Changements de représentation des structures de données dans Coq : le cas des entiers naturels

Nicolas Magaud & Yves Bertot

Projet Lemme, INRIA Sophia-Antipolis
2004, route des lucioles
06902 Sophia-Antipolis - France
{Nicolas.Magaud,Yves.Bertot}@sophia.inria.fr

Résumé

Dans le calcul des constructions inductives, des outils de calcul et de preuve sont associés à chaque type de données concret défini inductivement. Par conséquent, le choix d'une structure de données influence fortement le contenu des preuves. Nous proposons dans ce document une méthode pour passer facilement d'une représentation à une autre, en effectuant des traductions partiellement automatisées des preuves. Nous mettons cette méthode en œuvre sur les entiers naturels, pour lesquels il existe une représentation unaire (à la Peano) et une représentation binaire. Un prototype d'outil de traduction est développé dans le système *Coq* et a été expérimenté avec succès sur une dizaine de théorèmes d'arithmétique.

Mots-clé : transformation automatique de preuves, calcul des constructions, *Coq*

1. Introduction

Les exigences de qualité de logiciel pour les systèmes embarqués ont conduit à l'introduction de *méthodes formelles* permettant de spécifier de manière très rigoureuse les propriétés que doivent vérifier ces applications. Ces méthodes ont en outre pour but d'établir si un système est conforme ou non à sa spécification.
L'utilisation de systèmes d'aide à la preuve comme *Coq* [8, 2] entre dans ce cadre. De tels outils rendent possible la construction interactive de preuves formelles de théorèmes. A chaque pas de raisonnement effectué par l'utilisateur, la machine vérifie que celui-ci est valide.
Le développement de ces outils a conduit à la formalisation de nombreuses théories mathématiques parmi lesquelles l'arithmétique de Peano, les réels et des notions d'algèbre.

La modélisation d'un problème dans un système d'aide à la preuve conduit à faire des choix d'implantation pour la représentation effective des entités mathématiques considérées. Ces choix influent, non seulement sur la façon de définir des programmes manipulant ces données, mais aussi sur la manière de construire des preuves de propriétés concernant ces objets et ces opérations. La structure d'une preuve dans ce système est en effet fortement liée à celle des types de données concrets utilisés pour modéliser les objets manipulés. On peut être amené à vouloir changer la structure de données décrivant une entité mathématique; par exemple pour avoir un calcul plus efficace.

Le problème que nous nous posons est d'étudier dans quelle mesure, et à quelles conditions, il est possible de réutiliser les preuves déjà construites avant le changement de représentation pour établir les théorèmes relatifs aux opérations définies sur la nouvelle structure de données concrète.

Représentation des preuves Dans un système logique basé sur une théorie des types [14], l'isomorphisme de Curry-Howard donne une correspondance entre la logique étudiée et le λ-calcul typé sous-jacent. Les types sont les formules logiques et les λ-termes les preuves. Cet isomorphisme permet d'identifier l'interprétation de la flèche logique $\Rightarrow$ avec la flèche des types fonctionnels $\rightarrow$. De plus, la notion de produit dépendant $\forall x : A.(B\ x)$ est une généralisation de la flèche $A \Rightarrow B$. Lorsque le type $(B\ x)$ ne dépend pas de la valeur de x, on retrouve l'implication logique $A \Rightarrow B$.
Nous montrons, à titre d'exemple, comment construire une preuve de la transitivité de l'implication pour les formules propositionnelles P, Q et R :

$$(R \Rightarrow P) \Rightarrow (P \Rightarrow Q) \Rightarrow R \Rightarrow Q \tag{1}$$

Une preuve de cet énoncé peut être décrite par le λ-terme suivant :

$$\lambda th_1 : (R \Rightarrow P); th_2 : (P \Rightarrow Q); th_3 : R.(th_2\ (th_1\ th_3)). \tag{2}$$

La sémantique de Heyting-Kolmogorov donne aussi une interprétation calculatoire des démonstrations. On peut considérer le terme (2) comme une fonction qui, étant donné les preuves des trois propriétés $(P \Rightarrow Q)$, $(R \Rightarrow P)$ et R, construit une preuve de Q. La première étape est de calculer une preuve de P, au moyen des preuves respectives th_1 et th_3 de $(R \Rightarrow P)$ et de R; cette preuve est donnée par l'application $(th_1\ th_3)$. Il ne reste plus qu'à appliquer th_2 à ce terme de preuve pour obtenir la preuve annoncée de Q, à savoir $(th_2\ (th_1\ th_3))$.

Structure des preuves Un changement de représentation concrète d'une collection d'objets mathématiques consiste non seulement en la modification du type de données concret la décrivant, mais aussi en la construction de nouvelles fonctions de calcul sur ces objets.
Si les propriétés P', Q' et R' sont exactement les propriétés P, Q et R adaptées au nouveau type de données, on voit qu'il est possible de réutiliser facilement la preuve (2) pour prouver l'énoncé suivant :

$$(R' \Rightarrow P') \Rightarrow (P' \Rightarrow Q') \Rightarrow R' \Rightarrow Q' \tag{3}$$

Une preuve de cet énoncé est alors :

$$\lambda th'_1 : (R' \Rightarrow P'); th'_2 : (P' \Rightarrow Q'); th'_3 : R'.(th'_2\ (th'_1\ th'_3)). \tag{4}$$

Application aux entiers naturels Tout au long de ce document, nous considérons le cas des entiers naturels et du passage d'une représentation unaire (sous forme des entiers de Peano) :

$$nat := \mathsf{O} \mid \mathsf{S}(nat)$$

à une représentation sous forme d'entiers binaires.

$$bin := \mathsf{zero} \mid positif$$

$$positif := \mathsf{one} \mid 2 * x (x > \mathsf{zero}) \mid 2 * x + 1 (x > \mathsf{zero})$$

Cette transformation conduit à un changement radical de la structure des preuves par induction. En effet, le nombre de constructeurs passe de deux à quatre et il n'y a aucune correspondance entre les pas d'induction dans les deux types de données considérés. On ne peut pas réutiliser facilement la preuve d'un théorème dans une représentation pour établir le théorème équivalent dans l'autre représentation.

Pour être capable de réutiliser les preuves établies dans l'arithmétique de Peano, il faut disposer d'un principe d'induction non structurelle sur le codage binaire des entiers naturels; ce principe doit permettre de conserver la structure du raisonnement de départ.

Problèmes liés à la convertibilité Une autre difficulté apparaît s'il existe une relation d'équivalence sur les termes. Cette équivalence entre deux termes peut disparaître lors de leurs traductions vers une nouvelle représentation. Si dans notre calcul, P_A et P_B sont équivalents, le terme

$$\lambda th_1 : (P_A \Rightarrow Q); th_2 : (R \Rightarrow P_B); th_3 : R.(th_1 \ (th_2 \ th_3)) \tag{5}$$

est une preuve de l'énoncé suivant :

$$(P_A \Rightarrow Q) \Rightarrow (R \Rightarrow P_B) \Rightarrow R \Rightarrow Q \tag{6}$$

Supposons que les termes P'_A et P'_B, traductions de P_A et P_B dans la nouvelle structure de données, ne soient plus équivalents; il est nécessaire de modifier la structure du terme de preuve (5) et de remplacer l'équivalence implicite entre P_A et P_B par l'application d'un théorème $th'_4 : P'_B \Rightarrow P'_A$, qu'il faudra établir séparément. Une preuve de l'énoncé traduit de (6), à savoir

$$(P'_A \Rightarrow Q') \Rightarrow (R' \Rightarrow P'_B) \Rightarrow R' \Rightarrow Q'$$

est donc le terme suivant :

$$\lambda th'_1 : (P'_A \Rightarrow Q'); th'_2 : (R' \Rightarrow P'_B); th'_3 : R'.(th'_1 \ (th'_4 \ (th'_2 \ th'_3)))$$

Nous présentons dans ce document des résultats pratiques visant à faciliter le passage de la représentation unaire (*à la Peano*) des entiers à une représentation binaire dans le développement formel d'une théorie mathématique. Nous commençons par donner un aperçu des caractéristiques du système *Coq* et en particulier de la manipulation des types inductifs (Section 2). Nous montrons ensuite quelles opérations nous devons effectuer pour préparer la traduction automatique de preuves suivant ce changement de structure de données (Section 3). Dans un troisième temps, nous décrivons notre algorithme de traduction de preuves et présentons les résultats obtenus (Section 4). Nous décrivons ensuite quelques travaux reliés à notre étude (Section 5). Nous concluons finalement sur le travail réalisé avant de présenter les perspectives ouvertes par cette expérience.

2. Présentation du système Coq

Nous commençons par décrire rapidement le système d'aide à la preuve *Coq*, puis nous présentons les types inductifs dans ce système. Ceux-ci permettent la définition de types de données concrets et sont un élément central de notre étude.

2.1. Caractéristiques générales

Le système *Coq* est l'implantation d'un formalisme logique d'ordre supérieur : le Calcul des Constructions Inductives [2, Chap. 4]. Il s'agit d'un λ-calcul typé avec types dépendants [14], permettant en outre la définition de constructions inductives [12].
Coq propose un mode interactif de développement de preuve par chaînage arrière. On dispose d'un certain nombre de tactiques, applicables au but courant si certaines préconditions sont vérifiées. Elles permettent d'effectuer les pas élémentaires de preuve.
Dans ce calcul, les preuves sont des termes du λ-calcul comme les autres. L'objet dénotant la preuve d'un théorème est conservé; il est possible de vérifier une preuve par une simple opération de vérification de type.

2.2. Les types inductifs dans Coq

2.2.1. Définitions

L'extension inductive du calcul des constructions et son implantation *Coq* disposent d'une notion de types inductifs primitifs. Pour chaque type inductif, le système propose deux constructions primitives de filtrage (Cases) et de définition par point-fixe (Fixpoint); les schémas de récurrence sont définis à partir de celles-ci.

Ces types inductifs permettent de définir des collections d'objets mathématiques de la même manière que l'on définit des types de données concrets en ML. Par exemple, les entiers naturels *à la Peano* sont définis de la manière suivante dans *Coq* :

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

Cela signifie que l'on définit un objet nat, de sorte Set, dont les constructeurs sont O (zéro) et S (la fonction successeur $x \mapsto x+1$).
La définition d'un type inductif dans *Coq* provoque la construction *automatique* d'un *principe d'induction structurelle* et d'un opérateur de récursion permettant de définir des fonctions récursives. Le principe d'induction sur les entiers de Peano prend la forme suivante :

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

qui correspond exactement au principe de récurrence classique sur les entiers :

$$P\ (0)\ \wedge\ (\forall n \in \mathsf{nat}\ .\ P\ (n) \Rightarrow P\ (n+1))\ \Rightarrow\ \forall n \in \mathsf{nat}\ .\ P\ (n) \tag{7}$$

Non seulement ce principe d'induction est énoncé de manière automatique, mais une preuve de celui-ci est également construite automatiquement.

2.2.2. Convertibilité

Il existe dans le calcul des constructions inductives des règles de convertibilité qui identifient certains termes entre eux. Ces règles transcrivent la notion de $\beta\delta\iota$-équivalence, où la β-réduction correspond à l'application de fonctions, la δ-réduction au remplacement de noms de constantes par leurs définitions effectives. La ι-réduction définit les règles de réduction associées aux opérateurs Cases et Fixpoint.

Considérons par exemple la définition dans *Coq* de la fonction factorielle :

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

Les règles de ι-réduction correspondant à cette définition sont les suivantes :

$$(\mathsf{fact}\ \mathsf{O}) \rightarrow_\iota (\mathsf{S}\ \mathsf{O}) \qquad\qquad (\mathsf{fact}\ (\mathsf{S}\ p)) \rightarrow_\iota (\mathsf{mult}\ (\mathsf{fact}\ p)\ (\mathsf{S}\ p)))$$

Cela signifie que le système ne fait aucune distinction (autre que syntaxique) entre les deux termes (fact O) et (S O) . On dit que ces termes sont intentionnellement égaux, ou convertibles. Ils peuvent être utilisés indifféremment l'un à la place de l'autre.

Conséquences de la convertibilité La preuve de l'égalité (fact (3))=(6) dans un système comme *Coq* est immédiate. Les deux termes considérés (fact (3)) et (6) se réduisent à la même forme normale par les règles de réduction du calcul; ils sont donc intentionnellement égaux. Une telle preuve est dite implicite, puisque le raisonnement est en fait remplacé par du calcul et que le terme de preuve construit exprime simplement que (6)=(6), masquant que le membre gauche de l'équation a fait l'objet d'une réduction. L'application des règles de convertibilité ne construit aucun terme de preuve et se contente de modifier l'aspect syntaxique de l'énoncé à établir.
De la même manière, l'énoncé

$$\forall n \in \mathsf{nat} \quad (\mathsf{fact}\ (\mathsf{S}\ n)) = (\mathsf{mult}\ (\mathsf{fact}\ n)\ (\mathsf{S}\ n))$$

se prouve de façon immédiate par le calcul; en effet la ι-réduction du terme de gauche conduit au terme de droite. Il est important de savoir que cette preuve serait difficile si S n'était pas un constructeur, mais une simple fonction. Pour chaque fonction, les règles de ι-réduction dépendent de la structure du type manipulé.

Une présentation très complète de l'utilisation des types inductifs dans *Coq* est faite dans [7] et les fondements théoriques sont présentés dans [12].

3. Changement de structure de données concrète

Le but de cette partie est de présenter les éléments à mettre en place afin de pouvoir faire une traduction automatique des termes de preuves lors du passage de la représentation unaire à la représentation binaire des entiers. Nous commençons par étudier le passage du principe d'induction structurelle sur les entiers de Peano aux entiers binaires. Une fois ce principe disponible, nous étudions comment remplacer les règles de convertibilité (en particulier la ι-conversion) par des étapes de réécriture utilisant des théorèmes d'égalité.

3.1. Passage du principe d'induction

Nous cherchons à énoncer et démontrer un principe de raisonnement par induction sur les entiers binaires semblable au principe d'induction structurelle des entiers de Peano. Nous reprenons la notion d'entiers binaires utilisée dans l'implantation de la tactique Omega [2, Chap. 16] de *Coq*.

La définition des entiers binaires dans *Coq* est donné par les lignes suivantes :

```
Inductive positif : Set :=
      one: positif             (* 1 *)
    | pI: positif ->positif     (* 2x+1, x>0 *)
    | p0: positif ->positif.    (* 2x, x>0 *)

Inductive bin : Set := zero : bin | pos : positif -> bin.
```

Il est possible, et plus naturel, de définir les entiers binaires comme des listes de booléens. Cette représentation, bien que plus simple que celle adoptée, présente cependant l'inconvénient de ne pas donner une écriture unique de chaque entier naturel. Par exemple, 0100 et 100 sont deux représentations différentes du nombre quatre.

A la suite de ces définitions, les principes d'induction structurelle positif_ind et bin_ind sont automatiquement engendrés. On obtient finalement un principe d'induction sur les entiers binaires très différent du principe d'induction (7) dérivé de la définition des entiers de Peano. Ce nouveau principe d'induction comporte quatre cas mutuellement exclusifs : 0, 1, $2x+1$ $(x>0)$, $2x$ $(x>0)$.

$$P\ (0)\ \wedge\ P\ (1)\ \wedge\ (\forall p \in \mathsf{positif}\ .\ P\ (p) \Rightarrow P\ (2p+1))\ \wedge\ (\forall p \in \mathsf{positif}\ .\ P\ (p) \Rightarrow P\ (2p))$$
$$\Rightarrow\ \forall n \in \mathsf{bin}\ .\ P\ (n)$$

D'un point de vue ensembliste, les types bin et nat sont isomorphes (il est possible de construire une application bijective de l'un dans l'autre) ; mais les principes d'induction et opérateurs de récursion qui leur sont associés les distinguent fortement. La définition de fonctions simulant les constructeurs des entiers de Peano dans l'arithmétique binaire nécessite l'utilisation des opérateurs de filtrage et de définition par point-fixe. La fonction S' qui calcule le successeur d'un entier binaire s'exprime de la façon suivante :

```
Definition S' : bin -> bin :=
   [n:bin] Cases n of zero => (pos one)
            | (pos u) => (pos (aux_S' u))
           end.

Fixpoint aux_S' [n:positif] : positif :=
   Cases n of one (* 1 *)           => (p0 one)        (* 2*1 *)
            | (p0 t) (* 2*t t>0 *)   => (pI t)          (* 2*t+1 *)
            | (pI t) (* 2*t+1 t>0 *) => (p0 (aux_S' t)) (* 2*(t+1) *)
   end.
```

Grâce à l'ordre supérieur, nous allons énoncer et prouver un nouveau principe d'induction sur les entiers binaires. Ce nouveau principe suivra la structure des entiers de Peano et permettra ainsi de conserver la structure des preuves établies par récurrence dans l'arithmétique de Peano.

Procédé de construction

1. Nous commençons par construire deux fonctions

$$\mathsf{Nconvert} : \mathsf{bin} \rightarrow \mathsf{nat} \qquad \mathsf{anti\_Nconvert} : \mathsf{nat} \rightarrow \mathsf{bin}$$

   permettant le passage d'une représentation binaire à une représentation *à la Peano* des entiers naturels et vice-versa.
2. Dans un deuxième temps, nous établissons à l'aide des définitions précédentes et de quelques propriétés élémentaires sur l'addition des entiers binaires l'égalité suivante :

$$\forall a \in \mathsf{bin} \quad \mathsf{anti\_Nconvert}(\mathsf{Nconvert}(a)) = a$$

3. A partir de cette égalité, nous substituons dans la conclusion de l'énoncé du principe d'induction recherché (8) la variable n par $\mathsf{anti\_Nconvert}(\mathsf{Nconvert}(n))$.
4. Il est alors possible d'appliquer le principe d'induction sur les entiers de Peano nat_ind avec comme prédicat $Q = P \circ \mathsf{anti\_Nconvert}$ sur le terme $(\mathsf{Nconvert}(n))$ de type nat. Les buts résultant de cette application sont facilement établis en utilisant les hypothèses du principe d'induction attendu.

Nous disposons maintenant d'un principe d'induction non structurelle sur les entiers binaires. Son énoncé, qui exprime l'induction structurelle sur les entiers de Peano, est le suivant :

$$P\ (\mathsf{zero})\ \wedge\ (\forall n \in \mathsf{bin}\ .\ P\ (n) \Rightarrow P\ (\mathsf{S'}\ n))\ \Rightarrow\ \forall n \in \mathsf{bin}\ .\ P\ (n) \tag{8}$$

Les preuves procédant par induction dans l'arithmétique de Peano peuvent donc être réutilisées pour établir les théorèmes de base de l'arithmétique sur les entiers binaires. Il nous reste tout de même à fournir les éléments permettant de compenser la disparition de la convertibilité entre certains termes lors de ce changement de structures de données.

3.2. Représentation des fonctions

Pour être efficaces, les fonctions sur les entiers binaires doivent être définies de façon structurelle sur leurs arguments. En fait, leur définition ne fait que suivre la technique de calcul manuel sur les entiers en base 2. Dans l'arithmétique de Peano, la définition de l'addition est la suivante :

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

Elle conduit au fait que les termes (plus (S n) m) et (S (plus n m)) sont convertibles, puisque le premier se réduit en le second par simple ι-réduction. Dans la formalisation utilisant les entiers binaires, l'addition Bplus est définie par récursion structurelle sur les entiers binaires (de la même façon que l'addition dans le module `ZArith` de *Coq*) et les termes (Bplus (S' n) m) et (S' (Bplus n m)) ne sont pas convertibles. Malheureusement, les preuves bâties dans l'arithmétique de Peano utilisent de manière implicite ces propriétés de convertibilité. Si l'on veut pouvoir traduire les preuves sur l'arithmétique de Peano vers des preuves sur les entiers binaires, il est indispensable de produire des théorèmes d'égalité que l'on utilisera pour simuler par des réécritures la convertibilité implicitement utilisée dans les preuves originales.

Génération automatique des théorèmes d'égalité Il est possible, à partir de la définition de l'addition sur les entiers de Peano de construire automatiquement les énoncés des théorèmes d'égalité à établir sur l'addition des entiers dans l'arithmétique binaire.

$$\forall m \in \mathsf{bin} \quad (\mathsf{Bplus}\ \mathsf{zero}\ m) = m \tag{9}$$

$$\forall p{,}m \in \mathsf{bin} \quad (\mathsf{Bplus}\ (\mathsf{S'}\ p)\ m) = (\mathsf{S'}\ (\mathsf{Bplus}\ p\ m) \tag{10}$$

Il suffit pour cela d'exprimer pour chaque branche de l'opérateur (Cases) que l'application de la fonction (ici plus) au motif de filtrage est égale au membre droit de cette branche et de traduire l'énoncé ainsi obtenu vers l'opération d'addition sur les entiers binaires. De façon à pouvoir raisonner facilement sur de telles égalités, on prend soin d'appliquer la fonction considérée à tous ses arguments. De cette façon, on aura à établir et à manipuler des égalités sur des types non fonctionnels.

Pour chaque nouvelle opération sur les entiers binaires, on doit établir ces théorèmes d'égalité. Dans l'outil que nous proposons, ceux-ci sont stockés dans une base de données nommée simplification, et seront utilisés pour faire de la réécriture automatique. Ces égalités sont orientées de gauche à droite suivant ainsi le sens de la ι-réduction mise en jeu dans l'arithmétique de Peano.

3.3. Description d'isomorphismes

On relie chaque objet intervenant dans les preuves utilisant les entiers de Peano à un objet jouant le même rôle dans le développement relatif aux entiers binaires. Le tableau ci-dessous présente les composantes de base de cet isomorphisme. On peut remarquer que le constructeur O de l'arithmétique de Peano est traduit en le constructeur zero de l'arithmétique binaire, alors que le constructeur S est traduit en la fonction S' qui n'est plus un constructeur dans l'arithmétique binaire. L'isomorphisme ainsi construit doit être étendu à chaque fois qu'un nouveau théorème T_{nat} est traduit en un théorème T_{bin}.

Implantation Dans l'outil que nous proposons, au niveau du système *Coq*, l'isomorphisme est géré dans une table de hachage. On associe à chaque donnée dans l'arithmétique de Peano son

Type de données	nat	bin
Constructeurs dans l'arithmétique de Peano	O S	zero S'
Opérations de base	plus mult	Bplus Bmult
Principes d'induction	nat_ind	new_bin_ind
Théorèmes	plus_n_O	plus_n_O_bin

TAB. 1 – *Données de base fournies par l'isomorphisme*

équivalent dans l'arithmétique binaire. On dispose de deux commandes pour modifier et observer cet isomorphisme :

- "AddIso t_{nat} t_{bin} ." ajoute une liaison dans la table ;
- "CheckIso t_{nat}." permet de connaître (s'il en existe un) le terme de l'arithmétique binaire associé au terme de l'arithmétique de Peano t_{nat}.

4. Traduction systématique des termes de preuve

Nous disposons maintenant d'un principe d'induction non structurelle sur les entiers binaires qui imite le principe d'induction sur les entiers de Peano. De plus, nous avons montré comment gérer le problème de la convertibilité lors de ce changement de structure de données. Nous pouvons donc passer à la description du procédé de traduction adopté pour obtenir des théorèmes sur les entiers binaires à partir des théorèmes sur les entiers de Peano.

4.1. Exemple de terme de preuve à traduire

Dans la suite de ce document, nous considérons comme exemple de théorème à traduire plus_n_O :

$$\forall n \in \mathsf{nat} \quad n = (\mathsf{plus}\ n\ \mathsf{O}) \tag{11}$$

Le λ-terme suivant est une preuve de l'énoncé précédent :

$$\begin{array}{ll} \lambda n : \mathsf{nat}.(\mathsf{nat\_ind} & \lambda n0 : \mathsf{nat}.n0 = (\mathsf{plus}\ n0\ \mathsf{O}) \\ & (\mathsf{refl\_equal\ nat\ O}) \\ & \lambda n0 : \mathsf{nat}; H : (n0 = (\mathsf{plus}\ n0\ \mathsf{O})). \\ & \quad (\mathsf{f\_equal\ nat\ nat\ S}\ n0\ (\mathsf{plus}\ n0\ \mathsf{O})\ H)\ n) \end{array}$$

Le terme (refl_equal nat O) est une preuve de $\mathsf{O} = \mathsf{O}$. Par convertibilité, c'est aussi une preuve de $\mathsf{O} = (\mathsf{plus\ O\ O})$. De même,

$$\lambda n0 : \mathsf{nat}; H : (n0 = (\mathsf{plus}\ n0\ \mathsf{O})).\ (\mathsf{f\_equal\ nat\ nat\ S}\ n0\ (\mathsf{plus}\ n0\ \mathsf{O})\ H)$$

est une preuve de

$$\forall n0 : \mathsf{nat} \quad n0 = (\mathsf{plus}\ n0\ \mathsf{O}) \Rightarrow (\mathsf{S}\ n0) = (\mathsf{S}\ (\mathsf{plus}\ n0\ \mathsf{O}))$$

C'est aussi une preuve de $\forall n0 : \mathsf{nat} \quad n0 = (\mathsf{plus}\ n0\ \mathsf{O}) \Rightarrow (\mathsf{S}\ n0) = (\mathsf{plus}\ (\mathsf{S}\ n0)\ \mathsf{O})$ qui est le type attendu pour le troisième argument du principe de récurrence nat_ind.

4.2. Terme de preuve traduit structurellement

La première étape de la traduction d'un terme de preuve consiste en la traduction syntaxique du terme en suivant l'isomorphisme présenté dans la section 3.3. On parcourt simplement le terme en substituant les entités ayant une traduction par l'isomorphisme par leur équivalent dans la théorie résultat. A ce stade de la transformation, le terme n'est probablement pas typable, à cause de problèmes de convertibilité.

A l'issue de cette étape, notre exemple devient :

$$\begin{array}{ll} \lambda n : \mathsf{bin}.(\mathsf{new\_bin\_ind} & \lambda n0 : \mathsf{bin}.n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero}) \\ & (\mathsf{refl\_equal\ bin\ zero}) \\ & \lambda n0 : \mathsf{bin}; H : (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero})). \\ & \quad (\mathsf{f\_equal\ bin\ bin\ S'}\ n0\ (\mathsf{Bplus}\ n0\ \mathsf{zero})\ H)\ n) \end{array} \tag{12}$$

new_bin_ind est le principe d'induction (8) sur les entiers binaires suivant la structure des entiers de Peano.
Il faut maintenant transformer le terme ci-dessus (qui n'est pas typable en l'état) en une preuve de la traduction de l'énoncé (11), à savoir :

$$\forall n \in \mathsf{bin} \quad n = (\mathsf{Bplus}\ n\ \mathsf{zero}) \tag{13}$$

4.3. Double calcul de type

Lors de la première phase de traduction, nous avons ignoré tous les problèmes relatifs à la convertibilité. Certaines étapes de preuve étaient effectuées de manière implicite en utilisant les règles de réduction du calcul. Il nous faut maintenant déterminer, dans le terme de preuve, les positions où un problème de typage est apparu lors de la transformation. Nous définissons pour cela un double calcul de type, suivant le même principe que les travaux de Yann Coscoy [3] pour la traduction en langage naturel des preuves dans *Coq*.

4.3.1. Notions de type attendu et de type proposé

L'algorithme de traduction que nous présentons dans la suite utilise les notions de type attendu et de type proposé. Le type attendu T_A détermine l'information imposée par le contexte qui va recevoir un terme t. Le type proposé T_P est le type effectif du terme t synthétisé par le système.
Dans le cas d'une application $(t_1\ t_2)$, telle que le système de type de *Coq* infère les types suivants :

$$t_1 : T \to U \qquad \& \qquad t_2 : T'$$

Le type attendu pour t_2 est le type argument T du type fonctionnel $T \to U$, alors que le type proposé est le type inféré T' par le système pour t_2. Pour que le terme $(t_1\ t_2)$ soit typable, il faut que les types attendu et proposé soient convertibles. Si l'on interprète les types comme des formules logiques et les termes comme des preuves, le type attendu T correspond à une *obligation de preuve*. Il indique l'énoncé dont on doit fournir une preuve. Le type proposé T' est un *constat de preuve*. Il donne l'énoncé effectivement prouvé par le terme considéré (dans ce cas t_2). Pour que le raisonnement soit correct, il faut que l'implication $T' \Rightarrow T$ soit vraie.

4.3.2. Algorithme utilisé

La structure de données manipulée est celle des termes. Les termes du calcul que nous considérons ici sont les abstractions (λ-abstractions et types produits), les applications et les variables. D'autres termes existent (les constantes, l'opérateur de filtrage $\mathsf{Cases}\ldots$); ils ne sont pas présentés car l'algorithme ne fait que les retourner sans aucune modification.

$$\begin{array}{rl}\mathsf{terme} := & \mathsf{abstr}\ (\mathsf{variable}, \mathsf{terme}, \mathsf{terme})\ \mid\ \mathsf{prod}\ (\mathsf{variable}, \mathsf{terme}, \mathsf{terme}) \\ & \mathsf{appl}\ (\mathsf{terme}, \mathsf{liste\_de\_termes})\ \mid\ \mathsf{variables}\end{array}$$

A partir d'un terme t et d'un type T_A (le type attendu pour t), l'algorithme retourne un terme t' de type T_A et une liste d'énoncés de théorèmes à prouver pour que le terme t' soit bien typable et de type T_A. Ces théorèmes sont ceux destinés à remplacer la convertibilité entre les termes par de la réécriture sur les théorèmes d'égalité introduits à la section 3.2. Nous présentons informellement le fonctionnement de cet algorithme.
Initialement, le contexte (composé de liaisons variable $\rightarrow$ type) est vide. L'algorithme travaille par analyse structurelle du terme de preuve t en cours de traduction (par exemple le terme (12)).

Abstraction Dans le cas où t est une abstraction $\lambda x : A.t'$, on réduit le type attendu T_A pour t sous forme d'un type produit $\forall x : A'.(T'\ x)$. On utilise pour cela l'algorithme de mise en forme normale de tête faible. On ajoute dans le contexte le fait que x est de type A, puis on rappelle l'algorithme sur le terme t' avec le type attendu $(T'\ x)$. Les seuls cas traités pour l'instant sont celui où A et A' sont convertibles et celui où x n'apparaît pas dans le corps du produit. Le comportement de l'algorithme dans les autres cas est non spécifié et sera étudié ultérieurement. En supposant que le résultat retourné par cet appel récursif est un terme t'', l'algorithme reconstruit une abstraction $\lambda x : A.t''$ et compare alors le type T_P de ce terme au type attendu $T_A \equiv \forall x : A'.(T'\ x)$. Si ces types sont convertibles, il n'y a plus rien à faire et le résultat $\lambda x : A.t''$ est retourné. Dans le cas contraire, il faut utiliser un théorème établissant l'implication entre le type proposé T_P et le type attendu T_A; le système calcule un nouveau nom pour cette conjecture, par exemple H. Cette propriété devra être établie séparément pour terminer la traduction. Dans ce cas, le terme retourné est $(\mathsf{H}\ (\lambda x : A.t''))$.

Application Dans le cas où t est une application $(\mathsf{hd}\ t_1 \ldots t_n)$, le type T_{hd} de la tête hd de l'application est synthétisé par le système. Il n'y a pas de notion de type attendu pour la tête de l'application; il n'est en effet pas possible de déterminer le type attendu de la tête hd d'une application à partir du type attendu pour l'application elle-même $(\mathsf{hd}\ t_1 \ldots t_n)$. A partir du type T_{hd} synthétisé pour hd, on peut calculer le type attendu T_{A1} pour le premier argument (par la mise en forme normale de tête faible du type T_{hd}). On rappelle l'algorithme sur ce premier argument t_1. Trois résultats sont possibles:

t_1 **n'est pas typable.** Dans ce cas, l'algorithme est rappelé récursivement sur les données t_1 et T_{A1} et retourne un terme t'_1 convenable.

t_1 **est typable, mais de type non convertible avec** T_{A1}**.** Supposons que le type inféré par le système pour t_1 soit T_{P1}, on introduit alors une conjecture $\mathsf{H} : T_{P1} \Rightarrow T_{A1}$, et $(\mathsf{H}\ t_1)$ donne un terme de type T_{A1}.

t_1 **est typable et convertible avec le type attendu** T_{A1}**.**
Il n'y a plus rien à faire; le terme t_1 convient.

Une fois le premier terme de l'application traduit et bien typé (notons ce terme t'_1), on type le terme $(\mathsf{hd}\ t'_1)$ pour obtenir le type attendu T_{A2} pour le deuxième terme t_2. On itère le procédé jusqu'à avoir appliqué tous les termes à la tête de l'application en cours de traduction. Il faut ensuite typer la nouvelle application obtenue $(\mathsf{hd}\ t'_1 \ldots t'_n)$ pour vérifier si le type proposé T_P pour ce terme est compatible avec le type attendu T_A. Si oui, on retourne $(\mathsf{hd}\ t'_1 \ldots t'_n)$; dans le cas contraire, on retourne $(\mathsf{H}\ (\mathsf{hd}\ t'_1 \ldots t'_n))$ où H est l'énoncé $T_P \Rightarrow T_A$ que l'on devra établir séparément.

Variables Les variables ne sont pas transformées. Il est cependant important de savoir que le système *Coq* dispose de deux représentations des variables liées : les variables nommées et les indices de De

Bruijn. Lors de l'introduction d'une variable dans le contexte (au moment de la déstructuration d'une abstraction ou d'un produit), il est nécessaire de nommer la variable et de remplacer les indices de De Bruijn la dénotant dans le terme par son nom.

Les constantes, les types inductifs et leurs constructeurs ont déjà été traduits dans la phase de traduction structurelle du terme de preuve. Ils ne sont donc pas modifiés.

Raisonnement par cas avec l'opérateur Cases Le raisonnement par cas sur une structure peut être utilisé au moyen de la tactique Case. Au niveau du terme de preuve, son utilisation conduit à l'apparition de l'opérateur de filtrage primitif Cases. Lors du passage aux entiers binaires, on ne peut plus utiliser cet opérateur pour faire du raisonnement par cas suivant la structure des entiers de Peano. Cet opérateur ne permet en effet que de manipuler des objets de manière structurelle. On est donc amené à appliquer un théorème particulier pour reproduire ce raisonnement par cas :

$$P(\mathsf{zero}) \Rightarrow (\forall n : \mathsf{bin}\ P(\mathsf{S'}\ n)) \Rightarrow \forall n : \mathsf{bin}\ P(n)$$

Les conjectures utilisées pour faire coïncider type attendu et type proposé pour un terme peuvent contenir des variables libres (présentes dans le contexte de la preuve). Cependant, afin de pouvoir les établir effectivement (voir section 4.4) en utilisant les tactiques du système *Coq*, il est nécessaire de les clôturer par rapport à leurs variables libres.

4.3.3. Comportement de l'algorithme sur l'exemple

Dans notre exemple, seul le terme correspondant à la traduction du pas d'induction (troisième argument de new_bin_ind) nécessite l'utilisation d'une conjecture pour faire coïncider les types proposé et attendu. Le terme traduit structurellement

$$\lambda n0 : \mathsf{bin}; H : (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero}))\ .\ (\mathsf{f\_equal\ bin\ bin\ S'}\ n0\ (\mathsf{Bplus}\ n0\ \mathsf{zero})\ H)\ n)$$

a le type

$$\forall n0 : \mathsf{bin}\ (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero})) \Rightarrow (\mathsf{S'}\ n0) = (\mathsf{S'}\ (\mathsf{Bplus}\ n0\ \mathsf{zero}))$$

qui n'est pas convertible avec le type attendu

$$\forall n0 : \mathsf{bin}\ (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero})) \Rightarrow (\mathsf{S'}\ n0) = (\mathsf{Bplus}\ (\mathsf{S'}\ n0)\ \mathsf{zero})$$

L'algorithme engendre comme obligation de preuve l'énoncé suivant :

$$\begin{gathered}\forall n0 : \mathsf{bin}\ (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero})) \Rightarrow (\mathsf{S'}\ n0) = (\mathsf{S'}\ (\mathsf{Bplus}\ n0\ \mathsf{zero}))\\ \Rightarrow \forall n0 : \mathsf{bin}\ (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero})) \Rightarrow (\mathsf{S'}\ n0) = (\mathsf{Bplus}\ (\mathsf{S'}\ n0)\ \mathsf{zero})\end{gathered}$$

Si H_conv est une preuve de cet énoncé, le terme traduit structurellement précédent devient

$$(\mathsf{H\_conv}\ (\lambda n0 : \mathsf{bin}; H : (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero}))\ .\ (\mathsf{f\_equal\ bin\ bin\ S'}\ n0\ (\mathsf{Bplus}\ n0\ \mathsf{zero})\ H)))$$

Le terme de preuve global

$$\begin{array}{ll}\lambda n : \mathsf{bin}.(\mathsf{new\_bin\_ind} & \lambda n0 : \mathsf{bin}.n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero})\\ & (\mathsf{refl\_equal\ bin\ zero})\\ & (\mathsf{H\_conv}\ (\lambda n0 : \mathsf{bin}; H : (n0 = (\mathsf{Bplus}\ n0\ \mathsf{zero})).\\ & \quad (\mathsf{f\_equal\ bin\ bin\ S'}\ n0\ (\mathsf{Bplus}\ n0\ \mathsf{zero})\ H)))\ n)\end{array} \tag{14}$$

est alors typable et son type est $\forall n : \mathsf{bin}\ n = (Bplus\ n\ \mathsf{zero})$. Il ne reste plus qu'à établir l'obligation de preuve pour que la traduction soit complète.

4.4. Résolution des obligations de preuve engendrées

A chaque fois qu'un problème de convertibilité est détecté lors de la phase précédente, le système engendre un énoncé de théorème lui permettant d'obtenir le bon type. Ces énoncés sont ajoutés artificiellement au contexte pour terminer la traduction du terme de preuve; ils doivent cependant être établis formellement afin que la traduction soit complète. Dans nos expériences, ces conjectures ont toujours pu être établies au moyen de la tactique

Repeat (Intro ; AutoRewrite [simplification]) ; Prolog [] 3.

Cette tactique introduit une à une les hypothèses et tente à chaque fois de réécrire le but en utilisant l'ensemble des théorèmes d'égalité présent dans la base de données simplification construite à la section 3.2. La tactique Prolog permet de prouver automatiquement des théorèmes en utilisant les hypothèses locales et est capable de gérer des variables existentielles; cela permet d'appliquer des hypothèses dont certaines variables apparaissent dans les prémisses, mais pas dans la conclusion (typiquement des hypothèses de transitivité d'une relation binaire).

La tactique de réécriture automatique AutoRewrite dispose comme règles de réécriture des égalités orientées de gauche à droite simulant la convertibilité présente dans la représentation initiale des données. Ces règles simulent la ι-réduction sur les termes traduits. Comme une propriété de normalisation forte du calcul des constructions inductives pour les règles β, δ, ι a été établie [2, Chap. 4], on a la garantie que la réécriture automatique faite par la tactique AutoRewrite terminera.

4.5. Expérimentation et limitations

Nous avons implémenté cet algorithme en ML et intégré ce code au système *Coq*. Les expériences que nous avons menées ont montré que cette commande était tout à fait utilisable en pratique. Certains traitements particuliers ont toutefois été nécessaires afin de pouvoir traduire l'ensemble des théorèmes de la bibliothèque d'arithmétique mettant en jeu l'addition et la multiplication.

Injectivité et Séparabilité des constructeurs Les constructeurs d'un type de données ont des propriétés particulières par rapport aux fonctions habituelles. Ils sont injectifs et libres (c'est-à-dire que deux termes égaux ne peuvent pas être obtenus à l'aide de constructeurs différents). Les preuves de théorèmes dépendants de ces propriétés utilisent l'opérateur de filtrage sur le type de données concret considéré. Cependant celui-ci n'est utilisable que pour des raisonnements suivant la structure du terme. Il est donc nécessaire de recourir à un théorème simulant l'opérateur de filtrage. Cette technique nécessite l'établissement de règles de réduction relatives au principe d'induction new_bin_ind. Il faut être capable d'établir les égalités correspondant aux réductions suivantes :

$$(\mathsf{new\_bin\_ind}\ P\ \mathsf{h\_base}\ \mathsf{h\_rec}\ \mathsf{zero}) \ \rightarrow\ \mathsf{h\_base}$$

$$(\mathsf{new\_bin\_ind}\ P\ \mathsf{h\_base}\ \mathsf{h\_rec}\ (\mathsf{S'}\ u)) \ \rightarrow\ (\mathsf{h\_rec}\ u\ (\mathsf{new\_bin\_ind}\ P\ \mathsf{h\_base}\ \mathsf{h\_rec}\ u))$$

Nous n'avons pas établi ces égalités qui semblent difficiles, étant donnée la méthode utilisée pour prouver le principe d'induction non structurelle new_bin_ind sur les entiers binaires (voir section 3.1). La solution, plus pratique, que nous avons choisi est de construire des théorèmes bin_injection et bin_discriminate dont les énoncés sont les suivants :

$$\forall n,m \in \mathsf{bin}\ (\mathsf{S'}\ n) = (\mathsf{S'}\ m) \Rightarrow n = m \qquad\qquad \forall n \in \mathsf{bin}\ \neg \mathsf{zero} = (\mathsf{S'}\ n)$$

On les utilise à la place de la construction primitive Cases dans les preuves. Pour faire cela, il est nécessaire de savoir reconnaître dans les preuves les fragments utilisant l'injectivité et la non-confusion des constructeurs. Ces fragments de preuve étant construits automatiquement, c'est assez facile. Il suffit de chercher dans les termes de preuve les motifs suivants : (LET ? ? ? (eq_ind ...) ...) pour la discrimination des constructeurs et (LET ? ? ? (f_equal ...) ...) pour l'injectivité.

Cas des prédicats inductifs La définition d'un prédicat inductif représentant une propriété logique conduit automatiquement à la construction d'un principe de raisonnement sur sa structure. Dans le cas de la relation d'ordre $\leq$ sur les entiers binaires, nous définissons un ordre sous une forme facilement calculable en suivant la structure des entiers binaires LE, puis nous reprenons et adaptons la définition de l'ordre le_bin sur les entiers de Peano aux entiers binaires. A partir de ces données, il suffit d'établir deux théorèmes le_bin_vers_LE et LE_vers_le_bin établissant que ces deux relations sont équivalentes pour pouvoir continuer de raisonner sur les entiers binaires en utilisant le principe de raisonnement associé à la définition de l'ordre $\leq$ sur les entiers de Peano. Les preuves mettant en jeu la relation d'ordre $\leq$ sur les entiers de Peano peuvent donc elles aussi être facilement traduites. Cette partie n'a pas été implantée, mais testée à la main sur quelques exemples.

5. Travaux reliés

De nombreux travaux ont été effectués dans le domaine des transformations de preuves et de leurs applications. Madden [9] et Anderson [1] montrent comment utiliser les transformations automatiques de preuves pour optimiser les programmes extraits de ces preuves. Richardson [13] utilise cette technique pour changer les structures de données mises en jeu dans une preuve et le programme qui en est extrait. A partir de la donnée de la définition des entiers de Peano et de l'addition sur ce type de données, ainsi que de la définition des entiers binaires et des fonctions de passage d'une représentation à l'autre, il est possible de synthétiser un programme d'addition raisonnablement efficace sur les entiers binaires.

Un autre sujet de recherche connexe est l'utilisation du raisonnement par analogie dans les assistants de preuve. Il est en effet fréquent que des preuves mathématiques informelles contiennent des formules comme *"On démontrerait P de la même façon que l'on a établi Q"*. Dans ce cas, le mathématicien fournit un modèle de preuve pour le théorème P à établir et laisse le soin au lecteur d'adapter cette preuve pour en faire une preuve de l'énoncé P.
Régis Curien propose dans [4] une méthode pour rechercher dans une base de théorèmes un énoncé dont la preuve pourrait être réutilisée pour établir le nouvel énoncé considéré. Cette recherche se base sur la notion de filtrage d'ordre 2 sur l'énoncé du théorème dont on cherche à établir une preuve.
Erica Melis et Jon Whittle proposent dans [11] une technique pour construire la preuve d'un théorème à partir d'un modèle de preuve. Leur approche se distingue de la nôtre par le fait qu'ils travaillent à un plus haut niveau d'abstraction. Ils utilisent le concept de *proof-plans* (sortes de super-tactiques permettant une large automatisation des preuves) au sein du système CLAM.

Dans le domaine de la coopération entre assistants de preuve, Felty et Howe [6] montrent que l'on peut utiliser conjointement des développements effectués dans différents systèmes. Enfin, Denney [5] propose un mécanisme de traduction permettant de générer des scripts de preuve *Coq* à partir d'un enregistrement du déroulement d'une preuve dans le système HOL.

6. Conclusions et perspectives

6.1. Travail réalisé

Nous avons décrit dans ce document les méthodes à mettre en œuvre pour pouvoir réutiliser plus facilement, lors d'un changement de représentation concrète des entiers naturels, les bibliothèques de théorèmes existantes. Pour cela, nous avons construit un principe de raisonnement par induction non structurelle sur les entiers binaires. Celui-ci suit la structure des entiers de Peano. Disposer d'un tel principe permet la traduction structurelle des preuves. Cependant, la notion de convertibilité entre termes présente dans le système *Coq* complique cette traduction, il est nécessaire d'établir explicitement certaines égalités entre termes dans le développement utilisant les entiers binaires. Ces égalités étaient déjà présentes dans l'ancienne représentation mais sous forme calculatoire; elles étaient implicites au niveau des termes de preuve. Une fois le principe d'induction non structurelle sur les entiers binaires établi, et après avoir proposé une méthode de construction des lemmes d'égalité relatifs à la ι-conversion perdue, nous avons développé un traducteur automatique de preuves de théorèmes sur les entiers naturels. Nous avons traduit toute la bibliothèque de théorèmes sur les entiers de Peano utilisant l'addition et la multiplication vers les théorèmes équivalents dans l'arithmétique binaire. Nous présentons un exemple d'utilisation de notre commande de traduction de preuves ainsi que le code *Coq* des étapes préliminaires dans la version étendue [10] de ce papier.

6.2. Perspectives

L'intérêt d'un tel outil de traduction est de pouvoir facilement réutiliser toutes les preuves faites dans l'arithmétique de Peano pour établir les théorèmes correspondants dans l'arithmétique binaire. Les perspectives de recherche ouvertes par cette étude sont nombreuses. On peut tout d'abord regarder comment augmenter le degré d'automatisation de cette méthode, en particulier au niveau des égalités à établir pour compenser la perte de convertibilité entre certains termes.

L'approche présentée est relativement générale et ne dépend que peu du fait que l'on a travaillé sur les entiers naturels. On peut étudier comment cette approche se généralise à d'autres changements de structures de données. Le codage choisi pour les entiers binaires n'est pas le codage habituel sous forme de liste de booléens. Il serait intéressant de voir comment traduire des théorèmes de l'arithmétique de Peano vers des théorèmes utilisant cette structure de données. On sera alors confronté à un codage non injectif, ce qui posera de nouvelles questions comme le choix de représentants canoniques ou l'utilisation d'ensembles quotients.

Références

[1] Penny Anderson. Representing proof transformations for program optimization. In *Proceedings of the 12th International Conference on Automated Deduction*, volume 814, pages 575–589. LNAI, Springer-Verlag, 1994.

[2] The Coq Team. The Coq Proof Assistant Reference Manual – Version V6.3. Technical report, INRIA, July 1999. Version révisée distribuée avec Coq.

[3] Y. Coscoy. A natural language explanation for formal proofs. In C. Retoré, editor, *Proceedings of Int. Conf. on Logical Aspects of Computational Liguistics (LACL), Nancy*, volume 1328. Springer-Verlag LNCS/LNAI, September 1996.

[4] Régis Curien. *Outils pour la preuve par analogie.* PhD thesis, Université Henri Poincaré Nancy I, INRIA-Lorraine, 1995.

[5] Ewen Denney. A prototype proof translator from Hol to Coq. In *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs'2000*, LNCS. Springer, 2000.

[6] Amy P. Felty and Douglas J. Howe. Hybrid interactive theorem proving using Nuprl and Hol. In *Fourteenth International Conference on Automated Deduction*, volume 1249 of *LNCS*. Springer, 1997.

[7] Eduardo Giménez. A tutorial on recursive types in Coq. Rapport technique 221, INRIA, May 1998. disponible à http://coq.inria.fr/doc-eng.html.

[8] G. Huet, G. Kahn, and Ch. Paulin-Mohring. The Coq proof assistant - a tutorial, version 6.2.4. Rapport technique, INRIA, January 1999. Version révisée distribuée avec Coq.

[9] Peter Madden. The specialization and transformation of constructive existence proofs. In N.S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 131–148. Morgan Kaufmann, 1989. Also available from Edinburgh as DAI Research Paper 416.

[10] Nicolas Magaud et Yves Bertot. Changements de représentation des données dans le calcul des constructions inductives. Rapport de recherche RR-4039, INRIA, Octobre 2000.

[11] E. Melis and J. Whittle. Analogy in inductive theorem proving. In *Journal of Automated Reasoning*, volume 22, pages 117–147, 1999.

[12] 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.

[13] Julian Richardson. Automating Changes of Data Type in Functional Programs. In *Proceedings of KBSE-95, Boston USA, 12-15 November 1995*. IEEE Computer Society, 1995. Extended version available from Edinburgh as DAI Research Paper 767.

[14] Simon Thompson. Type Theory and Functional Programming. ISBN 0-201-41667-0, Addison-Wesley. 1991.