# Développement de procédures de décision certifiées.

Mémoire de magistère MMFAI

Pierre Corbineau

21 septembre 2001

# Table des matières

# Première partie

# Rapport de Magistère

# Introduction

Déjà trois années écoulées depuis mon entrée à l'École Normale Supérieure. Le temps de faire un bilan du chemin accompli pour mieux appréhender la suite.

Les expériences, plus que les connaissances, furent nombreuses. Ce mémoire se propose de vous présenter mes travaux passés, et mes travaux futurs.

Dans un premier temps, Je donnerai le détail de mon cursus, suivi d'une présentation moins administrative de mes trois années de magistère. Ensuite, je décrirai mon projet de recherche qui va se concrétiser bientôt sous la forme d'un sujet de thèse.

Enfin, le lecteur pourra trouver les rapports de mes stages de maîtrise, de magistère et de DEA, traces écrites de mes travaux.

# Chapitre 1

# cursus universitaire

## 1.1 Première année : 1998-1999

### 1.1.1 Licence d'informatique

- Algorithmique *J. Stern*
- Compilation *P. Cousot*
- Théorie de l'information *J. Vuillemin*
- Langages, calculabilité et complexité *P. Gastin*

### 1.1.2 Licence de mathématiques

- Logique *A. Louveau*
- Intégration et Probabilités 1 *F. Comets*
- Algèbre 1 *A. Beauville*

### 1.1.3 Maîtrise d'informatique

- Systèmes *J. Beigbeder*
- Sémantique des langages de programmation *P. Cousot*
- Réalisation de systèmes matériels *J. Vuillemin*
- Programmation orientée objet *P. Bertin, F. Bourdoncle*
- Calcul formel *D. Lazard*

Stage de maîtrise : *Preuve d'un programme impératif dans le système Coq* sous la direction de J. Despeyroux à l'INRIA Sophia-Antipolis.

## 1.2 Deuxième année : 1999-2000

### 1.2.1 Cours

Cours du DEA Systèmes Répartis sur les thèmes suivants :

- algorithmique répartie
- programmation de machines vectorielles et parallèles

– model-checking
– réseaux

### 1.2.2 Stage

Stage de magistère : *action abstractions for process algebra* sous la direction de R. Cleaveland à l'université SUNY at Stony Brook (NY, USA).

## 1.3 Troisième année : 2000-2001

DEA Programmation : Sémantique, Preuves et Langages

### 1.3.1 Tronc Commun

– Termes en logique du premier ordre *J.-P. Jouannaud*
– $\lambda$-calcul *J.-J. Lévy*
– Sémantique dénotationnelle *R. di Cosmo*
– Typage et Programmation *X. Leroy*
– Preuves en théorie des types *B. Werner*

### 1.3.2 Cours d'options

– Logique linéaire *R. di Cosmo, D. Kesner*
– Langages synchrones *M. pouzet*
– Calcul des constructions inductives *C. Paulin-Mohring,H. Herbelin et B.Barras*
– Introduction à l'interprétation abstraite *P. Cousot*
– Sémantique de jeux *V.Danos*
– Types et domaines *P.-L. Curien, G. Longo*

### 1.3.3 Stage

*Autour de la clôture de congruence avec Coq*, sous la direction de Jean Goubault-Larrecq, stage réalisé au LSV (ENS de Cachan).

# Chapitre 2

# Expériences, découvertes

Dès le début du magistère, c'est le développement de projets qui m'a le plus intéressé, et c'est cet aspect plus pratique de l'informatique qui m'a motivé pour choisir cette discipline par rapport aux mathématiques.

Pendant la première année,J'ai participé à plusieurs projets collectifs :

- développement d'un microprocesseur et de son assembleur pour le cours de Théorie de l'information. (avec B. Lerman)
- Design d'un circuit de cryptage CS-Cypher (avec S.Villemot et A.Kremer)
- Projet JAVA : Programmation d'un «jeu» client-serveur (collectif)

Cette première année m'a permis d'acquérir graduellement une assez bonne connaissance du système Unix. J'ai donc décidé de faire partie des tuteurs informatiques de l'ENS, pour pouvoir faire profiter les autres de mon expérience de ce système.

Le choix du stage de maîtrise a été motivé par l'envie de découvrir Coq, dont j'avais un peu entendu parlé dans mes cours. Il m'a montré comment fonctionnaient les unités de recherche.

Ma deuxième année fut placée sous le signe de l'informatique parallèle et distribuée. Au début de l'année, j'ai suivi à Jussieu quelques cours portant sur différents aspects du problème.

Ensuite je suis parti en stage aux États-Unis, où j'ai principalement travaillé sur le model-checking. Ce stage m'a en fait moyennement intéressé,mais m'a tout de même apporté énormément d'expérience concernant le fonctionnement du système universitaire américain, et m'a permis de visiter New-York avant la destruction des twin towers.

Comme la thématique de la vérification m'intéressait, j'ai choisi de m'inscrire au DEA Programmation pendant ma troisième année. J'ai aussi décidé de mettre à profit mon temps libre pour assurer la maintenance du réseau des élèves de l'ENS.

J'ai effectué mon stage de DEA à propos de procédures de décision et de réflexion avec Jean Goubault-Larrecq et j'ai soutenu mon mémoire le 17 septembre 2001.

# Chapitre 3

# Projet de recherche

La thématique vers laquelle je m'oriente est l'étude d'une méthodologie pour le développement de procédures de décision certifiées.

## 3.1 Environnements de preuve

De nombreux environnements de preuve sont développés dans la communauté informatique depuis pas mal de temps, ils ont tous en commun ceci : ils manipulent des objets représentant des propositions logiques en vue de permettre à l'utilisateur de réaliser des preuves. Les logiques sous-jascentes sont souvent trop puissantes pour pouvoir automatiser les preuves, aussi les preuves se font-elles de façon interactive.

Certains prouveurs, comme PVS, se contentent de réaliser des manipulations syntaxiques sur les termes, tandis que d'autres , comme Coq ou Isabelle, reposent sur des théories de types comme le Calcul des Constructions, et utilisent énormément l'isomorphisme de Curry-Howard.

Dans de tels formalismes les propositions logiques sont vues commes des types de termes représentant leur preuve. L'application d'une règle logique revient donc à ajouter un contexte au terme-preuve. La sureté de ce type de système ne repose plus que sur le type-checker.

## 3.2 Procédures de décision

On connait de nombreux fragments décidables des logiques usuelles, à commencer par la logique propositionnelle,ou l'arithmétique de Presburger, ou encore la théorie de l'égalité sans quantificateurs (clôture de congruence ). Des procédures de décision implantent ces algorithmes. Beaucoup d'environnements intègrent de telles procédures de décision à leur système de preuve.

Dans des systèmes comme PVS, le système se contente de croire aveuglément à ce que répond la procédure. Dans Coq, la procédure de décision doit fournir à Coq un terme-preuve, et c'est Coq qui va valider la preuve en vérifiant son type.

Un problème crucial se pose alors : comment être sûr de nos procédures de décision ?

Pour PVS, utiliser une procédure erronée passe totalement inaperçu, mais le système va alors considérer comme vrai une proposition fausse et devenir incohérent. Pour Coq, on sait qu'en cas de succès la preuve est garantie, mais bien sûr on peut avoir une procédure incomplète ou ne terminant pas, ou encore générant des preuves erronées que Coq va refuser.

Automatiser au maximum les prouveurs de théorèmes semble avoir été un enjeu suffisamment important ces dernières années pour qu'on sacrifie leur intégrité en leur adjoignant des packages non certifiés de model-checking ou de BDDs.

De plus pour Coq, la taille des preuves générées à vérifier est souvent un frein à l'intégration de nouvelles procédures de décision.

## 3.3 Réflexion et certification

Une approche possible dans cette démarche est la réflexion totale : au lieu de prouver une proposition correspondnant à une instance $x \in X$ du problème $P$, on code un programme $\Delta : x \to \text{bool}$ et on prouve $\forall x \in X, \Delta(x) = \text{true} \Rightarrow P(x)$.

La preuve de $P(x)$ se résumera donc à la réduction de $\Delta(x)$ et l'application du théorème, donc on ne fera que du type-checking et le terme de preuve sera de taille réduite. Avec la réflexion, on obtient donc une procédure de décision interne vérifiée et de terminaison garantie.

La réflexion souffre par contre d'un inconvénient majeur : dans Coq, il y a d'énormes contraintes sur les programmes, destinées à assurer la terminaison de ceux-ci. Ceci fait que plonger un programme dans l'environnememnt de preuves est très compliqué.

La réflexion partielle permet souvent elle aussi de réduire la taille des preuves : il s'agit de faire générer à un outil externe un script de preuve que l'environnement peut utiliser pour générer son terme-preuve.

Pour faire de la certification on peut aussi supposer l'existence de structures de données sans en fournir d'implantation.

## 3.4 Algorithme de Shostak

Une question classique concernant les procédures de décisions est : peut-on, à partir de procédures de décision sur des théories distinctes, obtenir une procédure de décision sur la théorie combinée ? Nelson et Oppen puis Shostak (*cf.* biblio. du rapport de DEA) ont proposé des méthodes. L'algorithme de Shostak, plus rapide que celui de Nelson et Oppen, a été publié sous plusieurs variantes à chaque fois incorrectes et ce depuis plus de 15 ans. Il serait donc intéressant d'arriver à certifier cet algorithme, quelle que soit la méthode employée.

# Conclusion

Après quelques hésitations, ces trois années de formation m'ont fait découvrir un panel assez large de domaines de recherche, et après quelques tentatives infructueuses, je pense avoir trouvé un thème correspondant à mes intérêts.

Tout ceci a été possible grâce à un nombre important de rencontres, au cours des stages notamment.

Je regrette par ailleurs que l'alternance des cours ait fait que je n'ai pas suivi le cours de Jean Goubault-Larrecq pendant ma première année.

Dans l'ensemble, je pense que le magistère a bien répondu à mes attentes, par la diversité de ses cours et l'originalité de son cursus.

Je remercie tous les membres des départements de mathématiques et d'informatique qui m'ont accompagné pendant ces trois années.

# Deuxième partie

# Rapport de Maîtrise

# Preuve d'un programme impératif dans le système Coq

Pierre Corbineau

Un stage réalisé à l'INRIA Sophia-Antipolis du 7 juin au 6 Août 1999, sous la direction de Joëlle Despeyroux.

# Introduction

Ce stage s'est déroulé au sein du projet Certilab, petit projet né de la scission en trois du projet Croap, et qui s'occupe de certification de logiciels.

Les premières semaines fûrent occupées à la découverte et à l'acquisition de la maîtrise de Coq, puis de la tactique Correctness. C'est à ce moment-là que la factorielle à été écrite et prouvée.

Ensuite il a fallu trouver un exemple de programme à prouver. En èpluchant la bibliographie sur la méthode B, plusieurs exemples se sont présentés et celui qui a été retenu est le système de controle d'ascenseurs.

Le reste du stage a été consacré à l'étude de cet exemple, au codage et à la preuve du programme choisi, qui a conduit à la découverte d'une petite erreur dans le B-book [1].

# Chapitre 4

# L'environnement d'aide à la preuve Coq

Coq est un environnement de preuve développé à l'INRIA. La première version, dûe à G. Huet et T. Coquand, date de 1984. Coq est basé sur la théorie des types, et plus précisément sur le *Calcul des Constructions Inductives* (CCI). Pour plus de détails sur le CCI, on se réfèrera à [3].

## 4.1 Description de l'utilisation du moteur de preuve

Le langage utilisé par Coq se sépare en deux parties :

*Gallina*, le langage des expressions logiques.

*The Vernacular*, pour les commandes et directives.

*Gallina* est un mélange des notations logiques usuelles avec des notations dérivées des langages fonctionnels. On trouvera ici la syntaxe des expressions les plus courantes.

| description | notation usuelle | syntaxe Coq |
|---|---|---|
| négation | $\neg A$ | `~A` |
| conjonction | $A \wedge B$ | `A/\B` |
| disjonction | $A \vee B$ | `A\/B` |
| implication | $A \Rightarrow B$ | `A->B` |
| équivalence | $A \Leftrightarrow B$ | `A<->B` |
| quantification universelle | $\forall x \in A, P$ | `(x :A)P` |
| quantification existentielle | $\exists x \in A, P$ | `Ex [x :A]P` |
| application | $f(x)$ | `(f x)` |
| abstraction | $\lambda(x : T).A$ | `[x :T]A` |
| fonction récursive | - | `Fix f [x :T] :T' :=A` |
| filtrage | - | `Cases x of A=>AA\|B=>BB end` |

FIG. 4.1: Syntaxe *Gallina* des expressions usuelles.

Les types des objets Coq sont aussi des expressions Gallina, et lorsque celle ci est un prédicat (de type Prop), alors l'objet est une preuve de ce prédicat.

*The Vernacular* est un ensemble de directives qui permet de :

1. Déclarer des variables d'un certain type. Si ce type est un prédicat, alors la déclaration pose ce prédicat en axiome, en effet un objet déclaré n'a pas de contenu mais seulement un type. C'est pourquoi les mots-clés de déclaration `Variable`, `Axiom` et `Hypothesis` sont équivalents.
   *exemple :*
   `Hypothesis a_positive :(gt a O).`
2. Définir des identificateurs dans l'environnement $\Gamma$. Les objets sont définis avec un contenu explicite : on peut ainsi définir un théorème en en fournissant une preuve explicite.
   *exemple :*
   `Definition deux :=(S(S O)).`
3. Definir un ensemble inductif. Cela se fait en donnant la liste des constructeurs de l'ensemble,avec leur type. Ensuite, Coq génère automatiquement les principes d'induction relatifs à cet ensemble.
   *exemple :*
   ```
   Inductive nat :Set :=
      O : nat
     |S : nat -> nat .
   ```
4. Definir une fonction récursive. Le mot-clé `Fixpoint` permet d'éviter d'utiliser le constructeur `Fix`. Les appels récursifs doit être effectué sur un sous terme de la variable sur laquelle porte la récursion.
   *exemple :*
   ```
   Fixpoint mod_2 [x :nat] : nat :=
     Cases x of
       (S (S n)) => (mod_2 n)
      |n         => n
     end .
   ```
5. Développer la preuve d'une proposition á l'aide de `Lemma` ou `Theorem`. Après avoir tapé `Theorem T :P.` , on passe en mode d'édition de preuve et Coq affiche le ou les buts à démontrer sous la forme $\frac{\mathcal{A}}{P}$ . Coq affiche au-dessus de la barre l'ensemble des hypothèses $\mathcal{A}$ et en-dessous le ou les buts (subgoals) à démontrer. Ces hypothèses peuvent être différentes pour chacun des sous-buts. L'utilisateur dispose aussi de l'environnement global $\Gamma$. L'édition de preuve consiste à modifier les hypotheses et les buts jusqu'à ce qu'il deviennent triviaux, et ce à l'aide de tactiques. On trouvera ci-après une liste des tactiques les plus fréquemment utilisées.

Dans ce qui précède, $\vdash$ désigne l'inférence de type ou, pour `Omega` , la démontrabilité (restreinte, car il faut "mâcher le travail").

Il existe deux autres tactiques fréquement utilisées, il s'agit de `Case` et `Induction` , qui s'appliquent aux types inductifs : résolution par cas, ou par induction.

De plus, Coq dispose d'une bibliothèque de théories assez importante, notamment des modules de logique classique, d'arithmétique unaire (`nat`), d'arithmétique binaire (`Z`), de théorie des ensembles et de résultats sur les relations binaires.

## 4.2 Exemple de preuve : transitivité de l'implication

Pour illustrer l'utilisation du moteur de preuve, voici un exemple simple de preuve en logique propositionnelle : La transitivité de l'implication.

```
Coq < Theorem trans_imp : (A,B,C :Prop)((A->B)/\(B->C))->(A->C).
1 subgoal

  ============================
   (A,B,C :Prop)(A->B)/\(B->C)->A->C

trans_imp < Intros A B C H H0.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : (A->B)/\(B->C)
  H0 : A
  ============================
   C
```

```
trans_imp < Elim H.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : (A->B)/\(B->C)
  H0 : A
  ============================
   (A->B)->(B->C)->C

trans_imp < Intros H1 H2.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : (A->B)/\(B->C)
  H0 : A
  H1 : A->B
  H2 : B->C
  ============================
   C

trans_imp < Apply H2.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : (A->B)/\(B->C)
  H0 : A
  H1 : A->B
  H2 : B->C
  ============================
   B

trans_imp < Apply H1.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : (A->B)/\(B->C)
  H0 : A
  H1 : A->B
  H2 : B->C
  ============================
   A
```

```
trans_imp < Exact H0.
Subtree proved!

trans_imp < Qed.
Intros A B C H H0.
Elim H.
Intros H1 H2.
Apply H2.
Apply H1.
Exact H0.

trans_imp is defined
```

Bien sûr il est possible d'aller plus vite dans ce cas, car il s'agit d'une tautologie :

```
Coq < Theorem trans_imp : (A,B,C :Prop)((A->B)/\(B->C))->(A->C).
1 subgoal

  ============================
   (A,B,C :Prop)(A->B)/\(B->C)->A->C

trans_imp < Tauto.
Subtree proved!

trans_imp < Qed.
Tauto.

trans_imp is defined
```

$$\frac{H{:}P\in\mathcal{A}}{P} \xrightarrow{\texttt{Exact H}} \texttt{Subtree Prooved !}$$

$$\frac{(?{:}P)\in\mathcal{A}}{P} \xrightarrow{\texttt{Assumption}} \texttt{Subtree Prooved !}$$

$$\frac{\mathcal{A}\vdash P}{P} \xrightarrow[\text{P prédicat sur les entiers, sans quantificateur}]{\texttt{Omega}} \texttt{Subtree Prooved !}$$

$$\frac{(\Gamma,\mathcal{A})\vdash(A,B{:}T)}{\neg(A=B)} \xrightarrow[\text{T type inductif, A et B avec des constructeurs distincts}]{\texttt{Discriminate}} \texttt{Subtree Prooved !}$$

$$\frac{\Gamma,\mathcal{A}}{P} \xrightarrow{\texttt{Absurd Q}} \frac{\Gamma,\mathcal{A}}{\neg Q}, \frac{\Gamma,\mathcal{A}}{Q}$$

$$\frac{\Gamma,\mathcal{A}}{P\wedge Q} \xrightarrow{\texttt{Split}} \frac{\Gamma,\mathcal{A}}{P}, \frac{\Gamma,\mathcal{A}}{Q}$$

$$\frac{\Gamma,\mathcal{A}}{P\vee Q} \xrightarrow{\texttt{Left}} \frac{\Gamma,\mathcal{A}}{P}$$

$$\frac{\Gamma,\mathcal{A}}{P\vee Q} \xrightarrow{\texttt{Right}} \frac{\Gamma,\mathcal{A}}{Q}$$

$$\frac{(\Gamma,\mathcal{A})\vdash(a{:}T)}{Ex[x{:}T]P} \xrightarrow{\texttt{Exists a}} \frac{\Gamma,\mathcal{A}}{P[x{:=}a]}$$

$$\frac{(H{:}A\rightarrow B)\in(\Gamma,\mathcal{A})}{B} \xrightarrow{\texttt{Apply H}} \frac{\Gamma,\mathcal{A}}{A}$$

$$\frac{\Gamma,\mathcal{A}}{P\rightarrow Q} \xrightarrow{\texttt{Intro H}} \frac{\Gamma\cup\{H{:}P\},\mathcal{A}\cup\{H{:}P\}}{Q}$$

$$\frac{\mathcal{A}}{B} \xrightarrow{\texttt{Cut A}} \frac{\mathcal{A}}{A\rightarrow B}, \frac{\mathcal{A}}{A}$$

$$\frac{(H{:}A=B)\in(\Gamma,\mathcal{A}}{C} \xrightarrow{\texttt{Rewrite H}} \frac{\Gamma,\mathcal{A}}{C[A{:=}B]}$$

$$\frac{((H{:}A=B),(G{:}C))\in(\Gamma,\mathcal{A})\ni(G{:}C)}{D} \xrightarrow{\texttt{Rewrite H in G}} \frac{\Gamma,\mathcal{A}\ni(G{:}C[A{:=}B])}{D}$$

$$\frac{(H{:}p)\in(\Gamma,\mathcal{A})}{Q} \xrightarrow{\texttt{Generalize H}} \frac{\Gamma,\mathcal{A}}{P\rightarrow Q}$$

$$\frac{\Gamma,\mathcal{A}\ni(H{:}\neg P)}{Q} \xrightarrow{\texttt{Elim H}} \frac{\Gamma,\mathcal{A}}{P}$$

$$\frac{(H{:}P\wedge Q)\in(\Gamma,\mathcal{A})}{R} \xrightarrow{\texttt{Elim H}} \frac{(\Gamma,\mathcal{A})}{P\rightarrow Q\rightarrow R}$$

$$\frac{(H{:}P\vee Q)\in(\Gamma,\mathcal{A})}{R} \xrightarrow{\texttt{Elim H}} \frac{\Gamma,\mathcal{A}}{P\rightarrow R}, \frac{\Gamma,\mathcal{A}}{Q\rightarrow R}$$

$$\frac{(H{:}Ex[x{:}T]P)\in(\Gamma,\mathcal{A})}{Q} \xrightarrow{\texttt{Elim H}} \frac{\Gamma,\mathcal{A}}{(x{:}T)(P->Q)}$$

$$\frac{\Gamma,\mathcal{A}}{P} \xrightarrow{\texttt{Replace A with B}} \frac{\Gamma,\mathcal{A}}{P[A{:=}B]}, \frac{\Gamma,\mathcal{A}}{A=B}$$

$$\frac{(H{:}P)\in\mathcal{A}}{P} \xrightarrow{\texttt{Clear H}} \frac{\Gamma,\mathcal{A}-\{H{:}P\}}{Q}$$

FIG. 4.2: Liste des principales tactiques

# Chapitre 5

# La tactique Correctness

Correctness a été développée par J.-C. Filliatre dans le cadre de sa thèse [4] au LRI (Paris XI) dont le sujet était *Preuve de programmes impératifs en théorie des types* . Il a soutenu sa thése le 12 juillet 1999, alors même que je me trouvais en stage.

Correctness n'est pas à proprement parler une tactique mais une commande Vernacular qui prend en argument un programme annoté et génère les obligations de preuve qui manquent à la certification du programme.

## 5.1 La syntaxe

Voici la syntaxe des programmes annotés :

Cette syntaxe est très proche de Caml-light, mais il y a quelques restrictions : les collisions de noms de variables dans un même domaine peuvent produire des résultats exotiques, et il est nécéssaire de spécifier le type du résultat des fonctions récursives.

De plus, pour les objets pouvant ne pas terminer ( `let rec` et `while ... do`), il est nécéssaire de spécifier un `variant` qui doit décroître strictement pour un ordre bien fondé, et Correctness génèrera les obligations de preuve correspondant à la décroissance stricte du variant et si l'ordre n'est pas un ordre classique (lt sur nat ou Zwf[1] sur Z), il faudra prouver qu'il est bien fondé.

Les pré- et post-conditions se placent à la fa con des triplets de Hoare, ou à l'aide de `assert` inséré dans un bloc.Toutefois,il convient de prendre certaines précautions car une post-condition ne s'applique qu'au bloc de programme le plus petit possible, c'est pourquoi il faut systématiquement rajouter des parenthèses autour des constructions telles que `let ... in`, car lorsqu'il manque des post-conditions, Correctness fournit des obligations non prouvables.

Correctness fournit aussi du sucre syntaxique, notamment pour les expressions arithmétiques qui sont retranscrites en type Z, et appelle á un usage massif d'`Omega` dans les preuves.

De plus, pour ce qui est des tests, Correctness peut calculer automatiquement, dans la plupart des cas, une valeur du type `{ b :bool | if b then A else B }` qui permet d'extraire les propositions correspondant aux valeurs vraie et fausse du test, et ce à partir de la disjonction constructive `{A}+{B}`, grâce à la fonction

[1] `(Zwf n a b)` est défini par $(n < a) \wedge (n < b) \wedge (a < b)$

`bool_of_sumbool`. Dans les cas trop compliqués,il faut annoter le test par if result then A else B . En outre, un test ne peut pas avoir d'effets de bord dans Correctness, sauf sur des références locales.

Enfin, pour travailler avec des tableaux, on dispose du type Array, Défini dans le fichier arrays.v fourni en annexe.

L'utilisateur dispose également d'un mécanisme de labels pour exprimer des relations entre les valeurs des variables à cet endroit et avant. La valeur `result` désigne la dernière valeur calculée (Programs définit le type unit : `Inductive Set unit := tt :Unit.`).

## 5.2 Le principe de fonctionnement

L'objet n'est pas ici d'être complet, on se réfèrera à [4] pour plus de détails.

Correctness génère des obligations de preuves qui ressemblent a celles de la logique de Hoare, pourtant le principe en est assez différent. Correctness contient un mécanisme d'inférence de type qui tient compte des effets de bord et permet ainsi de minimiser le nombre de variables intervenant dans les preuves.

Correctness construit ainsi par induction syntaxique une preuve de $\forall x, P(x) \Rightarrow \exists y : y = [\mathcal{P}]x \wedge Q(y)$, à l'aide d'une interprétation $\widehat{M}$ qui calcule à la fois l'environnement local, la traduction fonctionnnelle, et l'obligation de preuve relative à un morceau de programme $M$ .(voir figure 5.2

Les points d'interrogation sont des "trous" dans la preuve qui deviendront des obligations de preuve. Pour la boucle `while`, le terme $Y_{R,\pi}$ est un opérateur de point fixe qui prend en argument une relation $R$ et la preuve $\pi$ qu'elle est bien fondée. Les autres obligations sont alors : la décroissance de $\phi$, la conservation de l'invariant, et la validité initiale de l'invariant.

On remerquera que les règles fournies supposent effectivement l'absence d'effets de bord des tests.

## 5.3 Un exemple : une factorielle impérative

Pour illustrer plus précisément l'utilisation de Correctness, commen cons par un exemple simple : la preuve d'une factorielle impérative.

Pour écrire la spécification, écrivons la même fonction en Coq.

```
Fixpoint fact [n : nat] : nat :=
  Cases n of
    O => (S O)
    | (S m) => (mult n (fact m))
end.
```

Il faut aussi une version décidable de `lt` (cf **??**).

```
Definition lt_bool :=[x,y :nat](bool_of_sumbool?? (lt_ge_dec x y)).
```

Le programme :

```
Correctness i_fact fun (N :nat)->
let f=ref O in
```

```
let m=ref O in
 begin
  f :=(S O);
  m :=N;
  while (lt_bool O !m) do
    {
      invariant
        ((fact N)=(mult f (fact m)))
      variant
        m
      for
        lt
    }
    f :=(mult !f !m);
    m :=(pred !m)
  done;
 !f
end
{ (result = (fact N)) }.
```

On obtient alors trois buts à démontrer :

```
i_fact < Show 1.
subgoal 1 is :

  N : nat
  phi0 : nat
  f1 : nat
  m1 : nat
  Variant1 : phi0=m1
  Pre2 : (fact N)=(mult f1 (fact m1))
  resultb : bool
  Test1 : (lt (0) m1)
  ============================
   (lt (pred m1) m1)/\(fact N)=(mult (mult f1 m1) (fact (pred m1)))

i_fact < Show 2.
subgoal 2 is :

  N : nat
  ============================
   (fact N)=(mult (1) (fact N))

i_fact < Show 3.
subgoal 3 is :

  N : nat
  m1 : nat
  f1 : nat
  result : unit
```

```
Post2 : (fact N)=(mult f1 (fact m1))/\(ge (0) m1)
============================
 f1=(fact N)
```

Les deux premiers buts sont relatifs à l'invariant de boucle, ils expriment le fait qu'il est vérifié d'une itération à l'autre (subgoal 1) et avant la première (subgoal 2). Le troisième concerne le résultat (on remarquera dans les hypothèses l'invariant de boucle assorti de la condition de sortie. Dans ce cas précis, Correctness utilise par défaut l'ordre bien fondé `lt` sur `nat`, et nous évite cette preuve.

Une fois la preuve réalisée, il est possible d'en réaliser une extraction pour obtenir un programme OCaml correspondant. Ce fichier source est disponible en appendice.

| | | |
|---|---|---|
| *prog* | : := | { *predicate* } * *statement* [ { *predicate* } ] |
| *statement* | : := | *expression* |
| | \| | *identifier* := *prog* |
| | \| | *identifier* [ *expression* ] := *prog* |
| | \| | let *identifier* = ref *prog* in *prog* |
| | \| | if *prog* then *prog* [ else *prog* ] |
| | \| | while *prog* do *loop_annot* *block* done |
| | \| | begin *block* end |
| | \| | let *identifier* = *prog* in *prog* |
| | \| | fun *binders* -¿ *prog* |
| | \| | let rec *identifier* *binder* : *value_type* { variant *wf_arg* } = *prog* [ in *prog* ] |
| | \| | ( *prog* *prog* ) |
| *expression* | : := | *identifier* |
| | \| | ! *identifier* |
| | \| | *identifier* [ *expression* ] |
| | \| | *integer* |
| | \| | ( *expression*$^+$ ) |
| *block* | : := | *block_statement* [ ; *block* ] |
| *block_statement* | : := | *prog* |
| | \| | label *identifier* |
| | \| | assert { *predicate* } |
| *binders* | : := | ( *identifier*,...,*identifier* : *value_type* )$^+$ |
| *loop_annot* | : := | { invariant *predicate* variant *wf_arg* } |
| *wf_arg* | : := | *cic_term* [ for *cic_term* ] |
| *predicate* | : := | *cic_term* [ as *identifier* ] |

FIG. 5.1: Syntaxe des programmes annotés

$M \equiv \{P\}\ E\ \{Q\}$ :

$\quad \widehat{M} = \lambda \mathbf{x}.\lambda h : P.(\{\}, E[!x \leftarrow \mathbf{x}.x], ?)$

$M \equiv \{P\}\ x_0 := M_1\ \{Q\}$ :

$\quad \widehat{M} = \lambda \mathbf{x}.\lambda h : P. \text{let } (\mathbf{x_1}, v, q_1) = (\widehat{M_1}\ \mathbf{x}\ ?) \text{ in } (\mathbf{x_1} \oplus \{x_0 = v\}, \text{void}, ?)$

$M \equiv \{P\}\ M_1 ; M_2\ \{Q\}$ :

$\quad \widehat{M} = \lambda \mathbf{x}.\lambda h : P. \text{let } (\mathbf{x_1}, -, q_1) = (\widehat{M_1}\ \mathbf{x}\ ?) \text{ in}$

$\qquad \text{let } (\mathbf{x_2}, v, q_2) = (\widehat{M_2}\ (\mathbf{x} \oplus \mathbf{x_1})\ ?) \text{ in}$

$\qquad (\mathbf{x_1} \oplus \mathbf{x_2}, v, ?)$

$M \equiv \{P\}\ if\ E\ then\ M_1\ else\ M_2\ \{Q\}$ :

$\quad \widehat{M} = \text{if}(\widehat{E}\ \mathbf{x})\,\text{then}$

$\qquad \text{let}(\mathbf{x_1}, v, q_1) = ((\widehat{M_1}\ \mathbf{x}\ ?)\,\text{in}(\mathbf{x} \oplus \mathbf{x_1}, v, ?)$

$\quad\ else$

$\qquad \text{let}(\mathbf{x_1}, v, q_1) = ((\widehat{M_2}\ \mathbf{x}\ ?)\,\text{in}(\mathbf{x} \oplus \mathbf{x_1}, v, ?)$

$M \equiv \{P\}\ while_{\Phi,R}\ E\ do\ M_1\ done\ \{Q\}$

$\quad \widehat{M} = \lambda \mathbf{x}.((Y_{R,?}\ \lambda w.\lambda \mathbf{x}.\lambda \psi.\lambda h : P.$

$\quad\ \text{if}(\widehat{E}\ \mathbf{x})\,\text{then}$

$\qquad \text{let } (\mathbf{x_1}, -, q) = (\widehat{M_1}\ \mathbf{x}\ ?) \text{ in}(w\ (\mathbf{x} \oplus \mathbf{x_1})\ \phi(\mathbf{x} \oplus \mathbf{x_1})??)$

$\quad\ \text{else}$

$\qquad (\mathbf{x}, \text{void}, ?))\mathbf{x}\phi(\mathbf{x}))$

$M \equiv \{P\}\ let\ x_0 = ref\ M_1\ in\ M_2\ \{Q\}$

$\quad \widehat{M} = \lambda \mathbf{x}.\lambda h : P. \text{let } (\mathbf{x_1}, v_1, q_1) = (\widehat{M_1}\ \mathbf{x}\ ?) \text{ in}$

$\qquad \text{let } (\mathbf{x_2}, v_2, q_2) = (\widehat{M_2}\ (\mathbf{x} \oplus \mathbf{x_1} \oplus \{x_0 = v_1\})\ ?) \text{ in}$

$\qquad (\mathbf{x_1} \oplus (\mathbf{x_2} \setminus x_0), v_2, ?)$

$M \equiv \{P\}\ fun\ (x : V) \to M_1\ \{Q\}$

$\quad \lambda h : P.((\lambda x : \widehat{V}.\widehat{M_1}, ?$

$M \equiv \{P\}\ (M_1 M_2)\ \{Q\}$

$\quad \widehat{M} = \lambda \mathbf{x}.\lambda h : P. \text{let } (\mathbf{x_1}, a, q_1) = (\widehat{M_2}\ \mathbf{x}\ ?) \text{ in}$

$\qquad \text{let } (\mathbf{x_2}, f, q_2) = (\widehat{M_1}\ (\mathbf{x} \oplus \mathbf{x_1})\ ?) \text{ in}$

$\qquad \text{let } (\mathbf{x_3}, v, q_3) = (f\ a\ (\mathbf{x} \oplus \mathbf{x_1} \oplus \mathbf{x_2})\ ?) \text{ in}$

$\qquad (\mathbf{x_1} \oplus \mathbf{x_2} \oplus \mathbf{x_3}, v, ?)$

FIG. 5.2: Interprétation des programmes annotés

# Chapitre 6

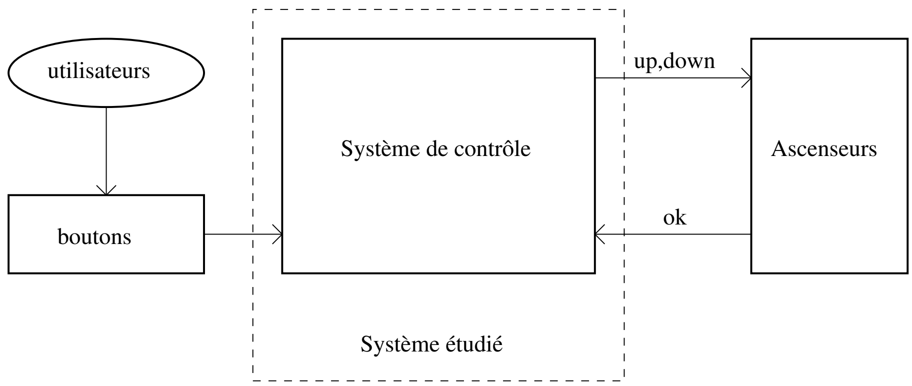
# Preuve sur un exemple

La tactique Correctness a déjà été utilisée pour réaliser les preuves de programmes “classiques” de l’algorithmique, en particulier des programmes de tri (voir l’article de J.-C. Filliâtre et N. Magaud [5]). Ici, l’objectif a été de certifier un programme de type plus proche des programmes utilisés par l’industrie. Mon choix s’est porté sur un algorithme pour une machine de contrôle d’un système d’ascenseurs [2], qui était fourni en exemple dans le B-book [1].

## 6.1 Les ascenseurs : Presentation

Le dispositif à programmer est la machine de commande des ascenseurs. On suppose qu’a chaque étage il y a deux boutons (en tout) : un pour monter et un pour descendre (excepté au rez-de-chaussée et au dernier étage). De même, dans chaque ascenseur, il y a un bouton pour chaque étage. La solution retenue est celle de minimiser le nombre de changements de sens, tout en faisant fonctionner les ascenseurs en concurrence : quand un utilisateurs appelle un ascenseur, tous les ascenseurs prennent en compte sa demande et le premier arrivé satisfait la requête.

Dans ce système, on suppose que les ascenseurs n’imforment pas le système de leur position mais signalent simplement le moment où l’action précédente est sur le point d’être terminée, et donc l’ascenseur est près à recevoir l’ordre suivant.



La spécification des ascenseurs a été écrite en B, c'est-à-dire sous la forme d'une machine virtuelle définie par son initialisation et ses transitions.

La méthode B est une méthode de vérification formelle de programmes basée sur la notion de raffinement, c'est-á-dire que les machines et opérations sont exprimées de fa con de plus en plus concrète jusqu'à l'obtention de l'implémentation finale, en prenant soin de préserver les propriétés de la spécification à chaque étape.

Voici le corps principal de la machine.

MACHINE

$Lift$

SETS

$LIFT$ ;
$DIRECTION = \{up, dn\}$

CONSTANTS

$ground, top$

PROPERTIES

$ground \in \mathrm{NAT} \ \wedge$
$top \in \mathrm{NAT} \ \wedge$
$ground < top$

DEFINITIONS

$FLOOR \mathrel{\widehat{=}} ground..top$

VARIABLES

$moving, floor, dir, in, out$

INVARIANT

$moving \subseteq LIFT \wedge$
$floor \in LIFT \rightarrow FLOOR \wedge$
$dir \in LIFT \rightarrow DIRECTON \wedge$
$in \in FLOOR \leftrightarrow DIRECTION \wedge$
$out \in LIFT \leftrightarrow DIRECTION \wedge$
$(ground \mapsto dn) \notin in \wedge$
$(top \mapsto up) \notin in \wedge$
$moving \ntriangleleft (out \cap floor) = \emptyset \wedge$
$in \cap \mathrm{ran}(moving \ntriangleleft (floor \otimes dir)) = \emptyset$

INITIALIZATION

$in, out, moving := \emptyset, \emptyset, \emptyset$
$floor, dir := LIFT \times \{ground\}, LIFT \times \{up\}$

FIG. 6.1: Corps principal du système

Les notations $A \mapsto B$ , $A \leftrightarrow B$ , $\mathrm{ran}(f)$ , $A \ntriangleleft (f)$ et $f \otimes g$ désignent respectivement l'ensemble des fonctions de A dans B, l'ensemble des relations (produit cartésien) entre A et B, l'image directe de f restreint au complémentaire de A, et la fonction obtenue en prenant pour images les couples d'images par f et g (f et g sont définies sur le même ensemble).

Les variables *floor* et *dir* représentent respectivement la direction et la position des ascenseurs. Les variables *in* et *out* représentent l'état (allumé ou éteint) des boutons aux étages et dans les ascenseurs.

L'invariant nous assure que la position des ascenseurs reste cohérente avec le modèle physique, mais aussi que les boutons censés ne pas exister ne sont jamais allumés, ou encore que lorsqu'un ascenseur est arrêté au bon étage et, si besoin est, dans la bonne direction, alors le bouton correspondant est éteint. Ceci nous assure de l'equivalence entre boutons allumés et requêtes non satisfaites.

Les opérations (transitions) qui suivent correspondent aux appuis sur les boutons.

OPERATIONS

Request_Floor$(l, f) \mathrel{\widehat{=}}$
PRE
$l \in LIFT \land$
$f \in FLOOR \land$
$(l \notin moving \Rightarrow floor(l) \neq f)$
THEN
$out := out \cup \{l \mapsto f\}$
END ;

Request_Lift$(f, d) \mathrel{\widehat{=}}$
PRE
$f \in FLOOR \land$
$d \in DIRECTION \land$
$(f, d) \neq (ground, dn)$
$(f, d) \neq (top, up)$
$(f, d) \notin \mathrm{ran}(moving \ntriangleleft (floor \otimes dir)$
THEN
$out := out \cup \{l \mapsto f\}$
END ;

Fig. 6.2: Opérations de l'utilisateur

On remarque que, pour préserver l'invariant, on a introduit le cas suivant : on ne peut pas appeler un ascenseur s'il y en a déjà un à l'étage, ou bien, une fois dans l'ascenseur, demander à aller à l'étage où il se trouve.

Maintenant, il faut définir certaines propriétés qui servent à déterminer quelle action effectuer.

DEFINITIONS

$attracted\_up(l) \mathrel{\widehat{=}} (dom(in) \cup out[\{l\}]) \cap ((floor(l)+1)..top) \neq \emptyset\,;$

$attracted\_dn(l) \mathrel{\widehat{=}} (dom(in) \cup out[\{l\}]) \cap (ground..(floor(l)-1)) \neq \emptyset\,;$

$can\_continue\_up(l) \mathrel{\widehat{=}} (l \mapsto floor(l)) \notin out \wedge$
$(floor(l) \mapsto dir(l)) \notin in \wedge$
$attracted\_up(l)\,;$

$can\_continue\_dn(l) \mathrel{\widehat{=}} (l \mapsto floor(l)) \notin out \wedge$
$(floor(l) \mapsto dir(l)) \notin in \wedge$
$attracted\_dn(l)\,;$

Fig. 6.3: Définition des propriétés intermédiaires

Enfin on trouvera à la page suivante les transitions de mouvement des ascenseurs. On notera que pour un ascenseur donné, il n'y a qu'une transition possible à un moment donné.

OPERATIONS (suite)

Continue_up($l$) $\widehat{=}$
PRE
$l \in moving \wedge$
$dir(l) = up \wedge$
$can\_continue\_up(l)$
THEN
$floor(l) := floor(l) + 1$
END ;
Continue_dn($l$) $\widehat{=}$ ... ;

Stop_up($l$) $\widehat{=}$
PRE
$l \in moving \wedge$
$dir(l) = up \wedge$
$\neg can\_continue\_up(l)$
THEN
$moving := moving - \{l\}$
$out := out - l \mapsto floor(l)$
$in := in - \{floor(l) \mapsto dir(l)\}$
END ;
Stop_dn($l$) $\widehat{=}$ ... ;

Depart_up($l$) $\widehat{=}$
PRE
$l \in LIFT - moving \wedge$
$dir(l) = up \wedge$
$attracted\_up(l)$
THEN
$moving := moving \cup \{l\}$
$floor(l) := floor(l) + 1$
END ;
Depart_dn($l$) $\widehat{=}$ ... ;

Change_up_to_down($l$) $\widehat{=}$
PRE
$l \in LIFT - moving \wedge$
$dir(l) = up \wedge$
$\neg attracted\_up(l) \wedge$
$attracted\_dn(l)$
THEN
$in := in - \{floor(l) \mapsto dn\}$
$dir(l) := dn$
END ;
Change_dn_to_up($l$) $\widehat{=}$ ... ;

END

Pour réaliser une bonne implémentation, il a fallu simuler les évènements extérieurs, mais aussi les évenements internes (signaux des ascenseurs). Comme le comportement du système ne dépend pas d'une notion de délai, mais uniquement d'une notion d'ordre séquentiel des évènements, les entrées seront simulées par une fonction `Z -> event`, où event est soit un bouton (couple (étage,direction) pour les boutons fixes et (ascenseur,étage ) pour les boutons dans les ascenseurs, ou alors un signal d'un ascenseur (voir fichier utils.v en annexe). Le calcul des valeurs de `attracted_up` et `attracted_dn` et réalisé par une boucle `while` car on ne dispose pas de primitives de type `for ... to` .

Le programme sera donc une boucle `while` avec un compteur de temps, dont l'implémentation est fournie en annexe (fichier lift.v).

## 6.2 Les objectifs de preuve

Pour certifier cette implémentation, il est nécéssaire de prouver plusieurs choses :

1. L'invariant principal est préservé par la boucle.
2. Lorsqu'un utilisateur appuie sur un bouton, il voit sa requête satisfaite en un nombre fini d'étapes.

Pour prouver ce dernier objectif, qui est une propriété de *liveness* du système, on va adopter la stratégie suivante : pour les boutons extérieurs, pour chaque ascenseur, on va exhiber une quantité qui décroît strictement pour un ordre bien fondé, et ce pour chaque bouton extérieur allumé et à chaque étape de l'ascenseur concerné. Pour les boutons intérieurs on adoptera la même stratégie pour les boutons situés dans cet ascenseur.

Pour que cette stratégie soit valable, il faut supposer que l'on effectue "assez" d'étapes d'ascenseur pour que la quantité puisse décroître, ce qui revient á supposer que "à chaque instant, pour un ascenseur donné, il existe un instant ultérieur pour lequelle l'évènement d'entrée sera le signal de cet ascenseur". Ainsi, si les ascenseurs continuaient à fonctionner indéfiniment (dans notre cas on doit fixer un instant d'arrêt arbitraire pour prouver que notre boucle `while` termine), chaque ascenseur effectuerait une infinité d'étapes, et donc un bouton restant allumé nous prouverait que l'ordre n'est pas bien fondé, d'où par absurdité, la propriété de *liveness* est vérifiée.

Le B-book [1] nous fournit une "distance" qui parait convenir à cette preuve, c'est la distance maximale que doit parcourir l'ascenseur ( étage f, direction d ) avant d'arriver à l'objectif ( étage g, direction e). Pour une définition plus formelle, voir figure 6.4.

Malheureusement, il se trouve que le B-book contient une erreur : $dist$ seule ne convient pas dans le cas de Stop_up et Stop_dn, car ces opérations ne changent ni la position, ni la direction de l'ascenseur, ce qui fait que $dist$ reste inchangée pour les boutons rèstés allumés (il est possible qu'il en reste : appel pour monter à partir des 1er et deuxième etages avec un seul ascenseur, il s'arrête d'abord au premier mais le bouton du deuxième reste allumé).

Etant donné que seule la valeur de $moving$ change (en dehors de $in$ et $out$) lors de Stop_up ou Stop_dn, on va utiliser le couple $(dist, moving(l))$ avec l'ordre lexicographique ( on prend $true > false$ pour les booléens ). En effet les oper-

| Condition | $d$ | $e$ | $dist(f, d, g, e)$ |
|---|---|---|---|
| $f > g$ | $up$ | $up$ | $2 \times (top - ground) - f + g + 2$ |
| | | $dn$ | $2 \times top - f - g + 1$ |
| | $dn$ | $up$ | $f + g + 1 - 2 \times ground$ |
| | | $dn$ | $f - g$ |
| $f = g$ | $up$ | $up$ | $0$ |
| | | $dn$ | $2 \times top - f - g + 1$ |
| | $dn$ | $up$ | $f + g + 1 - 2 \times ground$ |
| | | $dn$ | $0$ |
| $f < g$ | $up$ | $up$ | $g - f$ |
| | | $dn$ | $2 \times top - f - g + 1$ |
| | $dn$ | $up$ | $f + g + 1 - 2 \times ground$ |
| | | $dn$ | $2 \times (top - ground) - g + f + 2$ |

FIG. 6.4: Table des valeurs de la fonctions dist

| Condition | $d$ | $ddist(f, d, g)$ |
|---|---|---|
| $f > g$ | $up$ | $2 \times top - f - g + 1$ |
| | $dn$ | $f - g$ |
| $f = g$ | $up$ | $0$ |
| | $dn$ | $0$ |
| $f < g$ | $up$ | $g - f$ |
| | $dn$ | $f + g + 1 - 2 \times ground$ |

FIG. 6.5: Table des valeurs de la fonctions ddist

ations Depart_up et Depart_dn modifient la position de l'ascenseur et mettent *moving* à true.

Pour le cas des boutons intérieurs de l'ascenseur, la fonction sera la distance *ddist* (voir fig. 6.5). Il faudra toujours utiliser le produit lexicographique.

On pourra trouver en annexe les implémentations de la machine virtuelle et des fonctions distance.

# Conclusion

Le temps m'a manqué pour mener à bien la totalité des objectifs que je m'étais fixés, mais le degré d'avancement atteint permet de conclure à la faisabilité dans l'absolu de tels travaux.

Mais il ne faut pas se cacher que ce genre de preuve est long et fastidieux à réaliser, et ce d'autant plus que la moindre modification du code du programme peut s'avérer désastreuse pour les preuves en cours : tout le domaine de noms peut s'en retrouvé modifié, et ce malgré le confort relatif apporté par le mécanisme de labels de Correctness.

De plus, une utilisation efficace de Correctness nécéssite d'en connaître le fonctionnement détaillé, connaîssance que je ne possède pas encore.

Enfin, au stade oú je suis arrivé, environ 150 Mo sont nécéssaire pour un déroulement correct de la compilation, mais le temps de calcul reste raisonnable pour cette version (environ 5 min pour calculer les obligations de preuve). J'avais écrit une version où il n'y avait que des variables globales, et le calcul durait environ 45 min, ce qui laisse présager d'une évolution rapide du volume de calcul nécéssaire pour des projets de taille industrielle.

D'un point de vue plus technique, Correctness utilise des n-uplets sous forme de types inductifs appelés `Tuple_`$n$. Bien évidemment il ne sont pas tous définis par défaut, mais seulement pour les petites valeurs de n, ce qui crée des erreurs incompréhensibles pour le néophyte, et il serait souhaitable que Correctness définisse soi-même les n-uplets requis.

Pour conclure, disons qu'un projet de certification par Coq est réalisable, mais il faut garder à l'esprit que Correctness est un outil relativement jeune et qu'il y a des chances pour qu'il gagne en fiabilité et en clarté (plus de documentation) dans les années à venir.

# Remerciements

Tout d'abord je tiens à remercier Joëlle Despeyroux sans qui ce stage n'aurait pas eu lieu. Ensuite, je remercie Jean-Christophe Filliâtre pour sa disponibilité et sa patience en période de bouclage de thèse. Je remercie également Patrick Loiseleur de m'avoir débloqué alors que J.-C. Filliâtre était en vacances. Enfin, je remercie Francis Castagnac, l'ingénieur-systéme, pour ses dépannages trop fréquemment nécéssaires.

# Bibliographie

[1] J.-R. Abrial. *The B-book : assigning programs to meanings.* Cambridge University Press, 1996.

[2] J.-R. Abrial. *A lift control system, in [1]*, chapter 8.3, pages 358–369. Cambridge University Press, 1996.

[3] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* PhD thesis, Université Denis Diderot (Paris VII).

[4] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types.* PhD thesis, Université Paris-Sud, Juillet 1999.

[5] J.-C. Filliâtre and N. Magaud. Certification of Sorting Algorithms in the System Coq. In *Theorem Proving in Higher Order Logics : Emerging Trends*, 1999. To appear.

# Troisième partie

# Stage de deuxième année

# Action abstractions for process algebras

Pierre Corbineau-Rance Cleaveland
SUNY at Stony Brook

Spring-Summer 2000

# Introduction

We describe the syntax and semantics of ECCS to illustrate the use of the $\mu$-calculus model-checking of modal transitions systems, which we describe next, along with a new definition of formula satisfaction.

# Chapitre 7

# Enhanced CCS

In this chapter we define ECCS, an extension of the CCS syntax (and semantics) that will be easier to handle. You may notice that the restriction has disappeared from the syntax, it is because the renaming to an empty set has the same meaning and gives a more homogenous framework.

## 7.1 LTS, modal $\mu$-calculus

**Définition 7.1.1 (Labelled transition system (LTS))** *A LTS is a quadruple $T = (Q, p_0, A, (\stackrel{?}{\rightarrow}))$ where $Q$ is the set of states, $A$ is the set of actions, $p_0 \in Q$ is the initial state, and $(\stackrel{?}{\rightarrow})$ the labelled transition relation.*

The syntax of the modal $\mu$-calculus is described below

$$
\begin{array}{lcl}
\textit{formula} & ::= & \top \\
 & | & \bot \\
 & | & \neg \textit{formula} \\
 & | & \textit{formula} \wedge \textit{formula} \\
 & | & \textit{formula} \vee \textit{formula} \\
 & | & [\textit{action\_list}]\textit{formula} \\
 & | & \langle \textit{action\_list} \rangle \textit{formula} \\
 & | & \textit{Variable} \\
 & | & \mu\, \textit{Variable.formula} \\
 & | & \nu\, \textit{Variable.formula} \\
 & & \\
\textit{action\_list} & ::= & \textit{Action} \\
 & | & \textit{Action}, \textit{action\_list}
\end{array}
$$

**Définition 7.1.2 (interpetation of the modal $\mu$-calculus)** $[\![\top]\!]_\sigma = Q$
$[\![\bot]\!]_\sigma = \emptyset$
$[\![\neg\phi]\!]_\sigma = Q \setminus [\![\phi]\!]_\sigma$
$[\![\phi \wedge \psi]\!]_\sigma = [\![\phi]\!]_\sigma \cap [\![\psi]\!]_\sigma$
$[\![\phi \vee \psi]\!]_\sigma = [\![\phi]\!]_\sigma \cup [\![\psi]\!]_\sigma$
$[\![[S]\phi]\!]_\sigma = \{p \mid \forall q \in Q, \forall a \in S, p \stackrel{a}{\rightarrow} q \Rightarrow q \in [\![\phi]\!]_\sigma\}$
$[\![\langle S\rangle\phi]\!]_\sigma = \{p \mid \exists q \in Q : \exists a \in S : p \stackrel{a}{\rightarrow} q \wedge q \in [\![\phi]\!]_\sigma\}$
$[\![X]\!]_\sigma = \sigma(X)$
$[\![\mu X.\phi]\!]_\sigma = \mathrm{lfp}^{\subseteq}(S \mapsto [\![\phi]\!]_{\sigma[X \mapsto S]})$

*where* $\sigma[X \mapsto S]$ *maps* $X$ *to* $S$ *and any* $Y \neq X$ *to* $\sigma(Y)$.
$[\![\nu X.\phi]\!]_\sigma = \mathrm{gfp}^{\subseteq}(S \mapsto [\![\phi]\!]_{\sigma[X\mapsto S]})$

**Définition 7.1.3 ($\vdash$)** *Let* $\sigma_\emptyset$ *be the minimal environnement (i.e. mapping every variable to* $\emptyset$*), then for every closed modal* $\mu$*-calculus formula* $\phi$ *we write* $q \vdash \phi$ *whenever* $q \in [\![\phi]\!]_{\sigma_\emptyset}$*, and for a modal LTS* $T = (Q, p_0, A, (\xrightarrow{?}))$ *we write* $T \vdash \phi$ *whenever* $p_0 \in [\![\phi]\!]_{\sigma_\emptyset}$ *and* $T \nvdash \phi$ *otherwise. By extension we may write* $p \vdash_T \phi$ *(resp.* $p \nvdash_T \phi$*) if and only if* $p \in [\![\phi]\!]_{\sigma_\emptyset}$ *(resp.* $p \notin [\![\phi]\!]_{\sigma_\emptyset}$*).*

## 7.2 ECCS syntax and semantics

| | | |
|---|---|---|
| *process* | ::= | 0 |
| | \| | (*process*) |
| | \| | {*action_list*}.*process* |
| | \| | *process* + *process* |
| | \| | *process* \| *process* |
| | \| | *process*[*renaming_list*] |
| | \| | *Variable* |
| | \| | $\mu$ *Variable.process* |
| *action_list* | ::= | *action* |
| | \| | *action*, *action_list* |
| *action* | ::= | $\tau$ |
| | \| | *Label* |
| | \| | $\overline{Label}$ |
| *label_list* | ::= | *Label* |
| | \| | *Label*, *label_list* |
| *renaming_list* | ::= | *renaming* |
| | \| | *renaming*, *renaming_list* |
| *renaming* | ::= | {*label_list*}/*Label* |
| | \| | $\emptyset$/*Label* |

Here are the ECCS transition rules :

**prefix** $\dfrac{}{S.p \xrightarrow{a} p}\, a \in S$, **par** $\dfrac{p \xrightarrow{a} p'}{(p) \xrightarrow{a} (p')}$, **sum₁** $\dfrac{p \xrightarrow{a} p'}{p + q \xrightarrow{a} p'}$, **sum₂** $\dfrac{q \xrightarrow{b} q'}{p + q \xrightarrow{b} q'}$

**par₁** $\dfrac{p \xrightarrow{a} p'}{p \mid q \xrightarrow{a} p' \mid q}$, **par₂** $\dfrac{q \xrightarrow{b} q'}{p \mid q \xrightarrow{b} p \mid q'}$, **par₃** $\dfrac{p \xrightarrow{a} p', q \xrightarrow{b} q'}{p \mid q \xrightarrow{\tau} p' \mid q'}$ inverses$(a, b)$

**rec** $\dfrac{p[\mu X.p/X] \xrightarrow{a} p'}{\mu X.p \xrightarrow{a} p'}$

**rename** $\dfrac{p \xrightarrow{a} p'}{p[f] \xrightarrow{b} p'[f]}$ $b \in f(a)$ where $f(a) = \{a\}$ if not syntactically defined.

**Définition 7.2.1 (interpretation of a closed ECCS term)** *For every closed*

*ECCS term $p$ we will name $[\![p]\!]$ the LTS generated by this term with respect to the above semantics.*

## 7.3 relations between ECCS and standard CCS

There is a trivial way to convert a CCS expression to ECCS, but not the inverse, we will nevertheless try to give a kind of close emulation : $\Psi$ : ECCS $\mapsto$ CCS.

**Définition 7.3.1 ($\Psi$)** $\Psi(0) = 0$

$\Psi(S.p) = \overset{n}{\underset{i=1}{+}} s_i.\Psi(p)$ *where* $S = \{s_1, \ldots, s_n\}$

$\Psi((p)) = (\Psi(p))$

$\Psi(p + q) = \Psi(p) + \Psi(q)$

$\Psi(p \mid q) = \Psi(p) \mid \Psi(q)$

$\Psi(X) = X$

$\Psi(\mu X.p) = \mu X.\Psi(p)$

$\Psi(p[f]) = ((\Psi(p)[h]) \mid r_f) \setminus \widetilde{L}$

*where :*

$\widetilde{L} = \{\widetilde{l} \mid l \in L\}$

$h(l) = \widetilde{l}$

$r_f = \mu X.((\underset{l \in L}{+} \overline{\widetilde{l}}.(\underset{l' \in f(l)}{+} (l'.X))) + \widetilde{l}.(\underset{l' \in f(l)}{+} (\overline{l'}.X)))$.

$\Psi$ give us a reasonable way to emulate a ECCS process with a CCS one, thus ECCS doesn't add too much expressiveness.

# Chapitre 8

# Action Abstraction

Given a surjective function $\alpha$ from the set $L_C$ of concrete labels to the set $L_A$ of abstract ones, what can we do ?

## 8.1 Definition of the naive abstraction

We will canonically lift $\alpha$ to actions and sets, and we will now give the way we chose to extend it to processes. We define $\gamma$ as being its "inverse".

**Définition 8.1.1 (LTS abstraction)** *The abstraction of an LTS* $T = (Q, A, p_0, (\xrightarrow{?}))$ *is* $\alpha(T) = (Q, \alpha(A), p_0, (\underset{\alpha}{\xrightarrow{?}}))$*, where* $p \underset{\alpha}{\xrightarrow{a}} q \overset{\text{def}}{\Leftrightarrow} \exists b \in \gamma(a) : p \xrightarrow{b} q$.

**Définition 8.1.2 (naive process abstraction)** $\alpha^{\ominus}(0) = 0$
$\alpha^{\ominus}(S.p) = \alpha(S).\alpha^{\ominus}(p)$
$\alpha^{\ominus}(p + q) = \alpha^{\ominus}(p) + \alpha^{\ominus}(q)$
$\alpha^{\ominus}(p \mid q) = \alpha^{\ominus}(p) \mid \alpha^{\ominus}(q)$
$\alpha^{\ominus}(X) = X$
$\alpha^{\ominus}(\mu X.p) = \mu X.\alpha^{\ominus}(p)$
$\alpha^{\ominus}(p[f]) = \alpha^{\ominus}(p)[\alpha^{\ominus}(f)]$ *where* $\alpha^{\ominus}(f)(t) = \alpha(f(\gamma(t)))$

## 8.2 Basic results

**Définition 8.2.1 ($(\prec^{\ominus})$)** $p \prec^{\ominus} q \overset{\text{def}}{\Leftrightarrow} q = \alpha^{\ominus}(p)$

**Lemme 8.2.1 ([)** *s*imulation] $(\prec^{\ominus})$ is a simulation between $\alpha(\llbracket p \rrbracket)$ and $\llbracket \alpha^{\ominus}(p) \rrbracket$

**Proof:** By structural induction on the transition deduction tree.

**Définition 8.2.2 (Safety)** *An abstraction* $\alpha$ *is* safe *with respect to a renaming* $f$ *if, and only if :*

1. $\forall a \in L, \mid f(a) \mid \leq 1$
2. $\forall a, b \in L \times L, \alpha(a) = \alpha(b) \Rightarrow \alpha(f(a)) = \alpha(f(b))$

*An abstraction $\alpha$ is* safe *with respect to a process $p$ if, and only if it is safe with respect to all the renamings in $p$.*

**Lemme 8.2.2 ([)** partial reverse simulation] If $p$ is a closed ECCS terms, and $\alpha$ is safe with respect to $p$, then for all closed ECCS terms $q, q'$ and action $b \neq \tau$ such that $p \prec^{\ominus} q$ and $q \xrightarrow{b} q'$, there exists a closed ECCS term $p'$ such that $p \underset{\alpha}{\xrightarrow{a}} p'$ and $p' \prec^{\ominus} q'$.

**Définition 8.2.3 (nearly-bisimulation)** *We call* nearly-bisimulation *any relation between to LTS that is a simulation and a partial reverse simulation.*

## 8.3 Improvement of this abstraction

**Définition 8.3.1 (sort of a subterm)** *The sort of subterms of a closed ECCS term is defined as the least fixpoint of the pointwise monotonic function $F_{p_0}$ defined by structural induction on the syntax of terms as :*
$F_{p_0}(\rho)(0) = \emptyset$
$F_{p_0}(\rho)((p)) = F_{p_0}(\rho)(p)$
$F_{p_0}(\rho)(S.p) = \text{label}(S) \cup F_{p_0}(\rho)(p)$
$F_{p_0}(\rho)(p+q) = F_{p_0}(\rho)(p) \cup F_{p_0}(\rho)(q)$
$F_{p_0}(\rho)(p \mid q) = F_{p_0}(\rho)(p) \cup F_{p_0}(\rho)(q)$
$F_{p_0}(\rho)(V) = \rho(V)$
$F_{p_0}(\rho)(\mu V.p) = \rho(\mu V.p)$ *where* $\mu V.p)$ *binds* $V$ *in* $p_0$.
$F_{p_0}(\rho)(p[f]) = f(F_{p_0}(\rho)(p))$

Then, we may redefine the abstraction of renamings as :
$\alpha^{\oplus}(p[f]) = \alpha^{\oplus}(p)[\alpha^{\oplus}(f)]$ where $\alpha^{\oplus}(f)(t) = \alpha(f(\gamma(t) \cap \text{sort}(p)))$
Which allows to restrict the loss of information at the cost of a context dependance of the abstraction definition.

**Définition 8.3.2 ($\triangleleft$)** *$p \triangleleft q$ means that $\alpha^{\oplus}(p)$ and $q$ are syntactically equal except for restrictions and renamings, and that $\alpha^{\oplus}(p)$'s renamings are less undeterministic (i.e. smaller pointwise) than $q$'s.*

**Lemme 8.3.1 ([)** simulation] ($\triangleleft$) is a simulation between $\alpha(\llbracket p \rrbracket)$ and $\llbracket \alpha^{\oplus}(p) \rrbracket$.

**Définition 8.3.3 (Local safety)** *An abstraction $\alpha$ is* locally safe *with respect to a renaming $f$ in the sort $S$ if, and only if :*

1. $\forall a \in \text{label}(S), \mid f(a) \mid \leq 1$
2. $\forall a, b \in \text{label}(S) \times \text{label}(S), \alpha(a) = \alpha(b) \Rightarrow \alpha(f(a)) = \alpha(f(b))$

*An abstraction $\alpha$ is* safe *with respect to a process $p$ if, and only if it is safe with respect to every renaming in $p$, with respect to the sort of its scope..*

**Définition 8.3.4 (($\prec^{\oplus}$))** *$p \prec^{\oplus} q$ means that $p \triangleleft q$ and that every renaming $f$ in $\alpha^{\oplus}(p)$ renames an action $a$ to $\emptyset$ if and only if $g$, the corresponding renaming in $q$, does, or $a$ is not in the sort of $f$'s scope.*

**Lemme 8.3.2 ([)** nearly-bisimulation] If $\alpha$ is locally safe with respect to $p$, then ($\prec^{\oplus}$) is a nearly-bisimulation between $\alpha(\llbracket p \rrbracket)$ and $\llbracket \alpha^{\oplus}(p) \rrbracket$.

## 8.4 properties of nearly-bisimulations

**Lemme 8.4.1 (** *propns*

The set of $\mu$-calculus formulas closed by $\top$, $\bot$, $\wedge$, $\vee$, $\langle a \rangle$ with $a \neq \tau$, $[a]$, and $\mu$, over the set of abstract actions, is preserved by nearly-bissimulation, that is to say that if $T \vdash \phi$ and $T \ltimes T'$ where $\ltimes$ is a nearly-bisimulation then $T' \vdash \phi$.

# Chapitre 9

# Modal LTS based abstraction

## 9.1 Modal LTS, extended modal $\mu$-calculus,refinement

A modal LTS is a sextuple $T = (Q, p_0, A, (\xrightarrow[\text{must}]{?}), (\xrightarrow[\text{can}]{?}), (\xrightarrow[\text{may}]{?}))$ where $Q$ is the set of states, $p_0$ the initial state, and $(\xrightarrow[\text{must}]{?})$,$(\xrightarrow[\text{can}]{?})$ and $(\xrightarrow[\text{may}]{?})$ are three transitions relations over $Q$, such that $(\xrightarrow[\text{must}]{?}) \subseteq (\xrightarrow[\text{can}]{?}) \subseteq (\xrightarrow[\text{may}]{?})$.

The extended modal $\mu$-calculus syntax is described below :

$$
\begin{array}{lll}
\mathit{formula} & ::= & \top \\
 & | & \bot \\
 & | & \neg \mathit{formula} \\
 & | & \mathit{formula} \wedge \mathit{formula} \\
 & | & \mathit{formula} \vee \mathit{formula} \\
 & | & [\mathit{action\_star\_list}]\mathit{formula} \\
 & | & \langle \mathit{action\_star\_list} \rangle \mathit{formula} \\
 & | & \mathit{Variable} \\
 & | & \mu\, \mathit{Variable.formula} \\
 & | & \nu\, \mathit{Variable.formula} \\
 & & \\
\mathit{action\_star} & ::= & \mathit{Action} \\
 & | & \mathit{Action}^* \\
 & & \\
\mathit{action\_star\_list} & ::= & \mathit{action\_star} \\
 & | & \mathit{action\_star}, \mathit{action\_star\_list}
\end{array}
$$

An environment is a function mapping formula variables to couples of state subsets, all environnement are supposed to be *consistent*, which means that both sides of the couple must always have an empty interchapter. The *interpretation* $[\![\phi]\!]_\sigma$ of a formula $\phi$ in the environnement $\sigma$ is the couple of the subsets of Q containing all the states known to satisfy this formula (resp. all the states known not to satisfy it) when assumed that $\sigma(X)$ is the couple of sets of the states known to satisfy the formula $X$ (resp. the states known not to satisfy it). There can be some states "in the middle", for wich the satisfaction can't be prooved or refuted, so the is a rather "intuitionistic" interpretation of the $\mu$-calculus.

**Définition 9.1.1 ($(\sqcap), (\sqcup), (\sqsubseteq), (\sqsupseteq)$)**

$$(A, B) \sqcap (C, D) \stackrel{\text{def}}{=} (A \cap C, B \cup D)$$

$$(A, B) \sqcup (C, D) \stackrel{\text{def}}{=} (A \cup C, B \cap D)$$

$$(A, B) \sqsubseteq (C, D) \stackrel{\text{def}}{\Leftrightarrow} A \subseteq C \wedge B \supseteq D$$

$$(A, B) \sqsupseteq (C, D) \stackrel{\text{def}}{\Leftrightarrow} A \supseteq C \wedge B \supseteq D$$

$$(A, B) \subset\subset (C, D) \stackrel{\text{def}}{\Leftrightarrow} A \subseteq C \wedge B \subseteq D$$

**Définition 9.1.2 (interpetation of the extended modal $\mu$-calculus)** $[\![\top]\!]_\sigma = (Q, \emptyset)$
$[\![\bot]\!]_\sigma = (\emptyset, Q)$
$[\![\neg\phi]\!]_\sigma = ([\![\phi]\!]^-_\sigma, [\![\phi]\!]^+_\sigma)$
$[\![\phi \wedge \psi]\!]_\sigma = [\![\phi]\!]_\sigma \sqcap [\![\psi]\!]_\sigma$
$[\![\phi \vee \psi]\!]_\sigma = [\![\phi]\!]_\sigma \sqcup [\![\psi]\!]_\sigma$
$[\![X]\!]_\sigma = \sigma(X)$
$[\![\mu X.\phi]\!]_\sigma = \mathrm{lfp}^{\sqsubseteq}(S \mapsto [\![\phi]\!]_{\sigma[X \mapsto S]})$
*where $\sigma[X \mapsto S]$ maps $X$ to $S$ and any $Y \neq X$ to $\sigma(Y)$.*
$[\![\nu X.\phi]\!]_\sigma = \mathrm{gfp}^{\sqsubseteq}(S \mapsto [\![\phi]\!]_{\sigma[X \mapsto S]})$

To be sure of the existence of the fixpoints, we will restrict ourselves to formulas were occurences of variables bound by $\mu$ and $\nu$, are positive, i.e. nested in an even number of negations. Since we want those fixpoints to be computable we also restrict ourselves to finite-state modal LTSs. Then we proof the following lemma :

**Lemme 9.1.1 ([)** *m*onotonicity] Let $\phi$ be a formula where free variables appear either only in positive or negative occurences, and $\sigma$ and $\sigma'$ two environments such that for every positive free variable $X$, $\sigma(X) \sqsubseteq \sigma'(X)$ and for every negative free variable $Y$, $\sigma(X) \sqsupseteq \sigma'(X)$, then $[\![\phi]\!]_\sigma \sqsubseteq [\![\phi]\!]_{\sigma'}$ and both interpretations are consistent.

**Définition 9.1.3 ($\vdash$)** *Let $\sigma_\emptyset$ be the minimal environnement (i.e. mapping every variable to $(\emptyset, Q)$), then for every closed abstracted modal $\mu$-calculus formula $\phi$ we write $q \vdash^\sharp \phi$ whenever $q \in [\![\phi]\!]^+_{\sigma_\emptyset}$, and for a modal LTS $T = (Q, p_0, A, (\xrightarrow[\text{must}]{?}), (\xrightarrow[\text{can}]{?}), (\xrightarrow[\text{may}]{?}))$ we write $T \vdash^\sharp \phi$ whenever $p_0 \in [\![\phi]\!]^+_{\sigma_\emptyset}$ and $T \nvdash^\sharp \phi$ whenever $p_0 \in [\![\phi]\!]^-_{\sigma_\emptyset}$. By extension we may write $p \vdash^\sharp_T \phi$ (resp. $p \nvdash^\sharp_T \phi$) if and only if $p \in [\![\phi]\!]^+_{\sigma_\emptyset}$ (resp. $p \in [\![\phi]\!]^-_{\sigma_\emptyset}$).*

**Définition 9.1.4 (refinement)** *Given two modal LTSs $T = (Q, p_0, A, (\xrightarrow[\text{must}]{?}), (\xrightarrow[\text{can}]{?}), (\xrightarrow[\text{may}]{?}))$ and $T' = (Q', p'_0, A, (\xrightarrow[\text{must}]{?}'), (\xrightarrow[\text{can}]{?}'), (\xrightarrow[\text{may}]{?}'))$ over the same set* *of actions and $\rtimes$ a relation between their sets of states $Q$ and $Q'$. We say $\rtimes$ is a* refinement *if and only if for all $p \in Q$ and $p' \in Q'$ such that '$p \rtimes p'$ and for all $a \in A$ :*

- $\exists q \in Q : p \xrightarrow[\text{may}]{a} q \Rightarrow \exists q' \in Q' : p' \xrightarrow[\text{may}]{a}{}' q' \wedge q \rtimes q'$
- $\exists q' \in Q' : p \xrightarrow[\text{can}]{a}{}' q \Rightarrow \exists q \in Q : p \xrightarrow[\text{must}]{a} q' \wedge q \rtimes q'$
- $\exists q' \in Q' : p \xrightarrow[\text{must}]{a}{}' q \Rightarrow \exists q \in Q : p \xrightarrow[\text{must}]{a} q' \wedge q \rtimes q'$

*Then we say that $T$ is a refinement of $T$' if their exist a refinement $\rtimes$ between them such that their initial states are related by $\rtimes$. In such a case we will write $T \trianglelefteq T'$.*

**Lemme 9.1.2 ([)** *A* refinement has more information]

If $T \preceq T'$ then for any refinement relation $\rtimes$, let $\sigma$ and $\sigma'$ be two environments respectively over $T$ and $T'$ such that for all formula variables $X$, if $p \rtimes p'$ then $p' \in {\sigma'}^{+}(X) \Rightarrow p \in \sigma^{+}(X)$ and $p' \in {\sigma'}^{-}(X) \Rightarrow p \in \sigma^{-}(X)$. Then, if $p \rtimes p'$, $p' \in [\![\phi]\!]^{+}_{\sigma'} \Rightarrow p \in [\![\phi]\!]^{+}_{\sigma}$ and $p' \in [\![\phi]\!]^{-}_{\sigma'} \Rightarrow p \in [\![\phi]\!]^{-}_{\sigma}$.

## 9.2 formula abstraction, modal LTS abstraction, safety, optimality

In this chapter $\alpha$ will be a surjective map from a set of concrete actions to a set of abstract actions.

**Définition 9.2.1 (formula abstraction)** *The formula abstraction by an action abstraction $\alpha$ is rather straightforward : in all boxes and diamonds we replace $S$ by $\{b \mid \exists a \in S : b = \alpha(a)\} \cup \{b^* \mid \gamma(b) \subseteq S\}$, where $\gamma$ is the lifted inverse of $\alpha$.*

**Définition 9.2.2 (LTS modal abstraction)** *Let $T = (Q, p_0, A, (\xrightarrow{?}))$ be an LTS, then an abstraction of this LTS by the action abstraction $\alpha$ and the state abstraction $\alpha^\sharp$ is a modal LTS $T' = (\alpha^\sharp(Q), \alpha^\sharp(p_0), \alpha(A), (\xrightarrow[\text{must}]{?}), (\xrightarrow[\text{can}]{?}), (\xrightarrow[\text{may}]{?}))$.*

**Définition 9.2.3 (abstraction refinement ordering)** *Let $T = (Q, p_0, A, (\xrightarrow{?}))$ be an LTS, and $S$ and $S'$ be two abstractions of $T$ by the action abstraction $\alpha$ and the state abstraction $\alpha^\sharp$, then we write $S \trianglelefteq S'$ if $=$ is a refinement relation between $S$ and $S'$.*

Then we remark that $\trianglelefteq$ is a partial ordering of the abstractions of $T$.

**Lemme 9.2.1 ([)** *m*onotonicity of $[\![.]\!]_{\sigma_\emptyset}$] For every formula $\phi$, $[\![\phi]\!]_{\sigma_\emptyset}$ is a $\subset\subset$-monotonic function of the LTS abstraction ( ordered by $\trianglelefteq$).

**Définition 9.2.4 (LTS abstraction safety)** *$T' = (\alpha^\sharp(Q), \alpha^\sharp(p_0), \alpha(A), (\xrightarrow[\text{must}]{?}), (\xrightarrow[\text{can}]{?}), (\xrightarrow[\text{may}]{?}))$ is a* safe *abstraction of $T = (Q, p_0, A, (\xrightarrow{?}))$ by $\alpha$ and $\alpha^\sharp$ if and only if for all states $p$ and $q$ in $Q$ and all formulas $\phi$, $\alpha^\sharp(p) \vdash^\sharp \alpha(\phi) \Rightarrow p \vdash \phi$ and $\alpha^\sharp(p) \nvdash^\sharp \alpha(\phi) \Rightarrow p \nvdash \phi$.*

**Lemme 9.2.2 ([)** *s*afety conditions] With the same notations as above, if the following conditions hold, then the abstraction is safe :

- $\exists p \in \gamma^\sharp(q), a \in \gamma(b), p' \in \gamma^\sharp(q') : p \xrightarrow{a} p' \Rightarrow q \xrightarrow[\text{may}]{b} q'$
- $q \xrightarrow[\text{can}]{b} q' \Rightarrow \forall p \in \gamma^\sharp(q), \exists a \in \gamma(b), p' \in \gamma^\sharp(q') : p \xrightarrow{a} p'$
- $q \xrightarrow[\text{must}]{b} q' \Rightarrow \forall p \in \gamma^\sharp(q), a \in \gamma(b), \exists p' \in \gamma^\sharp(q') : p \xrightarrow{a} p'$

**Définition 9.2.5 (canonical abstration)** *With the same notations as for the safety, the canonical abstraction of $T$ by the action abstraction $\alpha$ and the state abstraction $\alpha^\sharp$ is the one defined by :*

- $\exists p \in \gamma^\sharp(q), a \in \gamma(b), p' \in \gamma^\sharp(q') : p \xrightarrow{a} p' \Leftrightarrow q \xrightarrow[\text{may}]{b} q'$
- $q \xrightarrow[\text{can}]{b} q' \Leftrightarrow \forall p \in \gamma^\sharp(q), \exists a \in \gamma(b), p' \in \gamma^\sharp(q') : p \xrightarrow{a} p'$
- $q \xrightarrow[\text{must}]{b} q' \Leftrightarrow \forall p \in \gamma^\sharp(q), a \in \gamma(b), \exists p' \in \gamma^\sharp(q') : p \xrightarrow{a} p'$

**Lemme 9.2.3 ([)** *w*eak optimality] **??** The canonical abstraction is *optimal* among those for which the safety conditions hold, which means the interpretation of every abstract formula $\phi$ is $\subset\subset$-maximal (i.e. it contains the most information).

## 9.3 Modal processes, syntax and semantics

We will now describe a computable state abstraction, i.e. a state abstraction that will allow us to deduce a safe modal LTS without computing the concrete one, which would improve the speed of model checking computations at the cost of not being able to proof some assertions.

$$
\begin{array}{lcl}
\mathit{process} & ::= & 0 \\
& | & (\mathit{process}) \\
& | & \mathit{action\_star\_list.process} \\
& | & \mathit{process} + \mathit{process} \\
& | & \mathit{process} \mid \mathit{process} \\
& | & \mathit{process}[\mathit{mod\_renaming}] \\
& | & \mathit{Variable} \\
& | & \mu\,\mathit{Variable.process} \\
\\
\mathit{action\_star\_list} & ::= & \mathit{action\_star} \\
& | & \mathit{action\_star}, \mathit{action\_star\_list} \\
\\
\mathit{action\_star} & ::= & \mathit{action} \\
& | & \mathit{action}^* \\
\\
\mathit{action} & ::= & \tau \\
& | & \mathit{Label}
\end{array}
$$

| | | |
|---|---|---|
| | $\mid$ | $\overline{Label}$ |
| *mod_renaming* | $::=$ | $[(renaming\_list),]^3(renaming\_list)$ |
| *renaming_list* | $::=$ | *renaming* |
| | $\mid$ | *renaming, renaming_list* |
| *renaming* | $::=$ | $\{label\_list\}/Label$ |
| | $\mid$ | $\emptyset/Label$ |
| *label_list* | $::=$ | *Label* |
| | $\mid$ | *Label, label_list* |

Here is the modal semantics :

$$\mathbf{weak}^{\mathrm{must}}\ \frac{p \xrightarrow[\mathrm{must}]{a} p'}{p \xrightarrow[\mathrm{can}]{a} p'},\quad \mathbf{weak}^{\mathrm{can}}\ \frac{p \xrightarrow[\mathrm{can}]{a} p'}{p \xrightarrow[\mathrm{may}]{a} p'},$$

$$\mathbf{pre}^{\mathrm{can}}\ \frac{}{S.p \xrightarrow[\mathrm{can}]{a} p}\ a \in S,\quad \mathbf{pre}^{\mathrm{must}}\ \frac{}{S.p \xrightarrow[\mathrm{must}]{a} p}\ a^* \in S,$$

$$\mathbf{par}^{\mathrm{may}}\ \frac{p \xrightarrow[\mathrm{may}]{a} p'}{(p) \xrightarrow[\mathrm{may}]{a} (p')},\quad \mathbf{par}^{\mathrm{can}}\ \frac{p \xrightarrow[\mathrm{can}]{a} p'}{(p) \xrightarrow[\mathrm{can}]{a} (p')},\quad \mathbf{par}^{\mathrm{must}}\ \frac{p \xrightarrow[\mathrm{must}]{a} p'}{(p) \xrightarrow[\mathrm{must}]{a} (p')},$$

$$\mathbf{sum}_{\mathbf{1}}^{\mathrm{may}}\ \frac{p \xrightarrow[\mathrm{may}]{a} p'}{p+q \xrightarrow[\mathrm{may}]{a} p'},\quad \mathbf{sum}_{\mathbf{1}}^{\mathrm{can}}\ \frac{p \xrightarrow[\mathrm{can}]{a} p'}{p+q \xrightarrow[\mathrm{can}]{a} p'},\quad \mathbf{sum}_{\mathbf{1}}^{\mathrm{must}}\ \frac{p \xrightarrow[\mathrm{must}]{a} p'}{p+q \xrightarrow[\mathrm{must}]{a} p'},$$

$$\mathbf{sum}_{\mathbf{1}}^{\mathrm{may}}\ \frac{q \xrightarrow[\mathrm{may}]{b} q'}{p+q \xrightarrow[\mathrm{may}]{b} q'},\quad \mathbf{sum}_{\mathbf{1}}^{\mathrm{can}}\ \frac{q \xrightarrow[\mathrm{can}]{b} q'}{p+q \xrightarrow[\mathrm{can}]{b} q'},\quad \mathbf{sum}_{\mathbf{1}}^{\mathrm{must}}\ \frac{q \xrightarrow[\mathrm{must}]{b} q'}{p+q \xrightarrow[\mathrm{must}]{b} q'},$$

$$\mathbf{par}_{\mathbf{1}}^{\mathrm{may}}\ \frac{p \xrightarrow[\mathrm{may}]{a} p'}{p \mid q \xrightarrow[\mathrm{may}]{a} p' \mid q},\quad \mathbf{par}_{\mathbf{1}}^{\mathrm{can}}\ \frac{p \xrightarrow[\mathrm{can}]{a} p'}{p \mid q \xrightarrow[\mathrm{can}]{a} p' \mid q},\quad \mathbf{par}_{\mathbf{1}}^{\mathrm{must}}\ \frac{p \xrightarrow[\mathrm{must}]{a} p'}{p \mid q \xrightarrow[\mathrm{must}]{a} p' \mid q},$$

$$\mathbf{par}_{\mathbf{1}}^{\mathrm{may}}\ \frac{q \xrightarrow[\mathrm{may}]{b} q'}{p \mid q \xrightarrow[\mathrm{may}]{b} p \mid q'},\quad \mathbf{par}_{\mathbf{1}}^{\mathrm{can}}\ \frac{q \xrightarrow[\mathrm{can}]{b} q'}{p \mid q \xrightarrow[\mathrm{can}]{b} p \mid q'},\quad \mathbf{par}_{\mathbf{1}}^{\mathrm{must}}\ \frac{q \xrightarrow[\mathrm{must}]{b} q'}{p \mid q \xrightarrow[\mathrm{must}]{b} p \mid q'},$$

$$\mathbf{par}_{\mathbf{3}}^{\mathrm{may}}\ \frac{p \xrightarrow[\mathrm{may}]{a} p', q \xrightarrow[\mathrm{may}]{b} q'}{p \mid q \xrightarrow[\mathrm{may}]{\tau} p' \mid q'}\ \mathrm{inverses}(a,b),\quad \mathbf{par}_{\mathbf{3}}^{\mathrm{must}}\ \frac{p \xrightarrow[\mathrm{must}]{a} p', q \xrightarrow[\mathrm{must}]{b} q'}{p \mid q \xrightarrow[\mathrm{must}]{\tau} p' \mid q'}\ \mathrm{inverses}(a,b),$$

$$\mathbf{par}_{\mathbf{3}}^{\mathrm{can_1}}\ \frac{p \xrightarrow[\mathrm{can}]{a} p', q \xrightarrow[\mathrm{must}]{b} q'}{p \mid q \xrightarrow[\mathrm{must}]{\tau} p' \mid q'}\ \mathrm{inverses}(a,b),\quad \mathbf{par}_{\mathbf{3}}^{\mathrm{can_2}}\ \frac{p \xrightarrow[\mathrm{must}]{a} p', q \xrightarrow[\mathrm{can}]{b} q'}{p \mid q \xrightarrow[\mathrm{must}]{\tau} p' \mid q'}\ \mathrm{inverses}(a,b),$$

$$\mathbf{rec}^{\mathrm{may}}\ \frac{p[\mu X.p/X] \xrightarrow[\mathrm{may}]{a} p'}{\mu X.p \xrightarrow[\mathrm{may}]{a} p'},\quad \mathbf{rec}^{\mathrm{can}}\ \frac{p[\mu X.p/X] \xrightarrow[\mathrm{can}]{a} p'}{\mu X.p \xrightarrow[\mathrm{can}]{a} p'},\quad \mathbf{rec}^{\mathrm{must}}\ \frac{p[\mu X.p/X] \xrightarrow[\mathrm{must}]{a} p'}{\mu X.p \xrightarrow[\mathrm{must}]{a} p'}$$

$$\textbf{rename}^{\text{may}}\ \frac{p \xrightarrow[\text{may}]{a} p'}{p[f,\_,\_,\_] \xrightarrow[\text{may}]{b} p'[f,\_,\_,\_]}\ b \in f(a)$$

where $f(a) = \{a\}$ if not syntactically defined,

$$\textbf{rename}^{\text{can}}\ \frac{p \xrightarrow[\text{can}]{a} p'}{p[\_,f,\_,\_] \xrightarrow[\text{can}]{b} p'[\_,f,\_,\_]}\ b \in f(a),$$

$$\textbf{rename}^{\text{must}}\ \frac{p \xrightarrow[\text{must}]{a} p'}{p[\_,\_,f,\_] \xrightarrow[\text{must}]{b} p'[\_,\_,f,\_]}\ b \in f(a),$$

$$\textbf{rename}^{\text{can}}_{\text{must}}\ \frac{p \xrightarrow[\text{can}]{a} p'}{p[\_,\_,\_,f] \xrightarrow[\text{must}]{b} p'[\_,\_,\_,f]}\ b \in f(a),$$

$$\textbf{rename}^{\text{must}}_{\text{can}}\ \frac{p \xrightarrow[\text{must}]{a} p'}{p[f,\_,\_,\_] \xrightarrow[\text{can}]{b} p'[f,\_,\_,\_]}\ b \in f(a),$$

## 9.4 process and formula abstractions

**Définition 9.4.1 (formula abstraction)** *The formula abstraction is rather straightforward : in all boxes and diamonds we replace $S$ by $\{b \mid \exists a \in S : b = \alpha(a)\} \cup \{b^* \mid \gamma(b) \subseteq S\}$.*

**Définition 9.4.2 (modal process abstraction)** $\alpha^\sharp(0) = 0$
$\alpha^\sharp(S.p) = (\alpha(S) \cup \{b^* \mid \gamma(b) \subseteq S\}).\alpha^\sharp(p)$
$\alpha^\sharp(p + q) = \alpha^\sharp(p) + \alpha^\sharp(q)$
$\alpha^\sharp(p \mid q) = \alpha^\sharp(p) \mid \alpha^\sharp(q)$
$\alpha^\sharp(X) = X$
$\alpha^\sharp(\mu X.p) = \mu X.\alpha^\sharp(p)$
$\alpha^\sharp(p[f]) = \alpha^\sharp(p)[f_{\text{may}}, f_{\text{can}}, f_{\text{must}}, f^{\text{can}}_{\text{must}}]$ *where :*

- $f_{\text{may}}(a) = \{b \mid \exists c \in \gamma(a) : \exists d \in \gamma(b) : d \in f(c)\}$
- $f_{\text{can}}(a) = \{b \mid \forall c \in \gamma(a), \exists d \in \gamma(b) : d \in f(c)\}$
- $f_{\text{must}}(a) = \{b \mid \forall d \in \gamma(b), \exists c \in \gamma(a) : d \in f(c)\}$
- $f^{\text{can}}_{\text{must}}(a) = \{b \mid \forall c \in \gamma(a), \forall d \in \gamma(b) : d \in f(c)\}$

**Lemme 9.4.1 (** *modabssafe* This abstraction is safe.

# Conclusion

Action and state abstractions provide a good framework to achieve the model-checking of huge or possibly infinite transition systems, and the use of modal LTSs allows us to reduce as much as possible the loss of information due to the abstraction. The refinement is a way building a strategy where we use finer and finer abstractions to try to proove a property. The modelchecking of modal LTS doesn't induce much overhead since the main difference with the normal one is that we hav to considere different transition relations. The ECCS abbstraction of CCS allows us to build modal LTSs from CCS terms without building the concrete LTS, which might reduce combinatorial state explosion provided we are able to find a clever way of abstrating actions.

# Quatrième partie

# Mémoire de DEA

# 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

## 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.

## 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 14).

# Chapitre 10

# le système Coq

## 10.1 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 $\lambda$-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.

## 10.2 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.
```

## 10.3 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`.

## 10.4 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.

## 10.5 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.

## 10.6 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 11

# La clôture de congruence

## 11.1 Algèbre de termes

### 11.1.1 Termes à arités multiples

**Définition 11.1.1 (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 11.1.2 (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 $T_\Sigma$ tel que $\forall (f,d) \in \Sigma, \forall (t_0, \ldots, t_{d_i-1}) \in T_\Sigma^{d_i}, f_i(t_0, \ldots, t_{d_i-1}) \in T_\Sigma$.*

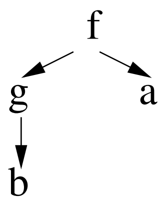



FIG. 11.1: exemple de terme : f(g(b),a)

**Définition 11.1.3 (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 11.1.4 (congruence)** *Soit $\Sigma$ une signature, une* congruence *sur $T_\Sigma$ 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 11.1.1 ()** *Soient une signature $\Sigma$ et un ensemble $A$ de couples de termes de $T_\Sigma$, il existe une une plus fine relation $\sim_A$ sur $T_\Sigma$ 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*

### 11.1.2 Termes curryfiés

**Définition 11.1.5 (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 $n-1$.*

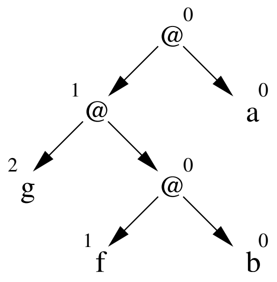



FIG. 11.2: exemple de terme : g(f(b),a)

**Définition 11.1.6 (congruence)** *Soit $\Sigma$ une signature, une* congruence *sur $T_\Sigma$ 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.

## 11.2 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.

### 11.2.1 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.

## 11.3 L'algorithme

### 11.3.1 Description rapide

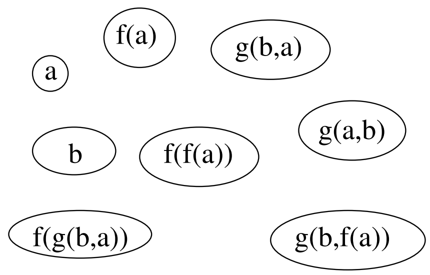
On suppose que l'on dispose d'une structure permettant de manipuler des classes d'équivalence : recherche du représentant canonique, fusion de classes. L'algorithme fonctionne ainsi : on construit une structure contenant tous les termes et sous-termes du problème dans des classes singletons, puis on 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.

### 11.3.2 Un exemple

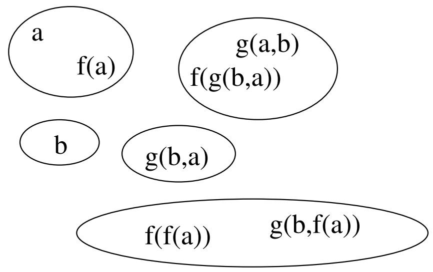
Prenons l'exemple de la couverture, soit le problème :

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

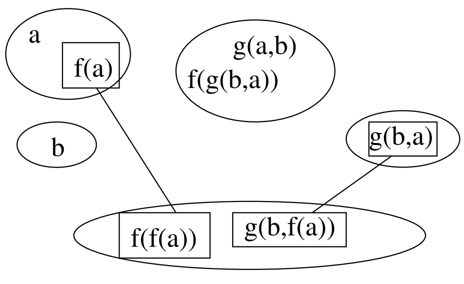
(exemple dû à Jean Goubault-Larrecq).



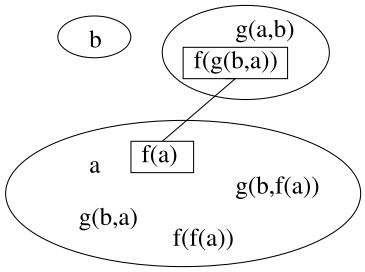

Tout d'abord, construisons l'ensemble des termes et sous termes, que nous plaçons dans des classes distinctes.



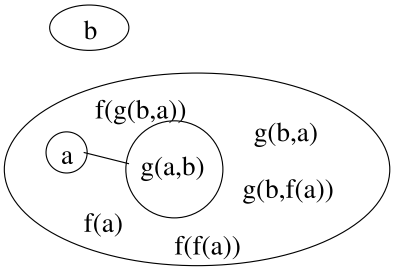

Ensuite nous fusionnons les classes contenant les membres des axiomes (notons que parfois plusieurs axiomes portent sur le même terme, c'est pourquoi on doit construire des singletons et fusionner ensuite).




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**.

### 11.3.3 Table de signatures

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

**Définition 11.3.1 (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* $sig(f(t_0, \ldots, t_{n-1})) = f(d(t_0), \ldots, d(t_{n-1}))$.

**Lemme 11.3.1 (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 12

# Réflexion totale : Implantation

## 12.1 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 12.2) 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.
```

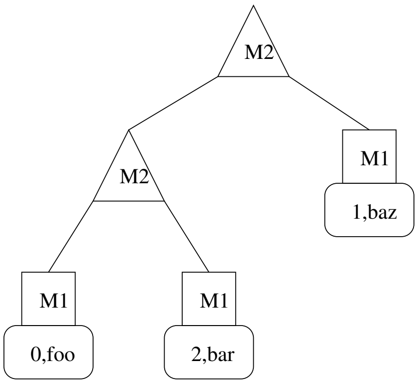
On dispose d'un type option pour coder les fonction partielles :

```
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.

## 12.2 UNION-FIND

### 12.2.1 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.

### 12.2.2 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).
```

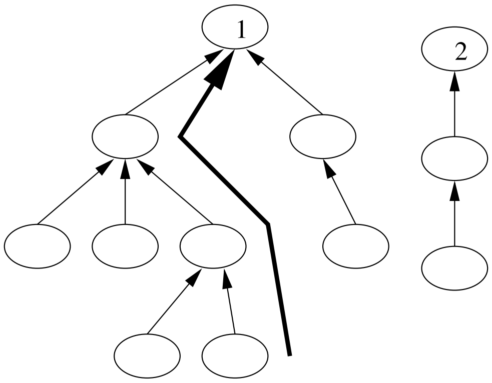



FIG. 12.1: l'opération FIND

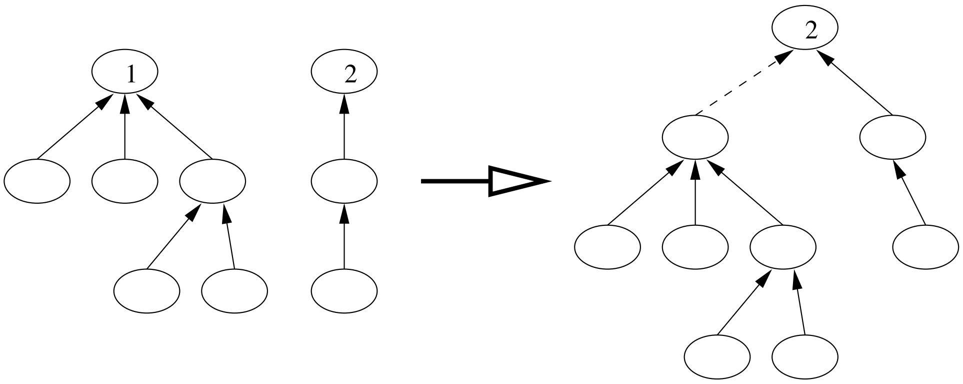



FIG. 12.2: l'opération UNION

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* 10.4 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 $\iota$-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».

## 12.3 Table de signatures

Comme on va représenter les nœuds par des `ad` (voir 12.1), 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.

## 12.4 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 12.3).

*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. 12.3: 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 12.4).

```
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. 12.4: Procédure de création de la structure

La boucle principale se décompose en deux parties, la première (figure 12.5) détermine les termes congruents, et la seconde (figure 12.6) 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 12.7) puis on referme la boucle (figure 12.8).

```
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. 12.5: 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. 12.6: Seconde sous-boucle

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

FIG. 12.7: 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. 12.8: Boucle principale

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

FIG. 12.9: Procédure complète

# Chapitre 13

# Certification

## 13.1 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*

## 13.2 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.

### 13.2.1 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 13.1 : 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* 13.2.2).

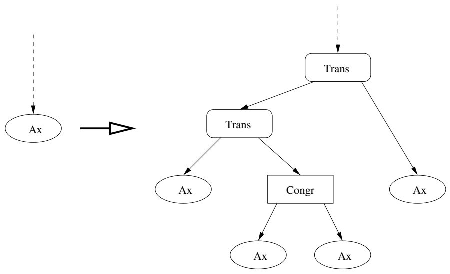



FIG. 13.1: Réalisation d'un congruence

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`.

### 13.2.2 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.

## 13.3 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.
```

### 13.3.1 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 13.2, 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 13.3.1 (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

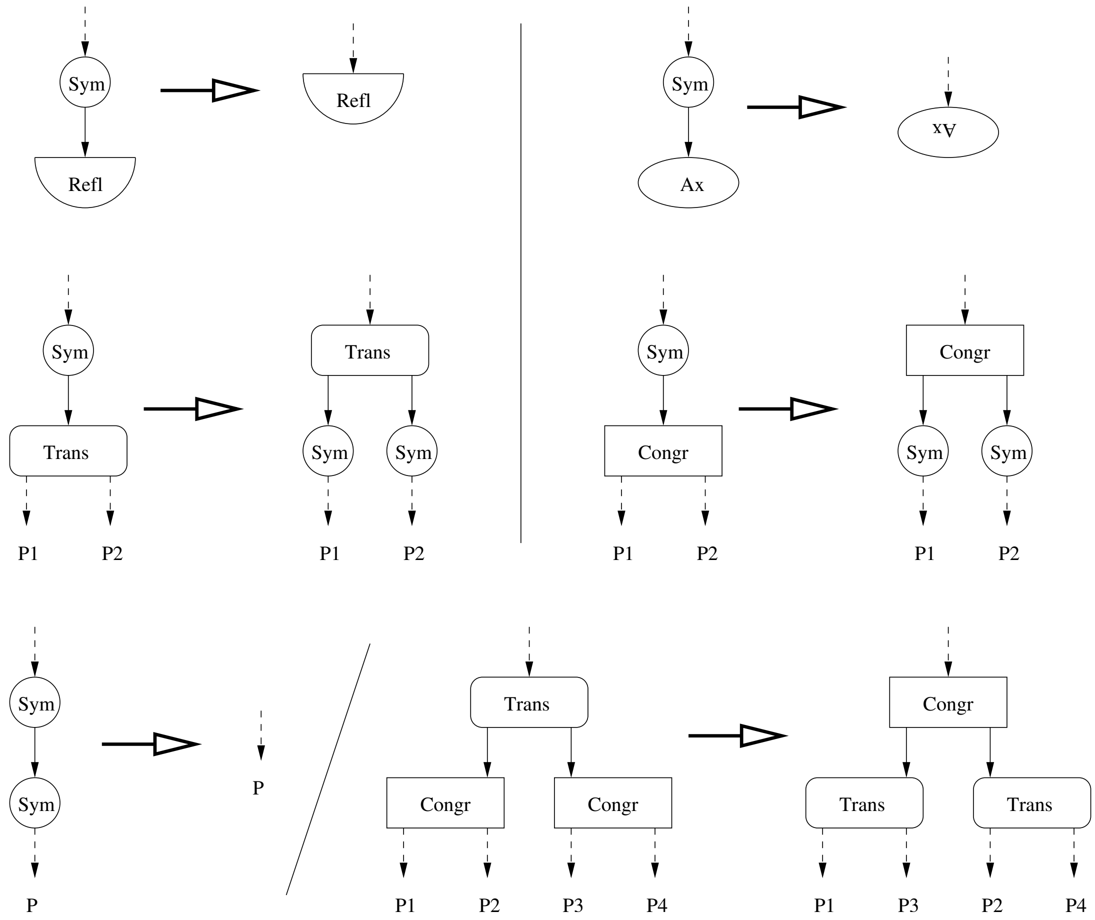



FIG. 13.2: Règles de normalisation

### 13.3.2 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 13.3)

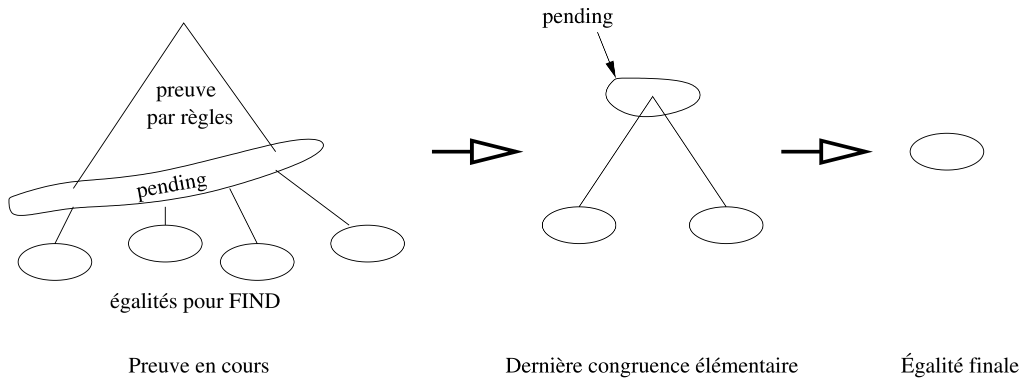



FIG. 13.3: Évolution d'une preuve

```
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`

## 13.4 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 14

# 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.

## 14.1 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 14.1).

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 14.2).

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

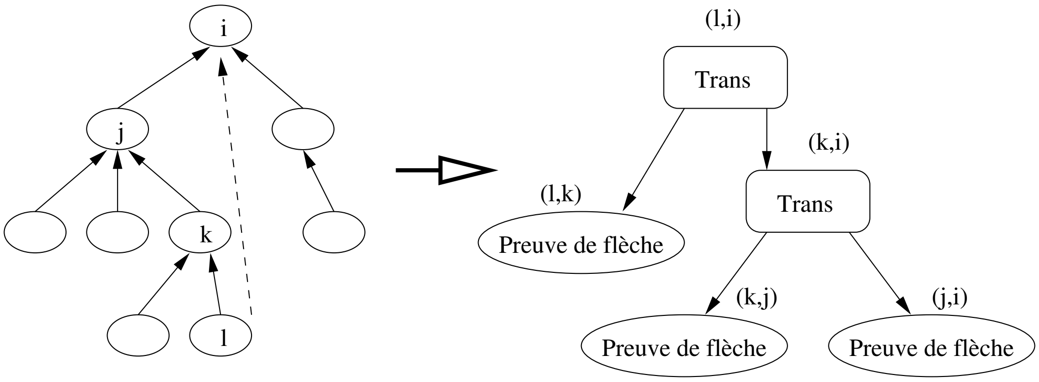



Fig. 14.1: génération de la preuve d'un chemin

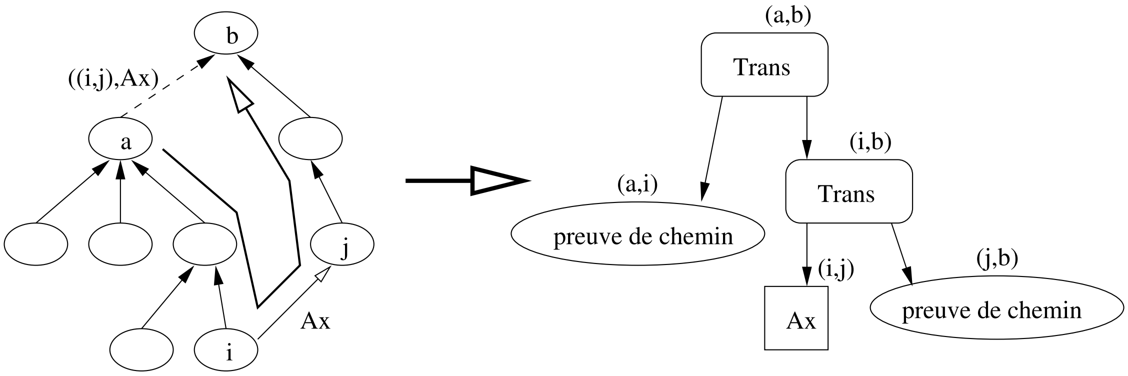



Fig. 14.2: génération de la preuve d'une flèche

antérieures, ce qui assure la terminaison.

Comme dans le mécanisme de normalisation (*cf* 13.2), on applique certaines règles de réduction des preuves pour de débarasser des règles de symétrie et réflexivité autant que possible.

## 14.2 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.

## 14.3 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.

## 14.4 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)))
```

## 14.5 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. 14.3: 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 fonctionnement 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.

# 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.