Notions de dépendances dans le système Coq

Olivier Pons

INRIA Sophia Antipolis

Réunion du Goupe Bip
Mardi 3 Mars 1998
Paris

Plan

- Notions générales sur les problèmes de dépendances.
- Schéma de fonctionnement du prouveur Coq
- L'idée Intuitive de calcul des dépendances
  - ∗ Dépendances entre théorèmes dans Coq
  - ∗ Idée du calcul sur un exemple
- Insuffisance de ce calcul des dépendances
  1. Normalisation et perte d'information
  2. Automatisme et "boite noire"
- Comment remédier à ces insuffisances
- Applications possibles
- Outils pour la propagation des modifications.
  - ∗ Le problème simple avec un algorithme Solution.
  - ∗ La version optimisée de l'algorithme.
- Bilan et perspectives

Introduction

- Maintenir la cohérence lors de modification sur un ou plusieurs composants.
- Au niveau des fichiers:
  Désormais archi-classique:
  - Make
  - Makedepend
  - Camldep,Coqdep
- A un niveau plus précis (fonction, théorème...)
  Beaucoup moins courant:
  - Sml-make

Schéma de fonctionnement du prouveur Coq

- **Principe**
  - ∗ Des objets
  - ∗ Des fonctions
  - ∗ Des énoncés et des preuves de théorèmes sur ces objets et ces fonctions.
  - ∗ Un lambda calcul d'ordre supérieur :
    le Calcul des Constructions Inductives.
  - ∗ Des tactiques :
    - · Raffinement
    - · Script de preuve
    - · Des tacticales

Exemples de développement : Les définitions

On importe la théorie `Le` dans laquelle est définie la notion d'ordre sur les entiers.

```
Require Le.
```

Quelques paramètres globaux

```
Parameter List_Dom:Set.
Local A := List_Dom.
```

Définition des listes

```
Inductive list : Set := nil : list | cons : A -> list -> list.
```

Définition de la fonction de concatenation:

```
Fixpoint app [l:list] : list -> list
      := [m:list]<list>Cases l of
                           nil =>  m
                       | (cons a l1) => (cons a (app l1 m))
                   end.
```

Exemples de développement : Les théoremes

```
Lemma app_ass : (l,m,n : list)(app (app l m) n)=(app l (app m n)).

Intros l m n ; Elim l.
(*
      2 subgoals

   l : list
   m : list
   n : list
   ============================
    (app (app nil m) n)=(app nil (app m n))

subgoal 2 is:
  (a:A)
   (l:list)
    (app (app l m) n)=(app l (app m n))
     ->(app (app (cons a l) m) n)=(app (cons a l) (app m n))
*)
```

Exemples de développement : Les théorèmes (suite)

```
Simpl;Trivial.
Simpl;Auto.
(*
1 subgoal

  l : list
  m : list
  n : list
  ============================
    (a:A)
     (l:list)
     (app (app l m) n)=(app l (app m n))
       ->(cons a (app (app l m) n))=(cons a (app l (app m n)))
*)

   Intros a l1 H;Rewrite <-H; Auto.
Qed.
```

Exemples de développement : Les théorèmes (suite)

- Ajout d'un théorème à la liste de `Hint`

```
Hint app_ass.
```

- L'associativite à gauche

```
Lemma ass_app : (l,m,n : list)(app l (app m n))=(app (app l m) n).
Proof.
        Auto.
Qed.
```

La représentation interne des preuves

Des termes du Calcul des Constructions Inductives.

```
 app_ass=
[l,m,n:list]
 (list_ind [l0:list](app (app l0 m) n)=(app l0 (app m n))
   (refl_equal list (app m n))
   [a:A]
    [l0:list]
     [H:(app (app l0 m) n)=(app l0 (app m n))]
      (eq_ind list (app (app l0 m) n)
        [l1:list](cons a (app (app l0 m) n))=(cons a l1)
        (refl_equal list (cons a (app (app l0 m) n)))
        (app l0 (app m n)) H) l)
     : (l,m,n:list)(app (app l m) n)=(app l (app m n))


ass_app =
[l,m,n:list]
 (sym_equal list (app (app l m) n) (app l (app m n)) (app_ass l m n))
     : (l,m,n:list)(app l (app m n))=(app (app l m) n)
```

Remarque sur la représentation interne

- On peut utiliser explicitement ces termes de preuve dans le script:

```
Lemma ass_app' : (l,m,n : list)(app l (app m n))=(app (app l m) n).
Proof
[l,m,n:list]
(sym_equal list (app (app l m) n) (app l (app m n)) (app_ass l m n)).
```

- Les termes sont stockés sous leur forme $\beta\iota$-normale.
  La réduction a lieu au moment du `Save`. Les fichiers compilés (.vo) ne contiennent que des termes normaux

Schéma de fonctionnement du prouveur Coq :Conclusion

Remarque

- Allier les avantages de la théorie des types (objets preuves) et des procédures de preuve (tactiques).
- Point de vue hybride (Objet preuve/Script de preuve) $\Longrightarrow$ contraintes nouvelles pour certains problèmes.

L'idée directrice

- Maintenir la cohérence des sources
- Modification minimum des sources

Idée Intuitive des dépendances

Exemple de L'associativité à gauche

- Coté Script

```
Lemma ass_app : (l,m,n : list)(app l (app m n))=(app (app l m) n).
Proof.
Intros.
Apply sym_equal.
Auto.
Save.
```

- Coté Theorie des type.

```
ass_app =
[l,m,n:list]
(sym_equal list (app (app l m) n) (app l (app m n)) (app_ass l m n))
     : (l,m,n:list)(app l (app m n))=(app (app l m) n)
```

Exemple Suite

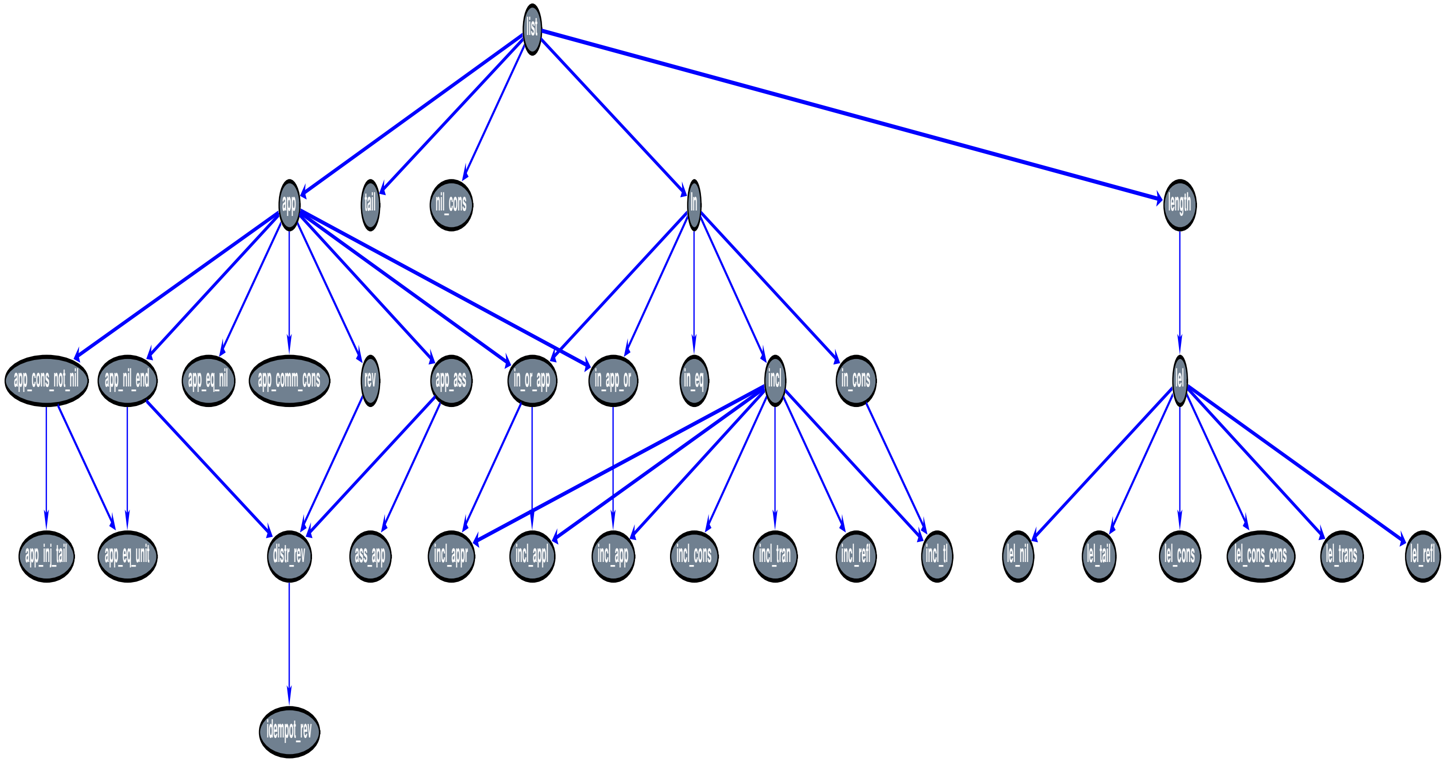
Identificateurs apparaissant dans le terme:
$\Longrightarrow$ nécessaires à la construction de la preuve.

- `sym_equal`,(symétrie de l'égalité)
- `list`, (définition inductive des listes)
- `app`, (function de concatenation des listes)
- app_ass, (associativité à droite de la concaténation).

Quelques questions

- L'information contenue dans le terme est elle un sur-ensemble de celle contenue dans le script ?
- Est-ce un calcul de dépendances suffisant pour une maintenance efficace des scripts ?

Exemple de graphe de dependance



Insuffisance de cette notion de dépendance

- Problème liés à la normalisation des termes de preuve
- Problème liés aux tactiques Automatiques

Exemple 1

Soit un lemme trivial sur les entiers naturels:

```
Lemma L11 :(x:nat)(x=O)->(pred (S x))=O.
Auto.
Save.
```

Utilisation de ce lemme dans un autre:

```
Lemma L12 :(A:Prop)A->A.
LApply (L11 O).
Auto.
Auto.
```

- Le terme de preuve avant normalisation

```
toto < Show Proof.
LOC:
Subgoals
Proof:
  ([_:(pred (S O))=O][A:Prop][H0:A]H0 (L11 O (refl_equal nat O)))
```

- Le λ-terme réduit:

```
Coq < Print toto.
toto = [A:Prop][H0:A]H0
     : (A:Prop)A->A
```

- La constante `L11` n'y apparaît plus

Exemple 2

Des théoremes qui n'apparaissent **ni dans le script ni dans le terme normalisé** mais **sont utilisés dans la preuve**.

- Encore un lemme trivial sur les entiers

```
Lemma L21:(x,y:nat)(plus x y)=(plus y x).
Intros n m ; Elim n ; Simpl ; Auto.
Intros y H ; Elim (plus_n_Sm m y) ; Auto.
Save.
Hint L21.
```

- Dans un autre lemme trivial, on fait un `Cut`, (qui s'avérera inutile)

```
Lemma L22 :(A:Prop)A->A.
Cut (x,y:nat)(plus x y)=(plus y x).
Auto.
Auto.
```

Exemple 2 (suite)

- Le sous but introduit par `Cut` est résolu par `Auto` utilisant le lemme précédent:

```
Proof: ([_:(x,y:nat)(plus x y)=(plus y x)][A:Prop][H0:A]H0
         [x,y:nat](L21 x y))
```

- Mais il n'est pas utile pour la preuve:

```
L22 = [A:Prop][H0:A]H0
     : (A:Prop)A->A
```

- `L21` n'est pas réellement utile à la preuve
- Le script de preuve du lemme `L22` ne pourra pas être rejoué si le lemme `L21` n'est pas présent dans la liste de `Hint`.

Exemple 3

Les "`Auto`" sont dépendants de l'environnement et du contexte:

- Enviromement de base:
  $plus\_assoc\_l,\ plus\_sym \in Environement$

  ```
  Lemma mon_lemme  : (n,m,p:nat)
  (plus n O)=n/\
  ((plus n (plus m p))=(plus m (plus n p))).

  Split;Auto;Rewrite (plus_assoc_l m n p).
   Rewrite (plus_sym m n); Auto.
  ```

- Ajout du lemme suivant à l'environement et à la liste des Hint:

  ```
  Lemma generalized_permute :
  (n,m,p:nat)
  (f:nat->nat->nat)
  (f_assoc_l :(n,m,p:nat) (f n (f m p))=(f (f n m) p))
  (f_sym :(n,m:nat)(f n m)=(f m n))
  ((f n (f m p))=(f m (f n p))).
  ```

- Le script de `mon_lemme` ne peut etre rejoué !

Exemple 3 (conclusion)

- `Auto` peut devenir plus "puissant" dans un environnement étendu.
- Cela peut compromettre la validité du reste du script

Quelques questions

- Quand le contexte augmente les fragments de preuve restant forment ils un sous ensembles des fragments de preuves précédent ?
- Est-ce que l'ancien script contient suffisamment d'information pour finir la preuve ?

Contre Exemple

```
Inductive A: Prop -> Prop -> Prop ->Prop :=
   A_intro: (a, b, c:Prop) a -> b -> c ->(A a b c).

Inductive B: Prop -> Prop -> Prop -> Prop ->Prop :=
   B_intro: (a, b, c, d:Prop) a -> b -> c -> d ->(B a b c d).

Parameters H1, H2, H3, H4:Prop.
Parameter th1:H1.    Hint th1.
Parameter th:(B H1 H2 H3 /\ H3 H3 /\ H3).
Parameters th2:H2; th4:H4.

Goal (A (B H1 H2 H3 /\ H3 H3 /\ H3) (H4 /\ H4) /\ (H4 /\ H4) True).
Split; Auto;
 (Split ; Auto; Apply [x, y:Prop] [h:x /\ y]h) Orelse Apply th.
        (* I *)       (* reste deux sous-buts: H4/\H4 *)
Split ; Apply th4.
Split ; Apply th4.
Qed.
```

Si on Hint `th2` et `th4` en `(* I *)` les deux sous-buts sont `H3/\H3`.

Solutions

1. Introduire différents types de dépendances
   - Dépendances Historiques (pour les Hints)
   - Dépendance Logiques
     - ∗ Dépendances Opaques (dépendance du type seul)
     - ∗ Dépendances Transparentes (dépendance du terme)
2. "Mixer ces types de dépendances "

Les "bons" calculs de dépendances pour la maintenance de source

- Calculer les dépendances logiques sur le terme non normalisé.
- Restreindre la liste des `Hint` utilisés par `Auto` et les tactiques associées. $Liste\_reduite =$

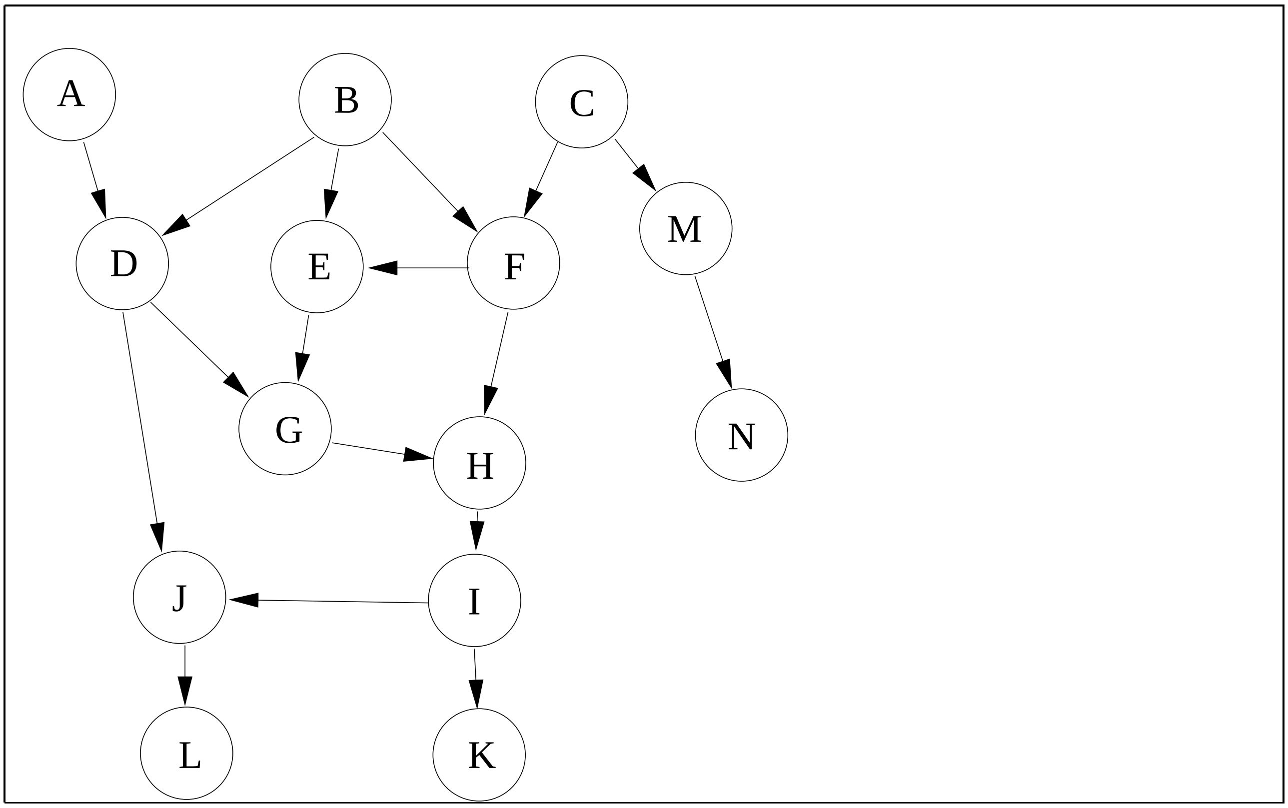
$$Liste\_de\_Hint \cap identificateurs\_dans\_le\_terme$$

  Les `Auto` deviennent des `Auto with Liste_reduite`.

Applications

- Théorie minimale
- Déplacement de Code
- Propagation des conséquences d'une modifications (changement d'axiomatique, changement de preuve, modification de terme fonctionel etc ...)

Propagation de modifications



Propagation de modifications

- Un algorithme de propagation interactif plus qu'un simple tri topologique
- A chaque étape un choix possible pour l'utilisateur
- Chaque noeud est modifié au plus une seule fois
- Tout noeud modifié ne l'est qu'après ceux dont il dépend
- Seuls les noeuds qui dépendent d'un noeud modifié sont considérés

Tri topologique

- C M N B F E A D G I J K L
- A B C D F E G H I K L M N

Bilan

- Réalisation d' outils d'aide aux developpeurs:
  - ∗ Visualiser les dépendances
  - ∗ Maintenir du code
- Des problèmes restent ouverts
  - ∗ Autres difficutés lies à un contexte:(Definition syntaxique, Coercion implicite)

Perspectives

- Développement des prototypes existants.
- Leur implementation en Ctcoq
- Liaison avec les outils de manipulation et de transformation de théorèmes
- Autres outils basés sur ces notions de dépendances. Par exemple des outils de métrique pour évaluer un développement ...