Autour de la clôture de congruence avec Coq

Stage de DEA réalisé au LSV (ENS de Cachan),

sous la direction de Jean Goubault-Larrecq

Pierre Corbineau

mars-septembre 2001

$$a=f(a)$$

$$g(a,b)=f(g(b,a))$$

$$g(b,f(a))=f(f(a))$$

$$g(a,b)=a\ ?$$

Remerciements

Mes remerciements vont en premier lieu à mon directeur de stage Jean Goubault-Larrecq pour son soutien, sa disponibilité et ses conseils. Ils vont ensuite à Jean-Christophe Filliâtre sans qui mon sujet de DEA n'existerait pas. Enfin ils vont à tous le personnel du LSV, chercheurs, thésards et personnels administratifs, qui m'ont accompagné pendant ces quelques mois.

Introduction

0.1 Sujet du stage

Le sujet du stage s'inscrit dans un projet plus ambitieux que que Jean-Christophe Filliâtre et Jean Goubault-Larrecq ont proposé : il s'agit de prouver l'algorithme de Shostak en Coq, et en particulier sa terminaison et sa complétude. Pour me familiariser avec les procédures de décision et les méthodes de réflexion, la clôture de congruence apparaissait naturellement comme un jalon intermédiaire sur cette route, et c'est ainsi que la formalisation en Coq de cet algorithme devint l'objectif de ce stage.

Ce rapport commence par quelques rappels concernant l'assistant à la preuve Coq. Ensuite il décrit théoriquement, puis concrètement l'algorithme de clôture de congruence et son implantation en Coq. Finalement, il expose les méthodes employées pour arriver à l'utilisation de la réflexion totale, puis partielle avec cet algorithme.

A l'occasion de ce dernier résultat, on présente la tactique `CC` qui résoud dans Coq les problèmes de clôture de congruence.

0.2 Équipe d'accueil

Le LSV (Laboratoire de Spécification et Vérification) est une structure récente : créé en octobre 1997 en réponse à un besoin de la part de l'ENS-Cachan, il devient une unité CNRS en janvier 1997. Ses thèmes de recherche gravitent autour des outils théoriques permettant d'augmenter la confiance de l'utilisateur dans les logiciels et matériels informatiques critiques.

Parmi toutes ces méthodes on peu citer :

- les méthodes d'abstraction
- les automates et réseaux de pétri
- les systèmes réactifs, temps réel et hybrides
- le *model-checking* et les logiques temporelles
- les méthodes de démonstration automatique
- les spécifications axiomatiques
- les résolutions de contraintes
- la vérification de protocoles cryptographiques

Le sujet sur lequel j'ai travaillé s'inscrit dans le thème de la démonstration automatique, et se rapproche notamment des travaux de Kumar Verma sur le *model-checking* avec BDD et de Xavier Rival sur les automates d'arbres.

Le LSV est formé d'une seule équipe comprenant dix membres permanents et autant de doctorants, plus six membres associés, c'est donc une petite structure ou il est facile de connaître tout le monde. La vie au laboratoire a été rythmée par des exposés hebdomadaires de membres invités ou extérieurs.

Mon intégration au laboratoire a été concrétisée par ma participation au séminaire annuel les 6 et 7 juin à Barbizon. Ce séminaire a été pour moi l'occasion de découvrir, au dela des problèmes purement théoriques et des thèmes de recherche actuels du LSV, les contraintes qu'impose la vie pratique sur une unité de recherche.

Une autre étape importante dans mon stage a été la participation au groupe de travail Coq qui s'est tenu au LRI le 5 juillet. Cette journée de travail a été pour moi l'occasion de rencontrer les équipes Coq du LRI et de l'INRIA Rocquencourt, et notamment Jean-Christophe Filliâtre, avec qui j'ai pu avoir des échanges fructueux. Cette réunion m'a permis de remettre en cause des choix d'implantation. Elle m'a aussi lancé dans l'écriture de la tactique `CC` (voir chapitre 5).

Chapitre 1

le système Coq

1.1 Principe général

Le système Coq est un moteur de preuve développé conjointement par l'INRIA et l'Université Paris-Sud. Il se présente sous la forme d'un environnement permettant la manipulations de termes typés qui représentant des programmes et des preuves et de leur types qui représentent les types des programmes et les propositions logiques.

Grâce à l'isomorphisme de Curry-Howard, on sait que ces deux aspects (preuves et programmes) sont isomorphes, néanmmoins les preuves on ceci de particulier qu'en fin de compte seul leur type (la proposition prouvée) compte, on appelle cela la *proof-irrelevance* : dans une démonstration, deux preuves de la même proposition sont librement interchangeables.

Il existe de nombreux système de termes typés basés sur le λ-calcul, et parmi les plus connus on peut citer le système F introduit par J.-Y. Girard et le Calcul des Constructions (T. Coquand, G. Huet).

Le systéme Coq est basé sur une extension de ce dernier : le Calcul des Constructions Inductives. Ce calcul est enrichi par l'ajout d'objets inductifs ressemblant aux types concrets de ML.

Dans Coq, on sépare les termes de bases dans deux sortes (les types de leur type) appelés `Set` et `Prop` qui servent à annoter les termes pour distinguer les programmes (dans `Set`) des preuves (dans `Prop`), et il existe une hiérarchie d'univers `Type` pour typer les sortes.

Pour construire les termes de base, on dispose de variables `x`, d'applications `(f x)`, d'abstractions `[x :A]P`, de produits `(x :A)B` et de sortes `Set`,`Prop` et `Type`, on notera que lorsque le produit n'est pas dépendant, on le note avec la flèche `A->B`.

Ces objets sont les seuls objets primitifs en Coq, avec les point fixes, les co-point-fixes et quelques extensions (substitutions explicites, enregistrements ...). Les connecteurs logiques, par exemple, sont définis à l'aide d'objets inductifs.

1.2 Objets inductifs

Un objet inductif est défini par un ensemble de constructeurs associés à leur type, et permettant d'écrire des termes dont le type est l'inductif.

Pour illustrer les types inductifs, nous allons donner des exemples qui nous serviront dans la suite :

```
Inductive bool  : Set :=  true : bool | false : bool

Inductive nat  : Set :=  O : nat | S : nat->nat

Inductive list [A : Set]  : Set :=
      nil : (list A) | cons : A->(list A)->(list A)
```

Le type list permet d'illustrer l'utilisation de types dépendants, ici dans `Set`, pour faire du polymorphisme «à la ML». Ici, `cons` sera de type `(A :Set)A->(list A)->(list A)`, ce qui nous amène à expliquer l'utilisation des `?` pour remplacer certains arguments : si on écrit `(cons ? O)`, Coq peut inférer tout seul le type de `O` qui est `nat` et il fera le remplacement de façon interne, ce qui permet d'éviter d'ecrire des arguments inutiles.

L'utilisations des objets inductifs se fait à l'aide de points-fixes :

```
Fixpoint plus [n :nat] : nat -> nat :=
   [m :nat]Cases n of
      O   => m
  | (S p) => (S (plus p m)) end.
```

1.3 Opérateurs logiques

Pour les proposition logiques, on a déjà l'implication (produit non dépendent) et la quantification universelle (produit dépendant), mais pour le reste, on va utiliser des objets inductifs.

```
Inductive and [A : Prop ; B : Prop]  : Prop :=  conj : A->B->A/\B

Inductive or [A : Prop ; B : Prop]  : Prop :=
      or_introl : A->A\/B | or_intror : B->A\/B

Inductive ex [A : Set ; P : A->Prop]  : Prop :=
      ex_intro : (x :A)(P x)->(Ex P)

Inductive True  : Prop :=  I : True

Inductive False  : Prop :=

Inductive eq [A : Set ; x : A]  : A->Prop :=  refl_equal : x=x
```

Pour comprendre comment fonctionnent ces opérateurs logiques, il faut regarder ce qu'on obtient en éliminant les constructeurs : une preuve de `and A B` donnera une preuve de `A` et une preuve de `B`, par exemple, et on constate aussi qu'une preuve par cas sur `False` permet de tout déduire : il n'y a aucun cas à considérer.

```
False_ind = [P :Prop ; f :False]<P>Cases f of end
```

On peut aussi construire dans `Set` de tels opérateurs :

```
Inductive sig [A : Set ; P : A->Prop]  : Set :=
      exist : (x :A)(P x)->(sig A P)

Inductive sumbool [A : Prop ; B : Prop]  : Set :=
      left : A->{A}+{B} | right : B->{A}+{B}
```

Ces opérateurs peuvent être utilisés dans des programmes car ils sont définis dans `Set`. Au nom de la *proof-irrelevance*, si on fait dépendre un programme d'un test sur `(or True True)`, dont la forme normale est soit `(or_introl I)` soit `(or_intror I)`, alors suivant la preuve choisie le résultat du programme changerait et donc `Set` deviendrait *irrelevent* et donc impropre pour effectuer des spécifications concrètes.

L'opérateur `sumbool` permet d'encapsuler dans un même objet un test rendant un booléen et la preuve que ce booléen «fait le bon choix». De plus, après extraction, `sumbool` se comporte exactement comme `bool`.

1.4 Notion d'accessibilité

Le calcul de Coq est fortement normalisant, et ceci passe par une condition de garde sur les points fixes : tout appel récursif doit être effectué sur un sous-terme strict du terme sur lequel est basée la récursion. Ce critère est très contraignant mais il offre l'avantage d'être purement syntaxique.

Le problème est que dans la plupart des algorithmes que l'on souhaite définir cette condition ne peut pas être réalisée «naturellement», et on voudrait utiliser une relation quelconque bien fondée telle que l'argument récursif «décroisse» pour cette relation. Pour contourner cela, on utilise le prédicat récursif d'accessibilité :

```
Inductive Acc [A : Set ; R : A->A->Prop]  : A->Prop :=
      Acc_intro : (x :A)((y :A)(R y x)->(Acc A R y))->(Acc A R x)
```

Une preuve de ce prédicat pour une relation et un élément de l'ensemble donné, sera la preuve de la finitude de toute chaine «décroissante» partant de l'élément en question pour cette relation.

On peut alors définir la bonne fondation :

```
Definition well_founded :=
[A :Set ; R :(A->A->Prop)](a :A)(Acc A R a)
     : (A :Set)(A->A->Prop)->Prop
```

L'intérêt du prédicat d'accessibilité est qu'il permet alors de définir des point fixes par induction bien-fondée : c'est le prédicat d'accessibilité qui va servir d'argument récursif.

En effet, par élimination d'un objet de type `(Acc A R x)`, on obtient un objet de type `(y :A)(R y x)->(Acc A R y)`, sous terme strict du précédent. Si on veut effectuer un appel récursif avec `y`, il faut juste utiliser cet objet avec une preuve de `(R x y)`, et on obtient un objet de type `(Acc A R y)`, permettant un appel récursif satisfaisant la condition de garde. (*cf* [2])

Comme `Acc` est dans `Prop`, après extraction, on obtient une fonction récursive telle qu'on la définirait dans ML mais en ayant une garantie de terminaison.

1.5 Le moteur de preuve

La fonctionnalité principale de Coq est le moteur de preuve, qui permet de construire des termes de façon interactive. Son affichage principal est constitué de la liste de l'environement courant et du type du terme à construire (le but à prouver et ses hypothèses) :

```
Coq < Theorem T :(A :Prop)(A\/False)->A.
1 subgoal

  ============================
   (A :Prop)A\/False->A
```

Ensuite l'utilisateur dispose de commandes appelées tactiques qui d'une part vont remplacer un «trou» de la preuve par un contexte contenant un ou plusieurs trous, et d'autre part vont remplacer le but courant par les types des «trous» de ce contexte : au début de la preuve le terme est simplement `?`.

Ici nous allons commencer par introduire les hypothèses :

```
T < Intros.
1 subgoal

  A : Prop
  H : A\/False
  ============================
   A

T < Show Proof.
LOC :
Subgoals
1 -> (A :Prop)A\/False->A
Proof : [A :Prop; H :(A\/False)](?1 A H)
```

Ensuite on distingue les deux cas pour `H` :

```
T < Elim H.
2 subgoals

  A : Prop
  H : A\/False
  ============================
   A->A

subgoal 2 is :
 False->A
```

Ici, Coq a créé deux sous-buts mais n'affiche que l'environement du premier.

```
T < Trivial.
1 subgoal

  A : Prop
  H : A\/False
  ============================
```

```
   False->A

T < Contradiction.
Subtree proved!
```

La tactique `Trivial` permet de résoudre les problèmes ou le terme preuve est une projection (ici l'identité), et `Contradiction` résoud les buts contenant `False` dans leurs hypothèses.

Le moteur de preuve permet d'écrire de façon semi-automatique des termes-preuves de taille importante en simplifiant la tâche du programmeur. Malgré tout il reste d'une utilisation assez lourde car il ne permet pas les ellipses habituelles des preuves sur papier.

1.6 Réflexion totale, Réflexion partielle

La réflexion est une méthode de preuve basé sur le principe suivant : «*Remplacer les preuves par des calculs*».

La réflexion totale consiste pour un problème P dont l'ensemble des instances est X, à implanter une procédure de décision $\Delta : X \to \text{bool}$ purement calculatoire et à prouver sa correction : $\forall x \in X, \Delta(x) = \text{true} \Rightarrow P(x)$.

Ainsi, on remplace une preuve explicite de $P(x)$ par le calcul de la forme normale de $\Delta(x)$, qui permet, par conversion, de prouver que $\Delta(x) = \text{true}$ et d'appliquer le théorème.

Le théorème de complétude : $\forall x \in X, P(x) \Rightarrow \Delta(x) = \text{true}$ n'a pas d'utilisation pratique en logique intuitioniste, mais en logique classique on utilise sa contraposée : $\forall x \in X, \Delta(x) = \text{false} \Rightarrow \neg P(x)$ pour des preuves négatives.

La réflexion partielle utilise une méthode différente pour arriver aux mêmes fins : elle a recours à un outil extérieur pour faire les calculs.

Cet outil fournit ensuite à l'environnement de preuve une trace permettant de reconstruire une preuve. Cette méthode a l'avantage d'être plus souple : elle permet l'utilisation de méthodes de calcul plus rapides que celles offertes par l'environnement de preuve (structures mutables ...).

Elle souffre cependant de deux inconvénients majeurs :

- L'outil peut ne pas terminer ou fournir des résultats incohérents (ce dernier cas ne remet cependant pas en cause l'intégrité de la preuve).
- La preuve doit être calculée explicitement, ce qui pose un problème de ressource en espace, et donc aussi en temps

La suite du rapport se compose des présentations sucessives de ces deux approches dans le cas de la clôture de congruence .

Chapitre 2

La clôture de congruence

2.1 Algèbre de termes

2.1.1 Termes à arités multiples

Définition 2.1.1 (signature)
Une signature *est un ensemble fini* $\Sigma = (f_i, d_i)_{i<n}, n \in \mathbb{N}$ *de symboles de constantes munis d'une arité entière.*

Définition 2.1.2 (terme)
Un terme *est un objet construit inductivement par l'application d'un symbole de constante* f_i *à* d_i *arguments, l'ensemble des termes est donc le plus petit ensemble* T_Σ *tel que* $\forall (f, d) \in \Sigma, \forall (t_0, \ldots, t_{d_i-1}) \in T_\Sigma^{d_i}, f_i(t_0, \ldots, t_{d_i-1}) \in T_\Sigma$.

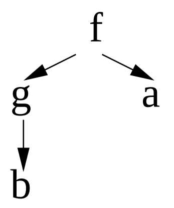


Fig. 2.1: exemple de terme : f(g(b),a)

Définition 2.1.3 (relation d'équivalence)
Une relation d'équivalence sur un ensemble T *est une relation binaire* $\approx$ *close par trois règles :*

- $\dfrac{}{t \approx t} t \in T$ *(réflexivité)*
- $\dfrac{t \approx u}{u \approx t} t, u \in T$ *(symétrie)*
- $\dfrac{t \approx u \wedge u \approx v}{t \approx v} t, u, v \in T$ *(transitivité)*

Définition 2.1.4 (congruence)
Soit Σ *une signature, une* congruence *sur* T_Σ *est une* relation d'équivalence $\approx$

close par la règle de congruence :

$$\frac{t_0 \approx u_0 \wedge \ldots \wedge t_{d-1} \approx u_{d-1}}{f(t_0, \ldots, t_{d-1}) \approx f(u_0, \ldots, u_{d-1})} (f, d) \in \Sigma \wedge t_0, \ldots, t_{d-1}, u_0, \ldots, u_{d-1} \in T_\Sigma$$

THÉORÈME 2.1.1 ()
Soient une signature Σ et un ensemble A de couples de termes de T_Σ, il existe une une plus fine relation $\sim_A$ sur T_Σ telle que $\sim_A$ soit une congruence contenant A, i.e. telle que $\forall (u, v) \in A, u \sim_A v$.

Démonstration : L'ensemble des congruences contenant A est non vide (il contient la relation pleine) et il est stable par intersection, il contient donc un plus petit élément pour l'inclusion qui est l'intersection de toutes les congruences contenant A. *qed*

2.1.2 Termes curryfiés

DÉFINITION 2.1.5 (TERME,ARITÉ)
Un terme *est soit une constante, il a alors l'arité de la constante, soit l'application d'un terme d'arité $n > 0$ à un terme d'arité 0, le terme ainsi formé a alors pour arité $n - 1$.*

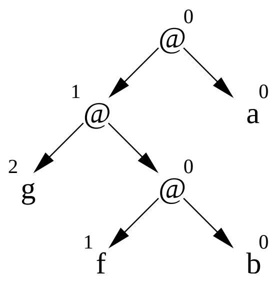


FIG. 2.2: exemple de terme : g(f(b),a)

DÉFINITION 2.1.6 (CONGRUENCE)
Soit Σ une signature, une congruence *sur T_Σ est une une* relation d'équivalence *$\approx$ ne liant que des termes de même arité et close par la règle de congruence :*

$$\frac{t \approx u \wedge v \approx w}{App(t, v) \approx App(u, w)} t, u, v, w \in T_\Sigma$$

(On remarque que la règle de congruence est compatible avec la restriction aux termes de même arité.)

On a également l'existence d'une plus petite congruence contenant un ensemble fini de couples de termes de même arité.

2.2 La clôture de congruence

Le problème de clôture de congruence consiste, étant donné un ensemble fini A d'axiomes (couples de termes clos supposés égaux), à déterminer, pour un couple (t, u) de termes, si $t \sim_A u$. Dans la suite on appellera *axiome* tout couple de A et *requête* le couple (t, u).

Ce problème est équivalent à l'existence d'une preuve de $t = u$, pour le système de preuve contenant les 3 règles de réflexivité, transitivité et symétrie, la règle de congruence et la règle d'axiome :

$$\frac{}{v = w}(v, w) \in A$$

Si l'on se restreint à des axiomes portant sur des termes d'arité 0, le problème de clôture de congruence sur les termes curryfiés est équivalent au problème sur les termes non curryfiés.

2.2.1 Aperçu historique

Le premier résultat important sur la clôture de congruence est dû a Robert Shostak [7], qui démontre que la décision d'un problème de clôture de congruence peut se faire en ne considérant la relation que sur l'ensemble (fini) des termes et sous termes des axiomes et de la requête.

Cette découverte permet d'en finir avec les méthodes par substitution, qui sont inefficaces car il n'existe pas de moyen simple de borner la profondeur des termes à générer pour obtenir un résultat.

Au lieu de procéder à des substitutions, Shostak propose de procéder par fusion de classes d'équivalences sur les termes et sous-termes du problème .

Nelson et Oppen [4] utilisent alors leurs travaux sur les structures UNION-FIND pour proposer une première implantation, mais c'est à Downey,Sethi et Tarjan que l'on doit l'introduction d'une *table des signatures* qui permet d'améliorer les performances de l'algorithme.

2.3 L'algorithme

2.3.1 Description rapide

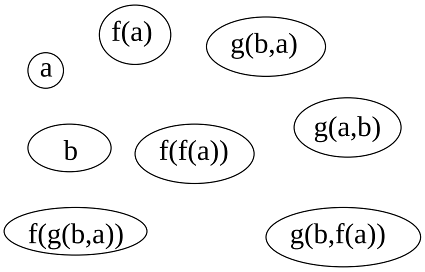
On suppose que l'on dispose d'une structure permettant de manipuler des classes d'équivalence : recherche du représentant canonique, fusion de classes. L'algorithme fonctionne ainsi : on construit une structure contenant tous les termes et sous-termes du problème dans des classes singletons, puis on fusionne les classes contenant les deux membres d'un axiome. Ensuite on fusionne les classes contenant deux termes congrus (c'est-à-dire deux termes de meme arité dont les sous termes sont deux à deux dans les mêmes classes d'équivalence), jusqu'à l'obtention d'un point fixe. Alors il suffit de regarder si les termes de la requête sont dans la même classe d'équivalence pour conclure.

2.3.2 Un exemple

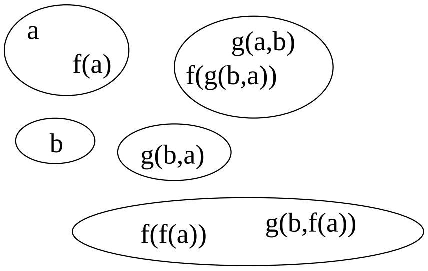
Prenons l'exemple de la couverture, soit le problème :

$$a = f(a) \wedge g(a, b) = f(g(b, a)) \wedge g(b, f(a)) = f(f(a)) \overset{?}{\vdash} g(a, b) = a$$

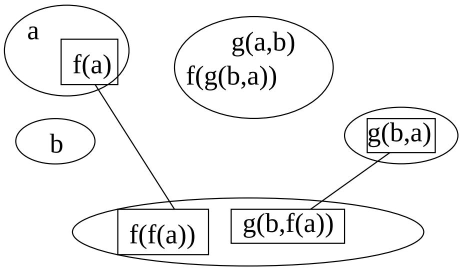
(exemple dû à Jean Goubault-Larrecq).



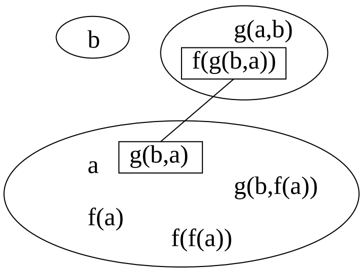
Tout d'abord, construisons l'ensemble des termes et sous termes, que nous plaçons dans des classes distinctes.



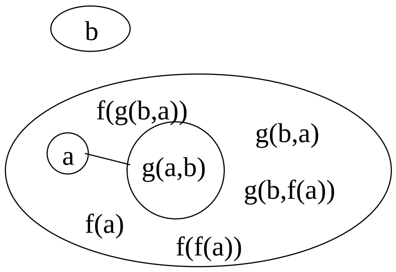
Ensuite nous fusionnons les classes contenant les membres des axiomes (notons que parfois plusieurs axiomes portent sur le même terme, c'est pourquoi on doit construire des singletons et fusionner ensuite).



Ensuite on identifie les termes congruents : a et $f(a)$ sont dans la même classe donc $f(a)$ et $f(f(a))$ sont congruents, et $g(b, a)$ et $g(b, f(a))$ le sont également.



Après fusion des classes, on s'aperçoit que maintenant a et $g(b, a)$ sont dans la même classe, donc $f(a)$ et $f(g(b, a))$ sont congruents.



On s'aperçoit qu'il ne reste plus de termes congruents qui ne soient dans la même classe après fusion, donc l'algorithme a fini son exécution.

On regarde les termes de la requête et on constate qu'ils sont dans la même classe d'équivalence, donc on répond **oui**.

2.3.3 Table de signatures

La table des signatures (voir [1]) est une structure annexe permettant une identification plus efficace des termes congruents.

Définition 2.3.1 (signature)
Étant donné une fonction d qui associe à chaque terme un index représentant la classe à laquelle il appartient, on définit la signature sig d'un terme non atomique par $sig(f(t_0, \dots, t_{n-1})) = f(d(t_0), \dots, d(t_{n-1}))$.

Lemme 2.3.1 (congruence)
Deux termes ayant même signature sont congruents

L'idée de Downey *et al.* est donc de construire et maintenir une table qui associe à une toute signature un terme dont c'est la signature, d'enlever les termes dont la signature change suite à une fusion de classes, et de ne consi'erer ensuite que ceux-ci pour trouver de nouvelles congruences.

La mise en oeuvre de cette table nécéssite la connaissance, pour chaque classe d'équivalence, des termes ayant un terme dans la classe comme sous-terme direct, c'est-à-dire des termes dont cette classe apparait dans la signature.

Pour simplifier la table de signatures, Downey *et al.* utilisent un codage des termes très proche du codage curryfié présenté plus haut, qui a l'énorme avantage de réduire la signature à un couple d'index, ce qui rend l'implantation de la table plus aisée.

Chapitre 3

Réflexion totale : Implantation

3.1 Structure d'adressage

Pour implanter la clôture de congruence en Coq, il nous fallait une structure capable de contenir deux systèmes de pointage orthogonaux : le système de pointage de la structure UNION-FIND (voir 3.2) et celui qui fait pointer les termes vers les sous-termes, en nous permettant un accès rapide à tous les nœuds.

Pour cela , j'ai utilisé la librairie `IntMap`, nouvellement rajoutée à la librairie standard de Coq [9], et tirée d'une contribution de Jean Goubault-Larrecq. Cette librairie implante une structure d'arbres binaires permettant de coder toute fonction de domaine fini sur les entiers.

Les entiers sont codés de façon binaire en réutilisant le type `positive` introduit par `ZArith` .

```
Inductive positive : Set :=
      xI : positive->positive
    | xO : positive->positive
    | xH : positive.

Inductive ad  : Set :=
      ad_z : ad
    | ad_x : positive->ad.
```

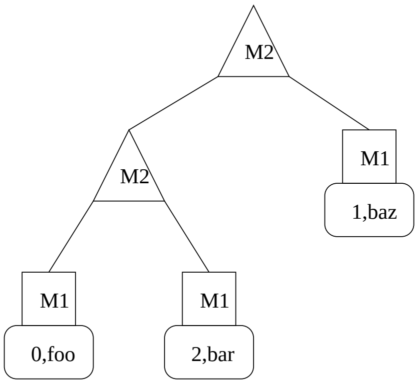
On dispose d'un type option pour coder les fonction partielles :

```
Inductive option [A : Set]  : Set :=
      NONE : (option A)
    | SOME : A->(option A).
```

Le type Map est alors défini de la façon suivante :

```
Inductive Map [A : Set]  : Set :=
      M0 : (Map A)
    | M1 : ad->A->(Map A)
    | M2 : (Map A)->(Map A)->(Map A).
```

Par exemple :



représente la fonction partielle :

$$\begin{cases} 0 & \mapsto & foo \\ 3 & \mapsto & baz \\ 10 & \mapsto & bar \end{cases}$$

On dispose alors des primitives suivantes :

```
MapGet : (A :Set)(Map A)->ad->(option A).
MapPut : (A :Set)(Map A)->ad->A->(Map A).
MapRemove : (A :Set)(Map A)->ad->(Map A).
MapCard : (A :Set)(Map A)->nat.
ad_alloc_opt : (A :Set)(Map A)->ad.
```

`MapGet`, `MapPut` et `MapRemove` permettent la lecture, l'ajout et le retrait d'éléments d'un arbre. `MapCard` calcule le nombre d'éléments d'un arbre et `ad_alloc_opt` calcule une adresse libre de l'arbre.

3.2 UNION-FIND

3.2.1 Principe de fonctionnement

L'algorithme UNION-FIND est basé sur une forêt dont chaque arbre représente une classe et chaque nœud un élément de la classe représentée par la racine de l'arbre. L'opération FIND devient alors une simple recherche de la racine d'un arbre, et UNION consiste à faire pointer la racine d'une classe sur celle d'une autre pour les faire fusionner. Ajouter une classe singleton ne pose aucune difficulté.

3.2.2 Implantation en Coq

Le problème principal est un problème de terminaison : en effet, FIND s'implante de façon récursive non structurelle, ce qui pose un problème à Coq qui ne sais pas gérer ce type de récursion, on va donc conserver avec la structure UNION-FIND une preuve de l'accessibilité de ses sommets.

On définit :

```
Inductive class [A :Set]  : Set :=
      Rep : A->(class A)
    | Eqto :ad->(class A).
```

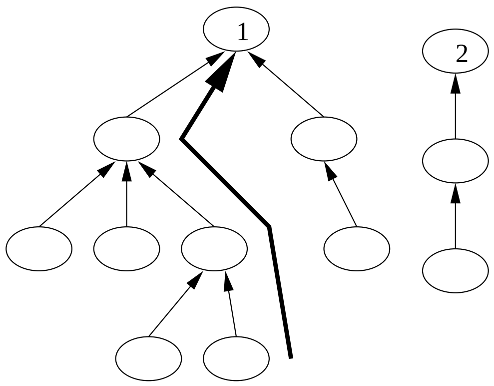


FIG. 3.1: l'opération FIND

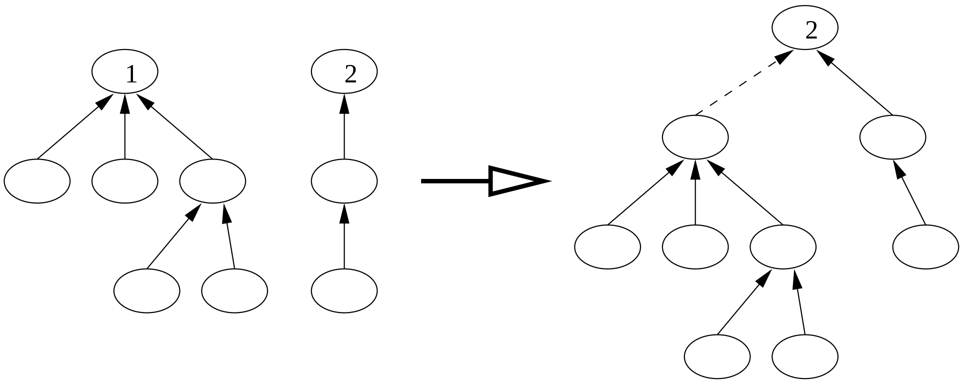


FIG. 3.2: l'opération UNION

Le type d'une structure UNION-FIND dont les nœuds contiennent une étiquette de type `B` et les classes une étiquette de type `A` sera `(Map (class A)*B)`.

Pour assurer la terminaison de FIND, on définit :

```
Definition father := [m :(Map (class A)*B)][j,i :ad]
  ((Cases (MapGet (class A)*B m i) of
  (SOME ((Eqto a),_))=>a=j
  | _ =>False.

Definition Wf_map :=[m :(Map (class A)*B)](i :ad)(Acc ad (father m) i).

Theorem find : (m :(Map (class A)*B))(i :ad)(Acc ad (father m) i)->(option ad).
```

La relation `father` exprime le fait qu'un nœud pointe sur une autre, et le prédicat `Wf_map` exprime le fait que cette relation est bien fondée. La fonction `find` peut donc être définie par induction structurelle sur le type `(Acc ad (father m) i)` (*cf* 1.4 et [2] pour plus de détails).

Pour pouvoir manipuler facilement les expressions avec find, il est utile de prouver que le résultat de `find` est indépendant de l'argument supplémentaire.

```
Lemma find_indep : (m :(Map (class A)*B))(i :ad)
        (acc1,acc2 :(Acc ad (father m) i))
        (find m i acc1)=(find m i acc2).
```

Lorsque l'on fait une preuve par récurrence, on a aussi besoin de pouvoir «déplier» le point fixe, et on utilise pour cela un cas particulier du théorème précédent, pour avoir un constructeur en tête et pouvoir appliquer une ι-réduction :

```
Lemma find_unfold : (m :(Map class*B))(i :ad)
        (H :(Acc ad (father m) i))
        (find m i H)=(find m i (Acc_intro??? (Acc_inv??? H))).
```

Pour UNION, il faut spécifier une fonction `comp : A->A->A` pour spécifier comment on compose les étiquettes de classes pour former la classe réunion. `union` est alors de type `(Map (class A)*B)->ad->ad->(Map (class A)*B)`.

```
Definition union :=[m :(Map (class A)*B)][i1,i2 :ad]
  Cases (option_sum? (MapGet? m i1)) of
    (inleft (exist ((Rep a1),b1 ) _))=>
      Cases (option_sum? (MapGet? m i2)) of
        (inleft (exist ((Rep a2),b2 ) _))=>
          (MapPut?
            (MapPut? m i2 ((Eqto i1),b2))
              i1 ((Rep (comp a1 a2)),b1))
      | _ => m
      end
  | _ => m
  end.
```

On prouve alors que `union` préserve la bonne fondation de la structure.

```
Theorem Wf_union : (m :(Map (class A)*B))(i,j :ad)
                (Wf_map m)->(Wf_map (union m i j)).
```

Et aussi le théorème fondamental de la structure UNION-FIND :

```
Theorem union_semantics :
  (m :(Map class*B))(H :(Wf_map m))(i,j :ad)[...]
  (a :ad)((find?? (Wf_union? i j H a)) =
    (Cases (find?? (H a)) of
        (SOME k) =>
           if (ad_eq k j) then (SOME? i)
           else (SOME? k)
      |  NONE =>(NONE ad)
                end)).
```

Ce théorème décrit de façon très précise ce qui se passe vis à vis de `find` lors d'une `union`.

Dans les implantations classiques de UNION-FIND , on utilise la «compression de chemins» : à chaque exécution de find on fait pointer directement le nœud testé sur la racine pour accélerer les accès ultérieurs. Ici, eût égard à la complexité de l'algorithme, on n'a pas utilisé de compression de chemin, on essaie seulement de faire les unions dans le sens «le plus avantageux».

3.3 Table de signatures

Comme on va représenter les nœuds par des `ad` (voir 3.1), une signature sera un couple de `ad`. La table de signature doit associer un terme à une signature, donc un `ad` à un couple d'`ad`.

Pour cela, plutôt que d'utiliser le type `(Map (Map ad))`, on utilise un codage injectif : `iad_conv : ad->ad->ad` qui utilise l'intercalage de bits (travail dû à Xavier Rival [6]).

Malheureusement un `(Map ad)` ne suffit pas car il faut pouvoir effacer des *termes* de la table de signature, sachant qu'alors on ne peut généralement pas recalculer la signature a laquelle était associée ce terme. On doit donc avoir une table réciproque de la précédente.

```
Definition stt :=(Map ad)*(Map ad*ad).
Definition empty_st :=((M0 ad),(M0 ad*ad)).
```

On définit alors les primitives suivantes : `query` pour chercher le terme associé à une signature, `enter` pour ajouter une signature et `delete` pour effacer un terme et la signature associée.

3.4 L'algorithme de clôture de congruence

L'algorithme de clôture de congruence est composée d'une boucle principale agissant principalement par «effets de bord» sur la structure UNION-FIND . On décrira d'abord comment cette structure est créée et ensuite ce que l'on en fait.

Les termes abstraits sont modélisés par :

```
Inductive term :Set :=
      Symb : ad->term
    | App : term->term->term.
```

Pour représenter les termes dans les nœuds on utilise le type suivant :

```
Inductive Set vert :=
      Leaf : vert
    | Node : ad->ad->vert.
```

Chaque terme est représenté de façon unique par l'adresse d'un nœud, et contient `Leaf` si c'est une constante et `(Node i j)` si c'est l'application du terme représenté par le noeud à l'adresse `i` au terme représenté par le noeud à l'adresse `j`.

Les nœuds contiendront donc un objet de type `vert` et les représentants de classes contiendront en plus la liste des super-termes (ainsi que la taille de cette liste pour plus d'efficacité mais on ne le montrera pas ici).

On commence par créer la structure en ajoutant les termes du problème de la façon suivante : on a une liste `syms` des adresses des nœuds représentant les termes déja ajoutés, et on ajoute les sous-termes récursivement avant d'ajouter leur application. La procédure `add` doit retourner l'adresse où elle a placé le numéro du terme (figure 3.3).

Pour plus de simplicité le code sera écrit dans un style impératif pour épargner au lecteur la syntaxe Coq parfois un peu aride et des détails techniques peu intéressants.

```
add(m,syms,t) : :=
    if (t,i)∈ syms  then
        return(i)
    else case t of
        Symb(a) :
            i := ad_alloc_opt(m)
            m[i] := (Rep(0,[]) ,Leaf) ;
            syms := (Symb(a) ,i) : :syms ;
            return(i)
        App(u,v) :
            j := add(u)
            k := add(v)
            i := ad_alloc_opt(m)
            m[i] := (Rep(0,[]) ,Node(j,k)) ;
            syms := (App(u,v) ,i) : :syms ;
ajouter i dans les listes de super-termes de classes de j et k
            return(i)
    fi
```

Fig. 3.3: Procédure d'addition de terme dans la structure

La création de la structure sera donc faite par ajouts successifs des couples de termes d'un axiome, puis union des classes des deux termes de l'axiome, et lorsque tous les axiomes ont été ajoutés, on ajoute les deux termes de la requête, et on garde en mémoire leurs adresses, puis, après exécution de l'algorithme, on regarde si les deux nœuds sont dans la même classe (figure 3.4).

```
create(axioms,s,t) : :=
    m := ∅ ;
    syms := []
    for each (u,v) ∈ axioms  do
        i := add(m,u,syms) ;
        j := add(m,v,syms) ;
        k := find(m,i) ;
        l := find(m,j) ;
        union(m,i,j)
    od ;
    i := add(m,u,syms) ;
    j := add(m,v,syms) ;
    return(m,i,j)
```

Fig. 3.4: Procédure de création de la structure

La boucle principale se décompose en deux parties, la première (figure 3.5) détermine les termes congruents, et la seconde (figure 3.6) fusionne leurs classes, en retournant la liste des termes dont la signature a changé, pour que la première recommence.

Pour l'implantation Coq, on définit séparément `combine_rec` et `process_rec` par induction sur les listes `pending` et `combine`. Ensuite, on définit leur composition «séquentielle» (figure 3.7) puis on referme la boucle (figure 3.8).

```
combine_rec(m,st,pending) : :=
    combine := ∅ ;
    for each v ∈ pending  do
        if query(st,v)=⊥  then
            enter(st,v)
        else
            combine := (v,query(st,v)) : :combine
        fi
    od ;
    return(combine)
```

Fig. 3.5: Première sous-boucle

```
process_rec(m,st,combine) : :=
    pending := ∅ ;
    for each (v,w) ∈ combine  do
        if find(m,v)≠find(m,w)  then
            if |list(m,find(m,v))| < |list(m,find(m,w))|  then
                for each u ∈ list(m,find(m,v))))  do
                    delete(u) ; pending := u : :pending ;
                union(m,find(m,w),find(m,v))
            else
                for each u ∈ list(m,find(m,w))))  do
                    delete(u) ; pending := u : :pending ;
                union(m,find(m,v),find(m,w))
            fi fi od ;
    return(pending)
```

Fig. 3.6: Seconde sous-boucle

```
cc_rec(m,st,pending) : :=
    combine := combine_rec(m,st,pending) ;
    pending := process_rec(m,st,pending) ;
    return(pending)
```

Fig. 3.7: Corps de la boucle principale

```
cc_prog(m) : :=
    st := empty_st ;
    pending := { i | MapGet(i) est un Node }
    while pending ≠ ∅  do
        pending := cc_rec(m,st,pending)
    od
```

Fig. 3.8: Boucle principale

```
decide_cc(axioms,u,v) : :=
    (m,i,j) := create(axioms,u,v) ;
    ccprog(m) ;
    return(find(m,i)=find(m,j))
```

Fig. 3.9: Procédure complète

Chapitre 4

Certification

4.1 Terminaison

L'argument de terminaison de l'algorithme est basé sur le nombre de classes d'équivalences présentes dans la structure, que l'on calcule avec :

```
Fixpoint uf_card [m :(Map (class A)*B)] : nat :=
    Cases m of
       (M1 _ ((Rep _),_)) => (S O)
      | (M2 m m') => (plus (uf_card m) (uf_card m'))
      | _ => O
    end.
```

On montre ensuite qu'une union fait décroître strictement le nombre de classes, plus précisément elle le fait décroitre d'un. (Les arguments manquants sont les hypothèses disant que les adresses sont bien des adresses de représentants distincts).

```
Theorem union_card : (m :(Map class*B))(i,j :ad)[...]
        ((uf_card m)=(S (uf_card (union m i j)))).
```

On s'aperçoit alors que la boucle principale `cc_rec`, si elle renvoie une liste de nœuds non vide, fait décroître strictement le nombre de noeuds de la structure : si la liste renvoyée est non vide, cela signifie qu'au moins une union a eu lieu est donc que le nombre de classes a diminué.

```
Theorem lth_cc : (uf :uft;H :(Wf_map?? uf))
  (st :stt;pending0 :(list ad))
  Cases (cc_rec uf H st pending0) of
    ((exist uf2 _),(_,pending))=>
      {pending=(nil ad)/\
        (le (uf_card?? uf2) (uf_card?? uf))}+
          {lt (uf_card?? uf2) (uf_card?? uf)}
        end.
```

On définite un ordre sur les structures qui compare leur nombre de classes :

```
Definition lt_card := [uf,uf0 :uft](lt (uf_card?? uf) (uf_card?? uf0)).
```

Cet ordre est bien fondé car il est l'image réciproque d'un ordre bien fondé (théorème `wf_inverse_image`).

```
Definition wf_lt_card :=
        (wf_inverse_image uft nat lt (uf_card nat*(list ad) vertex) lt_wf).
```

La boucle principale `cc_Prog` peut donc être définie par induction sur le type `(Acc uft lt_card ... )`.

Comme le calcul de Coq est fortement normalisant, le fait de pouvoir définir ainsi le point fixe suffit à assurer la terminaison de la partie principale de l'algorithme.*qed*

4.2 Correction

Pour la correction et la complétude, on va utiliser le point de vue «prouvabilité» du problème de clôture de congruence . Si in considère le type

```
Inductive Prf [l :(list term*term)] :term->Prop :=
       PrfRefl : (t :term)
         (Prf l t t)
     | PrfSym  : (t,u :term)
         (Prf l t u)->(Prf l u t)
     | PrfTrans :(t,u,v :term)
         (Prf l t u)->(Prf l u v)->(Prf l t v)
     | PrfCongr :(t,u,v,w :term)
         (Prf l t u)->(Prf l v w)->
           (Prf l (App t v) (App u w))
     | PrfAx :(n :nat)(t,u :term)
         (nth n l)=(SOME? (t,u))->(Prf l t u).
```

Le théorème de correction s'écrira :

```
Theorem cc_correct : (axioms :(list term*term);u,v :term)
    (decide_prb axioms u v)=true->
    (Prf axioms u v).
```

Pour prouver ce théorème, on va passer par une représentation des preuves interne à l'algorithme : au lieu d'utiliser des termes, on va utiliser des addresses :

```
Definition eq_opt :=[A :Set][ox,oy :(option A)]
  Cases ox oy of
    (SOME x) (SOME y)=> x=y
  | _ _ =>False
  end.

Inductive Eqv [uf :uft] : ad->ad->Prop :=
    Refl_eqv : (i :ad)(Eqv uf i i)
  | Trans_eqv : (i,j,k :ad)(Eqv uf i j)->(Eqv uf j k)->(Eqv uf i k)
  | Sym_eqv : (i,j :ad)(Eqv uf i j)->(Eqv uf j i)
  | Congr_eqv : (i,j,i1,i2,j1,j2 :ad)(ci,cj :(class nat*(list ad)))
        (trmi,trmj :term)
        (MapGet? uf i)=(SOME? (ci,((Node i1 i2),trmi)))->
        (MapGet? uf j)=(SOME? (cj,((Node j1 j2),trmj)))->
        (Eqv uf i1 j1)->(Eqv uf i2 j2)->(Eqv uf i j)
  | Ax_eqv : (wf :(Wf_map?? uf))(i,j :ad)
        (eq_opt? (find?? uf i (wf i)) (find?? uf j (wf j)))->(Eqv uf i j).
```

On remarque que la notion d'axiome a changé : est considéré comme axiome toute paire de nœuds situés dans la même classe d'équivalence. Les autres règles sont similaires à celles de `Prf`.

On note $\approx^n$ la relation induite par `Prf` sur les nœuds après n exécutions de la boucle `cc_rec`. On va montrer que $\approx^n$ satisfait les trois propriétés suivantes :

1. $\approx^0 \subseteq \equiv$ où $\equiv$ désigne la restriction de `Prf` aux termes présents dans la structure (Attention : les sous-preuves peuvent contenir des termes externes à la structure).
2. $\approx^{n+1} \subseteq \approx^n$
3. $\sim \subseteq \approx^\omega$ où $\sim$ représente la relation induite par la structure UNION-FIND à la fin de l'algorithme et $\approx^\omega$ la relation $\approx$ à la fin également.

Ainsi, on aura $\sim \subseteq \equiv$, *cqfd.*

Le troisième point est trivial : il suffit d'utiliser le constructeur `Ax_eqv`. Nous allons donc nous concentrer sur le premier et le second.

4.2.1 Correction de l'algorithme

On va montrer la correction grâce au théorème :

```
Theorem corr_cc : (uf :uft)(wf :(Wf_map?? uf))
    (st :stt)(pending :(list ad))
    (st_coh st)->(slist_complete uf)->
    (uf_coh uf)->(st_correct uf st)->
    Cases (cc_rec uf wf st pending) of
        ((exist uf0 _),_)=>
          (i,j :ad)
              (Eqv uf0 i j)->(Eqv uf i j)
    end.
```

Ce théorème va en fait réaliser une construction de preuve similaire à celle de la tactique du chapitre suivant : il va s'agir de remplacer les «axiomes» `Ax_eqv` par des preuves des congruences avant fusion des classes. La preuve va donc être construite «par le haut» suivant le schéma 4.1 : on va insérer les congruences calculées par `cc_rec` dans l'arbre de preuve en repoussant de plus en plus bas les axiomes, pour finir par obtenir des axiomes finaux prouvables sans congruence (*cf* 4.2.2).

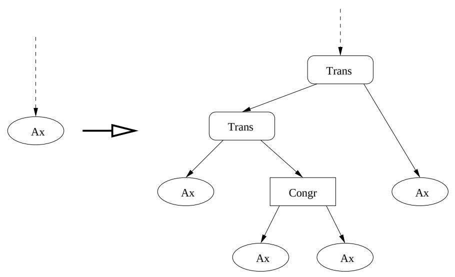


FIG. 4.1: Réalisation d'un congruence

Pour bien comprendre la figure, il faut comprendre qu'une preuve égalitaire est essentiellement une chaine transitive d'égalité, donc pour prouver que la fusion de classes n'introduit pas d'axiome non prouvable il faut couper la chaine à l'endroit de la congruence réalisée, et «recoller» les morceaux avec une preuve de congruence, qui elle même aura pour prémices des axiomes à cause de la correction de la table de signature : les sous termes sont équivalents pour UNION-FIND car ils ont même signature.. Pour réussir cette preuve il faut prouver d'abord quelques invariants concernant d'une part la cohérence interne de la structure UNION-FIND et de la table des signatures (préservation du pointage réciproque). Ensuite, il faut prouver que les signatures présentes dans la table des signatures restent toujours correctes. Toute ces preuves se font par induction sur `pending` et `combine`.

4.2.2 Correction de la création de structure

La correction de la création de structure, c'est le fait qu'une preuve sur les indices de la structure créée devient une preuve sur les termes du problème. En fait il suffit de prouver deux choses :
- les nœuds-termes pointent bien vers leurs sous-termes (préservation de la règle de congruence).
- les équivalences dans la structure UNION-FIND sont prouvables au sens de `Prf`.

Ces deux preuves sont réalisées par induction sur la taille du problème et des termes.

4.3 Complétude

Le théorème de complétude s'écrira :

```
Theorem cc_complete : (axioms :(list term*term);u,v :term)
    (Prf axioms u v)->
    (decide_prb axioms u v)=true.
```

4.3.1 Normalisation des preuves

Pour la complétude un problème nouveau se pose, c'est lui aui fait la nouveauté de l'algorithme lors de sa découverte : comme pour la correction, on va utiliser un format interne de preuve, mais comment transformer une preuve sur les temres en une preuves sur les indices ? On n'a aucune garantie que cette preuve ne fasse intervenir que des termes et sous-termes présents dans le problème .

La solution se présente sous la forme d'une *normalisation* des preuves, c'est-à-dire l'application d'un ensembles de transformations permettant d'obtenir une preuve *structurellement* interne. On modélise les preuves normales à l'aide de l'inductif `Nrm`.

```
Inductive NrmAtom [l :(list term*term)] : term->term->Prop :=
    NrmAx :(n :nat)(t,u :term)
      (nth n l)=(SOME? (t,u))->(NrmAtom l t u)
  | NrmSymAx : (n :nat)(t,u :term)
      (nth n l)=(SOME? (t,u))->(NrmAtom l u t).

Inductive Nrm  [l :(list term*term)] : term->term->Prop :=
```

```
    NrmTrans : (t,u,v :term)(NrmAtom l t u)->(Nrm l u v)->(Nrm l t v)
  | NrmTransCongr : (t,u,v,w,x,y :term)(Nrm l t u)->(Nrm l v w)->
        (NrmAtom l (App u w) x)->(Nrm l x y)->(Nrm l (App t v) y)
  | NrmCongrNil : (t,u,v,w :term)(Nrm l t u)->(Nrm l v w)->
                (Nrm l (App t v) (App u w))
  | NrmNil : (t :term)(Nrm l t t).
```

On s'assure d'abord que l'on peut normaliser correctement toute preuve, pour cela on construit des opérateurs ayant la même sémantique que les règles de `Prf` :

```
Theorem concat : (l :(list term*term);t1,t2,t3 :term)
    (p :(Nrm l t1 t2);q :(Nrm l t2 t3))(Nrm l t1 t3).

Theorem atom_reverse  : (l :(list term*term);t,u :term)
    (p :(NrmAtom l t u))(NrmAtom l u t).

Theorem reverse : (l :(list term*term);t,u :term)
    (p :(Nrm l t u))(Nrm l u t).
```

Puis on prouve que toute preuve peut être normalisée :

```
Theorem NrmPrf : (l :(list (term*term));t,u :term)
    (p :(Prf l t u))(Nrm l t u).
```

Les preuves sous forme normale ont la forme suivante : une liste, possiblement réduite à la liste vide (réflexivité), de preuves atomiques s'enchaînant par transitivité. Les preuves atomiques sont soit un axiome, soit son symétrique, soit une congruence dont les deux sous-preuves sont normales. De plus on exige (figure 4.2, en bas, à droite) qu'il n'y ait jamais deux congruences consécutives : le constructeur `NrmCongrNil` correspond à une congruence terminant la preuve, et le constructeur `NrmTransCongr` impose la présence d'un axiome après toute congruence.

Théorème 4.3.1 (Internalité)
Une preuve en forme normale de l'égalité de deux termes du problème ne fait intervenir que des termes présents dans le problème .

Démonstration : Par induction structurelle.

Si la preuve est une simple réflexivité elle concerne deux termes du problème donc c'est une preuve interne.

Les axiomes et leurs symétriques sont des preuves internes.

Reste le problème des congruences : les 2 termes reliés par une congruence sont soit aux extrémités de la chaîne, soit à côté d'une axiome ou symétrique d'axiome (pas de congruences consécutives) donc ces 2 termes sont présents dans le problème . Leurs sous termes sont donc aussi présents dans le problème et donc les prémices sont des sous-preuves d'égalités de termes du problème , elles-mêmes en forme normale. On peut donc leur appliquer l'hypothèse de récurrence. cqfd

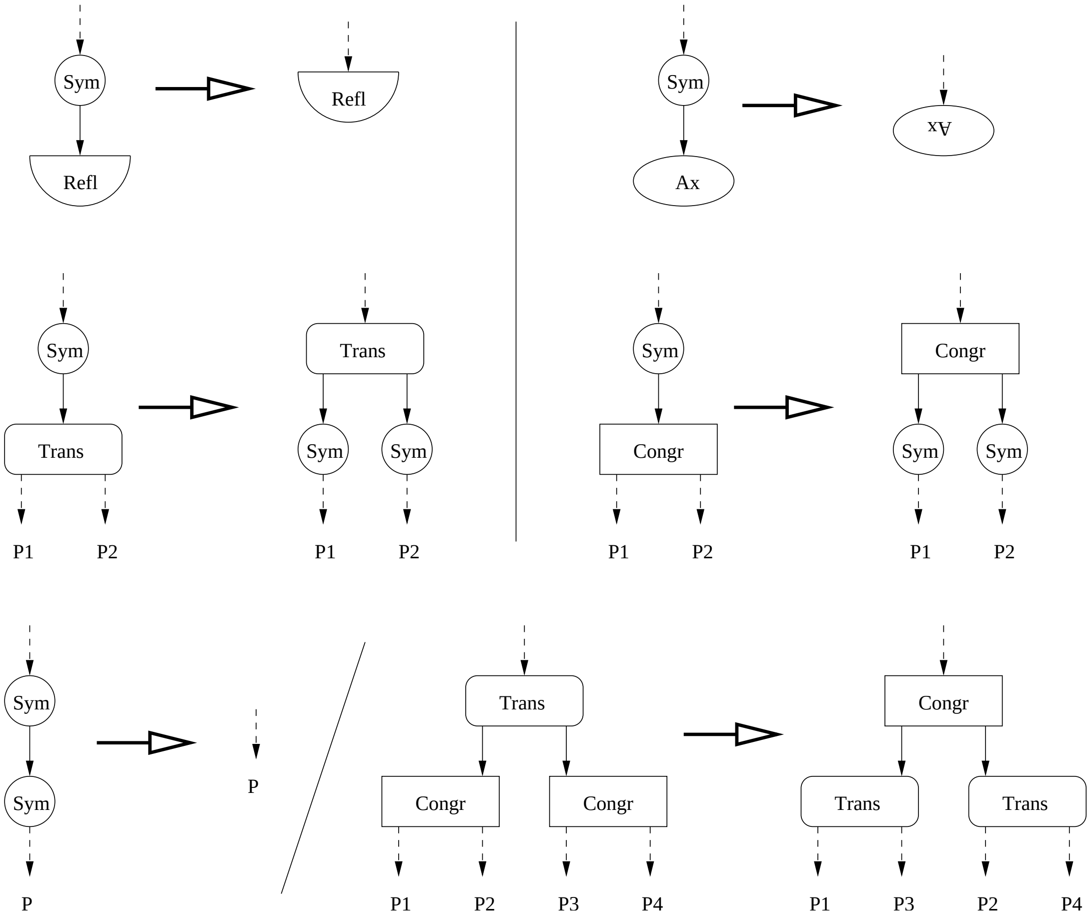


FIG. 4.2: Règles de normalisation

4.3.2 Preuve de complétude

Maintenant nous savons comment obtenir des preuves internes, mais comment modéliser ces preuves pour modéliser la complétude ? Il nous faut un inductif avec une contrainte pour que celui-ci nous permette de montrer l'égalité UNION-FIND à la fin de l'algorithme. Pour la correction cette contrainte était donnée par le nombre décroissant d'égalités UNION-FIND (en regardant le temps à l'envers) qui permettait de moins en moins de recours au constructeur `Ax_eqv`.

Pour la complétude l'invariant-clé est le suivant : lorsque deux termes ont même signature, soit ils sont dans la même classe, soit l'un des deux est dans `pending`. Ainsi, on peut construire un type de preuve interne `Eqvc` où les preuves de bases sont des congruences élémentaires, que l'on définit ainsi : une congruence élémentaire relie deux termes de même signature si l'un au moins des deux est dans `pending` (c'est le constructeur `Congr_n`). Le prédicat `uf_eqv` relie les termes situés dans la même classe d'équivalence.

```
Definition uf_eqv :=[uf :uft ;i,j :ad](wf :(Wf_map?? uf))
    (eq_opt? (find???? (wf i)) (find???? (wf j))).
```

```
Inductive  Constrained [uf :uft;st :stt;p :(list ad); i,j :ad] : Prop :=
    Cstr_b : (In i p)->(In j p)->(Constrained uf st p i j)
  | Cstr_l : (In i p)->(wf :Wf_map uf)(c :ad*ad)(t :ad)
      (signature uf wf j)=(SOME? c)->(query c st)=(SOME? t)->
      (eq_opt? (find???? (wf j)) (find???? (wf t)))->
      (Constrained uf st p i j).
  | Cstr_r : (In j p)->(wf :Wf_map uf)(c :ad*ad)(t :ad)
      (signature uf wf i)=(SOME? c)->(query c st)=(SOME? t)->
      (eq_opt? (find???? (wf i)) (find???? (wf t)))->
      (Constrained uf st p i j).

Inductive Eqvc [uf :uft;st :stt;p :(list ad)] : ad->ad->Prop :=
    Trans_l : (i,j,k :ad)
      (Eqvc uf st p i j)->(uf_eqv uf j k)->(Eqvc uf st p i k)
  | Trans_r : (i,j,k :ad)
      (uf_eqv uf i j)->(Eqvc uf st p j k)->(Eqvc uf st p i k)
  | Trans_b : (i,j,k :ad)
      (Eqvc uf st p i j)->(Eqvc uf st p j k)->(Eqvc uf st p i k)
  | Congr_n : (i,j,i1,i2,j1,j2 :ad)(ci,cj :(class nat*(list ad)))
      (trmi,trmj :term)
      (MapGet? uf i)=(SOME? (ci,((Node i1 i2),trmi)))->
      (MapGet? uf j)=(SOME? (cj,((Node j1 j2),trmj)))->
      (Constrained uf st p i j)->
      (uf_eqv uf i1 j1)->(uf_eqv uf i2 j2)->(Eqvc uf st p i j)
  | Congr_l : (i,j,i1,i2,j1,j2 :ad)(ci,cj :(class nat*(list ad)))
      (trmi,trmj :term)
      (MapGet? uf i)=(SOME? (ci,((Node i1 i2),trmi)))->
      (MapGet? uf j)=(SOME? (cj,((Node j1 j2),trmj)))->
      (Eqvc uf st p i1 j1)->(uf_eqv uf i2 j2)->(Eqvc uf st p i j)
  | Congr_r : (i,j,i1,i2,j1,j2 :ad)(ci,cj :(class nat*(list ad)))
      (trmi,trmj :term)
      (MapGet? uf i)=(SOME? (ci,((Node i1 i2),trmi)))->
      (MapGet? uf j)=(SOME? (cj,((Node j1 j2),trmj)))->
      (uf_eqv uf i1 j1)->(Eqvc uf st p i2 j2)->(Eqvc uf st p i j)
  | Congr_b : (i,j,i1,i2,j1,j2 :ad)(ci,cj :(class nat*(list ad)))
      (trmi,trmj :term)
      (MapGet? uf i)=(SOME? (ci,((Node i1 i2),trmi)))->
      (MapGet? uf j)=(SOME? (cj,((Node j1 j2),trmj)))->
      (Eqvc uf st p i1 j1)->(Eqvc uf st p i2 j2)->(Eqvc uf st p i j).
```

Ainsi, toute preuve devra contenir une sous-preuve avec une congruence élémentaire. On va ensuite montrer que `(Eqvc uf st pending i j)\/ (uf_eqv uf pending m i j)` est conservée par `cc_rec`. Comme `pending` est vide à la fin de l'algorithme on n'aura plus de `Eqvc` (théorème `Abs_Eqvc_nil`), donc on aura l'égalité que l'on voulait montrer (figure 4.3)

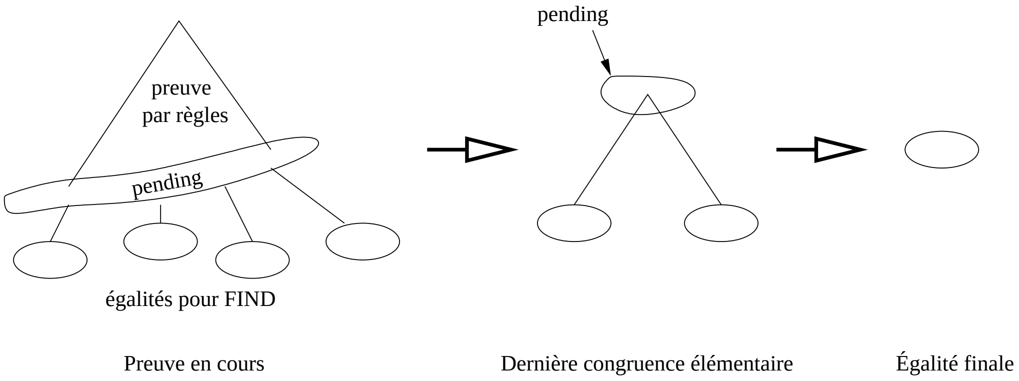


FIG. 4.3: Évolution d'une preuve

```
Theorem Abs_Eqvc_nil :
(uf :uft;st :stt;i,j :ad)(Eqvc uf (nil ad) i j)->False.

Theorem Or_Eqvc_nil :
  (uf :uft;st :stt;i,j :ad)
  ((Eqvc uf (nil ad) i j)\/(uf_eqv uf i j))->
  (uf_eqv uf i j).
```

Pour arriver à ce résultat, l'invariant clé est le suivant : à la fin d'une boucle, si deux nœuds sont de même signature, soit ils sont dans la même classe, soit tous les deux dans `pending`, soit l'un dans `pending` et l'autre dans la même classe que le terme de cette signature dans la table de signatures.

Prouver la conservation de l'invariant passe par deux sous-objectifs :
- Tous les nœuds du `pending` précédent on bien été fusionnés.
- Tous les nœuds dont la signature pourrait avoir changé sont dans `pending`

4.4 Réflexion totale : Bilan

La réalisation de preuves en Coq se révèle être un exercice pénible car contrairement à un programme en ML, toutes les fonctions doivent être totales, la partialité est alors modélisée par des types comme `option`.

Ceci ajouté à la lourdeur des preuves fait que par exemple la procédure `process_rec` contient à elle seule une douzaine de cas absurde liés aux filtrages sur `option`.

La seule preuve du théorème fondamental de UNION-FIND conduit à un script gigantesque. La manipulation des preuves d'accessibilité n'est pas toujours aisée à cause des dépendances, mais finalement l'utilisation massive du théorème d'*irrelevance* (`find_indep`)permet de vaincre les difficultés.

Curieusement, la terminaison s'est donc révélée étonnemment facile à prouver, et ceci grâce au caractère facilement observable du nombre de classes d'équivalence.

La correction et la complétude, par contre, s'avèrent plus difficiles à prouver en raison de l'utilisation de la table de signatures, et du fait que l'on utilise une structure «pseudo-mutable» qui impose de prouver un grand nombre d'invariants (structure des termes ...).

Enfin l'utilisation pratique de l'algorithme pour la réflexion s'avère très difficile : tout d'abord on doit pouvoir décider l'égalité syntaxique sur des termes Coq si on veut les entrer dans l'algorithme, ce qui parait très difficile. Et de toute façon le calcul des preuves d'accessibilité conduit à une explosion de la mémoire nécéssaire, ce qui fait que l'on épuise la mémoire pour le plus simple calcul.

Peut-être pourrait on obtenir du code réflexif qui arrive à mieux se réduire en utilisant un argument explicite de terminaison, sous la forme d'un entier que l'on passe comme argument à `find`, mais la preuve de validité de cette approche m'apparait assez complexe, car pour avoir un entier simple à calculer, il fut utilise rle cardinal d'une `map`. Mais alors il faudrait prouver que toute chaîne de la structure UNION-FIND est de longueur inférieure à ce cardinal.

Toutefois le code ML extrait se comporte bien et calcule assez rapidement. Cela amène une question concernant le système Coq : ne pourrait-on pas, par exemple normaliser le code extrait pour prouver l'égalité $\Delta(x) = true$? Cette méthode aurait l'avantage de supprimer tous ces calculs de preuves de terminaison.

Chapitre 5

Réflexion partielle : Une tactique Coq

En appliquant les principes de la réflexion partielle, j'ai implanté un algorithme de clôture de congruence en OCaml et je lui ai adjoint des annotations pour pouvoir ensuite générer des preuves.

En suite il a fallu réaliser un peu d'interfaçage pour transformer ce prototype en tactique, c'est-à-dire plonger dans les entrailles de Coq.

5.1 Génération de preuves

Les annotations ajoutées à la structure UNION-FIND portent deux informations supplémentaires sur les «pointeurs» :

```
type tag=Congr|Ax of identifier
type cl=Rep of int*(int list)|Eqto of int*(int*int*tag)
```

Le type `cl` est essentiellement le même que le type `Class` de Coq : on n'a ajouté que les indications d'union : les deux nœuds concernés (une union est réalisé sur les représentants, mais on ne sais plus quels sont les termes équivalents que l'on veux rendre égaux) et la raison : `Congr` pour une congruence ou `Ax n` pour l'axiome de nom n (`identifier` est une type interne de Coq).

La preuve est générée en utilisant les «flèches» de la structure UNION-FIND . On génère de façon mutuellement récursive les preuves d'une flèche et d'un «chemin». Si on sait prouver l'équivalence des deux extrémités d'une flèche, la preuve de l'équivalence des extrémités d'un chemin est la concaténation transitive des preuves des flèches qui le composent (voir figure 5.1).

Inversement, si on a une flèche, il faut prouver l'équivalence des extrémités de la flèche avec les nœuds dont l'équivalence a permis la liaison (ce sont des preuves de chemins ascendants), et il faut concaténer ces preuves avec celle de l'équivalence des deux noeuds : soit par un axiome , soit en allant chercher récursivement les preuves d´équivalence des sous-termes pour établir une congruence. C'est pour cela qu'il est nécéssaire de conserver de l'information sur les flèches (voir figure 5.2).

La terminaison de ce mécanisme est prouvable à l'aide de *timestamps* : si on date les flèches, la preuve d'un flèche utilise des sous preuves nécéssairement

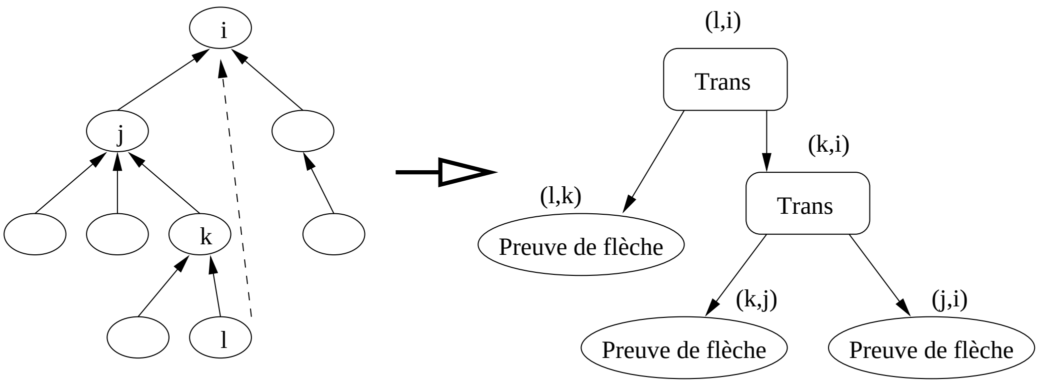


FIG. 5.1: génération de la preuve d'un chemin

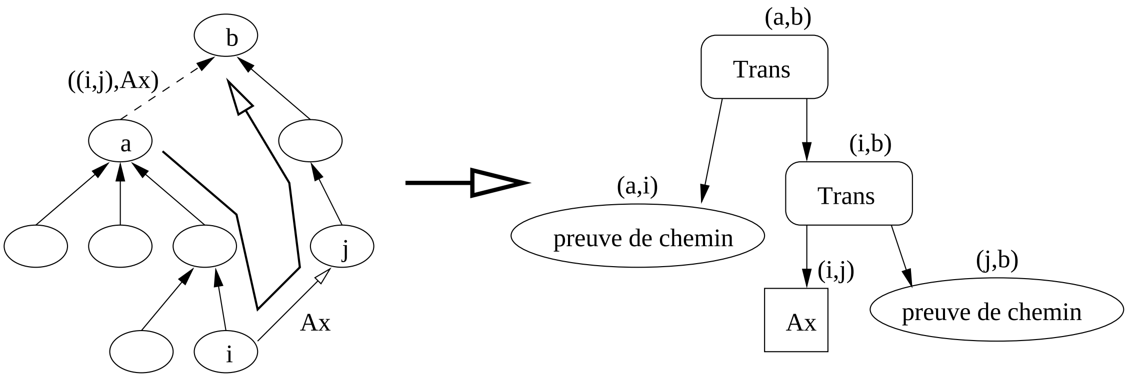


FIG. 5.2: génération de la preuve d'une flèche

antérieures, ce qui assure la terminaison.

Comme dans le mécanisme de normalisation (*cf* 4.2), on applique certaines règles de réduction des preuves pour de débarasser des règles de symétrie et réflexivité autant que possible.

5.2 Interfaçage avec Coq

La réalisation d'un tactique nécéssite l'implantation de trois choses :
- des procédures d'inspection de l'environnement en cours
- la procédure de décision et de génération de trace de preuve (voir plus haut)
- une fonction transformant cette trace en preuve

L'environnement est facile à inspecter, il suffit de vérifier que le but est une égalité, et de trouver toutes les hypothèses qui sont des égalités, ensuite on déstructure les termes du problème :

```
let t_make_app=(Array.fold_left (fun s t->App (s,t)))
```

```
let rec decompose_term t=
  match kind_of_term t with
    IsApp (f,args)->let targs=Array.map decompose_term args in
    (t_make_app (Symb f) targs)
  | _->(Symb t)
```

Pour plus de généralité, la tactique `CC` est capable d'utiliser indifféremment `eq : (A :Set)A->A->Prop` et `eqT : (A :Type)A->A->Prop`. Ceci permet de profiter de ce que tout objet de type `Set` ou `Prop` est aussi de type `Type`. d'ailleurs on dispose de deux théorèmes de conversion `eq2eqT` et `eqT2eq`.

Pour prouver la congruence, on utilise des lemmes intermédiaires :

```
Theorem Congr0 : (A,B :Type)(f,g :A->B)(x,y :A)f==g->x==y->(f x)==(g y).
Intros A B f g x y eq1 eq2;Rewrite eq1;Rewrite eq2;Reflexivity.
Save. Transparent Congr0.
...

Theorem Congr7 : (A,B :Set)(f,g :A->B)(x,y :A)f=g->x=y->(f x)=(g y).
Intros A B f g x y eq1 eq2;Rewrite eq1;Rewrite eq2;Reflexivity.
Save. Transparent Congr7.
```

Le choix de laisser `eq` et `eqT` coexister nous oblige à considerer huit cas suivant les sortes des sous termes.

Pour internaliser la preuve dans Coq, la méthode choisie a été d'utiliser des tactiques de haut niveau composées par les *tacticals* de L_{tac}, le langage de script pour les tactiques. Pour les règles, on utilise les tactiques internes suivantes :

Règle de démonstration	Tactique
réflexivité	`Reflexivity`
transitivité	`Transitivity`
symétrie	`Symmetry`
axiom	`Exact`
congruence	`Apply (Congr(dep) ...)`

Les trois premières tactiques présentent l'intérêt de pouvoir s'applique indifféremment à `eq` et `eqT`. Pour la règle d'axiome, on utilise éventuellement les théorèmes de conversion avant de donner l'axiome grâce á l'utilisation de `orelse`, et pour la congruence on regarde quel lemme il faut utiliser.

5.3 Types dépendants

Le problème est que congruence et types dépendents ne font pas bon ménage comme le montrent les exemples suivants :

```
Theorem dep :(A :Set)(P :A->Set)(f,g :(x :A)(P x))(x,y :A)
    (e :x=y)(e0 :(f y)=(g y))(f x)=(g x).

Theorem dep2 :(A,B :Set)
   (f :(A :Set)(b :bool)if b then bool else A->bool)
   (e :A==B)(f A true)=(f B true).
```

Ces théorèmes sont démontrables mais font intervenir des types dépendants : `CC` peut calculer un squelette de preuve, mais la preuve générée aura l'un des deux problème suivants : soit une transitivité avec un terme intermédiaire de type incompatible, soit une congruence ou les deux sous-termes gauches ont des types incompatibles.

Pour pouvoir résoudre un minimum de cas triviaux, on a quand même ajoute des lemmes comme :

```
Theorem Congr_dep3 :
    (A :Type ;P :(A->Set) ;f,g :((a :A)(P a)) ;x :A)f=g->(f x)=(g x).
```

En effet, si la sous-preuve de droite est une réflexivité, les deux fonctions auront des types compatibles bien que n'étant pas des type flèches. Ceci permet de résoudre des buts faisant intervenir des fonctions polymorphes simples, ou l'utilisation d'`eq_dep` n'est pas nécéssaire.

Ainsi on peut utiliser des fonctions polymorphes comme la fonction `length : (A :Set)(list A)->nat` à condition que les égalités ou elle intervient soient avec le même premier argument.

5.4 Un exemple

Pour citer un exemple, prenons celui de la couverture :

```
  A : Set
  a : A
  b : A
  f : A->A
  g : A->A->A
  H : a=(f a)
  H0 : (g b (f a))=(f (f a))
  H1 : (g a b)=(f (g b a))
  ============================
   (g a b)=a

T < CC.
Subtree proved!
```

```
T < Show Proof.
LOC :
Subgoals
Proof : [A :Set ;
        a,b :A ;
        f :(A->A) ;
        g :(A->A->A) ;
        H :(a=(f a)) ;
        H0 :((g b (f a))=(f (f a))) ;
        H1 :((g a b)=(f (g b a)))]
        (trans_eq A (g a b) (f (g b a)) a H1
          (trans_eq A (f (g b a)) (f a) a
            (Congr7 A A f f (g b a) a (refl_equal A->A f)
              (trans_eq A (g b a) (g b (f a)) a
                (Congr7 A A (g b) (g b) a (f a) (refl_equal A->A (g b))
                  H)
                (trans_eq A (g b (f a)) (f (f a)) a H0
                  (trans_eq A (f (f a)) (f a) a
                    (Congr7 A A f f (f a) a (refl_equal A->A f)
                      (sym_eq A a (f a) H)) (sym_eq A a (f a) H)))))
            (sym_eq A a (f a) H)))
```

5.5 Résultats

Pour tester la tactique, des essais ont été réalisés avec deux implantations différentes : l'une avec des tables de hachage (en conservant les indices) et l'autre avec des `map`. Les calculs ont été effectué sur un PC équipé d'un Processeur Intel Pentium III cadencé à 900 MHz et de 256 Mo de mémoire. Le résultats obtenus sont les suivants (un astérisque indique les essais où Coq est arrivé en manque de mémoire en lors de l'opération `Save` ou l'écriture du fichier compilé).

Nombre de nœuds	temps (hachage)	temps (`map`)
100	1.57s	1.56s
200	6.44s	6.39s
300	15.18s	15.19s
400	28.23s	28.28s
500	46.3s	46.27s
600	69.54s	69.75s
700	98.7s	99.07s
800	133.94s	134.18s
900	176.29s	176.47s
1000	226.62s	225.79s
1100	282.82s	282.39s
1200	346.35s*	346.72s*
1300	420.21s*	417.85s*

FIG. 5.3: Résultats des essais de la tactique CC.

On constate donc que la structure de stockage importe peu ici. Par contre, une tentative de suppression du système d'indices s'est avérée désastreuse : il y avait souvent besoin de comparer les termes malgré l'utilisation de hachage, ce qui fait que le temps de calcul explosait.

Malgré le problème d'incomplétude lié aux dépendances, la tactique `CC` donne des résultats très satisfaisants en terme de temps de calcul, et la simplification des traces de preuves donne des preuves de tailles et des structures très lisibles. Peut-être cette tactique sera-t-elle ajoutée au sytème Coq dans un futur proche.

Conclusion

La conclusion la plus immédiate est la suivante : dans le but d'implanter une tactique, la réflexion partielle est ici le moyen le plus adapté, malgré son défaut de terminaison, ou de terminaison correcte.

La réflexion totale est à réserver à des cas où la preuve de correction est effectivement plus courte que la preuve d'une instance moyenne du problème, car elle est beaucoup plus complexe dans sa mise en place que la réflexion partielle, voire impossible.

Pour une procédure de décision, la preuve de correction est souvent complexe, alors que les problèmes dépassent rarement une dizaine d'hypothèses, et le fait qu'on impose un langage fonctionnel pose des problème d'implantation : pallier l'absence de structures mutables, et d'exceptions est extrêmement contraignant.

En analysant le travail effectué, on s'aperçoit que beaucoup de temps est passé à prouver des lemmes plus liés à l'implantation qu'à l'algorithme lui-même, ce qui s'avère frustrant.

Cependant, une procédure de décision n'est certifiée que si on arrive a en obtenir une implantation réflexive, et dans ce but, avoir du code dont l'extraction peut s'exécuter rapidement est suffisant.

D'où une voie ouverte pour une certification moins fiable mais plus légère : partir de structures de données axiomatisées et prouver les algorithmes en se basant sur ces axiomes. Cet approche permet une bonne confiance dans le fonctionnnement de l'algorithme, qui peut être extrait par l'utilisation de liens (remplacer les axiomes par du code externe). Peut-être pourrait-ce être un bon compromis pour un algorithme qussi complexe que l'algorithme de Shostak.

Table des matières

Bibliographie

[1] Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpressions problem. *J. of the Association for Computing Machinery*, 27(4) :758–771, octobre 1980.

[2] E. Giménez. A tutorial on recursive types in Coq. Distribué comme partie de la documentation du système Coq., Mars 1996. Notes de cours rédigées pour la formation Coq qui s'est tenu à l'ENS-Lyon en Mars 1996.

[3] C. Gregory Nelson and Derek C. Oppen. A simplifier based on efficient decision algorithms. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages*, 1978.

[4] C. Gregory Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. of the Association for Computing Machinery*, 27(2) :356–364, avril 1980.

[5] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *TOPLAS : ACM Transactions on Programming Languages and Systems*, 1(2) :245–257, 1979.

[6] Xavier Rival and Jean Goubault-Larrecq. Experiments with finite tree automata in coq. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics.*, volume 2152 of *LNCS*. TPHOL'01, Springer, Septembre 2001.

[7] R. Shostack. An algorithm for reasoning about equality. *Communications of the ACM*, 21(2), juillet 1978.

[8] R. E. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1) :1–12, 1984.

[9] The Coq Development Team. *The Coq proof assistant reference manual.* INRIA-LRI-CNRS, `http ://pauillac.inria.fr/coq/doc/main.html`, 2001. version 7.0.