 Qu'est ce qu'une preuve de $\vdash (\exists x \in D) P(x)$?
 - 2 Que peut on faire (i.e. calculer) avec?
- Dans Coq (logique constructive, Curry-Howard) :
 - Le type `exists x : D, P(x)` est une **somme dépendante**
 - Un terme clos de ce type se $(\beta\delta\iota\zeta)$ -réduit sur un couple (x_0, p_0) , avec $x_0 : D$ et $p_0 : P(x_0)$.
 - Même chose pour le type `{x : D | P(x)}` (dans Type)
- **Remarque** : un résultat non trivial qui repose sur
 - La propriété de normalisation (faible) dans CIC
 - La propriété de *subject reduction*
 - La structure des formes canoniques

Première partie

Extraction(s) constructive(s) en Coq

Extraction constructive : le problème

- Une preuve constructive de

$$\forall \ell : \text{list nat}, \exists \ell' : \text{list nat}, (\text{sorted } \ell' \wedge \text{permut } \ell \ell')$$

est une fonction qui à toute liste ℓ associe un couple formé

- d'un **témoin** $\ell' : \text{list nat}$ (à extraire)
 - d'une **justification** $p : (\text{sorted } \ell' \wedge \text{permut } \ell \ell')$ (à jeter)
- **Problème** : éviter de perdre son temps à calculer la justification !
- Méthodes d'analyse statique pour distinguer dans le code
 - ce qui contribue à calculer le témoin (contenu informatif)
 - ce qui ne contribue pas à le calculer (contenu non informatif)
- **Remarque** : problème surtout sensible en *call-by-value* (Caml).
En *call-by-{name|need}*, il suffirait simplement d'ignorer la justification...

Deux schémas d'extraction pour Coq

- **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**
 $CIC \rightarrow \text{Haskell, SML, Caml...} + \text{Obj.magic}$
 - 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} }`

Extraction constructive et axiomes purement logiques

- Un effet secondaire intéressant :
 - Comme les termes de preuve sont effacés au cours de l'extraction, on peut extraire des programmes à partir de termes qui dépendent d'**axiomes purement logiques** (i.e. dans Prop) :
 - L'extraction reste compatible avec le **tiers-exclu** dans Prop...
... ou tout autre axiome (dans Prop), même **contradictoire** !
 - Correspond à un phénomène visible dans les modèles de CIC : une incohérence à un niveau de la hiérarchie d'univers ne se propage pas aux niveaux supérieurs :

$$\begin{aligned}(\text{forall } X : \text{Prop}, X) &\not\rightarrow (\text{forall } X : \text{Type}_1, X) \\(\text{forall } X : \text{Type}_i, X) &\not\rightarrow (\text{forall } X : \text{Type}_{i+1}, X)\end{aligned}$$

(les univers agissent comme des pare-feu logiques)

- Programme extrait \neq contenu calculatoire de la preuve
 \approx **squelette Martin-Lövien du terme source**
+ subtilités liées à l'extraction des Fixpoint gardés

Variantes pour l'extraction constructive

Autres *design* possibles pour l'extraction constructive :

- En jouant sur la séparation **informatif** / **non-informatif**
 - Dupliquer la sorte Prop : Prop-à-garder et Prop-à-jeter
 - Supprimer la distinction informatif / non informatif, et la remplacer par des techniques d'analyse statique au cas par cas.
- En basant l'extraction sur un **modèle de réalisabilité** plutôt que sur un **modèle de normalisation forte**
 - Une bien plus grande liberté dans le choix des réalisateurs

Par exemple, le remplacement (à la volée) des entiers unaires par des entiers binaires et des fonctions usuelles (+, ×, etc.) par leurs versions efficaces est justifié par le modèle de réalisabilité intuitionniste (*D-set model* de Streicher), pas par le modèle de normalisation.
 - Orthogonal à la séparation informatif / non informatif

Deuxième partie

Extraction de programme en logique classique

Non constructivité de la logique classique

- **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 (1/2)

- **Traduction négative** = une traduction $A \mapsto A^{\neg\neg}$ telle que

$$\mathcal{T}_c \vdash_{(\text{LK})} A \Rightarrow \mathcal{T}_i \vdash_{(\text{LJ})} A^{\neg\neg}$$

(\mathcal{T}_c = une théorie classique, \mathcal{T}_i = fragment intuitionniste de \mathcal{T}_c)

- En général, l'existence d'une traduction négative implique la **conservativité** de \mathcal{T}_c sur \mathcal{T}_i pour les formules Π_0^1 , Σ_0^1 et Π_0^2 :

$$\mathcal{T}_c \vdash (\forall x)(\exists y : \mathbb{N})D(x, y) \Rightarrow \mathcal{T}_i \vdash (\forall x)(\exists y : \mathbb{N})D(x, y)$$

où $D(x, y)$ est sans quantificateurs / à quantifications bornées / décidable / ...

- Exemples de formules Π_0^2 très utiles :
 - “La machine de Turing ZO termine sur tous ses inputs”
 - “Tout terme bien typé dans le formalisme MEU normalise”

Traductions négatives (2/2)

- Exemples de $(\mathcal{T}_c, \mathcal{T}_i)$ pour lesquels une telle traduction existe
 - Les arithmétiques : (PAN, HAN) $(1 \leq n \leq \omega)$
 - 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 : $(\text{CIC} + \text{EM}, \text{CIC})$ (conjecture : oui)
- L'existence d'une telle traduction implique :
 - La conservativité de \mathcal{T}_c sur \mathcal{T}_i pour les formules Π_0^2
 - L'équiconsistance des théories \mathcal{T}_c et \mathcal{T}_i
 - L'équi-**1-consistance** des théories \mathcal{T}_c et \mathcal{T}_i

Traduction négative et extraction de programmes

- Combinée avec Curry-Howard (intuitionniste) ou un modèle de réalisabilité intuitionniste (de \mathcal{T}_i), une traduction négative permet de faire de l'extraction classique sur les formules Σ_0^1 et Π_0^2 :

- $\bullet \pi : \forall^{\mathbb{N}}x \exists^{\mathbb{N}}y f(x, y) = 0$ (dans \mathcal{T}_c)

- $\bullet \pi x_0 : \exists^{\mathbb{N}}y f(x_0, y) = 0$ (dans \mathcal{T}_c , x_0 fraîche)

- $\bullet (\pi x_0)^{\neg\neg} : \neg_{\mathcal{R}} \forall^{\mathbb{N}}y \neg_{\mathcal{R}} f(x_0, y) = 0$ (dans \mathcal{T}_i , $\neg_{\mathcal{R}} A \equiv A \Rightarrow R$)
 $\quad : \forall^{\mathbb{N}}y (f(x_0, y) \Rightarrow R) \Rightarrow R$

- \bullet On pose $R \equiv \exists^{\mathbb{N}}y_0 f(x_0, y_0) = 0$ (Astuce de Friedman)

- $\bullet (\pi x_0)^{\neg\neg} : \underbrace{\forall^{\mathbb{N}}y (f(x_0, y) \Rightarrow \exists^{\mathbb{N}}y_0 f(x_0, y_0))}_{\exists\text{-intro}} \Rightarrow \exists^{\mathbb{N}}y_0 f(x_0, y_0)$

- $\bullet (\pi x_0)^{\neg\neg} (\exists\text{-intro}) : \exists^{\mathbb{N}}y_0 f(x_0, y_0)$ (dans \mathcal{T}_i)

- $\bullet \lambda x_0 . (\pi x_0)^{\neg\neg} (\exists\text{-intro}) : \forall^{\mathbb{N}}x_0 \exists^{\mathbb{N}}y_0 f(x_0, y_0)$ (dans \mathcal{T}_i)

- Berardi, Berger, Coquand, Kohlenbach, Schwichtenberg (MinLog), ...

Logique classique et continuations

- **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**

- **Exemple** : comment prouver

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

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


La réalisabilité classique (Krivine)

- **Réalisabilité classique** = méthode d'extraction de programme par traduction négative **reformulée en style direct** :

$$\text{Réal. classique} \approx (\text{CPS})^{-1} \circ (\text{Réal. intuit.}) \circ (\neg\neg\text{-trad.})$$

- 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 imprédictif)

Le λ_c -calcul (Krivine)

Syntaxe

Termes $t, u ::= x \mid \lambda x. t \mid tu \mid \underbrace{\alpha \mid \dots}_{\text{instructions}} \mid k_\pi$
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 $tu \star \pi \quad \gamma \quad t \star u \cdot \pi$

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

Call/cc $\alpha \star t \cdot \pi \quad \gamma \quad t \star k_\pi \cdot \pi$

Resume $k_\pi \star t \cdot \pi' \quad \gamma \quad t \star \pi$

...

...

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

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

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

La question existentielle, le retour

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** $p' \Vdash \neg A(n)$

Troisième partie

Le modèle de réalisabilité étendu à pICC



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)
- ↪ Nécessité d'interpréter les types de pCIC par des objets plus complexes que les valeurs de fausseté
- Une source d'inspiration (intuitionniste) :
 - Les ω -ensembles [Hyland 82, Longo-Moggi 90]
 - Les D -ensembles [Steicher 91]
 - Les λ -ensembles [Altenkirch 93]
- ↪ Modèles de réalisabilité intuitionnistes de (p)CIC

La structure de Π -ensemble

Définition (Π -ensemble)

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**)

- Intuition : $v \perp_X \pi$ signifie “ π réfute v dans X ”
- $v^{\perp_X} =$ ensemble des réfutations de v dans X
- Réalisabilité : $t \Vdash v \in X \equiv t \in (v^{\perp_X})^{\perp}$
- Π -ensemble pointé = Π -ensemble avec point distingué
- Produit cartésien d'une famille de Π -ensembles
(généralisation de l'implication et de la quantification universelle)

Éléments d'interprétation

- Les ingrédients habituels :
 - Univers de pCIC interprétés par des univers ensemblistes
 - Fonctions ensemblistes codées par leur trace (\neq graphe) pour forcer l'identification $(x \in |X| \mapsto \bullet) = \bullet$ (**objet-preuve**)
 \rightsquigarrow sémantique de la “proof-irrelevance”
- Propositions interprétées par des **Π -ensembles dégénérés** :
 - Support : $|X| = \{\bullet\}$ (singleton objet-preuve)
 - Réfutation locale réduite à une valeur de fausseté : $\bullet^{\perp x} \subseteq \Pi$
 - Réalisabilité : $t \Vdash \bullet \in X$ noté $t \Vdash X$
- Sortes interprétées par des **Π -ensembles grossiers** :
 - $\perp_X = \emptyset$ (\rightsquigarrow éléments du support réalisés par tous les termes)

Correction

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

Extraction et adéquation

- L'interprétation $M \mapsto \llbracket M \rrbracket$ associe une **dénotation** $\llbracket M \rrbracket \in \llbracket T \rrbracket$ à chaque terme $M : T \dots$ mais **aucun réalisateur** $t \Vdash \llbracket M \rrbracket \in \llbracket T \rrbracket$

Ainsi les preuves $M : T : \text{Prop}$ sont interprétées par \bullet (proof-irrelevance), sans mentionner un réalisateur particulier $t \Vdash \bullet \in \llbracket T \rrbracket$

- À côté de la fonction d'interprétation $M \mapsto \llbracket M \rrbracket$ on définit une **fonction d'extraction** $M \mapsto M^*$ de pCIC dans λ_c

Invariant (adéquation) : Si $M : T$, alors $M^* \Vdash \llbracket M \rrbracket \in \llbracket T \rrbracket$

- Définition de la fonction d'extraction
 - Sur le PTS sous-jacent : **projection vers le λ -calcul** (en traduisant les types par n'importe quel λ -terme)
 - Autres constructions : cf plus loin

Extraction de constantes (1/2)

- À chaque constante $c : T$ (avec équations) on doit associer
 - Une dénotation $\llbracket c \rrbracket \in \llbracket T \rrbracket$ (satisfaisant les équations)
 - Un réalisateur $c^* \Vdash \llbracket c \rrbracket \in \llbracket T \rrbracket$
 - Pas d'autres contraintes sur c^* : **transparence de réalisation**
- **Exemple** : Si $c : T$ est un **type** ($T \equiv \text{Prop/Set/Type}$) :
 - $\llbracket c \rrbracket$ doit être un Π -ensemble
 - dont le support est l'équivalent ensembliste de c dans le modèle (Par exemple : $\llbracket \text{nat} \rrbracket = \mathbb{N}$)
 - dont la relation de réfutation locale **conditionne la représentation des données de type c dans le code extrait** (entiers unaires ou binaires?)
 - $c^* =$ n'importe quel λ -terme (car $\llbracket T \rrbracket = \Pi$ -ensemble grossier)

Extraction de constantes (2/2)

- Si c est une **fonction**, par ex. : $\text{plus} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$
 - Le choix de $\llbracket c \rrbracket$ est (en général) imposé par les équations de c ,
ici : $\llbracket \text{plus} \rrbracket = +_{\mathbb{N}}$
 - Mais on a souvent plusieurs choix possibles pour c^* ...
(Si entiers unaires : addition par récursion à gauche ou à droite?)
- Si $c : T$ est un **axiome** ($T : \text{Prop}$) :
 - $\llbracket c \rrbracket = \bullet$ (pas d'autre choix possible)
 - $c^* =$ n'importe quelle quasi-preuve $t \Vdash \bullet \in \llbracket T \rrbracket$ (s'il en existe)
- Même chose si c est un **théorème** ($c := M : T : \text{Prop}$)
 - On peut choisir : $c^* = M^*$... (réalisateur par défaut)
 - Ou prendre une autre quasi-preuve $c^* \Vdash \bullet \in \llbracket T \rrbracket$
 \rightsquigarrow Permet d'introduire des **réalisateurs optimisés**

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)

Le modèle générique

- Le modèle de réalisabilité classique (dans PA2, ZF, pCIC) est paramétré par un ensemble saturé $\perp\!\!\!\perp \subseteq \Lambda \star \Pi$
 - Les lemmes de correction et d'adéquation valent pour tout $\perp\!\!\!\perp$
 - Mais la dénotation d'un terme dépend (en général) de $\perp\!\!\!\perp \dots$
- Pour prédire le comportement des réalisateurs : le **générique**

$$\perp\!\!\!\perp_g = \mathcal{C} \left(\bigcup_{t \in \text{QP}} \text{thd}(t \star \alpha_t) \right)$$

- QP = ensemble des **quasi-preuves** (= termes sans continuations)
 - $t \mapsto \alpha_t$ = bijection entre les fonds de pile et les quasi-preuves
 - $\text{thd}(p) = \{p' : p \succ^* p'\}$ (**fil** de p)
 - Hypothèse : aucune instruction ne touche au fond de pile
- Les propriétés de terminaison des programmes extraits se démontrent (par l'absurde) avec $\perp\!\!\!\perp = \perp\!\!\!\perp_g$

Quatrième partie

Le schéma d'extraction



Extraction sur le PTS sous-jacent

- Une projection $M \mapsto M^*$ des termes de pI λ CC 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 $M : T : \text{Prop}/\text{Set}/\text{Type}$ (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 pICC, 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 :

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

Extraction du filtrage

- 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 .fix_n_i ($1 \leq i \leq n$) sont définies par

$$\begin{aligned} \text{.fix_n_i } F_1 \cdots F_n &:= \\ F_i &(\text{.fix_n_1 } F_1 \cdots F_n) \cdots (\text{.fix_n_n } F_1 \cdots F_n) \end{aligned}$$

- Dans l'extracteur, les instructions .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 :

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

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

Exemples (1/2)

- Les booléens

```
true      :=  $\lambda e_1 e_2 . e_1$   
false     :=  $\lambda e_1 e_2 . e_2$   
bool%case :=  $\lambda z . z$ 
```

- Le type option (monomorphe) :

```
None      :=  $\lambda e_1 e_2 . e_1$   
Some      :=  $\lambda x e_1 e_2 . e_2 x$   
option%case :=  $\lambda z . z$ 
```

- Les listes (polymorphes) :

```
nil       :=  $\lambda _ e_1 e_2 . e_1$   
cons      :=  $\lambda _ x y e_1 e_2 . e_2 x y$   
list%case :=  $\lambda z . z$ 
```

Exemples (2/2)

- 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 :=  $\lambda z . z$ 
```

- Encore plus inefficace 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
 - nat 42 ($=_{\beta} \lambda k.k\ 42$) = entier-programme = **entier paresseux**
 - Primitives sur les entiers-données :

```
int_succ n k    > k (n + 1)
```

```
int_pred n k    > k (n - 1)
```

```
int_null n k0 k1 > k0|1 (suivant que n 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 (λn. m (λm. int_plus n m nat))
```

```
Coq.Init.Datatypes.mult n m :=  
  n (λn. m (λm. int_mult n m nat))
```

```
Coq.Init.Datatypes.pred n :=  
  n (λn. int_null n (nat 0) (int_pred n 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

Les commandes d'extraction classique (1/2)

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

- `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 qualid1 ... qualidn`.
Ajoute à la liste courante les λ_c -définitions nécessaires pour réaliser les objets de l'environnement `qualid1 ... qualidn`, ainsi que tous les objets dont ceux-ci dépendent

Les commandes d'extraction classique (2/2)

- 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

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

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

Propriété satisfaite par la fonction `ex_proof%witness` :

- pour tous objets sémantiques $\vec{x} : \vec{T}$
- pour tous réalisateurs $\vec{t} \Vdash \vec{x} : \vec{T}$
- pour tous λ_c -termes k_1, k_2

il existe un objet sémantique $y : U$ et un réalisateur $v \Vdash \bullet : P y$

- `ex_proof%witness` $\vec{t} \Vdash y : U$
- `dec_proof` \vec{t} (`ex_proof%witness` \vec{t}) $\Vdash \text{left}(\bullet) : \{P y\} + \{\neg P y\}$
- `sumbool%case` (`dec_proof` \vec{t} (`ex_proof%witness` \vec{t})) $k_1 k_2 \succ^* k_1 v$