Lot 5.1

Technologie de vérification

Ajouter la réécriture au noyau de Coq

Combinaison à des systèmes de vérification automatique de terminaison

Description :	Ce document est le rapport de stage de DEA de Sébastien Hinderer au LORIA, intitulé “Certification des preuves de terminaison par interprétations polynomiales”, encadré par Frédéric Blanqui et soutenu le 22 juin 2004 [15]. Dans ce stage, Sébastien a développé une bibliothèque Coq permettant de certifier les preuves de terminaison par interprétations polynomiales. De plus, à partir du prototype de Coq avec récriture [7], il a développé une extension de Coq V8.0 lui permettant de définir des systèmes de règles de récriture, de demander à CiME de trouver une interprétation polynomiale, et de générer automatiquement la preuve de correction de cette interprétation. Tout cela est disponible sur `http://www.loria.fr/~blanqui/termin.html`.
Auteur(s) :	Sébastien Hinderer
Référence :	Averroes / Lot 5.1 / Fourniture 5 / V1.0
Date :	8 octobre 2004
Statut :	validé
Version :	1.0

Réseau National des Technologies Logicielles
Projet subventionné par le Ministère de la Recherche et des Nouvelles Technologies

CRIL Technology, France Télécom R&D, INRIA-Futurs, LaBRI (Univ. de Bordeaux – CNRS), LIX (École Polytechnique, CNRS) LORIA, LRI (Univ. de Paris Sud – CNRS), LSV (ENS de Cachan – CNRS)

Historique

22 juin 2004	V 0.1	rapport de stage de DEA
8 octobre 2004	V 1.0	mise au format Averroes

Table des matières

Chapitre 1

Introduction

Depuis plusieurs années, les programmes informatiques sont utilisés dans des domaines vitaux ou économiquement importants. Il semble donc indispensable de pouvoir en certifier le bon fonctionnement. La certification d'un programme consiste à rechercher sa spécification (c'est-à-dire l'ensemble des propriétés mathématiques que ce programme doit vérifier), puis à prouver que le programme est correct vis-à-vis de cette spécification.

Pour aider à la réalisation de cette double tâche, des outils particuliers, les systèmes d'aide à la preuve, ont été mis au point. Ces systèmes permettent de définir des fonctions, ainsi que de démontrer des théorèmes et des propriétés de ces fonctions.

Les systèmes d'aide à la preuve existants diffèrent entre eux tant par le cadre logique sous-jacent (les axiomes et les règles autorisés pour faire les démonstrations), que par le degré d'automatisation des preuves qu'ils fournissent. Certains sont complètement automatiques et donc limités (prouver un théorème est indécidable en général), tandis que d'autres, au pouvoir d'expression plus important, fonctionnent de manière interactive, c'est-à-dire que les définitions et les preuves sont construites grâce à un dialogue entre l'utilisateur et le programme d'aide à la preuve. La coopération, et même l'intégration de ces deux types d'outils est évidemment souhaitable. Elle est d'ailleurs l'objet de nombreux travaux, dont ce stage fait partie.

Le programme `Coq` [11], dont il sera ici question, entre dans la catégorie des logiciels d'aide à la preuve interactifs. Bien que le pouvoir d'expressivité de `Coq` soit déjà très important, ce système souffre d'un certain nombre de limitations. D'une part, l'utilisateur ne peut pas toujours définir ses fonctions ou ses prédicats comme il le souhaiterait ; il doit en effet en donner une définition inductive, c'est-à-dire ne faisant intervenir que du filtrage sur les constructeurs et des appels récursifs sur des sous-termes. D'autre part, l'utilisateur ne peut pas indiquer au système d'autres règles de simplification que celles permises par les définitions inductives. Cela l'oblige dans les preuves à descendre à un niveau de détail souvent inintéressant et lui fait perdre du temps.

Cet état de fait a conduit un certain nombre de chercheurs à étudier des extensions du Calcul des Constructions Inductives [19] (cadre logique sur lequel `Coq` est basé) qui permettraient de s'affranchir de ces limitations, notamment en autorisant des définitions de fonctions ou de prédicats à l'aide de règles de réécriture, paradigme plus simple et plus puissant que les règles inductives qui sont un cas particulier de réécriture [3, 5, 14, 23].

Cependant, les conditions à vérifier pour qu'une telle extension reste décidable et valide sont plus complexes qu'avec des définitions inductives et, si l'on veut profiter pleinement de toutes les potentialités de la réécriture, il peut être intéressant de faire appel à des outils externes de vérification de la confluence (l'ordre d'un calcul n'a pas d'importance) et de la terminaison (les calculs terminent toujours), deux propriétés nécessaires pour assurer la décidabilité et la validité du système. Un tel système ne serait alors plus tout à fait fiable. Une manière de contourner cela est de vérifier dans le système lui-même que les résultats fournis par les outils externes sont bien corrects. A cette fin, il est nécessaire de formaliser les critères utilisés par ces outils.

L'objectif de ce stage est ainsi de formaliser le critère des interprétations polynomiales [10], qui est un critère très efficace et utilisé par la plupart des systèmes de preuve de terminaison actuels, tels que `CiME` [9].

Ce stage s'inscrit dans le cadre du projet RNTL Averroes [1], dans lequel `CiME` a déjà été utilisé pour réaliser un prototype de `Coq` avec réécriture [6], et constitue le dernier objectif du lot 5.1. Ce stage s'inscrit également dans un début de coopération informelle avec Femke van Raamsdonk de l'Université Libre d'Amsterdam dans le but de créer une bibliothèque `Coq` de théorèmes sur la réécriture et la terminaison en particulier. C'est ainsi que l'une de ses étudiantes, Nicole de Kleijn, a réalisé une formalisation partielle du critère RPO (Recursive Path Ordering [13]) l'année dernière [12], tandis qu'un autre étudiant, Adam Koprowski, cherche actuellement à formaliser la preuve de HORPO (Higher Order RPO) [17], l'extension de RPO aux termes du λ-calcul simplement typé.

Des travaux dans le même esprit ont déjà été réalisés :
- dans le domaine de la preuve d'égalité : dans sa thèse [18], Nguyen utilise `ELAN` [8] pour trouver une preuve d'égalité dont la trace permet de construire un terme-preuve qui est vérifié par `Coq` ;
- concernant la formalisation de théorèmes de réécriture en `Coq` : dans [20, 21], Rouyer s'intéresse à la formalisation de quelques algorithmes de réécriture en `Coq`. Cette formalisation est ancienne, et basée sur une structure de termes curryfiés qui n'est pas appropriée pour la définition d'une interprétation pour laquelle le nombre d'arguments doit être connu.

Dans un premier chapitre, nous présentons en détail le critère de terminaison par interprétations polynomiales dont nous nous proposons de formaliser la preuve. Dans un second chapitre, nous nous attacherons à donner une brève présentation de `Coq` avant de voir, au chapitre 4, comment le théorème à démontrer a été formalisé, en étudiant les problèmes posés par cette formalisation, et les choix qui ont été faits pour les résoudre. Enfin, le chapitre 5 décrit la mise en œuvre d'une interface avec `CiME` permettant, depuis le système `Coq`, la recherche d'une interprétation polynomiale et la construction de la preuve de terminaison d'un système de réécriture utilisant cette interprétation.

[1] `http://www-verimag.imag.fr/AVERROES/`

Chapitre 2

Critère de terminaison par interprétations polynomiales

L'objectif de ce stage est de formaliser dans `Coq` la preuve qu'il suffit, pour vérifier qu'un système de réécriture termine, d'associer un polynôme à chaque symbole de la signature, et de s'assurer que ces polynômes satisfont certaines conditions aisément vérifiables automatiquement (la positivité de leurs coefficients, principalement). Avant d'entrer plus avant dans les détails de la formalisation proprement dite, nous commençons par donner une description plus en profondeur du théorème que nous cherchons à prouver. À cette fin, nous rappelons quelques définitions relatives aux systèmes de réécriture. Toutefois, seules les notions directement utiles à notre formalisation seront présentées ici. Le lecteur désireux d'en apprendre davantage sur la réécriture pourra par exemple se référer à [2] et [22], ouvrages de référence dans ce domaine.

2.1 Systèmes de réécriture

Nous commençons donc par donner quelques définitions relatives aux systèmes de réécriture.

2.1.1 Signatures

Une signature est la donnée d'un ensemble de symboles et d'une fonction qui à chaque symbole associe un entier naturel appelé *arité* de ce symbole [1]. La plupart du temps, l'ensemble des symboles est fini, mais rien n'interdit qu'il soit infini.

Exemple 1 *Pour travailler en théorie des groupes, on peut prendre une signature constituée de l'ensemble de symboles* $\{0, +, -\}$*, où l'arité de* 0 *est* 0*, l'arité de* $-$ *est* 1*, et l'arité de* $+$ *est* 2.

Dans toute la suite, nous supposerons fixée une signature Σ.

2.1.2 Termes

Soit V un ensemble dénombrable appelé ensemble de variables. On peut alors définir l'ensemble $T(\Sigma, V)$ des termes construits à partir de Σ et V par les clauses suivantes :

- une variable est un terme ;
- pour tout symbole $f \in \Sigma$ d'arité n, si $t_1, \ldots, t_n$ sont des termes, alors $f(t_1, \ldots, t_n)$ est un terme.

Autrement dit, l'ensemble $T(\Sigma, V)$ contient V et est clos par application des symboles de Σ.

[1] L'arité d'un symbole est le nombre d'arguments attendu par ce symbole. Par exemple, l'opérateur d'addition $+$ a deux arguments, il est donc d'arité 2.

Remarque 1 *Comme l'ensemble des variables importe peu, nous n'y ferons plus référence et supposerons par la suite qu'on s'est donné un ensemble V fixé. Une signature Σ ayant été fixée précédemment, l'ensemble $T(\Sigma, V)$ sera noté T dans la suite.*

2.1.3 Contextes

Il est souvent utile de pouvoir travailler sur un sous-terme d'un terme. Pour ce faire, il est nécessaire de formaliser précisément ce que signifie « être un sous-terme ». L'introduction de la notion de contexte facilite grandement l'expression de cette relation.

Intuitivement, un contexte est un terme avec un trou. Plus formellement, les contextes, tout comme les termes, se définissent par induction :
– le contexte vide noté [ ] est un contexte ;
– pour tout contexte C, pour tout symbole f d'arité n, pour tout i, si $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$ sont des termes, alors $f(t_1, \ldots, t_{i-1}, C, t_{i+1}, \ldots, t_n)$ est un contexte.

Si C est un contexte et t un terme, on note $C[t]$ le terme obtenu en remplissant le « trou » de C avec le terme t. Par exemple, si C est le contexte vide, $C[t]$ représente le terme t lui-même.

La relation « s est un sous-terme de t » peut alors se définir ainsi : s est un sous-terme de t ssi il existe un contexte C tel que $t = C[s]$.

Par exemple, x est un sous-terme de $f(x, y)$, car si $C = f([], y)$, on a bien l'égalité $f(x, y) = C[x]$.

2.1.4 Substitutions

Une substitution est une fonction des variables vers les termes, c'est-à-dire de V vers T. Une fois définie sur les variables, cette fonction peut se prolonger naturellement pour être définie sur tous les termes. L'application d'une substitution σ à un terme t, notée $t\sigma$, consiste à remplacer chaque occurrence d'une variable v apparaissant dans t par le terme $\sigma(v)$.

2.1.5 Règles et systèmes de réécriture, terminaison

Maintenant que nous avons défini toutes les notions de base dont nous avons besoin, nous allons introduire les notions fondamentales de règles et de systèmes de réécriture. Nous expliquerons ensuite ce que signifie précisément la relation « t_1 se réécrit en t_2 », et définirons la terminaison d'un système de réécriture.

Une règle de réécriture est un couple (l, r) de termes que l'on note généralement $l \to r$. Pour qu'une règle $l \to r$ soit admise, il est de plus requis que l ne soit pas une variable, et que r ne contienne que des variables apparaissant aussi dans l.

Un système de réécriture est un ensemble de règles de réécriture. Bien que rien n'interdise de considérer des ensembles infinis de règles, nous nous contenterons dans la suite d'utiliser des ensembles finis, mais ceci ne change pas grand chose à la formalisation.

On dit qu'un terme t_1 se réécrit en un terme t_2 par la règle $l \to r$ ssi il existe un contexte C et une substitution σ tels que $t_1 = C[l\sigma]$ et $t_2 = C[r\sigma]$.

Cette définition signifie que si un terme possède un sous-terme qui est une instance de l, alors on peut remplacer ce sous-terme par une instance de r.

Si R est un système de réécriture, on dit que t_1 se réécrit en t_2 par R ssi R contient une règle $l \to r$ telle que t_1 se réécrit en t_2 par $l \to r$ selon la définition précédente.

Un système de réécriture R induit donc une relation R' sur l'ensemble T des termes (il s'agit de la relation $t_1 R' t_2$ ssi t_1 se réécrit en t_2 par R).

On dit que le système de réécriture R termine ssi la relation R' qu'il engendre est bien fondée [2]. En d'autres termes, R termine si, quelque soit la manière dont on réécrit un terme, on se ramène toujours, en un nombre fini d'étapes, à un terme qui ne peut plus être réécrit.

[2] Rappelons qu'une relation $>$ sur un ensemble E est bien fondée ssi pour tout $x \in E$ il n'existe qu'un nombre fini d'éléments $x_1, \ldots, x_n$ tels que $x > x_1 > x_2 > \ldots > x_n$.

Le problème consistant à déterminer si un système de réécriture donné termine est indécidable. Néanmoins, l'intérêt de la réécriture appliquée à des domaines tels que la recherche de preuves ou le calcul symbolique ont motivé la recherche de critères (conditions décidables et suffisantes) qui permettent, dans certains cas, de savoir si un système de réécriture donné termine ou non, et s'il termine, d'exhiber une preuve de terminaison.

Les techniques utilisées pour établir la terminaison d'un système de réécriture peuvent être classées selon différentes catégories :

les méthodes incrémentales et modulaires : elles consistent à déduire la terminaison d'un système de réécriture de celle de sous-systèmes plus petits ;

les méthodes transformationnelles : elles consistent à transformer le système de réécriture de sorte que sa terminaison puisse se déduire de celle du système transformé. L'une des méthodes de cette catégorie est celle des paires de dépendances, décrite dans [1] ;

les méthodes syntaxiques : elles consistent à construire un ordre bien fondé sur les termes, et à montrer que la relation R' engendrée par un système de réécriture R est incluse dans cet ordre, ce qui prouve que R' est bien fondée, et donc que R termine. La plus connue de ces méthodes est RPO (Recursive Path Ordering), décrite dans [13] ;

les méthodes sémantiques : elles consistent à interpréter chaque symbole f d'arité n par une fonction n-aire sur un domaine D à préciser, et à comparer les termes en les interprétant et en comparant leurs interprétations grâce à une relation bien fondée sur D. L'une des méthodes les plus connues de cette catégorie est celle par interprétations polynomiales, que nous nous attacherons à présenter par la suite.

Remarque 2 *Pour prouver la terminaison d'un système de réécriture, on peut être amené à combiner les méthodes vues précédemment. On peut par exemple commencer par ramener la preuve de terminaison d'un système de réécriture à celles de systèmes plus petits, à l'aide de méthodes incrémentales. On pourra ensuite prouver la terminaison des systèmes ainsi obtenus grâce aux méthodes syntaxiques ou sémantiques, et en les ayant au préalable transformés par une méthode transformationnelle.*

2.2 Interprétations

Comme nous l'avons annoncé précédemment, nous portons maintenant notre attention sur les méthodes sémantiques permettant de prouver la terminaison d'un système de réécriture, et plus particulièrement sur la méthode par interprétations polynomiales. Nous commençons par définir la notion d'interprétation monotone bien fondée, qui permet d'énoncer un théorème général sur la terminaison des systèmes de réécriture. Dans une seconde partie, nous définissons une classe particulière d'interprétations monotones bien fondées, les interprétations polynomiales, et nous montrons que les hypothèses du théorème de terminaison se ramènent, dans certains cas, à des vérifications de positivité des coefficients de polynômes.

2.2.1 Interprétations monotones bien fondées

Une interprétation est la donnée d'un ensemble D appelé domaine de l'interprétation, et d'une fonction I qui à tout symbole f d'arité n associe une fonction n-aire I_f sur D. Pour interpréter un terme (c'est-à-dire lui associer un élément de D), nous devons, en plus des symboles de fonction, savoir comment assigner une valeur (un élément de D) aux variables, d'où le recours au concept classique de valuation.

Une valuation est une fonction qui à toute variable associe un élément de D.

L'interprétation $[\![t]\!]^I_\xi$ d'un terme t se définit alors par induction sur t :

- pour toute variable x, $[\![x]\!]^I_\xi = \xi(x)$;
- pour tout symbole f d'arité n, pour tous termes $t_1, \ldots, t_n$.
  $[\![f(t_1, \ldots, t_n)]\!]^I_\xi = I_f([\![t_1]\!]^I_\xi, \ldots, [\![t_n]\!]^I_\xi)$

Remarque 3 *Une substitution n'est en fait qu'une valuation particulière. L'interprétation sous-jacente a pour domaine l'ensemble des termes lui-même, et l'application d'une substitution σ à un terme t revient en fait à interpréter t sous la valuation σ. Cette remarque sera exploitée dans notre formalisation, pour définir la notion de substitution à partir de celle, plus générale, d'interprétation.*

Une interprétation monotone bien fondée est une interprétation dont le domaine est muni d'une relation bien fondée (notée ici $>_D$), et pour laquelle toutes les fonctions d'interprétation sont monotones en chacune de leurs variables. Rappelons qu'une fonction f d'arité n est monotone en sa i-ième variable ssi
$\forall x, y, a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$
$x > y \Rightarrow f(a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_n) > f(a_1, \ldots, a_{i-1}, y, a_{i+1}, \ldots, a_n)$. La relation $>_D$ permet de construire une relation $>_I$ sur l'ensemble T des termes : $t >_I u$ ssi pour toute valuation ξ, $[\![t]\!]^I_\xi >_D [\![u]\!]^I_\xi$. On dira qu'une règle de réécriture $l \to r$ est compatible avec une interprétation I si $l >_I r$.

Nous pouvons maintenant énoncer le théorème principal de terminaison par interprétations :

Théorème 1 *Soit R un système de réécriture. S'il existe une interprétation monotone bien fondée I telle que pour toute règle $l \to r \in R$, $l >_I r$, alors R termine.*

Démonstration 1 *La monotonie des symboles de fonction assure que l'ordre $>_I$ est stable par contexte, c'est-à-dire que pour tout contexte C, pour tous termes t_1 et t_2, $t_1 >_I t_2 \Rightarrow C[t_1] >_I C[t_2]$. Par l'absurde, supposons qu'il existe une suite infinie (t_n) de termes telle que $t_0 \to t_1 \to t_2 \to \ldots \to t_n \to \ldots$. Alors en utilisant la stabilité par contexte de $>$ on a $t_0 >_I t_1 >_I t_2 >_I \ldots >_I t_n >_I \ldots$. Par définition de $>_I$, ceci signifie que pour toute valuation ξ, $[\![t_0]\!]^I_\xi >_D [\![t_1]\!]^I_\xi >_D [\![t_2]\!]^I_\xi >_D \ldots$, ce qui est impossible puisque $>_D$ est bien fondée par hypothèse, ce qui prouve la terminaison de R.*

Remarque 4 *Ce théorème découle en réalité de deux autres résultats. Le premier, connu sous le nom de critère de Manna-Ness, affirme que si la relation engendrée par un système de réécriture est incluse dans un ordre de réduction (relation bien fondée, stable par substitution et contexte), alors ce système de réécriture termine. Le second est que pour toute interprétation monotone bien fondée I sur un domaine D, l'ordre $>_I$ construit à partir de $>_D$ est un ordre de réduction.*

2.2.2 Interprétations polynomiales

Un polynôme à n indéterminées $X_1, \ldots, X_n$ est une somme finie de termes de la forme $ax_1^{k_1} \ldots x_n^{k_n}$, où les k_i sont des entiers naturels et a est un coefficient, élément d'un ensemble à préciser.

Une interprétation polynomiale est une interprétation monotone bien fondée sur $\mathbb{Z}^+$, et telle que chaque symbole de fonction f d'arité n est interprété par un polynôme P_f à n indéterminées à coefficients dans $\mathbb{Z}$. La relation bien fondée utilisée est l'ordre usuel sur les entiers. Dans une interprétation polynomiale, les termes pourront, eux aussi, être interprétés par des polynômes. On notera par la suite P_t le polynôme associé au terme t.

Comme nous l'avons indiqué au début de cette section, dans le cas des interprétations polynomiales, les hypothèses du théorème 1 se ramènent à des tests de positivité de coefficients de polynômes :
- pour que les fonctions polynomiales soient bien définies sur $\mathbb{Z}^+$, il suffit que les polynômes soient à coefficients positifs ou nuls ;
- si un polynôme P est à coefficients positifs ou nuls, et si le coefficient du monôme X_i dans P est strictement positif, alors P est monotone en sa i-ième indéterminée ;
- pour vérifier qu'une règle $l \to r$ est compatible avec l'ordre engendré par l'interprétation, il suffit de s'assurer que $P_l - P_r - 1$ est à coefficients positifs ou nuls.

Nous pouvons donc maintenant énoncer précisément le théorème que nous nous proposons de démontrer dans `Coq` :

Théorème 2 *Soit Σ une signature, I une fonction qui à tout symbole $f \in \Sigma$ d'arité n associe un polynôme I_f à n indéterminées à coefficients dans $\mathbb{Z}$, et R un système de réécriture sur Σ. Si pour tout $f \in \Sigma$ d'arité n, I_f est à coefficients positifs ou nuls, et pour tout $i \leq n$ le coefficient de X_i dans I_f est strictement positif, et si pour toute règle $l \rightarrow r \in R$, $P_l - P_r - 1$ est à coefficients positifs, alors R termine.*

La construction d'une preuve de terminaison comportera en réalité deux étapes distinctes : d'une part, la construction d'une interprétation polynomiale, qui implique la preuve de monotonie des polynômes, et, d'autre part, la vérification de compatibilité des règles avec l'ordre engendré par l'interprétation.

Chapitre 3

Présentation de Coq

Avant d'étudier plus précisément comment le théorème 2 énoncé au chapitre précédent a été formalisé dans `Coq`, nous présentons brièvement cet outil, pour donner une intuition de son pouvoir d'expressivité. Une présentation plus exhaustive de `Coq` peut être trouvée dans son manuel de référence [11]. Une autre, très complète et pédagogique, a été développée dans [4].

Le programme `Coq`, développé au sein du projet LogiCal de l'INRIA, est ce que l'on appelle un logiciel d'aide à la preuve, ou encore un « assistant de preuve ». Ceci signifie que son fonctionnement est basé sur un dialogue permanent entre l'utilisateur qui entre des définitions, des théorèmes et leurs preuves, et le système, chargé de vérifier la correction des objets ainsi construits.

Cette vérification est effectuée en utilisant le formalisme du Calcul des Constructions Inductives [19]. Il s'agit d'un système de types (ensemble de règles permettant d'attribuer un type à des λ-termes) permettant d'exprimer et de raisonner aussi bien s-r des fonctions que des prédicats, et qui permet notamment d'exprimer les propositions et les preuves de la logique d'ordre supérieur.

3.1 Types inductifs

De même que les langages de programmation fonctionnels comme ML, `Coq`permet de définir des types de données inductifs.

Exemple 2 *Dans `Coq`, les entiers de Peano sont définis comme suit :*

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

Cette définition indique que `O` est un entier naturel, et que, étant donné un entier naturel, on construit son successeur grâce à la fonction `S`.

Lorsque des types inductifs tels que `nat` sont définis, le système `Coq` se charge de construire des principes de récurrence qui vont permettre de raisonner avec (faire des calculs sur) des objets de types inductifs.

Exemple 3 *Lorsque l'on définit le type `nat`, `Coq` construit automatiquement le principe de récurrence usuel sur les entiers*

```
nat_ind
     : forall P : nat -> Prop,
       P O -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

qui permet, comme on le fait habituellement en mathématiques pour démontrer qu'un prédicat P est vrai pour tous les entiers, de montrer que P est vrai pour `O` et que si P est vrai pour un entier, alors il l'est pour son successeur.

x

Remarque 5 *Comme nous le verrons en Section 4.2.2, il arrive que les principes de raisonnement générés automatiquement par* `Coq` *ne puissent être utilisés pour travailler sur les objets définis par induction. Il est alors nécessaire de définir soi-même les principes de récurrence dont on a besoin.*

Signalons par ailleurs que `Coq` offre la possibilité de définir des structures (enregistrements), qui sont en réalité représentées dans le système par des types inductifs à un constructeur. Ainsi, la déclaration

```
Record point : Set := mkPoint {
  x : nat;
  y : nat
}.
```

revient à définir un type inductif `point` a un constructeur `mkPoint` qui, à partir de deux arguments de type `nat`, construit un objet de type `point`. Sont également définies deux fonctions `x` et `y` qui permettent d'accéder aux coordonnées d'un point donné.

3.2 Types polymorphes

Il est également possible de définir des types inductifs polymorphes. Voici par exemple la définition `Coq` des listes polymorphes présentes dans les langages fonctionnels classiques :

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

Cette définition stipule que pour tout type de donnée `A`, la liste vide `(nil A)` est une liste d'éléments de `A`, et qu'il est possible de construire une liste d'éléments de `A` à partir d'une liste déjà construite et d'un nouvel objet de type `A`.

La liste contenant les entiers 1, 2 et 3 peut donc être représentée par le terme `Coq` `(cons nat 1 (cons nat 2 (cons nat 3 (nil nat))))`. Cependant, il semble évident que le premier argument de `cons`, ici `nat`, devrait pouvoir être inféré par le système à partir du type du second argument, tandis que l'argument de `nil` devrait pouvoir être calculé par le système, au moins dans certains cas, d'après le contexte dans lequel `nil` est rencontré. `Coq` dispose donc d'un mécanisme d'arguments implicites. Il permet, dans certains cas, d'éviter de spécifier tous les arguments d'une application, laissant au système le soin de les inférer. Ainsi, après avoir donné au système les commandes

```
Implicit Arguments cons [A].
Implicit Arguments nil [A].
```

nous pouvons réécrire la liste précédente en omettant `nat` :
`(cons 1 (cons 2 (cons 3 nil)))`. `Coq` dispose en outre de notations qui permettent d'alléger encore l'écriture de certains termes, ces notations pouvant être définies par l'utilisateur.

3.3 Types dépendants

Contrairement aux types inductifs et polymorphes vus précédemment, les types dépendants n'existent pas dans les langages fonctionnels classiques. Un type dépendant est un type qui peut être paramétré non seulement par un autre type (cas des types polymorphes vu précédemment), mais aussi par des valeurs. Cette nouvelle forme de paramétrisation permet d'établir des contraintes sur les données incluses dans le type considéré, et ainsi d'avoir des types plus sûrs.

Exemple 4 *Grâce aux types dépendants, il est possible de définir le type des vecteurs de longueur n (listes comportant exactement n éléments). La définition prend la forme suivante :*

```
Inductive vector (A : Set) (n : nat) : Set :=
  | Vnil  : vector A 0
  | Vcons : A -> forall n : nat, vector A n -> vector A (S n)
```

Cette définition indique que `Vnil A` *est un vecteur d'éléments de* `A` *de taille* 0*, et que si* `a` *est de type* `A`*,* `n` *de type* `nat` *et* `v` *de type* `vector A n`*, alors* `Vcons a n v` *est de type* `vector A (S n)`*.*

Ici aussi, le mécanisme des arguments implicites présenté précédemment permet d'alléger les notations.

3.4 Preuves et tactiques

Comme nous l'avons indiqué plus haut, les preuves sont construites de manière interactive. La première étape de la construction d'une preuve consiste à entrer l'énoncé du théorème à prouver, et déclenche l'ouverture du mode d'édition de preuves de `Coq`. Celui-ci présente à l'utilisateur le but à prouver, en séparant les hypothèses de la conclusion par une barre horizontale.

L'utilisateur construit sa preuve en partant de la conclusion, et en revenant aux hypothèses. L'arbre de preuve est construit pas à pas, par l'application de *tactiques*. Une tactique peut être vue comme une fonction qui, lorsqu'elle est appliquée à un but B donné, génère une liste (éventuellement vide) de buts dont le système saura déduire B.

Une fois la preuve terminée, elle est vérifiée par le système, puis, si elle est correcte, elle est sauvegardée et ajoutée à l'environnement courant. Le théorème ainsi prouvé peut désormais être utilisé dans des preuves ultérieures.

Le système `Coq` est fourni avec un grand nombre de tactiques, qui permettent d'automatiser un certain nombre de preuves telles que les preuves d'égalités dans les anneaux et les corps (`ring` et `field`), les preuves d'inégalités linéaires en arithmétique (`omega`), et les preuves en logique propositionnelle (`intuition`). D'autres tactiques, telles que `auto`, fonctionnent en recherchant, dans des bases de données construites par l'utilisateur, un lemme susceptible de résoudre le but en cours.

Enfin, le système `Coq` dispose d'un langage nommé `Ltac` et permettant à l'utilisateur d'écrire ses propres tactiques, et donc d'automatiser d'autres preuves. Outre des combinaisons de tactiques existantes, ce langage permet également de faire du filtrage sur le but, et donc de choisir quelle tactique appliquer en fonction de la forme de celui-ci.

Chapitre 4

Formalisation des interprétations polynomiales dans Coq

Après avoir présenté `Coq`, nous allons maintenant expliquer plus en détails comment les différents concepts et théorèmes présentés au chapitre 2 ont été formalisés. Pour illustrer notre propos, nous donnerons les définitions `Coq` des principaux types utilisés par notre formalisation. Celle-ci est disponible dans son intégralité à l'adresse `http://www.loria.fr/~blanqui`, et a été développée en utilisant la version 8.0 de `Coq` (dernière version publiée). Nous commençons par présenter la structure de vecteur introduite à la section 3.3. Les sections suivantes décrivent la formalisation des notions de réécriture, des polynômes et des outils permettant d'établir la compatibilité d'une règle avec l'ordre engendré par une interprétation polynomiale. L'avant-dernière section de ce chapitre est consacrée aux tactiques qui ont été écrites pour automatiser les preuves de terminaison. Enfin, le chapitre se termine par la présentation complète de la construction de la preuve de terminaison pour un système de réécriture simple sur les groupes, utilisant les tactiques et théorèmes vus aux sections précédentes.

4.1 Vecteurs

Comme nous l'avons signalé précédemment, le calcul de l'interprétation d'un terme nécessite que l'on connaisse tous les arguments à passer aux fonctions qui interprètent les symboles. En d'autres termes, le calcul de l'interprétation du terme $f(t_1, \ldots, t_n)$ n'est possible que si l'on dispose bien de n sous-termes, où n est l'arité de f. C'est pour cette raison que nous avons été amenés à nous intéresser aux vecteurs, qui permettent justement de garantir que cette condition sera vérifiée, alors que des listes, n'offrant aucune garantie quant à leur longueur, ne le permettent pas.

Le type `vector`, introduit en section 3.3, est défini dans le module `Bvector` de la bibliothèque standard de `Coq`. Ce module fournit aussi quelques fonctions élémentaires sur les vecteurs. Néanmoins, comme son objectif est surtout la définition et la manipulation de vecteurs de bits, le nombre de fonctions disponibles pour manipuler des vecteurs d'un type quelconque est assez réduit. En outre, la bibliothèque standard de `Coq` ne fournit aucun théorème sur les vecteurs. Nous avons donc été conduits à définir un certain nombre de fonctions, de prédicats et de théorèmes nous permettant de manipuler plus aisément les vecteurs, structure qui, comme on le verra par la suite, intervient à plusieurs reprises dans notre formalisation.

Les principales fonctions que nous avons écrites sont les suivantes :

Vcast : permet de transformer un vecteur de taille m en un vecteur de taille n si $m = n$;

Vnth : permet d'extraire le i-ième élément d'un vecteur de taille n. Les cases des vecteurs étant numérotées de 0 à $n - 1$, il faut fournir à cette fonction une preuve que $i < n$ pour lui permettre de renvoyer un résultat ;

Vbreakn : permet de couper un vecteur de longueur $n_1 + n_2$ en deux vecteurs, l'un de longueur n_1, l'autre de longueur n_2 ;

Vforall : il s'agit d'un prédicat permettant d'exprimer que tous les éléments d'un vecteur vérifient une propriété ;

En plus de ces fonctions, plusieurs théorèmes les concernant ont été prouvés. D'autres fonctions du module `Bvector` ont été réécrites et simplifiées, pour tenir compte des nouvelles fonctionnalités offertes par le système `Coq` (arguments implicites notamment). Ce travail devrait être intégré à la bibliothèque standard de `Coq` à plus ou moins brève échéance.

4.2 Réécriture

Nous allons maintenant montrer comment les diverses notions décrites en 2.1 ont été formalisées dans `Coq`. Certaines notions peuvent être spécifiées assez facilement, et ont une définition `Coq` très proche de leur définition abstraite. Pour d'autres notions, plusieurs possibilités s'offraient à nous pour les représenter dans `Coq`, et il a donc fallu en choisir une. Pour chacune de ces notions, nous nous efforcerons de montrer les différentes options possibles, et d'expliquer pourquoi notre choix s'est porté sur l'une d'entre elles en particulier.

4.2.1 Signature

Dans les travaux relatifs à la formalisation de réécriture dans `Coq` présentés en introduction, Rouyer [20, 21] et de Kleijn [12] représentent une signature par l'ensemble des entiers naturels. Chaque symbole est alors représenté par un entier, et l'on autorise des arités quelconques. Par ailleurs, cette formalisation autorise un nombre infini de symboles.

Il nous a cependant semblé que cette façon de procéder ne conviendrait pas pour notre formalisation. En effet, si nous voulons profiter de la richesse du système de types de `Coq` pour assurer que chaque terme de la forme $f(t_1, \ldots, t_n)$ a le bon nombre de sous-termes, il nous faut donner l'arité de chaque symbole. En outre, la plupart des signatures qu'on est amené à utiliser ne comportent qu'un nombre fini de symboles. Se donner *à priori* un ensemble infini de symboles ne paraît donc pas souhaitable, d'autant plus que nous avons à démontrer des propriétés quantifiées universellement sur les symboles (par exemple : pour tout symbole, le polynôme qui lui est associé est à coefficients positifs). Si nous avions adopté la solution précédente, nous aurions donc dû faire en sorte que la propriété à démontrer soit vraie pour les symboles qui n'étaient pas effectivement utilisés, ce qui aurait constitué un surcroît de travail bien inutile.

Nous avons donc choisi une autre manière de formaliser les signatures, qui nous permet de définir des signatures finies ou infinies, et de préciser l'arité de chaque symbole. La définition `Coq` de signature que nous utilisons est la suivante :

```
Record Signature : Type := mkSignature {
  symbol :> Set;
  arity : symbol -> nat
}.
```

Cette définition permet que `symbol` soit n'importe quel ensemble, tandis que `arity` définit une fonction de cet ensemble vers les entiers naturels, associant à chaque symbole son arité.

Définir une signature consistera donc à construire d'abord un ensemble de symboles, puis à définir la fonction arité sur cet ensemble, puis enfin à regrouper ces éléments au sein d'une structure de type `Signature`.

4.2.2 Termes

Tout comme pour les signatures, il existe plusieurs possibilités pour représenter des termes du premier ordre dans `Coq`. La question est essentiellement de savoir comment représenter les sous-termes d'un terme dont le symbole de tête est une fonction. La solution adoptée par les formalisations existantes consiste à utiliser une liste de termes. Cette solution a néanmoins un inconvénient : on ne peut pas être sûr par typage qu'un terme ayant pour racine un symbole f d'arité n aura exactement n sous-termes (*i.e.* sera bien formé). Or, être sûr qu'un terme a le bon nombre de sous-termes se révélera indispensable pour certaines étapes de la formalisation, par exemple lorsqu'il s'agit de calculer l'interprétation d'un terme.

L'utilisation de listes de termes ne semble donc pas souhaitable ici. C'est pourquoi, nous lui avons préféré l'utilisation de vecteurs, qui permettent de s'assurer par typage qu'un terme est syntaxiquement correct. La définition des termes se présente alors comme suit :

```
Inductive term (Sig : Signature) : Set :=
  | Var : variable -> term
  | Fun : forall f : symbol Sig, vector term (arity f) -> term.
```

Cette définition affirme qu'un terme peut être construit soit à l'aide du constructeur `Var` qui attend un argument de type `variable` (c'est-à-dire `nat`), soit à partir du constructeur `Fun`, auquel il faut fournir comme arguments un symbole de fonction, et un vecteur de termes de longueur l'arité de ce symbole.

Insistons bien sur le fait que nous n'avons besoin d'aucune vérification supplémentaire sur les objets de type `term` pour garantir leur correction. C'est le système de typage de `Coq` qui assure que les termes sont corrects : il est impossible de construire un terme mal formé, car les termes mal formés sont rejetés par le système de typage de `Coq`.

Remarque 6 *Comme nous l'avions indiqué en section 5, il arrive que les principes de raisonnement sur les types inductifs générés automatiquement par* `Coq`*, bien que corrects, ne permettent pas de raisonner effectivement sur les types inductifs. Par exemple, lors de la déclaration du type* `term` *vue précédemment, le principe d'induction* `term_ind` *généré par* `Coq` *est le suivant :*

```
term_ind
     : forall P : term -> Prop,
       (forall v : variable, P (Var v)) ->
       (forall (f : Sig) (v : vector term (arity f)), P (Fun f v)) ->
       forall t : term, P t
```

Avec ce principe de raisonnement, il est impossible de prouver certaines propriétés sur les termes. Plus précisément, c'est l'étape de récurrence qui n'est pas démontrable, puisqu'on ne dispose d'aucune hypothèse sur le vecteur `v` *de sous-termes à partir de laquelle déduire* `P(Fun f v)`*. Il a donc été nécessaire de définir manuellement un principe de récurrence permettant de raisonner sur les termes, et dont voici le type :*

```
term_ind
     : forall (P : term -> Prop) (Q : forall n : nat, vector term n -> Prop),
       (forall v : variable, P (Var v)) ->
       (forall (f : Sig) (v : vector term (arity f)),
        Q (arity f) v -> P (Fun f v)) ->
       Q 0 Vnil ->
       (forall (t : term) (n : nat) (v : vector term n),
        P t -> Q n v -> Q (S n) (Vcons t v)) -> forall t : term, P t
```

Ce principe de récurrence permet de démontrer qu'une propriété `P` *est vraie pour tout terme en s'aidant d'une propriété* `Q` *des vecteurs de termes. Les 4 hypothèses à démontrer pour prouver que* `P` *est vraie pour tout terme* `t` *sont les suivantes :*

1. `P` *est vraie pour tout terme de la forme* `Var v`*, où* `v` *est de type* `nat`*;*
2. *soit* `f` *un symbole de fonction, et soit* `v` *un vecteur comportant exactement* `arity f` *termes. Alors* `Q v` *implique* `P (Fun f v)`*;*
3. `Q` *est vraie pour le vecteur nul* `Vnil`*;*
4. *soit* `t` *un terme,* `n` *de type* `nat` *et* `v` *un vecteur de* `n` *termes. Alors* `P t` *et* `Q v` *implique* `Q (Vcons t v)`*.*

4.2.3 Contextes

La définition des contextes (termes à trous), est assez proche de celle des termes. Elle fait cependant apparaître une difficulté supplémentaire. Dans la définition des contextes que nous avons donnée en 2.1.3, nous avons affirmé que si f est un symbole, $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$ des termes et C un contexte, alors $f(t_1, \ldots, t_{i-1}, C, t_{i+1}, \ldots, t_n)$ est aussi un contexte. Les points de suspension traduisent ici le fait qu'on ne peut pas donner une liste explicite des termes de t_1 à t_{i-1}, car cette liste dépend de i, tandis que la liste des termes de t_{i+1} à t_n dépend à la fois de i et de n.

Il va de soi qu'un système tel que `Coq` ne permet pas d'utiliser des points de suspension, comme on le fait généralement dans une définition plus informelle. La solution passe, encore une fois, par l'utilisation de vecteurs de termes. Une première idée consiste à dire que si f est un symbole d'arité n, v_i et v_j des vecteurs de termes de longueurs respectives i et $n-i-1$ et C un contexte, alors $f(v_i, C, v_j)$ est aussi un contexte (car $i+1+(n-i-1)=n$).

Cette solution n'est cependant pas celle que nous avons choisie. En effet, si l'on procède ainsi, on est obligé, pour connaître la longueur du vecteur v_j, de calculer $n-i-1$, et donc d'utiliser la soustraction sur les entiers naturels. Pour que celle-ci soit bien définie, on serait en outre obligé de supposer que $i \leq n$, ce qui, bien que vrai, ne ferait qu'alourdir une définition déjà complexe.

Nous utilisons donc une définition légèrement plus simple, puisqu'elle ne fait pas intervenir la soustraction sur les entiers naturels. Pour éviter de recourir à cette opération, nous introduisons deux entiers naturels i et j, et demandons que la somme $i+j+1$ soit égale à l'arité du symbole f. La définition `Coq` des contextes est finalement la suivante :

```
Inductive context (Sig : Signature) : Set :=
  | Hole : context
  | Cont : forall f : symbol Sig,
         forall i j : nat, i + S j = (arity f) ->
         Terms i -> context -> Terms j -> context.
```

Le constructeur `Hole` représente ici le contexte vide, tandis que `Cont` permet de construire un contexte récursivement, à partir d'un contexte déjà construit. Quant à `Terms`, il s'agit d'une abréviation pour `vector (term Sig)`.

4.2.4 Interprétations et substitutions

Comme nous l'avons indiqué à la section 3, les substitutions ne sont que des interprétations particulières. Nous commençons donc par expliquer comment nous définissons les interprétations en `Coq`, puis nous montrons comment définir les substitutions à partir du cadre plus général.

Une interprétation peut être représentée par une structure `Coq` contenant deux champs : le domaine de l'interprétation, et une fonctionnelle qui à chaque symbole d'arité n associe une fonction n-aire sur ce domaine. Voici la définition `Coq` de cette structure :

```
Record interpretation (Sig : Signature) : Type := mkInterpretation {
  domain : Set;
  symb_int : forall f : symbol Sig, naryFunction domain (arity f)
}.
```

L'identificateur `naryFunction` désigne une constante que nous avons définie, et qui représente le type des fonctions n-aires sur un domaine, ce domaine et n étant des paramètres à passer à `naryFunction`.

Nous avons aussi défini la notion de valuation associée à une interprétation : c'est une fonction des variables vers le domaine de l'interprétation. Enfin, il s'agit de définir la fonction d'interprétation qui à un terme et une valuation donnés associe un élément du domaine d'interprétation. Bien que la définition mathématique de cette fonction soit assez simple, son écriture en `Coq` pose quelques problèmes pratiques, car les critères utilisés par `Coq` pour décider si une fonction termine (et donc pour l'accepter) sont assez restrictifs. On ne peut donc pas écrire la fonction d'interprétation aussi intuitivement qu'on le ferait dans un langage de programmation fonctionnel traditionnel. Il est nécessaire d'écrire deux fonctions récursives imbriquées : l'une sur les termes, l'autre sur les vecteurs de termes. Cependant, comme la fonction d'interprétation des vecteurs de termes est définie à l'intérieur de la fonction d'interprétation des termes, elle n'est pas visible de l'extérieur de cette fonction. Donc, si l'on souhaite disposer, dans le reste du développement, d'une fonction d'interprétation des vecteurs de termes, on est obligé de dupliquer la définition de cette fonction, le premier exemplaire de celle-ci se trouvant à l'intérieur de la fonction d'interprétation des termes, le second à l'extérieur, utilisable dans le reste du développement.

Remarque 7 *A moyen terme, l'implantation dans Coq de critères de terminaison plus souples, permettant de définir ce type de fonctions d'une manière plus intuitive, est envisagée. Ce stage participe d'ailleurs à la mise en place d'une telle extension. On sait en effet que l'on peut étendre Coq par tout système de réécriture du premier ordre terminant, (localement) confluent, non-dupliquant et linéaire-gauche [16].*

Les interprétations monotones bien fondées, quant à elles, sont définies comme suit :

```
Record WFMinterpretation (Sig : Signature) : Type := mkWFMInterpretation {
  WFMI_Interp : interpretation Sig;
  WFMI_a : domain WFMI_Interp;
  WFMI_R : relation (domain WFMI_Interp);
  WFMI_RWF : wellfounded WFMI_R;
  WFMI_symb_monotone :
    forall f : symbol Sig, Monotone WFMI_R (symb_int WFMI_Interp f)
}.
```

Quelques explications quant aux champs de cette structure :
- `WFMI_a` permet de s'assurer que le domaine de l'interprétation est non vide. Si le domaine d'interprétation était vide, aucune valuation ne pourrait être construite, et l'ordre sur les termes engendré par une interprétation monotone bien fondée ne pourrait être défini correctement ;
- `WFMI_R` est la relation sur le domaine d'interprétation à partir de laquelle celle sur les termes est construite, et `WFMI_RWF` est une preuve que `WFMI_R` est bien fondée ;
- `WFMI_symb_monotone` est une fonction qui a tout symbole associe la preuve que la fonction par laquelle il est interprété est monotone.

Enfin, comme nous l'avons signalé au début de cette section, les substitutions sont construites comme des interprétations particulières. On définit une interprétation dont le domaine est l'ensemble des termes, et la fonctionnelle qui à un symbole de fonction associe une fonction n-aire est le constructeur `Fun`. Les substitutions sont alors les valuations sur cette interprétation, et, pour une substitution σ, le calcul de $t\sigma$ revient à calculer l'interprétation de t sous la valuation σ dans l'interprétation ainsi construite.

4.2.5 Règles et systèmes de réécriture

La représentation des règles et des systèmes de réécriture a évolué en cours de formalisation. Nous avions en effet choisi d'être aussi général que possible, puis nous avons constaté que ces choix, bien que théoriquement intéressants, rendaient malaisée l'utilisation pratique des structures de données que nous avions définies. Nous examinons donc chacune de ces deux entités en montrant, pour chacune, comment sa représentation a évolué durant la formalisation.

Règles de réécriture Notre premier choix a consisté à représenter une règle comme un couple de termes, sans lui imposer les conditions habituellement requises, à savoir que le membre gauche ne doit pas être une variable, et que toutes les variables apparaissant dans le membre droit doivent figurer aussi dans le membre gauche. Ces conditions, qui ne nous semblaient alors pas indispensables, ce sont révélées nécessaires par la suite, lorsque nous avons eu à calculer les polynômes associés aux termes. Elles permettent en effet de s'assurer que le membre droit est interprété par un polynôme ne nécessitant pas plus d'indéterminées que le membre gauche. Une règle de réécriture est donc définie de la façon suivante (avec `varlist` désignant une fonction qui calcule la liste des variables figurant dans un terme, et `incl` représentant l'inclusion sur les listes) :

```
Record rule (Sig : Signat-re) : Set := mkRule {
  lft : term Sig;
  rht : term Sig;
  rule_is_correct : (forall x : variable, lft <> (Var Sig x)) /\
  incl (varlist rht) (varlist lft).
}.
```

Remarque 8 *La formalisation actuelle n'utilise pas l'hypothèse de correction qui affirme que le membre gauche ne doit pas être une variable. Néanmoins, comme d'une part cette condition est facile à vérifier, et comme d'autre part elle interviendra nécessairement dans des développements futurs utilisant cette définition des règles de réécriture, il nous a semblé préférable de la mentionner dès à présent.*

Systèmes de réécriture La formalisation initiale des systèmes de réécriture s'inspirait de celle des signatures. Nous avions en effet choisi de représenter un système de réécriture en se donnant un ensemble abstrait qui servait à indexer les règles, et une fonction qui, à chaque élément de cet ensemble, associait une règle. Cette manière de procéder avait l'avantage de permettre de représenter aussi bien des systèmes possédant un nombre fini de règles, que des systèmes avec un nombre infini de règles. Nous avons néanmoins choisi de renoncer à cette formalisation, car elle rendait la vérification de propriétés sur les règles plus complexe que d'autres représentations, dans la mesure où il fallait démontrer cette propriété pour tout élément de l'ensemble des indices, ce qui pouvait se révéler difficile à automatiser, par exemple si la démonstration se faisait par récurrence.

Ainsi, puisque d'une part la plupart des systèmes de réécriture utilisés en pratique ne comportent qu'un nombre fini de règles, et puisque, d'autre part, une formalisation tenant mieux compte de cet état de fait rendait plus facile la preuve de nos théorèmes sur les systèmes de réécriture, il nous a semblé préférable d'abandonner la représentation choisie initialement, au profit d'un autre, mieux adaptée. Nous représentons donc un système de réécriture par une liste de règles. La vérification de la compatibilité d'un système de réécriture avec une interprétation polynomiale consiste donc à vérifier que tous les éléments d'une liste ont une certaine propriété, ce qui implique la vérification d'une conjonction finie de propositions, cette vérification étant facile à automatiser, pourvu que chaque propriété élémentaire puisse être vérifiée automatiquement.

4.3 Polynômes

Comme nous l'avons vu en section 2.2.2, les conditions que doivent vérifier les polynômes pour que l'on puisse les utiliser dans la construction d'une interprétation polynomiale peuvent être ramenées à des conditions suffisantes sur leurs coefficients. Il s'agit maintenant de prouver dans `Coq` les résultats précédemment énoncés.

Par ailleurs, nous avons vu qu'une règle $l \to r$ est compatible avec l'ordre engendré par une interprétation polynomiale si $P_l - P_r > 0$ en tout point, c'est-à-dire si $P_l - P_r - 1 >= 0$ en tout point. Il est donc nécessaire, d'une part de pouvoir calculer P_l et P_r, et, d'autre part, de pouvoir montrer que si $P_l - P_r - 1$ est à coefficients positifs, alors $P_l - P_r - 1 >= 0$ en tout point.

Il semble donc indispensable de disposer d'une représentation des polynômes qui permette de réaliser les calculs dont nous avons besoin, ainsi que de prouver les théorèmes permettant de déduire des propriétés des fonctions polynomiales de celles des coefficients des polynômes.

À notre connaissance, la seule formalisation des polynômes dans `Coq` qui a été réalisée jusqu'à présent est celle de la bibliothèque C-CoRN (Constructive `Coq` Repository at Nijmegen) (*Cf.*

`http://vacuumcleaner.cs.kun.nl/c-corn/`). L'objectif de cette bibliothèque est de formaliser un grand nombre de théorèmes mathématiques dans plusieurs domaines (algèbre, analyse réelle et complexe, topologie...), et ceci en logique intuitionniste. Cependant, les seuls polynômes qui ont été formalisés sont ceux à une indéterminée. Nous ne pouvons donc pas espérer réutiliser ce travail. En outre, la taille de cette bibliothèque et sa durée de compilation (près d'une demie-heure sur une machine rapide) nous dissuadent d'y recourir.

Nous avons donc choisi de développer notre propre formalisation des polynômes. Nous commençons par discuter de la représentation des polynômes, avant de décrire la formalisation que nous avons effectuée. Cette formalisation se divise en trois parties : d'abord la définition des polynômes à coefficients dans $\mathbb{Z}$ et des opérations s'y rapportant, puis l'étude des polynômes à coefficients positifs et de leurs propriétés, et enfin la définition des polynômes monotones, c'est-à-dire des polynômes engendrant des fonctions polynomiales monotones pour la relation d'ordre usuelle sur les entiers naturels.

4.3.1 Représentation des polynômes

Comme nous l'avons indiqué précédemment, le calcul du polynôm associé à un terme nécessite la définition de la composée de polynômes. Or, cette opération n'a de sens que si l'on connaît le nombre d'indéterminées des polynômes que l'on souhaite composer. En effet, si P est un polynôme à k indéterminées, l'opération $P(P_1, \ldots, P_m)$ n'a de sens que si $m = k$. Nous allons donc chercher à décrire des polynômes avec un nombre donné d'indéterminées. Autrement dit, le type des polynômes dont nous avons besoin est un type dépendant, paramétré par un entier, à savoir le nombre d'indéterminées.

Nous cherchons donc à représenter un polynôme à n indéterminées, c'est-à-dire une somme de monômes de la forme $ax_1^{k_1} \ldots x_n^{k_n}$. Les monômes ainsi définis peuvent en réalité être dissociés en deux parties : d'une part leur coefficient, ici a, et, d'autre part, le produit $x_1^{k_1} \ldots x_n^{k_n}$. Comme ce dernier produit sera souvent considéré comme un objet isolé, nous choisissons, contrairement à ce qui est fait habituellement, de lui réserver l'appellation de monôme. Pour nous, un monôme est donc un objet de la forme $x_1^{k_1} \ldots x_n^{k_n}$, le produit $ax_1^{k_1} \ldots x_n^{k_n}$ ne recevant aucune appellation particulière. Reste maintenant à choisir une représentation pour les monômes, à déterminer quel doit être le type des coefficients, et finalement à choisir une représentation pour les polynômes.

Les monômes ayant la forme $x_1^{k_1} \ldots x_n^{k_n}$, où les k_i sont des entiers, il semble naturel de les représenter par des vecteurs d'entiers de longueur n. Si i est un entier compris entre 0 et $n-1$, la i-ième case du vecteur v représentant le monôme $x_0^{k_0} \ldots x_{n-1}^{k_{n-1}}$ contiendra k_{i+1}.

Le type à donner aux coefficients est, quant à lui, plus sujet à discussion. Une première idée qui pourrait paraître séduisante serait de prendre pour coefficients des entiers naturels. L'intérêt d'un tel choix serait que les polynômes ainsi construits seraient à valeurs dans $\mathbb{N}$ par construction, ce qui dispenserait de toute vérification supplémentaire quant à la positivité des polynômes.

Néanmoins, comme nous l'avons signalé précédemment, la vérification de la compatibilité d'une règle $l \rightarrow r$ avec une interprétation polynomiale nécessite de pouvoir calculer $P_l - P_r - 1$, et donc de pouvoir faire des soustractions sur les polynômes, c'est-à-dire sur leurs coefficients. La soustraction n'ayant pas de bonnes propriétés sur $\mathbb{N}$, cette première idée doit être abandonnée, au profit de coefficients dans $\mathbb{Z}$, ensemble sur lequel la soustraction a les propriétés que l'on en attend habituellement. Nous pourrons donc calculer facilement la différence de deux polynômes, la contrepartie étant que la positivité des polynômes ne sera plus vraie par typage, mais devra faire l'objet de vérifications supplémentaires.

Reste à choisir la représentation finale des polynômes. Là encore, plusieurs possibilités se présentent. L'une d'entre elles consiste à représenter un polynôme par une fonction qui à tout monôme associe un coefficient. Cette représentation abstraite peut sembler séduisante, du fait de la simplicité avec laquelle elle permet de décrire certaines opérations telles que l'addition de polynômes. Elle s'avère cependant inutilisable dès qu'il s'agit d'évaluer un polynôme, c'est-à-dire d'en donner la valeur en un point particulier. En effet, pour évaluer un polynôme, il est nécessaire d'en connaître tous les coefficients, ce qui est impossible si un polynôme est représenté par une fonction des monômes vers les coefficients, car une telle fonction n'est pas à domaine fini. Il faut donc choisir une représentation plus concrète et plus aisée à utiliser pour le calcul. L'idée la plus simple est alors de représenter un polynôme par une liste de couples (*coefficient*, *monome*). Cette représentation est simple, et permet de spécifier aisément toutes les opérations sur les polynômes, même l'évaluation.

L'inconvénient principal de cette représentation est qu'elle n'assure pas que les polynômes soient représentés de manière canonique, c'est-à-dire qu'un monôme apparaisse une fois au plus. D'autres représentations garantissant une forme canonique ont été envisagées puis abandonnées, car elles n'utilisaient pas un type inductif, et donc ne permettaient pas un raisonnement facile sur les polynômes.

Nous représentons donc un polynôme à n indéterminées par une liste de couples dont le premier élément est un coefficient (élément de $\mathbb{Z}$), et le second est un vecteur d'entiers de longueur n contenant les exposants des indéterminées pour ce monôme.

4.3.2 Polynômes à coefficients dans $\mathbb{Z}$

La première partie de la formalisation des polynômes traite des polynômes à coefficients dans $\mathbb{Z}$ en général. Elle se divise en plusieurs sections, dont nous résumons brièvement le contenu :

- les définitions : on y définit les monômes et les polynômes, et on y prouve que l'égalité sur les monômes est décidable, résultat que l'on réutilisera à plusieurs reprises par la suite. Sont également définis des monômes et polynômes particuliers qui interviennent fréquemment dans la formalisation : le monôme où seule l'indéterminée i apparaît, avec une puissance 1 (c'est un vecteur dont tous les éléments valent 0 sauf le i-ième qui vaut 1), le polynôme nul, le polynôme constant. Dans toute la suite, nous noterons M_i le polynôme défini par $M_i = x_i$. Enfin, on définit une fonction qui, étant donné un monôme M et un polynôme P renvoie le coefficient de M dans P. Il est à noter que la valeur renvoyée par cette fonction est correcte même si P n'est pas sous forme canonique, puisque la fonction calcule la somme des coefficients de chacune des occurrences de M dans P. Cette fonction fournit de plus un premier exemple d'utilisation de la décidabilité de l'égalité sur les monômes, puisqu'on a besoin pour la calculer de comparer des monômes et de prendre une décision en fonction du résultat de cette comparaison ;
- l'addition : on commence par définir l'addition d'un couple (*coefficient*, *monome*) à un polynôme, puis on définit l'addition des polynômes P_1 et P_2 en itérant la fonction précédente sur tous les éléments de la liste P_1. Il est à remarquer que l'addition aurait pu être spécifiée plus simplement en concaténant les listes P_1 et P_2, mais cette façon de faire n'aurait pas conservé la forme canonique, alors que l'algorithme que nous avons choisi la respecte ;
- la multiplication : elle aussi est définie par étapes. On commence par définir une multiplication sur les monômes (ceci consiste simplement à ajouter terme à terme les vecteurs contenant les puissances des indéterminées), avant de définir la multiplication d'un monôme par un polynôme, puis enfin la multiplication de deux polynômes. Cette section comporte aussi une fonction permettant d'élever un polynôme donné à une certaine puissance, fonction utilisée dans le calcul de la composée de polynômes ;
- la composition de polynômes : on commence par définir la composée d'un monôme M à n indéterminées et des polynômes $P_1, \ldots, P_n$, que l'on passe naturellement à la fonction en les stockant dans un vecteur de polynômes de taille n. Si $M = x_1^{k_1} \ldots x_n^{k_n}$, alors $M(P_1, \ldots, P_n) = P_1^{k_1} \ldots P_n^{k_n}$, d'où l'utilité de la fonction puissance sur les polynômes, définie précédemment. La composée de polynômes s'écrit ensuite assez aisément ;
- l'évaluation de polynômes : il s'agit de calculer la fonction polynomiale associée à chaque polynôme. En d'autres termes, si P est un polynôme à n indéterminées et à valeurs dans A, on cherche à construire la fonction de A^n dans A qui, étant donnés n éléments $a_1, \ldots, a_n$ de A calcule $P(a_1, \ldots, a_n)$. Par la suite, nous appellerons cette fonction qui a un polynôme associe une fonction polynomiale la fonction d'évaluation des polynômes. Comme précédemment, on commence par définir la fonction d'évaluation sur les monômes, avant de la définir sur les polynômes ;
- une section contenant divers lemmes relatifs à l'évaluation de polynômes. Il s'agit surtout de

montrer qu'elle est compatible avec les opérations usuelles sur les polynômes, c'est-à-dire, par exemple, que la fonction polynomiale associée à une somme de polynômes est égale à la somme des fonctions polynomiales.

4.3.3 Polynômes à coefficients positifs

Nous avons vu précédemment qu'il était important de pouvoir exprimer le fait qu'un polynôme est à coefficients positifs, et de disposer de théorèmes concernant ce type de polynômes.

La formalisation en `Coq` de la propriété de positivité des coefficients peut se faire de deux manières non équivalentes. La première possibilité consiste à dire qu'un polynôme P est à coefficients positifs ssi pour tout monôme M, le coefficient de M dans P est supérieur ou égal à 0. Cette définition a l'avantage d'être correcte même pour les polynômes qui ne sont pas sous forme canonique, mais elle s'avère difficile à utiliser comme hypothèse dans les démonstrations et à vérifier pour un polynôme donné.

La deuxième possibilité consiste à dire qu'un polynôme est à coefficients positifs ssi tous les coefficients apparaissant dans la liste qui le représente sont positifs. Ainsi, selon cette définition, $2x - x$ n'est pas à coefficients positifs, car -1 n'est pas positif. Cette seconde définition est cependant correcte lorsque les polynômes sont en forme canonique, et même lorsqu'ils contiennent des monômes apparaissant plusieurs fois, pourvu que ce soit toujours avec des coefficients positifs.

Il faut néanmoins remarquer que les deux définitions précédentes coïncident lorsque les polynômes sont sous forme canonique, c'est-à-dire pour les polynômes où chaque monôme est autorisé à apparaître au plus une fois. Cette définition a en outre l'avantage d'être facile à utiliser dans les démonstrations, et à vérifier automatiquement pour des polynômes donnés.

En plus des deux définitions énoncées précédemment, le développement autour des polynômes à coefficients positifs a essentiellement consisté à prouver que les opérations d'addition, de multiplication et de composition préservent la positivité des coefficients. L'évaluation de polynômes sur $\mathbb{Z}^+$ a aussi été définie. Il s'agit d'une fonction qui à un polynôme à n indéterminées et à une preuve de positivité de ses coefficients (selon la seconde définition) associe une fonction n-aire sur $\mathbb{Z}^+$. C'est cette fonction qui est utilisée pour construire une interprétation à partir d'une interprétation polynomiale. Les résultats relatifs à la positivité des coefficients d'une composée de polynômes permettent, quant à eux, d'évaluer les polynômes associés aux termes lors de la vérification de la compatibilité des règles de réécriture avec une interprétation polynomiale.

4.3.4 Polynômes monotones

Il nous reste à voir comment les théorèmes reliant la monotonie des fonctions polynomiales aux propriétés sur les coefficients ont été établis. Rappelons que si un polynôme P est à coefficients positifs, et si, pour toute indéterminée i, le coefficient de M_i dans P est strictement positif, alors la fonction polynomiale associée à P est monotone.

Pour prouver ce théorème, nous avons été amenés à introduire deux définitions de la monotonie des polynômes. La première définition affirme qu'un polynôme P est monotone ssi ses coefficients sont positifs ou nuls, et si pour tout i le coefficient du monôme M_i dans P est strictement positif. Cette définition offre l'avantage d'être facile à vérifier sur un polynôme donné. En revanche, elle est difficile à exploiter dans les preuves, dans la mesure où elle n'apporte aucun renseignement sur la structure du polynôme, puisque le monôme M_i peut figurer plusieurs fois, avec des coefficients différents dans la liste représentant le polynôme. Nous avons donc introduit une seconde définition de la monotonie des polynômes, qui apporte une information plus facile à exploiter quant à la structure du polynôme. Nous disons qu'un polynôme P est monotone ssi ses coefficients sont positifs ou nuls, et si pour tout i il existe deux polynômes P_1 et P_2 et un coefficient $c > 0$ tels que $P = P_1 + cM_i + P_2$.

La démonstration du théorème principal se fait alors en trois étapes :

- on commence par montrer que si un polynôme est monotone d'après la définition 1, alors il l'est aussi d'après la définition 2 ;

– on montre que si un polynôme est monotone au sens de la définition 2, alors la fonction polynomiale associée est monotone pour la relation $<$. La démonstration de ce résultat utilise le fait que si un polynôme est monotone pour la définition 2, alors la fonction polynomiale qui lui est associée est monotone pour la relation $\leq$;
– on conclut, d'après les deux théorèmes précédents, que si un polynôme est monotone au sens de la définition 1, alors la fonction polynomiale qui lui est associée est monotone pour la relation bien fondée $<$ sur les entiers positifs.

4.4 Vérification de la compatibilité des règles de réécriture

Dans les sections précédentes, nous avons montré comment nous représentons les divers objets utilisés pour construire les interprétations polynomiales et prouver les résultats liés aux systèmes de réécriture en général. Nous allons maintenant montrer comment la vérification de la compatibilité d'une règle avec une interprétation polynomiale a été ramenée à la vérification de la positivité des coefficients d'un polynôme.

Plus formellement, nous allons donner quelques éléments d'information sur la preuve en `Coq` du lemme suivant : soit $l \to r$ une règle de réécriture et IP une interprétation polynomiale.
Si $P_l - P_r - 1$ est à coefficients positifs, alors $l \to r$ est compatible avec l'ordre engendré par IP.

Le premier problème posé par la démonstration de ce lemme est la définition de P_l et P_r, polynômes associés au membre gauche et au membre droit de la règle de réécriture, respectivement. En effet, si l'on appelle I l'interprétation monotone bien fondée engendrée par IP, tout ce que nous pouvons calculer pour l'instant, c'est l'interprétation d'un terme t dans I, qui se présente sous la forme d'une fonction de l'ensemble des valuations sur I vers le domaine de I. En d'autres termes, l'interprétation d'un terme t dans I est une fonction qui, étant donnée une valuation, retourne l'interprétation de t pour cette valuation particulière. Mais rien ne permet de dire que la fonction d'interprétation est en fait la fonction d'évaluation d'un polynôme. Ceci doit donc être démontré, avec, comme préalable, la définition précise de la manière dont on calcule le polynôme associé à un terme. Nous commençons donc par expliquer comment calculer le polynôme associé à un terme, puis nous étudions plus précisément la preuve du lemme énoncé précédemment.

4.4.1 Calcul du polynôme associé à un terme

Comme nous l'avons indiqué précédemment, le type des polynômes est paramétré par le nombre d'indéterminées. Pour pouvoir associer un polynôme à un terme, il faut donc connaître au préalable le nombre d'indéterminées du polynôme que l'on cherche. Ces constatations préliminaires suggèrent un calcul en deux étapes :
– on calcule le nombre n d'indéterminées nécessaires à la construction du polynôme ;
– on construit récursivement un polynôme à n indéterminées associé au terme t.

Le nombre d'indéterminées nécessaire pour les polynômes n'est autre que le nombre de variables apparaissant dans un terme. Cependant, comme nous avons choisi de représenter les variables par des entiers naturels, il nous a paru plus simple, plutôt que de rechercher le nombre de variables apparaissant dans un terme, de rechercher le plus grand entier utilisé pour représenter une variable dans un terme. Ce nombre étant un majorant du nombre de variables, il peut être utilisé à sa place pour déterminer le nombre d'indéterminées à utiliser pour construire un polynôme associé à un terme.

Nous allons maintenant montrer comment on calcule le polynôme associé à un terme t dont la variable maximale est n (autrement dit, toutes les variables apparaissant dans t sont des entiers inférieurs à n). Le polynôme P_t associé à t a $n+1$ indéterminées et se définit alors par induction sur la structure de t :
– si $t = x_i$, alors $P(_t$ est le polynôme M_i ;
– si $t = f(t_1, \ldots, t_n)$, alors $P_t = P_f(P_{t_1}, \ldots, P_{t_n})$, c'est-à-dire que P_t est la composée du polynôme associé au symbole f dans l'interprétation par les polynômes associés aux sous-termes $t_1, \ldots, t_n$.

La fonction Coq qui construit les polynômes associés aux variables a besoin d'un argument lui prouvant que i est bien inférieur à n. Cependant, si nous essayons de calculer directement le polynôme associé à un terme t, nous ne disposerons pas d'une telle preuve, et il sera donc impossible de construire les polynômes associés aux variables. Nous avons donc été amenés à construire une structure de données intermédiaire, calquée sur celle des termes, mais qui contient de plus les preuves dont nous avons besoin pour la construction des polynômes. La définition Coq de cette structure est la suivante :

```
Inductive bterm (Sig : Signature) (n : nat) : Set :=
  | TVar : forall (x : nat), x<=n -> bterm n
  | TFun : forall (f : Sig), vector (bterm n) (arity f) -> bterm n.
```

Ce nouveau type de termes, que nous appellerons par la suite « termes bornés », est donc un type similaire à celui des termes définis à la section 4.2.2, mais paramétré par un entier `n`, et tel que toutes les variables apparaissant dans un terme soient inférieures à `n`.

Remarque 9 *Le problème concernant la définition du principe de récurrence sur les termes, évoqués en section 4.2.2, se retrouve pour la structure de termes bornés. On est, encore une fois, obligé de définir manuellement le principe de récurrence dont on a besoin pour pouvoir raisonner sur les termes bornés.*

Une fois les termes bornés définis, il s'agit d'écrire deux fonctions : l'une prend comme arguments un terme t, un entier n, et une preuve que toutes les variables de t sont majorées par n, et renvoie le terme borné correspondant. La seconde fonction, qui permet de calculer le polynôme associé à un terme borné, est une transcription en Coq de la définition par récurrence donnée précédemment.

Remarquons une propriété importante du développement ainsi conçu : il est possible d'associer à un terme un polynôme à un nombre arbitraire d'indéterminées, pourvu que le nombre d'indéterminées soit supérieur ou égal au plus grand nombre naturel représentant une variable apparaissant dans le terme. Cette propriété s'avère indispensable pour comparer des polynômes associés aux termes de règles dans lesquels il n'y a pas le même nombre de variables à gauche qu'à droite.

Soit par exemple la règle $x + (-x) \to 0$. Le terme de gauche comporte une variable, et donc on peut lui associer un polynôme à une indéterminée. Le terme de droite, quant à lui, ne contient aucune variable, donc on pourrait lui associer un polynôme à 0 indéterminée, c'est-à-dire un polynôme constant. Néanmoins, si nous procédions ainsi, nous ne pourrions calculer $P_l - P_r$, car les polynômes P_l et P_r seraient de types différents, et vouloir calculer $P_l - P_r$ n'aurait aucun sens.

Néanmoins, grâce à la remarque précédente, nous pouvons considérer le polynôme associé au terme 0 comme un polynôme à une indéterminée, ce qui permet le calcul de $P_l - P_r$.

Pour conclure cette partie sur le calcul des polynômes associés aux termes, signalons que, comme le laisse entrevoir l'exemple précédent, le nombre d'indéterminées qu'il convient d'utiliser pour calculer les polynômes associés aux termes d'une règle n'est pas l'indice de la plus grande variable apparaissant dans chacun des termes, mais bien l'indice de la plus grande variable apparaissant dans le membre gauche la règle. Pour pouvoir associer un polynôme de même type au membre droit de la règle, il faut alors s'assurer qu'il comptera bien moins de variables que le membre gauche. Ceci est évidemment vrai par définition des règles de réécriture, et c'est ce point particulier qui nous a conduit, comme nous l'avons signalé plus tôt, à affiner notre définition d'une règle de réécriture, en exigeant que l'ensemble de variables apparaissant dans le membre droit d'une règle soit inclus dans l'ensemble de variables apparaissant dans le membre gauche.

4.4.2 Preuve du lemme

Le lemme que nous souhaitons prouver a pour conclusion qu'une règle $l \to r$ est compatible avec l'ordre engendré par une interprétation polynomiale. Plus précisément, il s'agit de montrer que, pour toute valuation ξ, $[\![l]\!]_\xi > [\![r]\!]_\xi$. Or l'hypothèse du lemme porte sur les polynômes associés à l et r. Il est donc nécessaire de pouvoir passer des interprétations telles qu'elles figurent dans la conclusion à des expressions où figurent des polynômes.

Intuitivement, il s'agit d'exprimer l'idée que l'interprétation d'un terme t sous la valuation ξ n'est rien d'autre que l'évaluation du polynôme associé à t en un point particulier. Cette formulation laisse entrevoir la nouvelle difficulté qui se présente. Il s'agit de définir en quel point il faut évaluer le polynôme associé à t pour obtenir le même résultat (élément de $\mathbb{Z}^+$) que lorsqu'on interprète t sous la valuation ξ. Le point en question (qui est en fait un élément de $((\mathbb{Z}^+)^n)$, où n est le nombre d'indéterminées du polynôme associé à t), dépend de la valuation ξ. Plus précisément, il s'agit du point défini par un vecteur de taille n dont chaque emplacement i contient la valeur de ξ en i.

Nous avons donc eu besoin d'introduire une fonction qui, étant donnés une valuation ξ et un entier n, calcule le vecteur contenant les valeurs de cette valuation sur les entiers inférieurs à n.

Une fois cette fonction définie, nous avons pu démontrer l'égalité dont nous avions besoin entre interprétation d'un terme et évaluation du polynôme associé. Sans entrer dans les détails, signalons simplement que cette preuve est complexe et nécessite encore la preuve de plusieurs résultats intermédiaires.

Reste finalement à prouver le lemme de compatibilité. Une fois que l'on dispose des théorèmes précédents, la preuve de ce lemme s'écrit assez facilement en remplaçant les interprétation des termes par les polynômes associés. On doit ensuite montrer que $P_l(v) > P_r(v)$, ce qui se ramène à prouver que $P_l(v) - P_r(v) - 1 \geq 0$. Or une condition suffisante pour que cette inégalité soit vraie pour tout v est que le polynôme $P_l - P_r - 1$ soit à coefficients positifs, ce qui termine la démonstration du lemme de compatibilité, puisque cette propriété constitue l'hypothèse du lemme.

4.5 Automatisation

À ce stade, chaque étape de la preuve de terminaison d'un système de réécriture par interprétation polynomiale (construction de l'interprétation puis vérification de la correction et de la compatibilité des règles) a été ramenée à des vérifications faciles sur des polynômes et des listes. Pour ce qui est de la correction des règles, il faut vérifier que le membre gauche n'est pas une variable, et que la liste des variables du membre droit est incluse dans la liste des variables du membre gauche. Concernant les polynômes, on veut vérifier qu'ils sont à coefficients positifs, et qu'ils contiennent bien certains monômes, lorsqu'il s'agit d'en prouver la monotonie.

Pour automatiser toutes ces vérifications, des tactiques `Coq` ont été écrites en utilisant le langage `Ltac` présenté à la section 3.4. Les tactiques qui ont été écrites sont de deux types :
- des tactiques de bas niveau, chargées de construire les preuves d'inclusion de listes et de positivité de coefficients ;
- des tactiques de plus haut niveau, qui ne font qu'appeler les tactiques précédentes ou des tactiques `Coq` prédéfinies, et qui ont surtout pour vocation de rendre l'écriture des preuves plus rapide.

Nous allons maintenant présenter brièvement les tactiques que nous avons été amenés à écrire, avant de nous intéresser plus particulièrement à l'une d'entre elles, qui illustre bien les possibilités offertes par le langage `Ltac`. Les principales tactiques mises au point sont donc les suivantes :
- `incltac` : permet de prouver l'inclusion d'une liste `l1` dans une liste `l2`. Cette tactique procède par induction sur `l1` en appelant une tactique `intac` chargée quant à elle de montrer l'appartenance d'un élément à une liste, et qui fonctionne par induction sur la liste ;
- `ruletac` : permet de vérifier la correction d'une règle. Rappelons que la proposition de correction d'une règle se présente sous la forme d'une conjonction de deux propositions : l'une affirmant que le membre gauche n'est pas une variable, l'autre que les variables du membre droit figurent toutes dans le membre gauche. `ruletac` dissocie la conjonction en ces deux propositions, prouve la première grâce aux tactiques prédéfinies de `Coq`, et appelle `incltac` sur la seconde ;
- `buildtrs` : construit un système de réécriture à partir d'une liste de couples de termes, chaque couple de terme représentant une règle. Pour chaque couple, une règle est construite

en utilisant `ruletac` ;
- `postac` : prouve la positivité d'un nombre. Bien que `omega`, la tactique résolvant les inégalités linéaires de l'arithmétique, aurait pu être utilisée ici, nous avons préféré nous en passer, car, compte tenu de la forme très simple des buts à résoudre, il ne paraissait pas très efficace d'appeler une tactique aussi lourde. Nous procédons donc en appliquant directement les lemmes de la bibliothèque standard de `Coq` dont nous avons besoin ;
- `montac` : la tactique permettant de prouver la monotonie d'un polynôme, dont le fonctionnement est expliqué plus en détail ci-dessous ;
- `buildpolint` : permet de construire une interprétation polynomiale. Cette tactique prend en argument une fonction qui à chaque symbole associe un polynôme l'interprétant, et se charge d'appeler les tactiques nécessaires pour s'assurer que les polynômes vérifient les hypothèses de positivité de coefficients et de monotonie ;
- `provetermination` : prouve la terminaison d'un système de réécriture en utilisant l'interprétation polynomiale qui lui est donnée en argument. Cette tactique procède en appliquant le théorème de terminaison et en appelant les tactiques chargées de vérifier automatiquement les hypothèses de ce théorème.

Comme nous l'avions annoncé précédemment, nous allons maintenant présenter plus en détails le fonctionnement de la tactique `montac`, qui permet d'automatiser la preuve de monotonie des polynômes, nécessaire à la construction des interprétations polynomiales. Plus précisément, si l'on souhaite prouver la monotonie d'un polynôme P à n indéterminées, il s'agit d'automatiser la preuve de la proposition suivante : pour tout $i < n$, le coefficient de M_i dans P est strictement positif. Pour un $i < n$ donné, la preuve est facile. Il suffit de simplifier les expressions et de prouver une inégalité triviale entre deux nombres entiers. La difficulté à laquelle nous avons été confrontés a plutôt consisté à générer les bons sous-buts, et à éliminer ceux pour lesquels $i \geq n$. C'est cette tâche que le langage de tactiques de `Coq` a permis d'automatiser. Voici le code de la tactique `montac` :

```
Ltac montacrec :=
  match goal with
    | H:lt ?i O |- _ => elimtype False; apply (lt_n_O _ H)
    | H:lt ?i (S ?k) |- _ => destruct i;
      [(simpl; omega) | (generalize (lt_S_n _ _ H); intro; montacrec)]
    |  _ => idtac
  end.

Ltac montac := intros i H; simpl in H; montacrec.
```

L'essentiel de la construction des preuves est fait par la tactique `montacrec`, qui est basée sur du filtrage sur le but et un raisonnement par cas sur `i`. Comme le montre la définition de `montacrec`, trois cas peuvent se présenter :

- soit le contexte contient une hypothèse affirmant qu'un certain entier est inférieur à 0, et ce but doit être éliminé comme absurde. Ceci se fait en utilisant un lemme de la bibliothèque standard de `Coq` qui affirme qu'aucun entier n'est inférieur à 0 ;
- si le contexte contient une hypothèse de la forme $i < k+1$, alors on raisonne par cas sur `i` : soit $i = 0$ et alors on peut utiliser des tactiques `Coq` prédéfinies pour vérifier que le coefficient du monôme correspondant dans le polynôme est strictement positif, soit i est le successeur d'un autre nombre, et alors nous relançons récursivement `montacrec` ;
- enfin, si aucune hypothèse ne convient, on ne fait rien.

La tactique `montac` se contente de mettre le contexte sous une forme que la tactique `montac` sait traiter, avant d'appeler cette dernière tactique.

4.6 Exemple : preuve de terminaison d'un système de réécriture

Nous allons maintenant montrer comment les théorèmes et tactiques définis aux sections précédentes permettent d'automatiser la construction d'une preuve de terminaison pour un système de réécriture donné. Soit $\Sigma = \{0, +, -\}$ la signature utilisée pour manipuler des groupes. 0 est une constante, c'est-à-dire d'arité 0, + est d'arité 2, et − est d'arité 1. Notre objectif est de prouver dans Coq la terminaison du système de réécriture suivant : $\{x + 0 \to x, x + (-x) \to 0\}$.

Pour y parvenir, nous commençons par définir la signature de travail :

```
Inductive Gsymbol : Set :=
  | Gzero : Gsymbol
  | Ginv : Gsymbol
  | Gplus : Gsymbol.

Definition Garity (s : Gsymbol) :=
  match s with
    | Gzero => 0
    | Ginv => 1
    | Gplus => 2
  end.

Definition GroupSig := mkSignature Gsymbol Garity.
```

Nous définissons ensuite le système de réécriture dont nous souhaitons prouver la terminaison. Pour rendre cette définition plus lisible, nous construisons progressivement les termes qui interviennent dans les règles de réécriture, et leur attribuons un nom.

```
Definition ZERO := (@Fun GroupSig Gzero Vnil).
Definition x := (@Var GroupSig O).
Definition T1 := (@Fun GroupSig Gplus (Vcons x (Vcons ZERO Vnil))).
Definition Ix := (@Fun GroupSig Ginv (Vcons (@Var GroupSig O) Vnil)).
Definition T2 := (@Fun GroupSig Gplus (Vcons x (Vcons Ix Vnil))).
```

Le symbole `@` est utilisé pour indiquer à Coq que l'on souhaite expliciter tous les arguments d'une application, même si certains ont été définis comme étant implicites. Une fois définis les termes dont nous avons besoin, la construction du système de réécriture se fait via la tactique `buildtrs` présentée en section 4.5. Notons que cette construction est complètement automatique, et ne requière donc aucune intervention de l'utilisateur.

```
Definition Gtrs : trs GroupSig.
buildtrs ((T1, x) :: (T2, ZERO) :: nil).
Defined.
```

Il s'agit maintenant de construire l'interprétation polynomiale que nous allons utiliser pour prouver la terminaison du système de réécriture. L'interprétation que nous allons utiliser est la suivante : $P_0 = 0$, $P_- = X$ et $P_+ = X + Y + 1$. Comme pour les systèmes de réécriture, nous commençons par définir quelques polynômes en leur associant un nom, pour rendre les définitions plus lisibles.

```
Definition PZero := (P0 O).
Definition Pinv := (1, Vcons (S O) Vnil) :: nil.
Definition M1 := Vcons (S O) (Vcons O Vnil).
Definition M2 := Vcons O (Vcons (S O) Vnil).
Definition Pplus := (1, M1) :: (1, M2) :: (1, (Vcons O (Vcons O Vnil))) :: nil.
```

Nous pouvons maintenant définir la fonction qui à chaque symbole associe le polynôme qui l'interprète :

```
Definition Gpolynomials (s : symbol GroupSig) :=
  match s as x return polynomial (@arity GroupSig x) with
    | Gzero => PZero
    | Ginv => Pinv
    | Gplus => Pplus
  end.
```

La construction de l'interprétation polynomiale proprement dite se fait alors de la manière suivante :

```
Definition GPI : polynomialInterpretation GroupSig.
buildpolint Gpolynomials.
Defined.
```

Ici encore, la construction de l'interprétation polynomiale est complètement automatisée, et aucune intervention de l'utilisateur n'est nécessaire.

Nous pouvons enfin prouver la terminaison du système de réécriture considéré en utilisant l'interprétation polynomiale que nous venons de définir :

```
Theorem Gtrs_terminates : trsTerminates Gtrs.
provetermination GPI.
Qed.
```

Ici également, la construction de la preuve est totalement automatique.

Nous sommes donc maintenant en mesure de construire une preuve complète de terminaison d'un système de réécriture, et cela, sans que l'utilisateur ait à intervenir pour prouver des résultats intermédiaires.

Chapitre 5

Recherche automatique d'une interprétation polynomiale

En section 4.6, nous avons vu comment on pouvait construire dans `Coq` la preuve de terminaison d'un système de réécriture du premier ordre, en utilisant le critère des interprétations polynomiales, précédemment démontré. Cet exemple a montré que, si les tactiques qui ont été définies rendent les preuves extrêmement courtes (une ligne par preuve), la construction d'objets tels que les signatures et les termes demeure fastidieuse. En outre, à ce stade, l'utilisateur qui désire prouver la terminaison d'un système de réécriture en utilisant une interprétation polynomiale est toujours obligé de la fournir à Coq. Cette tâche qui, nous l'avons vu, est complexe, semble d'autant plus ingrate qu'il existe des outils capables de construire automatiquement une interprétation polynomiale à partir d'un système de réécriture dont on souhaite prouver la terminaison.

Dans ce chapitre, nous allons montrer comment une solution permettant d'une part de simplifier la définition des systèmes de réécriture, et, d'autre part, d'éviter à l'utilisateur d'avoir à rechercher et à définir une interprétation polynomiale a été mise en place.

La simplification de la définition de systèmes de réécriture passe par l'ajout à `Coq` de commandes permettant de spécifier des symboles définis par des règles de réécriture, et des règles définissant ces symboles. Ce travail d'enrichissement de la syntaxe de `Coq`, qui avait été mené à bien avant le début de ce stage, ne sera pas présenté ici. Nous nous contenterons de monter, au fur et à mesure des besoins, quelles sont les commandes utilisées pour définir des symboles et des règles.

Nous portons donc notre attention sur l'étude de la solution mise en œuvre pour éviter à l'utilisateur d'avoir à chercher et à définir une interprétation polynomiale permettant de prouver la terminaison d'un système de réécriture. Cette solution passe par la mise au point d'un mécanisme permettant à `Coq` de communiquer avec `CiME` [9], un outil permettant notamment la recherche d'interprétations polynomiales.

Dans une première section, nous nous attacherons à décrire avec précision le schéma de fonctionnement de l'interface entre `Coq` et `CiME` que nous cherchons à mettre au point. Nous nous intéresserons ensuite aux différentes possibilités permettant de faire communiquer `Coq` et `CiME`, et expliquerons comment les impératifs de portabilité et de simplicité nous ont amenés à en choisir une en particulier. Enfin, nous donnerons quelques éléments d'information concernant la mise en œuvre de l'interface entre `Coq` et `CiME`, et conclurons ce chapitre en revenant à l'exemple présenté en section 4.6, et nous verrons comment la construction d'une preuve de terminaison peut être simplifiée, grâce aux facilités syntaxiques permettant la définition de systèmes de réécriture, à la puissance de `CiME`, et à l'utilisation d'une interface construisant seule les objets et théorèmes intervenant dans la preuve de terminaison.

5.1 Schéma de fonctionnement

Comme nous l'avons indiqué au chapitre 1, l'ajout de réécriture à `Coq` doit permettre de définir des fonctions à l'aide de règles, la définition d'une fonction étant acceptée par `Coq` si l'ensemble des règles termine.

Ceci motive l'idée d'ajouter à `Coq` une commande `Termin` prenant un symbole en argument et permettant de prouver la terminaison de l'ensemble des règles définissant ce symbole. La commande `Termin` devra donc fonctionner de la manière suivante :

- rechercher l'ensemble des règles où figure le symbole passé en paramètre ;
- vérifier que l'ensemble de règles ainsi construit ne fait intervenir que des termes algébriques (application de fonctions ou de constantes à des termes algébriques ou des variables), car ces termes sont les seuls acceptés par `CiME` ;
- calculer la signature et l'ensemble de variables associés au système de réécriture défini précédemment ;
- envoyer à `CiME` la signature, l'ensemble de variables et le système de réécriture à étudier ;
- recevoir l'interprétation polynomiale renvoyée par `CiME` ;
- utiliser cette interprétation pour construire la preuve de terminaison du système de réécriture initial en déclarant tous les objets nécessaires (signature, système de réécriture, interprétation polynomiale), et en appelant les tactiques adéquates.

5.2 Communication entre `Coq` et `CiME`

Le schéma de fonctionnement présenté précédemment met en évidence la nécessité de disposer d'un mécanisme de communication bidirectionnelle entre `Coq` et `CiME`. Il faut en effet pouvoir envoyer de l'information à `CiME`, et recevoir les résultats qu'il fournit.

Dans la mesure où `CiME` et `Coq` sont tous deux écrits en `OCaml`, plusieurs solutions peuvent être envisagées pour permettre à ces deux programmes de communiquer. Une première solution consiste à intégrer le code de `CiME` au programme `Coq` et à utiliser `CiME` comme une sorte de bibliothèque. L'envoi d'informations à `CiME` consisterait alors simplement à appeler les fonctions déclarant les objets que l'on souhaite envoyer à `CiME`. De même, la recherche d'une interprétation polynomiale se résumerait à appeler directement les fonctions de `CiME` chargées de la réalisation de cette tâche. Cette solution a donc l'avantage non négligeable d'être relativement simple à mettre en œuvre.

En revanche, une telle approche présente l'inconvénient important d'être extrêmement sensible tant aux évolutions de `Coq` qu'à celles de `CiME`. Que l'on souhaite bénéficier d'améliorations disponibles dans une version plus récente de `CiME`, et tout le travail d'intégration de `CiME` à `Coq` sera à recommencer. De la même façon, il faudra réintégrer `CiME` à chaque nouvelle version de `Coq`.

Une seconde approche consiste à ne pas tenir compte du fait que `Coq` et `CiME` sont écrits dans le même langage, et à faire communiquer un processus `Coq` et un processus `CiME` indépendants. Cette approche peut être mise en œuvre sans que l'on ait à modifier `CiME`, car ce programme fonctionne de la même manière qu'un interpréteur de commandes, c'est-à-dire qu'on peut lui fournir des commandes sur son entrée standard, et qu'il affiche ses résultats sur sa sortie standard.

Cette approche a l'inconvénient d'être plus complexe à mettre en øuvre que la première, dans la mesure où elle nécessite l'écriture d'un analyseur syntaxique capable d'extraire l'information dont nous avons besoin des messages affichés par `CiME`. Malgré ce surcroît de complexité, cette solution nous paraissant plus élégante et plus pérenne que la précédente, c'est celle que nous avons choisi de mettre en œuvre.

Voici par exemple une cession `CiME` qui permet d'avoir un aperçu du travail nécessaire, tant pour préparer les messages à envoyer à `CiME`, que pour réaliser l'analyse syntaxique de ceux qu'il produit, afin de déterminer quelle est l'interprétation polynomiale à utiliser dans la preuve de terminaison.

```
$ cime
Welcome to CiME version 2.02 - Built on 22/12/2003 16:03:50

CiME> #quiet;
Verbose level is now 0

CiME> let S = signature "0 : constant; + : infix binary; - : prefix unary; ";
S : signature = <signature>
CiME> let X = vars "x";
X : variable_set = <variable set>

CiME> let R = TRS S X "x+0->x; x+(-x)->0; ";
R : (S,X) TRS = { x + 0 -> x,
                  x + -(x) -> 0 } (2 equation(s))

CiME> termination R;
Entering the termination expert. Verbose level = 0
Trying to solve the following constraints:
{ V_0 + 0 > V_0 ;
V_0 + -(V_0) > 0 ;
}
(2 termination constraints)
Search parameters: linear polynomials, coefficient bound is 1.
Solution found for these constraints:
[0] = 0;  [+](X0,X1) = X1 + X0 + 1;  [-](X0) = X0;
Termination proof found.
- : unit = ()

CiME> #quit;
Quitting.
$
```

5.3 Mise en œuvre

Le code mettant en œuvre le schéma de fonctionnement décrit en section 5.1 se présente sous la forme de 5 modules qui ont été ajoutés au code source de `Coq`.

Les deux premiers modules sont un analyseur lexical et un analyseur syntaxique, écrits respectivement en `ocamllex` et `ocamlyacc`, équivalents pour `OCaml`des classiques `lex`et `yacc`. Ces deux modules permettent le traitement des messages renvoyés par `CiME`et l'extraction d'une interprétation polynomiale.

Un troisième module fournit une interface de plus haut niveau avec `CiME`. Les fonctions qu'il exporte sont de plusieurs types :

- des fonctions permettant de contrôler le processus `CiME`, c'est-à-dire d'en lancer l'exécution et de l'arrêter ;
- des fonctions fournissant une interface de communication générique, c'est-à-dire permettant d'envoyer une commande à `CiME` et de recevoir sa réponse ;
- des fonctions de plus haut niveau permettant de déclarer des objets tels que les signatures, les ensembles de variables et les systèmes de réécriture, et de rechercher une interprétation polynomiale prouvant la terminaison d'un système de réécriture.

Un quatrième module est chargé de l'appel des tactiques `Coq` nécessaires à la construction des preuves de terminaison. Il utilise des facilités fournies par `camlp4`, un préprocesseur pour `OCaml`, pour construire des objets de type `tactic` (directement applicables à un but), à partir du nom d'une tactique.

Enfin, le module principal est chargé de l'implantation du schéma de fonctionnement de la section 5.1 proprement dit. Il définit sa propre représentation des termes algébriques, vers laquelle les termes du calcul des constructions (type `constr`) sont convertis dès l'appel de la commande `Termin`. Si un terme non algébrique est rencontré, une erreur est générée, et la commande `Termin` échoue. Si tous les termes sont algébriques et si les règles de réécriture sont correctes, on procède à une phase de renommage des variables. Celle-ci est rendue indispensable par le fait que, dans `Coq`, les variables sont représentées par des entiers qui sont en réalité des indices de De Bruijn. Or, aucune garantie n'est fournie quant à la positivité de ces entiers. Et, comme notre formalisation est basée sur une représentation des variables par des entiers naturels, il est nécessaire de les renommer pour s'assurer que toutes les variables seront représentées par des entiers positifs. La signature associée au système de réécriture est ensuite calculée. Une fois toutes ces données disponibles, elles sont envoyées à `CiME`, pour permettre la récupération d'une interprétation polynomiale. Enfin, plusieurs objets sont ajoutés à l'environnement `Coq` courant pour construire la preuve de terminaison :

- un type inductif correspondant à l'ensemble des symboles considéré ;
- une signature indiquant l'arité de chaque symbole ;
- une constante représentant le système de réécriture ;
- une constante correspondant à l'interprétation polynomiale ;
- le théorème de terminaison proprement dit.

5.4 Exemple d'utilisation

Nous revenons maintenant à l'exemple de la section 4.6. Nous allons montrer comment la preuve construite manuellement peut maintenant être obtenue grâce à l'interface développée pendant ce stage. Voici la session `Coq` permettant d'établir cette preuve de terminaison :

```
Coq < Section groups.
Coq < Variable G : Set.
G is assumed
Coq < Symbol Gzero : G.
Gzero is assumed
Coq < Symbol Ginv : G -> G.
Ginv is assumed
Coq < Symbol Gplus : G -> G -> G.
Gplus is assumed
Rules [x : G] {
  (Gplus x Gzero) => x;
  (Gplus x (Ginv x)) => Gzero
}.
Rules accepted.

Coq < Termin Gplus.
SymbSet_1 is defined
SymbSet_1_rect is defined
SymbSet_1_ind is defined
SymbSet_1_rec is defined
Sig_1 is defined
Trs_1 is defined
PI_1 is defined
Trs_1_termination_proof is defined
```

Chapitre 6

Conclusion

L'ajout de réécriture aux assistants de preuve est souhaitable, car il permet de définir plus de fonctions, et rend les preuves d'égalités plus aisées. Il pose néanmoins des problèmes, notamment celui de vérifier que les fonctions définies par règles terminent. Une solution consiste à faire appel à des outils externes de vérification de la terminaison, mais cela suppose, pour conserver au système toute sa fiabilité, d'être en mesure de vérifier les preuves de terminaison qu'ils fournissent au sein même de l'assistant de preuve.

Le travail réalisé pendant ce stage est une première étape en ce sens, puisqu'il a consisté à formaliser un critère de terminaison de systèmes de réécriture : les interprétations polynomiales. En outre, l'interface avec `CiME`qui a été proposée a permi de s'assurer que les théorèmes et tactiques qui ont été mis au point permettent effectivement de vérifier des résultats fournis par un outil externe, et cela sans aide de l'utilisateur.

Ce développement est constitué de 81 définitions de fonctions et types et de 117 théorèmes ou lemmes, répartis dans 20 fichiers, pour un total de 2817 lignes. Il se distingue des travaux antérieurs concernant la formalisation de réécriture dans `Coq` par le fait que, pour la première fois à notre connaissance, les termes du premier ordre ont été formalisés en utilisant des vecteurs, ce qui permet d'en garantir la correction par typage. Une bibliothèque définissant les polynômes à n indéterminées a également été réalisée, et nous semble assez générale pour pouvoir être réutilisée dans d'autres applications.

À l'issu de ce stage, plusieurs pistes restent à explorer. L'une d'entre elles consiste à étendre la bibliothèque dont nous avons commencé le développement, en y intégrant les preuves d'autres critères de terminaison tels que RPO (Recursive Path Ordering) [13], les paires de dépendance décrites dans [1], mais aussi des preuves de critères permettant de vérifier la confluence, une autre propriété importante des systèmes de réécriture. La manière dont la réécriture peut être ajoutée efficacement aux assistants de preuve en général, et à `Coq` en particulier, constitue une autre direction à explorer. Enfin, la mise en place de mécanismes génériques permettant à `Coq` de communiquer avec des outils externes non connus à l'avance faciliterait grandement la coopération entre programmes dans la construction de preuves, autre domaine restant à explorer.

Bibliographie

[1] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236 :133–178, 2000.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1997.

[3] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization in the algebraic-λ-cube. *Journal of Functional Programming*, 7(6) :613–660, 1997.

[4] Y. Bertot and P. Castéran. *Coq'Art : The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer, 2004.

[5] F. Blanqui. Definitions by rewriting in the Calculus of Constructions, 2004. To appear in Mathematical Structures in Computer Science.

[6] F. Blanqui. Extension de coq avec récriture. projet rntl Averroes, lot 5, fourniture 2, 2004. Available on `http://www.loria.fr/~blanqui/`.

[7] F. Blanqui. Prototype d'extension du système Coq. Projet Averroes, lot 5.1, fourniture 3, 2004.

[8] P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. *ELAN User Manual*. INRIA Nancy, France, 2000. `http://www.loria.fr/ELAN/`.

[9] E. Contejean, C. Marché, B. Monate, and X. Urbain. CiME version 2, 2000. `http://cime.lri.fr/`.

[10] E. Contejean, C. Marché, A.-P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. Technical Report 1382, LRI, Orsay, France, 2004.

[11] Coq-Development-Team. *The Coq Proof Assistant Reference Manual – Version 8.0*. INRIA Rocquencourt, France, 2004. `http://coq.inria.fr/`.

[12] N. de Kleijn. Well-foundedness of RPO in Coq. Master's thesis, Free University of Amsterdam, 2003.

[13] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17 :279–301, 1982.

[14] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31 :33–72, 2003.

[15] S. Hinderer. Certification des preuves de terminaison par interprétations polynomiales. Master's thesis, Université Heni Poincaré, Nancy, France, 2004.

[16] J.-P. Jouannaud and M. Okada. Abstract Data Type Systems. *Theoretical Computer Science*, 173(2) :349–391, 1997.

[17] J.-P. Jouannaud and A. Rubio. The Higher-Order Recursive Path Ordering. In *Logic in Computer Science*, 1999.

[18] Q.-H. Nguyen. *Calcul de réécriture et automatisation du raisonnement dans les assistants de preuve*. PhD thesis, Université Henri Poincaré, Nancy, France, 2002.

[19] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993. LIP research report 92-49.

[20] J. Rouyer. Développement de l'algorithme d'unification dans le Calcul des Constructions avec types inductifs. Technical Report 1795, INRIA Lorraine, France, 1992.

[21] J. Rouyer. *Développements d'algorithmes dans le Calcul des Constructions*. PhD thesis, Centre de Recherche en Informatique de Nancy, France, 1994.

[22] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

[23] D. Walukiewicz-Chrząszcz. Termination of rewriting in the Calculus of Constructions. *Journal of Functional Programming*, 13(2) :339–414, 2003.