

Extraction de programmes à partir de preuves classiques en Coq

Alexandre Miquel

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

23 avril 2009 – VERIMAG (Grenoble)

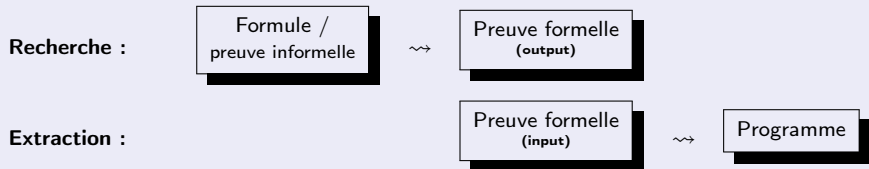
La question existentielle (en logique)

- 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** ?
(en fait : extraire un **programme** qui calcule le témoin)

- Extraction de programme \neq Recherche (ou construction) de preuve



- 1 Extraction de programmes en logique constructive
- 2 Traductions négatives et extraction classique
- 3 La réalisabilité classique
- 4 Le schéma d'extraction classique en Coq
- 5 Exemples

- 1 Extraction de programmes en logique constructive
- 2 Traductions négatives et extraction classique
- 3 La réalisabilité classique
- 4 Le schéma d'extraction classique en Coq
- 5 Exemples

Extraction constructive : le dictionnaire

- Sémantique de Brouwer-Heyting-Kolmogorov
- Correspondance de Curry-Howard

Théorie de la démonstration

Proposition (formule)
Preuve (dérivation)

$A \wedge B$, $A \vee B$, $A \Rightarrow B$

Règle de déduction
Vérificateur de preuve

Élimination des coupures
Preuve sans coupure

Preuve d'un lemme
Théorie (énoncés et preuves)

Programmation fonctionnelle

Type de données
Programme (ou donnée)

$A \times B$, $A + B$, $A \rightarrow B$

Règle de typage
Vérificateur de type

Évaluation (calcul)
Valeur

Sous-programme
Module (interface et implém.)

L'extraction de programme en logique constructive

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

$$\vdash \exists x A(x) \quad \rightsquigarrow \quad t \quad + \quad \vdash A(t)$$

- 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 :

$$\begin{aligned} p &: (\forall x : A) (\exists y : B) P(x, y) \\ &: \Pi x : A . \Sigma y : B . P(x, y) \quad \rightsquigarrow \quad f : A \rightarrow B \end{aligned}$$

Stratégies d'élimination du code mort

- **Dans HA** (MINLOG – Schwichtenberg *et al.*, Munich)

- ▶ Élimination des preuves des **formules de Harrop** (= code mort)

$$\begin{aligned} A &::= \perp \mid e = e \mid A \wedge A \mid A \Rightarrow A \mid \forall^{\sigma} x A \mid \exists^{\sigma} x A \\ H &::= \perp \mid e = e \mid H \wedge H \mid A \Rightarrow H \mid \forall^{\sigma} x H \end{aligned}$$

(Intuition : formule de Harrop = formule à au plus une preuve)

- **Dans le calcul des constructions inductives** (Coq)

- ▶ Élimination de code mort basée sur la stratification :
On garde ce qui est dans Set/Type, on jette ce qui est dans Prop.
- ▶ Nécessite de distinguer 3 quantificateurs existentiels !

★ $\text{exists } x : T, P x$	(dans Prop, avec $T : \text{Type}$ et $P x : \text{Prop}$)
★ $\{x : T \mid P x\}$	(dans Type , avec $T : \text{Type}$ et $P x : \text{Prop}$)
★ $\{x : T \& P x\}$	(dans Type , avec $T : \text{Type}$ et $P x : \text{Type}$)

Recherche de preuve ou extraction : un exemple

- Pour tous $x, y, z, n \in \mathbb{N}$, soit :

$$F(x, y, z, n) = \begin{cases} 1 & \text{si } x > 0, y > 0, n > 2 \text{ et } x^n + y^n = z^n \\ 0 & \text{sinon} \end{cases}$$

Théorème (Wiles, Taylor) : $(\forall x, y, z, n \in \mathbb{N}) F(x, y, z, n) = 0$ (Π_1^0)

- **Recherche de preuve :** > 350 ans
 - ▶ Preuve très complexe (utilise plusieurs branches des mathématiques)
 - ▶ À quand le terme de preuve p en Coq ?
- **Extraction de programme :** aucun intérêt !
 - ▶ Comportement calculatoire prédictible : $p \times y \ z \ n \succ^* \text{id}$
 - ▶ Calculatoirement, on peut remplacer p par $p' \equiv \lambda x y z n . \text{id}$

\rightsquigarrow Point de vue de la **réalisabilité**

Typage et réalisabilité

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

Typage ($t : A$)

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

- Analyse syntaxique des termes
- Au minimum semi-décidable
- Justification simple : dérivation
 \rightsquigarrow Recherche de preuve

Réalisabilité ($t \Vdash A$)

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

- Analyse calculatoire des termes
- Fortement indécidable
- Justification externe (preuve)
 \rightsquigarrow Extraction de programme

Adéquation : Correction pour le typage \Rightarrow Correction pour la réalisabilité

Historiquement, la réalisabilité a été introduite en **logique** (Kleene'45) comme une relation $t \Vdash A$ entre des **programmes** et des **formules**

Plan

- 1 Extraction de programmes en logique constructive
- 2 Traductions négatives et extraction classique**
- 3 La réalisabilité classique
- 4 Le schéma d'extraction classique en Coq
- 5 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 ($X \neq \emptyset$) :

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

R -traduction (historiquement : A -traduction)

- **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 $A \mapsto A^*$ t.q. $\mathcal{T}_c \vdash_{(LK)} A$ entraîne $\mathcal{T}_i \vdash_{(LJ)} A^*$

- **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

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

Exemples et contre-exemples

- Exemples de $(\mathcal{T}_c, \mathcal{T}_i)$ pour lesquels une R -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 : $(\text{MLTT} + \text{tiers-exclu}, \text{MLTT})$

où MLTT = Théorie des types de Martin-Löf (et ses variantes)

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

- 1 \mathcal{T}_c et \mathcal{T}_i sont **équiconsistantes** (cas : $R \equiv \perp$)
- 2 \mathcal{T}_c est **conservative** sur \mathcal{T}_i pour les formules Σ_1^0, Π_2^0 (astuce de Friedman)
- 3 \mathcal{T}_c et \mathcal{T}_i sont **équi-1-consistantes**

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

• 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 $\lambda k. \text{Right} (\lambda x. k (\text{Left}(x)))$

- 1 Extraction de programmes en logique constructive
- 2 Traductions négatives et extraction classique
- 3 La réalisabilité classique**
- 4 Le schéma d'extraction classique en Coq
- 5 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 (CC^ω = calcul des constructions avec univers) [CSL'07]
- ▶ dans pCIC  (pCIC = CIC – Set imprédicatif)

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

Le λ_c -calcul (Krivine)

Syntaxe

Termes	$t, u ::= \underbrace{x \mid \lambda x . t \mid tu}_{\text{quasi-preuves}} \mid \overbrace{\alpha \mid \dots}^{\text{instructions}} \mid k_\pi$
Piles	$\pi ::= \alpha \mid u \cdot \pi \quad (u, \pi \text{ clos})$
Processus	$p, q ::= t \star \pi \quad (t, \pi \text{ clos})$

k_π = (constante de) continuation associée à π , α = (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	$k_\pi \star t \cdot \pi' \succ t \star \pi$
	$\dots \qquad \qquad \qquad \dots$

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 $t \Vdash A$ ($\equiv t \in |A|$)

Structure du quantificateur existentiel

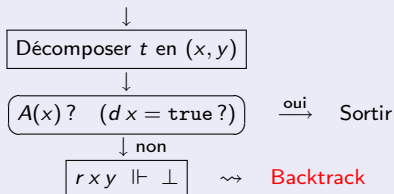
En réalisabilité intuitionniste et classique, un réalisateur de $\exists^{\mathbb{N}}x A(x)$ contient un couple **témoin/justification** (n, t) , avec $n \in \mathbb{N}$ et $t \Vdash A(n)$

- En réalisabilité intuitionniste
 - ▶ On a la garantie que n vérifie $A(n)$
 - ▶ La justification $t \Vdash A(n)$ est donc superflue (code mort)
- En réalisabilité classique
 - ▶ Aucune garantie que n vérifie $A(n)$! (possibilité de **faux-témoin**)
 - ▶ La justification $t \Vdash A(n)$ n'offre aucune garantie... (**parapreuve**)
... mais contient le matériel nécessaire pour backtracker
- Comment obtenir des témoins fiables?

Recherche de témoin par essai/erreur

- Les ingrédients :

- ▶ Une preuve (classique) $t : (\exists x : \mathbb{N}) A(x)$
- ▶ 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 $\neg A(n)$



Implémentation

$t (M (\lambda xy . d x (\text{stop } x) (r x y)))$
(M = opérateur de mise en mémoire)

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 Traductions négatives et extraction classique
- 3 La réalisabilité classique
- 4 Le schéma d'extraction classique en Coq
- 5 Exemples

Extraction sur le PTS sous-jacent

- Une projection $M \mapsto M^*$ des termes de pICC 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)

- Par défaut, les structures de données (co)inductives sont traduites par les codages habituels en λ -calcul
- Le filtrage est traduit à l'aide d'un **combinateur de filtrage** `I%case` associé à chaque type (co)inductif `I` :

`I%case` (terme à déstructurer) (branches de filtrage...)

- Les (co)points-fixes sont traduits en introduisant des combinateurs de point fixe `.fix_n_i` ($1 \leq i \leq n$) :

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

- Par défaut, les axiomes sont traduits par des constantes inertes

Exemple

- Les entiers naturels (codage par défaut) :

```
Coq.Init.Datatypes.O      :=  $\lambda e_1 e_2 . e_1$   
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 de constante, l'extracteur consulte un fichier de définitions "override.lco" pour voir si cette constante y est définie.
 - ▶ Si oui, la définition correspondante est placée dans le code extrait
 - ▶ Sinon, l'extracteur produit une définition suivant le schéma par défaut
- En pratique le mécanisme d'*override* sert à :
 - ▶ Changer la représentation de certains inductifs (nat)
 - ▶ Réaliser certains axiomes (par défaut réalisés par des constantes inertes)
↪ réalisation des axiomes de **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” :

$$\text{nat } n \ k \ := \ k \ n$$
$$\text{Coq.Init.Datatypes.O} \quad := \ \text{nat } 0$$
$$\text{Coq.Init.Datatypes.S } n \ := \ n \ (\lambda n. \text{int_succ } n \ \text{nat})$$
$$\text{Coq.Init.Datatypes.nat\%case } z \ e_0 \ e_1 \ := \\ z \ (\lambda n. \text{int_null } n \ e_0 \ (\text{int_pred } n \ (\lambda p. e_1 \ (\text{nat } p))))$$

- 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. int_plus n m nat))
```

```
Coq.Init.Datatypes.mult n m :=  
  n (\lambda n. m (\lambda m. int_mult n m nat))
```

```
Coq.Init.Datatypes.pred n :=  
  n (\lambda n. 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

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

```
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

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.
- Classical Extraction Save "*file.lco*".
- Classical Extract *qualid*₁ ... *qualid*_n.
- 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 Traductions négatives et extraction classique
- 3 La réalisabilité classique
- 4 Le schéma d'extraction classique en Coq
- 5 Exemples

Le principe du minimum et ses conséquences

Principe du minimum

Si X est un type de données habitué :

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

Preuve. Par l'absurde, avec une récurrence 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 généralement : $(\forall f, g : \mathbb{N} \rightarrow \mathbb{N}) (\exists x : \mathbb{N}) f(x) \leq f(g(x))$

Preuve. On prend le point x donné par le principe du minimum.

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

Démo