

Retour d'expérience sur l'utilisation de Coq

Éric Jaeger

Journée SSURF – 29 juin 2010

Contexte

Quelques développements Coq, dont certains complexes

- ▶ Volume total important
- ▶ Beaucoup de concepts à manipuler
- ▶ Peu ou pas d'utilisation des bibliothèques

Quelques “gadgets” Coq se révèlent alors indispensables

- ▶ Simplicité : paramètres implicites, coercitions, réflexion...
- ▶ Lisibilité : notations
- ▶ Efficacité : optimisations, inductions *ad hoc*
- ▶ Automatisation : tactiques

L'objectif est de fournir un petit retour d'expérience sur ces gadgets, du point de vue d'un utilisateur “naïf”, n'appartenant pas à la communauté Coq

Quelques sources d'information

Coq Reference Manual

Interactive Theorem Proving and Program Development (Coq'Art),
Y. Bertot and P. Castéran

Certified Programming with Dependent Types (CPDT), *A. Chiplala*

Coq-Club

Plan

- 1 **Simplicité**
- 2 Lisibilité
- 3 Efficacité (dans les preuves)
- 4 Automatisation (des preuves)
- 5 Quel rapport avec FOCALIZE ?

Paramètres implicites

COQ peut compléter certains termes à trous, par exemple dans un résultat tel que

Theorem *eqrefl* : $\forall (T : \mathbf{Type})(t : T), t = t$

T peut être déduit de t , comme dans **Check** (*eqrefl* _ t)

Implicit Arguments *eqrefl* [T] permet d'omettre les _

Bonne pratique : rendre implicites les paramètres qui pourront être déduits dans les cas d'utilisation les plus fréquents

- ▶ **apply** (*eqtrans* H_{12} H_{23}) est mieux que **apply** (*eqtrans* T x_1 x_2 $H_{x_1 R x_2}$ x_3 $H_{x_2 R x_3}$) ou encore **apply** (*eqtrans* _ _ $H_{x_1 R x_2}$ _ $H_{x_2 R x_3}$)
- ▶ Pour les cas rares où un paramètre doit être ré-explicité, il reste possible d'utiliser le @ devant l'identifiant

Remarque : utiliser aussi la syntaxe *trans* ($x := \dots$) ($2 := \dots$) ...

Paramètres implicites

Remarque : le langage GALLINA de COQ ne donne pas d'instruction permettant de retrouver le type d'un terme

- ▶ La commande vernaculaire **Check** permet d'afficher le type d'un terme, mais pas de l'utiliser dans une définition
- ▶ **type of** est une tactique interprétée dans la boucle interactive ou au moment de la compilation, mais non dynamique

Bref, on ne peut pas écrire **Definition** $T := \text{type_of } t$

Il est pourtant possible à travers un usage "dévoyé" des paramètres implicites de définir cette fonction, qui est une projection

Definition $\text{type_of}(T : \text{Type})(t : T) := T$.

Implicit Arguments $\text{type_of}[T]$.

Coercitions

Coq permet de déclarer une forme de *cast* automatique, toute valeur de type T peut être considérée comme de type T' modulo l'application d'une fonction f avec **Coercion** $f : T \rightarrow T'$

Cela permet de rendre implicites les conversions “évidentes” – de faire des abus de notation standard

- ▶ Pour un langage de termes qui sont des expressions ou des prédicats, **Inductive** $T := TofE : E \rightarrow T \mid TofP : P \rightarrow T$, permettre de laisser les $TofE$ ou $TofP$ implicites

- ▶ Utiliser les booléens comme des propositions logiques

Definition $bool_true(b : \mathbf{bool}) : \mathbf{Prop} := b = \mathbf{true}$ puis

Coercion $bool_true : \mathbf{bool} \rightarrow \mathbf{Sortclass}$ (par exemple)

Coercitions

Attention à quelques subtilités cependant

Definition $bofn1(n:\mathbb{N}) := \text{match } n \text{ with } 0 \Rightarrow \text{false} \mid _ \Rightarrow \text{true} \text{ end}$

Coercion $bofn1:\mathbb{N} \rightarrow \text{bool}$

$\text{true}=0$ est bien formé... mais pas $0=\text{true}$!

Definition $bofn2(n:\mathbb{N}) := \text{match } n \text{ with } 0 \Rightarrow \text{true} \mid _ \Rightarrow \text{false} \text{ end}$

Coercion $bofn2:\mathbb{N} \rightarrow \text{bool}$

Est-ce que $\text{true}=0$ est prouvable ?

En pratique, on peut rapidement se perdre avec trop de coercitions

- ▶ Les utiliser... sans abuser
- ▶ **Set Printing Coercions** pour rendre explicites les transformations implicites faites par COQ

Logique vs calcul

Il est fréquent de décrire des prédicats dont on peut monter ensuite qu'il sont décidables ; cette preuve de décidabilité est un programme qui peut être extrait

En pratique, cependant

- ▶ On peut directement écrire une fonction qui renvoie un booléen, implémentant un tel prédicat : le code est maîtrisé, les simplifications (calculs) en Coq sont plus lisibles et efficaces, etc.
- ▶ De même on peut préférer écrire une fonction qui calcule y à partir de x plutôt que d'utiliser une preuve existentielle sur une relation – ne serait-ce que pour maîtriser la valeur y utilisée

A noter qu'on peut définir des méta-relations en Coq entre un prédicat et une fonction booléenne qui le décide, etc.

Logique vs calcul

Mais on peut aussi se contenter d'utiliser uniquement les programmes, et combiner avec les coercitions pour avoir des notations intuitives (comme en FOCALIZE)

Inductive $Pofb : \mathbf{bool} \rightarrow \mathbf{Prop} := pofb : Pofb \mathbf{true}$.

Coercion $Pofb : \mathbf{bool} \rightarrow \mathbf{Sortclass}$.

Fixpoint $even(n : \mathbb{N}) \{ \mathbf{struct} \ n \} : \mathbf{bool} :=$
 $\mathbf{match} \ n \ \mathbf{with} \ 0 \Rightarrow \mathbf{true} \mid S \ 0 \Rightarrow \mathbf{false} \mid S \ (S \ n') \Rightarrow even \ n' \ \mathbf{end}$.

Theorem $even_SS_n : \forall (n : \mathbb{N}), even (S (S \ n)) \rightarrow even \ n$.

Proof. $\mathbf{intros} \ n \ H; \mathbf{inversion_clear} \ H; \mathbf{apply} \ pofb. \ \mathbf{Qed}$.

Anonymisation

Appliquons ce qui précède avec $Fresh: T \rightarrow V$ une fonction qui pour un terme renvoie une variable, avec une preuve que $Fresh(t)$ est toujours fraîche dans T (i.e. $Fresh_free: \forall (t: T), Fresh(t) \setminus T$)

En pratique, cela peut parfois être lourd : dans une preuve pour laquelle on a besoin de $v \setminus (t_1 + t_2) * t_3$, on se retrouve à manipuler $Comb(Fresh\ t_1, Comb(Fresh\ t_2, t_3)) \setminus (t_1 + t_2) * t_3$

Trucs

- ▶ Utiliser **set** ($v := Fresh\ (t_1 + t_2) * t_3$)
- ▶ Utiliser un théorème d'anonymisation de la fonction

Theorem $fresh: \forall (t: T), \{v \mid v \setminus t\}$.

Proof. **intros** t ; **exists** $(Fresh\ t)$; **apply** $Fresh_free$. **Qed.**

Ensuite **destruct** $(fresh\ t)$ **as** $[v\ H]$ pour avoir v et $H: v \setminus t$. . . C'est exactement l'inverse de l'extraction à partir d'une preuve !

Plan

- 1 Simplicité
- 2 **Lisibilité**
- 3 Efficacité (dans les preuves)
- 4 Automatisation (des preuves)
- 5 Quel rapport avec FOCALIZE ?

Notations

COQ permet de définir “à la volée” des notations quelconques utilisant des symboles Unicode (en UTF-8 par exemple)

- ▶ infixe, préfixe, etc.
- ▶ en associant une priorité

Cela permet d'être beaucoup plus proche des notations usuelles – bref, c'est un “gadget” indispensable !

Par exemple :

Notation `''T''` := (**true**).

Notation `''⊥''` := (**false**).

Notation `''b1 ^ b2''` := (**andb** *b1 b2*) (**at level 80, right associativity**).

Notations

On peut aller au-delà en combinant avec la possibilité d'avoir des paramètres implicites et des coercitions

Definition $Subsets(A: \mathbf{Set}) := A \rightarrow \mathbf{Prop}$.

Notation $"' \uparrow' A"$:= $(Subsets A)$ (no associativity, at level 50).

Inductive $Inter(A: \mathbf{Set})(S1 S2: \uparrow A): \uparrow A :=$

$inter: \forall (a: A), S1 a \rightarrow S2 a \rightarrow (Inter A S1 S2) a$.

Notation $"S1 \cap S2"$:= $(Inter _ S1 S2)$ (left associativity, at level 25).

Definition $Relation(A B: \mathbf{Set}) := \uparrow (A * B)$.

Infix $" \leftrightarrow "$:= $Relation$ (at level 30).

Notations

Dans BICOQ, on définit le type des prédicats et des expression du B, puis le type des termes du B, des notations, des coercitions... et des fonctions qui utilisent les notations dans les *pattern matchings*¹

Fixpoint $BPdepth(P:\Pi)\{\mathbf{struct} P\} : \mathbb{N} :=$

match P with $\neg P' \Rightarrow S(BPdepth P') \mid \dots$ **end**

with $BEdepth(E:\Sigma)\{\mathbf{struct} E\} : \mathbb{N} :=$

match E with $E1 \mapsto E2 \Rightarrow S(\max(BEdepth E1, BEdepth E2) \mid \dots$ **end**

Definition $BTdepth(T:\Theta) : \mathbb{N} :=$

match T with $TofE E' \Rightarrow BEdepth E' \mid TofP P' \Rightarrow BPdepth P'$ **end**.

Notation " $\mathcal{D}(' T')$ " $:= (BTdepth T)$.

$\mathcal{D}(E)$, $\mathcal{D}(P)$ et $\mathcal{D}(T)$ sont tous des termes bien formés et bien typés.

¹Mais pas les coercitions...

Notations

Cela permet par exemple, au final, pour ce théorème du B-BOOK

$$\exists x \cdot (P \Rightarrow Q) \Leftrightarrow (\forall x \cdot P \Rightarrow \exists x \cdot Q)$$

Theorem 1.3.4

D'avoir comme code COQ correspondant

```
Theorem bbt_1_3_4 : forall (i:I)(p q:Π), ⊢(∃(i•p⇒q)⇔(∀(i•p)⇒∃(i•q))).
```

Avec des paramètres explicites, sans coercitions et sans notations, le même énoncé serait illisible... et beaucoup plus volumineux

Plan

- 1 Simplicité
- 2 Lisibilité
- 3 Efficacité (dans les preuves)**
- 4 Automatisation (des preuves)
- 5 Quel rapport avec FOCALIZE ?

Factorisation des constructeurs

Petit truc qui se révèle plus utile qu'on ne pourrait le penser : autant que possible factoriser les constructeurs dans les définitions

Fixpoint $Lift(h:\mathbb{N})(t:T)\{\mathbf{struct} t\}:T :=$
match t **with**
 | $\lambda t' \Rightarrow \lambda(Lift\ h+1\ t')$
 | $t1@t2 \Rightarrow (Lift\ h\ t1)@(Lift\ h\ t2)$
 | $\chi n \Rightarrow \mathbf{if} d \leq n \mathbf{then} \chi(n+1) \mathbf{else} \chi n$ $\chi(\mathbf{if} d \leq n \mathbf{then} (n+1) \mathbf{else} n)$
end.

Par exemple si on s'intéresse dans une preuve à $\mathcal{D}(Lift\ t)$, la profondeur d'un terme lifté, avec t une variable

- ▶ la première forme impose de raisonner par cas sur $d \leq n$
- ▶ la seconde forme permet immédiatement de conclure que le résultat est une variable

Induction

La définition d'un type inductif, en Coq, s'accompagne de la génération de principes d'induction structurelle associés.

Mais ce n'est pas adapté à tous les usages

- ▶ Lorsque des définitions inductives sont imbriquées
- ▶ Lorsqu'une induction forte est nécessaire
- ▶ Lorsque le sous-terme d'induction n'est pas un sous-terme du terme principal mais une variante
- ▶ Lorsque le sous-terme d'induction n'est pas du même type que le terme principal
- ▶ Lorsque ce n'est pas la bonne stratégie d'un point de vue sémantique

Une stratégie raisonnablement efficace et simple à mettre en œuvre : prouver la récurrence forte sur la base d'une mesure pour un type quelconque, et utiliser ce principe avec des mesures *ad hoc*

Induction

Principe de récurrence forte pour une mesure sur une famille de types dépendants quelconque

$$\forall (T : \mathbf{Type})$$

$$(D : T \rightarrow \mathbf{Type})$$

$$(M : \forall (t : T), D t \rightarrow \mathbb{N})$$

$$(Q : \forall (t : T), D t \rightarrow \mathbf{Prop})$$

$$(\forall (t : T)(d : D t), (\forall (t' : T)(d' : D t'), M(d') < M(d) \Rightarrow Q t' d')) \Rightarrow Q t d \Rightarrow$$

$$\forall (t : T)(d : D T), Q t d$$

Pour l'utiliser, on définit une mesure et une relation d'accessibilité (inductive) compatible avec la mesure et surjective dans le sous-ensemble des termes sur lesquels on veut faire la preuve

Plan

- 1 Simplicité
- 2 Lisibilité
- 3 Efficacité (dans les preuves)
- 4 **Automatisation (des preuves)**
- 5 Quel rapport avec FOCALIZE ?

Le langage Ltac

Le langage LTAC permet d'écrire des tactiques, *i.e.* des programmes utilisés pour construire des termes de preuve

- ▶ Un des grands avantages de COQ
- ▶ L'alternative est d'écrire des modules OCAML pour COQ

LTAC permet

- ▶ d'écrire des tactiques de preuve
- ▶ d'écrire des fonctions de calcul
- ▶ d'utiliser ou décrire des *tacticals* (**repeat** par exemple)

Par contre il ne peut être utilisé que dans le contexte d'une preuve

En pratique, LTAC reste peu documenté (cf. **unify** par exemple) et assez difficile à maîtriser ; l'effort reste cependant payant

$\mathcal{L}tac$ et calcul

Calcul de la longueur d'une liste en LTAC

Require Import *List*.

Ltac *LG l* :=

match *l* **with** *nil* \Rightarrow *O* | **cons** ?*h* ?*l'* \Rightarrow **let** *L* := *LG l'* **in constr** :(*S L*) **end**.

Quelques essais

Definition *l* : **list** \mathbb{N} := 2 :: 1 :: 0 :: **nil**.

Definition *s* := *LG l*. **Erreur** alors que avec **length** cela fonctionne

Theorem *test* : $\forall (l : \text{list } \mathbb{N}), \dots$

Proof.

intros *l'*; **let** *L* := *T_length l'* **in pose** *L* **Erreur**

let *L* := *T_length l* **in pose** *L* **Erreur**

let *L* := *T_length* (2 :: 1 :: 0 :: **nil**) **in pose** *L*.

Cette dernière ligne crée une hypothèse $n := 3 : \mathbb{N}$


Pattern matching en Ltac

Plusieurs formes de commandes **match** en Ltac

- ▶ Le **match** sur un terme, comme dans l'exemple précédent des listes, pour lequel les *patterns* ne sont pas forcément exhaustifs, linéaires. . . ni même des *patterns* en fait

match *T1 with T2* \Rightarrow **true end** retourne la valeur booléenne **true** en cas d'égalité, et un échec en cas d'inégalité

- ▶ Le **match** sur le but dans une preuve : lorsqu'il y a unification entre le but et un *pattern*, la tactique associée est exécutée ; si elle échoue il y a *backtracking* et poursuite sur le même *pattern*, puis sur les *patterns* suivants²

²À noter les variantes **reverse** et **lazy**, non abordées ici. 

Mappings en $\mathcal{L}tac$

Objectif : appliquer une tactique à toutes les hypothèses, puis à toutes les paires d'hypothèses

Attention, c'est plus difficile qu'il n'y paraît !

Première tentative

```
Ltac map T :=
```

```
match goal with
```

```
| [H : _] - _] ⇒ revert H; map T; intro H; T H
```

```
| _ ⇒ idtac
```

```
end.
```

Peu robuste : $T H$ peut échouer, il peut être nécessaire de changer le nom de H si T introduit de nouvelles hypothèses, etc.

Mappings en $\mathcal{L}tac$

Ltac *map* T :=

match goal with

| $[H : _] - _] \Rightarrow$ **revert** H ; *map* T ; **let** $H' :=$ **fresh** H **in intro** H' ; **try** (T H')

| $_ \Rightarrow$ **idtac**

end.

map **ltac** : (**fun** $X \Rightarrow$ **clear** X) va effacer toutes les hypothèses de la première à la dernière... sauf celles qui sont liées

Cette version de *map* est robuste à l'introduction d'hypothèses par la tactique passée en paramètre : elle ne va pas boucler ni provoquer des conflits de noms

Par contre, il ne faut pas introduire des hypothèses en tête de la conséquence ! Il y aurait perte de traçabilité entre le **revert** et le **intro**...

Mappings en $\mathcal{L}tac$

Le **vrai** problème de ce(tte) *tactical* apparaît en l'utilisant pour appliquer une tactique à toutes les paires

```
Ltac map2 T := map (fun X => map (fun Y => T X Y))
```

Si on utilise *map2* avec *idtac* tout va bien, on affiche l'ensemble des paires d'hypothèses ; il y a par contre de curieux dysfonctionnements avec d'autres tactiques, notamment avec

```
fun X Y =>
  let TX := type of X in
  let TY := type of Y in
  idtac "(" X" : " TX" , " Y" : " TY" )"
```

En effet, on montre que quand on tente d'exécuter *T H1 H2* pendant la double récursion, l'une des deux hypothèse a été "mise de côté" par un **revert**

Mappings en $\mathcal{L}tac$

Seconde tentative : parcourir les hypothèses avec un effet de bord (le **revert**) se révélant une mauvaise idée, on effectue une copie des hypothèses puis on travaille sur les copies

Code non détaillé, juste un petit sous-ensemble

```
Ltac count inb ieq :=
```

```
let rec c n := match goal with
```

```
| [H :  $\_ - \_$ ]  $\Rightarrow$  revert H; c (S n); intro H
```

```
|  $\_ \Rightarrow$  pose (inb := n); assert (ieq : inb = n); [apply refl_equal |]
```

```
end in c 0.
```

count $H1 H2$ crée deux hypothèses $H1 := n : \mathbb{N}$ et $H2 : H1 = n$, avec n le nombre d'hypothèses ; $H2$ permet de mémoriser la valeur de n même si $H1$ est généralisé

Problème : travailler sur des copies n'est pas satisfaisant si la tactique appliquée cherche notamment à effacer certaines hypothèses

Mappings en $\mathcal{L}tac$

Troisième tentative : construire la liste des identifiants des hypothèses du contexte, puis appliquer la tactique à l'ensemble de ces identifiants

Problème : dans une instruction de la forme

match goal with $[H : T \mid - _] \Rightarrow \dots$, H n'est pas un identifiant mais un terme de type T ; une liste de tels "identifiants" est mal typée

Quatrième tentative : construire la liste des couples $(T, H : T)$ correspondant à l'ensemble des hypothèses, puis appliquer la tactique au contenu de cette liste

Courage, c'est la dernière, et elle semble fonctionner !

Mappings en $\mathcal{L}tac$

Inductive context :=

| *ctx_nil*: context

| *ctx_cns*: $\forall (T : Type), T \rightarrow context \rightarrow context.$

Ltac *ctx_mem* $T t c$:=

match *c* with

| *ctx_nil* \Rightarrow **false**

| *ctx_cns* ? T' ? t' ? c' \Rightarrow

match T with

| T' \Rightarrow **match** t with t' \Rightarrow **true** | $_ \Rightarrow$ *ctx_mem* $T t c'$ **end**

| $_ \Rightarrow$ *ctx_mem* $T t c'$

end

end.

Mappings en $\mathcal{L}tac$

```

Ltac build_ctx acc :=
match goal with
| [H :?T | - _] =>
  match ctx_mem T H acc with false => build_ctx (ctx_cns T H acc) end
| _ => acc
end.

```

La ligne en rouge est un *pattern matching* non exhaustif; soit $H:T$ est ajoutée au contexte, soit une erreur provoque un *backtracking*

```

Ltac map T :=
let c := build_ctx ctx_nil in
let rec mh c :=
  match c with ctx_cns ?t' ?h' ?c' => mh c'; T h' | ctx_nil => idtac end in
  mh c.

```

Mappings en $\mathcal{L}tac$

Ltac *map2* T :=

let c := *build_ctx* ctx_nil **in**

let rec mp $c1$ $c2$:=

match $c1$ **with**

| $ctx_cns ?t1' ?h1' ?c1'$ \Rightarrow

match $c2$ **with**

| $ctx_cns ?t2' ?h2' ?c2'$ \Rightarrow

mp $c1$ $c2'$; **match** $h1'$ **with** $h2'$ \Rightarrow **idtac** | $_$ \Rightarrow T $h1'$ $h2'$ **end**

| ctx_nil \Rightarrow mp $c1'$ $c1'$

end

| ctx_nil \Rightarrow **idtac**

end in

mp c c .

Mappings en $\mathcal{L}tac$

On peut ensuite définir des notations associées aux tactiques... On suppose que l'on a un contexte $H_1:T_1 \dots H_n:T_n$

Tactic Notation "exec" **tactic**(T) "on" "hyps" := $map \mathbf{ltac}:(T)$.
 Exécute $T H_n$, etc. puis $T H_1$

Tactic Notation "exec" **tactic**(T) "on" "hyps" "except" **ident**(I) :=
 $map \mathbf{ltac}:(\mathbf{fun} X \Rightarrow \mathbf{match} X \mathbf{with} I \Rightarrow \mathbf{idtac} \mid _ \Rightarrow T X \mathbf{end})$.
 Exécute $T H_n$, etc. puis $T H_1$ sauf $T I$

Tactic Notation "exec" **tactic**(T) "on" "hyps" "in" **constr**(E) := ...
 Exécute $T H_n$ si $H_n:E$, etc. puis $T H_1$ si $H_1:E$

Tactic Notation "exec" **tactic**(T) "on" "pairs" := ...
 Exécute $T H_{n-1} H_n$, $T H_{n-2} H_n$, $T H_{n-2} H_{n-1}$, etc. i.e. T sur tous les couples (H_i, H_j) avec $i < j$

Mappings en $\mathcal{L}tac$

Tactic Notation *"remove" "duplicates"* :=
`exec(fun X Y => try(unify(type_of X)(type_of Y); first[clear Y | clear X]))`
on pairs.

Pour tous les couples $H_i : T$ $H_j : T$ du contexte, tente d'effacer H_i , et en cas échec tente d'effacer H_j

Tactic Notation *"remove" "all" "instances" "of" constr(E)* := ...
 Élimine toute les instances de E , i.e. les $H : E$, $H : E x$, $H : E x y$, etc.

Tactic Notation *"new" constr(E) "as" ident(I)* := ...
 Crée une hypothèse $I : E$ s'il n'existe aucune hypothèse $H : E$

Tactic Notation *"close" "by" "transitivity" constr(E)* := ...
 Pour un résultat de transitivité E , génère automatiquement la clôture en exploitant toutes les hypothèses disponibles

Quelques observations sur Ltac

LtAC est un langage puissant, malheureusement peu documenté ;
développer une tactique est souvent délicat

- ▶ Effets de bords
- ▶ Ordre supérieur et types dépendants
- ▶ *Pattern matching* riche
- ▶ Sémantique avec *backtracking*

Quelques regrets

- ▶ On peut manipuler des termes ou des buts, mais pas les deux
(par exemple **clear** *H*; **cons** *H L*)
- ▶ On ne peut pas parcourir les hypothèses (de manière simple)
- ▶ On ne peut pas marquer des hypothèses

Plan

- 1 Simplicité
- 2 Lisibilité
- 3 Efficacité (dans les preuves)
- 4 Automatisation (des preuves)
- 5 **Quel rapport avec FOCALIZE ?**

Petite analyse rapide

Les gadgets décrits ici sont très utiles en COQ ... Sont-ils intéressants pour FOCALIZE ?

- ▶ Paramètres implicites : justifié dans un cadre d'ordre supérieur avec types dépendants, pas très intéressant pour FOCALIZE
- ▶ Notations : très intéressants ; FOCALIZE permet déjà d'utiliser des symboles préfixes ou infixes comme noms de fonctions, mais pas de définir à la volée des notations quelconques en UTF-8, avec des valeurs inférées (cf. $S_1 \cap S_2 \equiv \text{inter_} S_1 S_2$)
- ▶ Coercitions : à voir, il faut comprendre le rapport avec la notion d'héritage et de paramétrisation ; par exemple dans certains cas une valeur d'une espèce paramètre pourrait être vue comme une valeur de l'espèce courante (identifiant de variable et variable)

Petite analyse rapide

- ▶ Logique vs calcul : déjà bien intégré dans FOCALIZE (pas de réelle distinction entre **bool** et **Prop**)
- ▶ Anonymisation : peu intéressant en FOCALIZE en raison de la méthodologie (espèces abstraites) et l'automatisation des preuves
- ▶ Induction : à creuser, une solution pragmatique pourrait être de proposer une espèce avec une notion de mesure et un schéma d'induction associé ; à l'utilisateur de proposer la mesure
- ▶ Langage de tactique : inutile en FOCALIZE – ZENON fait le travail