Proposition de syntaxe pour Coq

B. Barras

29 avril 2002

1 Grammaire des tactiques

La réflexion de la rénovation de la syntaxe des tactiques n'est pas encore aussi poussée que pour les termes (section 2), mais cette section vise à énoncer les quelques principes que l'on souhaite suivre.

- Réutiliser les mots-clés de la syntaxe des termes (i.e. en minuscules) pour les constructions similaires de tactiques (let_in, match, and, etc.). Le connecteur logique `and` n'étant que rarement utilisé autrement que sous la forme $\wedge$ (sauf dans le code ML), on pourrait dégager ce mot-clé.
- Les arguments passés aux tactiques sont principalement des termes, on préconise l'utilisation d'un symbole spécial (par exemple l'apostrophe) pour passer une tactique ou une expression (AST). L'idée étant que l'on écrit plus souvent des tactiques prenant des termes en argument que des tacticals.

1.1 Arguments de tactiques

La syntaxe actuelle des arguments de tactiques est que l'on parse par défaut une expression de tactique, ou bien l'on parse un terme si celui-ci est préfixé par ' (sauf dans le cas des variables). Cela est gênant pour les utilisateurs qui doivent écrire des ' pour leurs tactiques.

À mon avis, il n'est pas souhaitable pour l'utilisateur de l'obliger à marquer une différence entre les tactiques "primitives" (en fait "système") et les tactiques définies par Ltac. En effet, on se dirige inévitablement vers une situation où il existera des librairies de tactiques et il va devenir difficile de savoir facilement s'il faut ou non mettre des '.

1.2 Bindings

Dans un premier temps, les "bindings" sont toujours considérés comme une construction du langage des tactiques, mais il est intéressant de prévoir l'extension de ce procédé aux termes, puisqu'il s'agit simplement de construire un nœud d'application dans lequel on donne les arguments par nom ou par position, les autres restant à inférer. Le principal point est de trouver comment combiner de manière uniforme ce procédé avec les arguments implicites.

⟨*tactic*⟩	::=	⟨*tactic*⟩ & ⟨*tactic*⟩	(THEN)
	|	[⟨*tactic*⟩ | ... | ⟨*tactic*⟩]	(PAR)
	|	⟨*ident*⟩ ⟨*tactic-arg*⟩*	(APPLY)
	|	fun	(FUNCTION)
	|	⟨*simple-tactic*⟩	
⟨*tactic-arg*⟩	::=	⟨*constr*⟩	
	|	' ⟨*tactic*⟩	
⟨*simple-tactic*⟩	::=	Apply ⟨*binding-term*⟩	
	|	⟨*elim-kw*⟩ ⟨*binding-term*⟩	
	|	⟨*elim-kw*⟩ ⟨*binding-term*⟩ using ⟨*binding-term*⟩	
	|	Intros ⟨*intro-pattern*⟩	
⟨*elim-kw*⟩	::=	Elim | Case | Induction | Destruct	

FIG. 1 – Grammaire des tactiques

⟨*binding-term*⟩	::=	⟨*constr*⟩ with ⟨*binding*⟩*
⟨*binding*⟩	::=	⟨*constr*⟩

FIG. 2 – Grammaire des bindings

Il est toutefois important de réfléchir dès maintenant à une syntaxe pour éviter de rechanger encore la syntaxe.

Intégrer la notation with aux termes peut poser des problèmes puisque ce mot-clé est utilisé pour le filtrage : comment parser (en LL(1)) l'expression :

```
Cases x with y ...
```

Soit on trouve un autre mot-clé, soit on joue avec les niveaus de priorité en obligeant a parenthéser le with des "bindings" :

```
Cases (x with y) with (C z) => ...
```

ce qui introduit un constructeur moralement équivalent à une application situé à une priorité totalement différente (les "bindings" seraient au plus haut niveau alors que l'application est à un niveau bas).

1.3 Enregistrements

Il faudrait aménager la syntaxe des enregistrements dans l'optique d'avoir des enregistrements anonymes (termes de première classe), même si pour l'instant, on ne dispose que d'enregistrements définis a toplevel.

Exemple de syntaxe pour les types d'enregistrements :

```
{ x1 : A1;
  x2 : A2(x1);
  _ : T;    (* Pas de projection disponible *)
  y;        (* Type infere *)
  ...       (* ; optionnel pour le dernier champ *)
}
```

Exemple de syntaxe pour le constructeur :

```
{ x1 = O;
  x2 : A2(x1) = v1;
  _ = v2;
  ...
}
```

Quant aux dépendences, une convention pourrait être de considérer les champs non annotés par le type comme non dépendants.

Plusieurs interrogations :

- l'ordre des champs doit-il être respecté ? sinon, que faire pour les champs sans projection ?
- autorise-t-on `v1` a mentionner `x1` (comme dans la définition d'un module), ce qui se comporterait comme si on avait écrit `v1` à la place. Cela pourrait être une autre manière de déclarer les dépendences

La notation pointée pour les projections pose un problème de parsing, sauf si l'on a une convention lexicale qui discrimine les noms de modules des projections et identificateurs : `x.y.z` peut être compris comme `(x.y).z` ou texttt x.(y.z).

2 Grammaire des termes

2.1 Quelques principes

1. Diminuer le nombre de niveaux de priorité en regroupant les règles qui se ressemblent : infixes, préfixes, lieurs (constructions ouvertes à droite), etc.
2. Éviter de surcharger la signification d'un symbole (ex : `( )` comme parenthésage et produit dans la V7).
3. Faire en sorte que les membres gauches (motifs de Cases, lieurs d'abstraction ou de produits) utilisent une syntaxe compatible avec celle des membres droits (branches de Cases et corps de fonction).

$$
\begin{array}{rcll}
\langle \textit{paren-constr} \rangle & ::= & \langle \textit{cast-constr} \rangle \;,\; \langle \textit{paren-constr} \rangle & (\textsc{pair}) \\
 & | & \langle \textit{cast-constr} \rangle & \\
 & & & \\
\langle \textit{cast-constr} \rangle & ::= & \langle \textit{constr} \rangle \;:\; \langle \textit{cast-constr} \rangle & (\textsc{cast}) \\
 & | & \langle \textit{constr} \rangle & \\
 & & & \\
\langle \textit{constr} \rangle & ::= & \langle \textit{appl-constr} \rangle \; \langle \textit{infix} \rangle \; \langle \textit{constr} \rangle & (\textsc{infix}) \\
 & | & \langle \textit{prefix} \rangle \; \langle \textit{constr} \rangle & (\textsc{prefix}) \\
 & | & \langle \textit{constr} \rangle \; \langle \textit{postfix} \rangle & (\textsc{postfix}) \\
 & | & \langle \textit{appl-constr} \rangle & \\
 & & & \\
\langle \textit{appl-constr} \rangle & ::= & \langle \textit{appl-constr} \rangle \; \langle \textit{appl-arg} \rangle + & (\textsc{apply}) \\
 & | & \texttt{@} \; \langle \textit{global} \rangle \; \langle \textit{simple-constr} \rangle + & (\textsc{expl-apply}) \\
 & | & \langle \textit{simple-constr} \rangle & \\
 & & & \\
\langle \textit{appl-arg} \rangle & ::= & \texttt{@} \; \langle \textit{int} \rangle \; := \; \langle \textit{simple-constr} \rangle & (\textsc{impl-arg}) \\
 & | & \langle \textit{simple-constr} \rangle & \\
 & & & \\
\langle \textit{simple-constr} \rangle & ::= & \langle \textit{atomic-constr} \rangle & \\
 & | & (\; \langle \textit{paren-constr} \rangle \;) & \\
 & | & \langle \textit{match-constr} \rangle & \\
 & | & \langle \textit{fix-constr} \rangle &
\end{array}
$$

Fig. 3 – Grammaire des termes

$$
\begin{array}{rcll}
\langle \textit{prefix} \rangle & ::= & \mathsf{!}\ \langle \textit{binder} \rangle + \ . & (\text{PROD}) \\
& | & \mathsf{fun}\ \langle \textit{binder} \rangle + \ \Rightarrow & (\text{LAMBDA}) \\
& | & \mathsf{let}\ \langle \textit{ident} \rangle\ \langle \textit{binder} \rangle * \ = \langle \textit{constr} \rangle\ \mathsf{in} & (\text{LET}) \\
& | & \mathsf{if}\ \langle \textit{constr} \rangle\ \mathsf{then}\ \langle \textit{constr} \rangle\ \mathsf{else} & (\text{IF-CASE}) \\
& | & \mathsf{eval}\ \langle \textit{red-fun} \rangle\ \mathsf{in} & (\text{EVAL}) \\
\\
\langle \textit{infix} \rangle & ::= & \rightarrow & (\text{IMPL}) \\
\\
\langle \textit{atomic-constr} \rangle & ::= & \_ & \\
& | & ?\langle \textit{int} \rangle & \\
& | & \langle \textit{sort} \rangle & \\
& | & \langle \textit{global} \rangle & \\
\\
\langle \textit{binder} \rangle & ::= & \langle \textit{ident} \rangle & (\text{INFER}) \\
& | & (\ \langle \textit{ident} \rangle\ \langle \textit{type} \rangle\ ) & (\text{BINDER}) \\
\\
\langle \textit{type} \rangle & ::= & :\ \langle \textit{constr} \rangle & \\
& | & \epsilon &
\end{array}
$$

FIG. 4 – Grammaires annexes aux termes

2.2 Présentation de la grammaire

La grammaire des termes (correspondant à l'état `barestate`) est décrite figures 3 et 4. On constate par rapport aux précédentes versions de Coq d'importants changements de priorité, le plus marquant étant celui de l'application qui se trouve désormais juste au dessus[1] des constructions fermées à gauche et à droite.

La grammaire des noms globaux est la suivante :

$$
\begin{array}{rcl}
\langle \textit{global} \rangle & ::= & \langle \textit{ident} \rangle \\
& | & \langle \textit{ident} \rangle.\langle \textit{global} \rangle
\end{array}
$$

Le _ dénote les termes à synthétiser. Les métavariables sont reconnues au niveau du lexer pour ne pas entrer en conflit avec le ? de l'existentielle.

Les opérateurs infixes ou préfixes sont tous au même niveau de priorité du point de vue de Camlp4. La solution envisagée est de les gérer à la manière de Yacc, avec une pile (voir discussions plus bas). Ainsi, l'implication est un infixe normal ; la quantification universelle et le let sont vus comme des opérateurs préfixes avec un niveau de priorité plus haut (i.e. moins lieur). Il subsiste des problèmes si l'on ne veut pas écrire de parenthèses dans :

[1] La convention est de considérer les opérateurs moins lieurs comme "au dessus", c'est-à-dire ayant un niveau de priorité plus élevé (comme c'est le cas avec le niveau de la grammaire actuelle des termes).

```
A -> (!x. B -> (let y = C in D))
```

La solution proposée est d'analyser le membre droit d'un infixe de manière à autoriser les préfixes et les infixes de niveau inférieur, et d'exiger le parenthésage que pour les infixes de niveau supérieurs.

En revanche, à l'affichage, certains membres droits seront plus lisibles s'ils n'utilisent pas cette astuce :

```
(fun x => x) = fun x => x
```

La proposition est d'autoriser ce type d'écritures au parsing, mais l'afficheur écrit de manière standardisée en mettant quelques parenthèses superflues : $=$ serait symétrique alors que $\rightarrow$ appellerait l'afficheur de priorité élevée pour son sous-terme droit.

Les priorités des opérateurs primitifs sont les suivantes (le signe $*$ signifie que pour le membre droit les opérateurs préfixes seront affichés sans parenthèses quel que soit leur priorité) :

symbole	priorité
!	$200\, R*$
fun	$200\, R*$
let	$200\, R*$
if	$200\, R$
eval	$200\, R$
$\rightarrow$	$90\, R*$

Il y a deux points d'entrée pour les termes : $\langle$*constr*$\rangle$ et $\langle$*simple-constr*$\rangle$. Le premier peut être utilisé lorsqu'il est suivi d'un séparateur particulier. Dans le cas où l'on veut une liste de termes séparés par un espace, il faut lire des $\langle$*simple-constr*$\rangle$.

Les constructions fix et cofix (voir aussi figure 5) sont fermées par end pour simplifier l'analyse. Sinon, une expression de point fixe peut être suivie par un in ou un and, ce qui pose les mêmes problèmes que le "dangling else" : dans

```
fix f1 x {x} = fix f2 y {y} = ... and ... in ...
```

il faut définir une stratégie pour associer le and et le in au bon point fixe.

Un autre avantage est de faire apparaitre que le fix est un constructeur de terme de première classe et pas un lieur :

```
fix f1 ... and f2 ...
in f1 end x
```

Les propositions précédentes laissaient `f1` et `x` accolés, ce qui est source de confusion lorsque l'on fait par exemple `Pattern (f1 x)`.

Les corps de points fixes et co-points fixes sont identiques, bien que ces derniers n'aient pas d'information de décroissance. Cela fonctionne puisque l'annotation est optionnelle. Cela préfigure des cas où l'on arrive à inférer quel est

$$
\begin{array}{rcll}
\langle \textit{fix-expr} \rangle & ::= & \textsf{fix}\ \langle \textit{fix-decls} \rangle\ \langle \textit{fix-select} \rangle\ \textsf{end} & (\textsc{fix}) \\
 & | & \textsf{cofix}\ \langle \textit{cofix-decls} \rangle\ \langle \textit{fix-select} \rangle\ \textsf{end} & (\textsc{cofix}) \\
 & & & \\
\langle \textit{fix-decls} \rangle & ::= & \langle \textit{fix-decl} \rangle\ \textsf{and}\ \langle \textit{fix-decls} \rangle & \\
 & | & \langle \textit{fix-decl} \rangle & \\
 & & & \\
\langle \textit{fix-decl} \rangle & ::= & \langle \textit{ident} \rangle\ \langle \textit{binder} \rangle +\ \ \langle \textit{type} \rangle\ \langle \textit{annot} \rangle\ =\ \langle \textit{constr} \rangle & \\
 & & & \\
\langle \textit{annot} \rangle & ::= & \textsf{\{}\ \langle \textit{ident} \rangle\ \textsf{\}} & \\
 & | & \epsilon & \\
 & & & \\
\langle \textit{fix-select} \rangle & ::= & \textsf{in}\ \langle \textit{ident} \rangle & \\
 & | & \epsilon &
\end{array}
$$

Fig. 5 – Grammaires annexes des points fixes

l'argument qui décroit structurellement (en particulier dans le cas où il n'y a qu'un seul argument).

La construction Case peut-être considérée comme obsolète. Quant au Match de la V6, il disparaît purement et simplement.

De manière globale, l'introduction de définitions dans les termes se fait avec le symbole =, et le := est réservé aux définitions au niveau vernac. Il y avait un manque de cohérence dans la V6, puisque l'on utilisait = pour le let et := pour les points fixes et les commandes vernac.

2.3 Infixes

2.3.1 Infixes extensibles

Le problème de savoir si la liste des symboles pouvant apparaître en infixe est fixée ou extensible par l'utilisateur reste à voir.

Notons que la solution où les symboles infixes sont des identificateurs que l'on peut définir paraît difficilement praticable : par exemple `Logic.eq` n'est pas un opérateur binaire, mais ternaire. Il semble plus simple de garder des déclarations infixes qui relient un symbole infixe à un terme avec deux "trous". Par exemple :

infixe	identificateur
=	`Logic.eq _ ?1 ?2`
==	`JohnMajor.eq _ ?1 _ ?2`

La syntaxe d'une déclaration d'infixe serait par exemple :

$$
\begin{array}{rcll}
\langle \mathit{match\text{-}expr} \rangle & ::= & \textsf{match}\ \langle \mathit{case\text{-}items} \rangle\ \langle \mathit{case\text{-}type} \rangle\ \textsf{with}\ \langle \mathit{branches} \rangle\ \textsf{end} & (\textsc{match}) \\
 & | & \textsf{match}\ \langle \mathit{case\text{-}items} \rangle\ \textsf{with}\ \langle \mathit{branches} \rangle\ \textsf{end} & (\textsc{infer-match}) \\
\\
\langle \mathit{case\text{-}items} \rangle & ::= & \langle \mathit{case\text{-}item} \rangle\ \&\ \langle \mathit{case\text{-}items} \rangle & \\
 & | & \langle \mathit{case\text{-}item} \rangle & \\
\\
\langle \mathit{case\text{-}item} \rangle & ::= & \langle \mathit{constr} \rangle\ \langle \mathit{pred\text{-}pattern} \rangle & (\textsc{dep-case}) \\
 & | & \langle \mathit{constr} \rangle & (\textsc{nodep-case}) \\
\\
\langle \mathit{case\text{-}type} \rangle & ::= & \Rightarrow\ \langle \mathit{constr} \rangle & \\
 & | & \epsilon & \\
\\
\langle \mathit{pred\text{-}pattern} \rangle & ::= & \textsf{as}\ \langle \mathit{ident} \rangle\ :\ \langle \mathit{constr} \rangle & \\
\\
\langle \mathit{branches} \rangle & ::= & |\ \langle \mathit{patterns} \rangle \Rightarrow \langle \mathit{constr} \rangle\ \langle \mathit{branches} \rangle & \\
 & | & \epsilon & \\
\\
\langle \mathit{patterns} \rangle & ::= & \langle \mathit{pattern} \rangle\ \&\ \langle \mathit{patterns} \rangle & \\
 & | & \langle \mathit{pattern} \rangle & \\
\\
\langle \mathit{pattern} \rangle & ::= & \ldots &
\end{array}
$$

FIG. 6 – Grammaires annexes du filtrage

```
Infix "=" 50 := Logic.eq _ ?1 ?2;
```

2.3.2 Gestion des précédences

Les infixes peuvent être soit laissé à Camlp4, ou bien (comme ici) considérer que tous les opérateurs ont la même précédence et gérer soit même la recomposition des termes à l'aide d'une pile (comme Yacc).

2.4 Extensions de syntaxe

2.4.1 Litéraux numériques

La proposition est de considerer les litéraux numériques comme de simples identificateurs. Comme il en existe une infinité, il faut un nouveau mécanisme pour leur associer une définition. Par exemple, en ce qui concerne `Arith`, la définition de 5 serait `S` 4. Pour `ZArith`, 5 serait `xI` 2.

Comme les infixes, les constantes numériques peuvent être qualifiées pour indiquer dans quels module est le type que l'on veut référencer. Par exemple (si on renomme `Arith` en `N` et `ZArith` en `Z`) : `N.5`, `Z.5`.

$$
\begin{array}{lcl}
\langle global \rangle & ::= & \ldots \\
 & | & \langle int \rangle
\end{array}
$$

2.4.2 Nouveaux lieurs

$$
\begin{array}{lcll}
\langle constr \rangle & ::= & \ldots & \\
 & | & \mathsf{ex}\ \langle binder \rangle+\ .\ \langle constr \rangle & (\textsc{ex}) \\
 & | & \mathsf{ex}\ \langle binder \rangle+\ .\ \langle constr \rangle\ ,\ \langle constr \rangle & (\textsc{ex}2) \\
 & | & \mathsf{ext}\ \langle binder \rangle+\ .\ \langle constr \rangle & (\textsc{ex}\mathrm{T}) \\
 & | & \mathsf{ext}\ \langle binder \rangle+\ .\ \langle constr \rangle\ ,\ \langle constr \rangle & (\textsc{ex}\mathrm{T}2)
\end{array}
$$

Pour l'instant l'existentielle n'admet qu'une seule variable, ce qui oblige à écrire des cascades de ex.

Pour parser les existentielles avec deux prédicats, on peut considérer & comme un infixe intermédiaire et l'opérateur existentiel en présence de cet infixe se transforme en `ex2`.

2.4.3 Nouveaux infixes

Précédences des opérateurs infixes (les plus grands associent moins fort) :

identificateur	module	infixe/préfixe	précédence
`iff`	Logic	$\longleftrightarrow$	100
`or`	Logic	$\vee$	80 R
`sum`	Datatypes	$+$	80 R
`and`	Logic	$\wedge$	70 R
`prod`	Datatypes	$*$	70 R
`not`	Logic	˜	60 L
`eq _`	Logic	$=$	50
`eqT _`	Logic_Type	$=$	50
`identityT _`	Data_Type	$=$	50
`plus`	Peano	$+$	40 L
`Zplus`	fast_integer	$+$	40 L
`minus`	Minus	$-$	40 L
`Zminus`	zarith_aux	$-$	40 L
`Zopp`	fast_integer	$-$	40 L
`mult`	Peano	$*$	30 L
`Zmult`	fast_integer	$*$	30 L

Notons qu'il faudrait découper `Logic_Type` en deux car celui-ci définit deux égalités, ou alors les mettre dans des modules différents.

2.5 Exemples

```
Definition not (A:Prop) := A->False;
Inductive eq (A:Set) (x:A) : A->Prop :=
    refl_equal : eq A x x;
Inductive ex (A:Set) (P:A->Prop) : Prop :=
    ex_intro : !x. P x -> ex A P;
Lemma not_all_ex_not : !(P:U->Prop). ~(!n. P n) -> ?n. ~ P n;
Fixpoint plus n m : nat {struct n} :=
  match n with
    O     => m
  | (S k) => S (plus k m)
  end;
```

2.6 Questions ouvertes

Voici les points sur lesquels la discussion est particulièrement ouverte :

- choix d'autres symboles pour les quantificateurs ! et ?. En l'état actuel des discussions, on garderait le ! pour la qunatification universelle, mais on choisirait quelquechose comme `ex` pour l'existentielle, afin de ne pas suggérer trop de symétrie entre ces quantificateurs (l'un est primitif, l'autre pas).

- syntaxe particulière pour les `sig`, `sumor`, etc.
- la possibilité d'introduire plusieurs variables du même type est pour l'instant supprimée au vu des problèmes de compatibilité de syntaxe entre les membres gauches et membres droits. L'idée étant que l'inference de type permet d'éviter le besoin de déclarer tous les types.

2.7 Autres extensions

2.7.1 Lieur multiple

L'écriture de types en présence de polymorphisme est souvent assez pénible :

```
Check !(A:Set) (x:A) (B:Set) (y:B). P A x B y;
```

On pourrait avoir des déclarations introduisant à la fois un type d'une certaine sorte et une variable de ce type :

```
Check !(x:A:Set) (y:B:Set). P A x B y;
```

Noter que l'on aurait pu écrire :

```
Check !A x B y. P A (x:A:Set) B (y:B:Set);
```

3 Syntaxe des tactiques

3.1 Questions diverses

Changer "Pattern nl c ... nl c" en "Pattern [nl] c ... [nl] c" pour permettre des chiffres seuls dans la catégorie syntaxique des termes.

Par uniformité remplacer "Unfold nl c" par "Unfold [nl] c" ?

Même problème pour l'entier de Specialize (ou virer Specialize ?) ?

3.2 Questions en suspens

`EAuto` : deux syntaxes différentes pour la recherche en largeur et en profondeur ? Quelle recherche par défaut ?

Remarques pêle-mêle (HH)

Renommer Variable/Hypothesis hors section en Parameter/Axiom.

Renommer les `command0`, `command1`, ... `lcommand` etc en `constr0`, `constr1`, ... `lconstr`.

Autoriser la syntaxe

```
Variable R (a : A) (b : B) : Prop.
Hypotheses H (a : A) (b : B) : Prop; Y (u : U) : V.
Variables H (a : A) (b : B), J (k : K) : nat; Z (v : V) : Set.
```

Renommer eqT, refl_eqT, eqT_ind, eqT_rect, eqT_rec en eq, refl_equal, etc. Remplacer == en =.

Enlever le Export de Bool, Arith et ZARith de Ring quand inapproprié et l'ajouter à côté des Require Ring.

Remplacer les noms Coq.omega.Omega par Coq.Omega ...

Remplacer AddPath par Add LoadPath (ou + court)

Remplacer Save. par Qed.

Remplacer Implicit Arguments On/Off par Set/Unset Implicit Arguments.