# Une procédure de décision réflexive pour un fragment de l'arithmétique de Presburger

P. Crégut

*France Télécom R&D - DTL/TAL*
*22301 Lannion CEDEX, France*
`pierre.cregut@francetelecom.com`

## 1. Introduction

Les outils de preuve, même lorsqu'ils reposent sur des logiques puissantes, ont besoin de procédures de décision pour décharger les utilisateurs des tâches fastidieuses. Si on souhaite préserver le fait que la confiance dans les preuves ne repose que sur la correction du noyau du prouveur, il faut exécuter les preuves trouvées par la procédure de décision avec ce noyau.

Malheureusement, les preuves automatiques et manuelles ont souvent des structures assez différentes. Les premières reposent le plus souvent sur de longues suites de réécritures pour lesquelles les assistants de preuve comme Coq sont peu efficaces. Aussi les temps d'exécution des preuves construites automatiquement peuvent rapidement devenir prohibitifs même si la procédure de décision elle même est très efficace.

Une solution pour augmenter la vitesse des preuves est d'utiliser la réflexion, c'est à dire interpréter la preuve comme un calcul directement sur une forme syntaxique du but à prouver. Ainsi on échange des réécritures coûteuses du fait du nombre de vérifications de typage qu'elles demandent par des réductions qui sont primitives et effectuées avec toute la vitesse du noyau du démonstrateur. Les vérifications intermédiaires disparaissent : elles ont été prouvées une fois pour toute par les théorèmes de correction de la procédure réflexive.

Cet article présente la traduction d'une procédure de décision pour Coq codée initialement en Objective Caml, le langage d'implantation du prouveur, en une procédure de décision réflexive exprimée dans le calcul des constructions inductives vu comme un langage fonctionnel pur.

Le but est de montrer que le développement d'une procédure réflexive bien qu'encore coûteux à partir d'une version classique est devenu une alternative raisonnable pour améliorer les performances même pour des procédures très complexes et d'identifier les principales difficultés de la traduction.

Nous présentons la bibliothèque utilisée pour construire la procédure de décision réflexive, puis le principe de la procédure de décision et enfin les détails de l'implémentation de la procédure nommé Omega. Nous concluons par quelques résultats expérimentaux.

## 2. Une bibliothèque pour la réflexion

### 2.1. La réflexion

Le calcul des constructions inductives (CIC) est un lambda-calcul typé augmenté avec des types inductifs et code un calcul des prédicats en utilisant le principe de Curry Howard :

- les propositions sont codées comme des types dans une sorte particulière `Prop` ;

– les preuves sont des objets de ces types. Une proposition est vraie si et seulement si son type est habité par au moins une preuve.

La règle du modus ponens sur les propositions correspond à l'application des preuves et le typage garantit la correction des théorèmes construits. La réflexion est un moyen d'échanger des combinaisons complexes d'objets dont la correction repose sur le typage par des calculs (réductions) pour lesquels le noyau de Coq est optimisé. En pratique, la réflexion nécessite :

– un type inductif $S$ (ou un ensemble de types mutuellement inductifs et parmi eux un type distingué) qui code la syntaxe des prédicats. Nous supposerons que $S$ contient une représentation de vrai et faux (**True** et **False**) et des constructeurs logiques de base.
– une *fonction d'interprétation* $[\![-]\!] : S \to \texttt{Prop}$ qui traduit les éléments de $S$ en des propositions dans Prop (il définit la sémantique des termes de $S$) ; on notera $\lceil - \rceil$ la fonction inverse ;
– une *fonction de réflexion* $\mathcal{R} : A \to S$ qui génère de nouveaux termes $S$ interprétables comme des théorèmes valides grâce à un *théorème de correction* qu'il faut prouver :

$$\textbf{Correction :}\ \ \forall a : A.[\![\mathcal{R}(a)]\!]$$

En pratique, $A$ est souvent $S$ et $a$ est directement lié au résultat attendu :

– $\mathcal{R} : a \mapsto t\lceil = \rceil a$ où $t$ peut être interprété comme la forme réduite de $a$,
– $\mathcal{R} : a \mapsto a$ si $a$ peut être prouvé par une méthode de décision ou $a \mapsto$ **True** sinon,
– $\mathcal{R} : a \mapsto t\lceil \to \rceil a$ où $t$ est plus simple à prouver que $a$.

Souvent, seule une partie du but à prouver est réellement utile pour la preuve et donc seule celle-ci doit être réifiée. Mais à l'intérieur même des parties utiles, certains sous-termes ne sont pas nécessaires et pire ne font pas partie du fragment du calcul des constructions qui a été choisi comme réifiable. Ces termes sont abstraits du but et codés comme des variables ($\texttt{Var} : V \to S$)

On introduit une fonction *environnement* $e : (v : V) \to D(v)$ et $[\![t]\!]_e$ l'interprétation de $t$ dans l'environnement $e$ est définie telle que $[\![\texttt{Var}\ \ v]\!]_e = e(v)$. Le théorème de correction est modifié pour prendre $e$ en paramètre : $\forall e.\forall a.[\![\mathcal{R}(a)]\!]_e$ Pour que $e$ soit totale même sur un accès incorrect, il est nécessaire que tous les types des variables utilisées soient habités.

## 2.2. Représentation des buts

Le présupposé est qu'il est plus facile de traduire des procédures de décision existantes si le cadre réflexif est similaire au cadre normal de développement. Nous avons donc reproduit la gestion des buts classiques aux démonstrateurs de type LCF adoptée par Coq.

Une fonction réflexive est l'équivalent d'une tactique. Chaque fonction prend un but en argument et peut en créer plusieurs. Le contexte de preuve est composé d'une liste de buts et est interprété comme leur conjonction.

$[\![(g :: l)]\!]$ = $[\![g]\!]$ ∧ $[\![l]\!]$ $\qquad$ $[\![[\,]]\!]$ = True
Definition tactic : goal ⇒ (list goal)

Chaque but est codé comme une paire faite d'une liste d'hypothèses et d'une conclusion (En pratique, pour Omega la conclusion est toujours **False** et le type utilisé est une liste d'hypothèses).

Definition goal = (list proposition) × proposition
$[\![([h_1;...;h_n],c)]\!]$ = ($\mathtt{h}_1$ ∧ ... ∧ $\mathtt{h}_n$) ⇒ c

Comme les tactiques, les fonctions réflexives peuvent être composées :

$$(t_1\ \ \texttt{then}\ \ t_2)\ \ g = \texttt{flatten}(\texttt{map}\ \ t_2\ \ (t_1\ \ g))$$

La fonction reduce simplifie les contextes de preuve en éliminant les sous buts résolus :

$$\begin{array}{lcl} \texttt{reduce}(g :: l) & = & \textbf{if}\ \ g = \textbf{True}\ \ \textbf{then}\ \ \texttt{reduce}\,l\ \ \textbf{else}\ \ g :: \texttt{reduce}\,l \\ \texttt{reduce}([\,]) & = & [\,] \end{array}$$

## 2.3. Les tactiques valides

Dans cette section, nous supposons :
- que les tactiques ne génèrent qu'un seul but ;
- que les tactiques ne manipulent que les hypothèses et pas la conclusion ;
- que les tactiques prennent un argument complémentaire *trace*.

Nous pouvons raffiner dans ce cas le sens du critère de correction d'une tactique : Soit $R'$ tel que $R((hyps, concl), trace) = (R'(hyps, trace), concl)$, alors le critère peut être réécrit en

$$\text{valid\_hyp} : \forall e, t, h. [\![h]\!]_e \to [\![R'(h,t)]\!]_e$$

Si la tactique ajoute une hypothèse $R' : h \mapsto r(h) :: h$, la correction se réduit à vérifier **valid1 :** $[\![h]\!] \to [\![r(h)]\!]$. Un exemple très important de fonction valide est (`nth_hyp` $i$ $h$) qui rend le $i^{ème}$ élément de la liste $h$ s'il existe et **True** sinon. Finalement une fonction $s$ n-aire est stable si $\forall p_1 \dots p_n . [\![p_1]\!] \wedge \dots \wedge [\![p_n]\!] \Rightarrow [\![s(p_1, \dots, p_n)]\!]$. Ces fonctions sont transformées en tactiques vérifiant **valid1** par le combinateur :

$$\texttt{apply\_n}\ i_1 \dots i_n\ f\ h = (f(\texttt{nth\_hyp}\ i_1\ h) ... (\texttt{nth\_hyp}\ i_n\ h))\ ::\ h$$

**Réécritures** Une classe de fonctions stables $s$ unaires utile est quand $s(p) = p$ au sens de Leibnitz. On appelle une telle fonction une conversion de $p$ et on peut bâtir de nouvelles conversions à partir de conversions élémentaires avec des combinateurs comme en LCF (ou HOL). Des combinateurs utiles sont réécrire la $i^{ème}$ branche d'un terme, appliquer différentes fonctions de réécriture aux différentes branches d'un terme, appliquer séquentiellement des réécritures.

L'intérêt des conversions sur les tactiques de réécriture de Coq est de permettre le partage entre plusieurs conversions du parcours du terme à la recherche des points d'application des modifications.

## 2.4. Utilisation de traces

Sur ce point, l'implantation choisie pour les procédures de décision diffère de l'implantation directe en Coq de la procédure de décision pour des raisons d'efficacité. Une procédure de décision génère normalement directement une suite de tactiques prises parmi des tactiques élémentaires $t_1, \dots t_n$ et reliées par des combinateurs comme `then`.

Nous avons préféré avoir une tactique générique unique paramétrée par une trace qui est un type inductif décrivant les traces élémentaires à appliquer.

```
Inductive trace :=  THEN : trace -> trace -> trace  ... | Tᵢ : Xᵢ -> trace
Fixpoint R [t: trace] := [g: goal] Cases t of
    (THEN t1 t2) ->  (then (R t1) (R t2) g)
    ...
 | (Tᵢ x) -> (Rᵢ x g) end
```

La raison de ce choix est que la combinaison de fonctions construit des termes très gros qui sont longs à typer alors que les traces sont des termes légers dont le typage est rapide. Il y a naturellement un léger surcoût dû à son interprétation, mais ce surcoût est inférieur à celui du typage.

De plus l'expression de la correction du moteur est plus simple. En reprenant le cadre général, on pose $A = S \times \texttt{trace}$. Nous considérons la classe des fonctions réflexives $\mathcal{R} : (g,t) \mapsto (R(g,t) \to g)$. La correction de $\mathcal{R}$ implique :

$$\forall e. \forall (g,t). [\![R(g,t)]\!]_e \Rightarrow [\![g]\!]_e$$

$t$ doit être convenablement choisi pour que $R(g,t)$ soit [ ] si $[\![g]\!]$ est prouvable.

# 3. Architecture globale de la tactique Omega

La procédure de décision traduite est une procédure de décision sur le fragment sans quantificateur de l'arithmétique de Presburger (plus précisément avec des variables universellement quantifiées). Elle a été développée par W. Pugh [Pug92] pour résoudre des systèmes générés par l'optimisation de la compilation de programmes sur des machines vectorielles.

Pour illustrer les différentes étapes, on utilisera le problème suivant comme exemple :

$$\forall x, y, z : Z. y < 0 \vee z = 3 \Rightarrow (x = 2 \wedge z \geq 0) \vee x > 4 \Rightarrow x > 0$$

## 3.1. Procédure de recherche d'une solution

**Simplification des buts** La grammaire décrivant les buts initiaux est :

$$
\begin{array}{l}
\texttt{G := P} \rightarrow \texttt{G | P} \\
\texttt{P := } X \texttt{ | E | P} \vee \texttt{P | P} \wedge \texttt{P | } \neg \texttt{ P | P} \rightarrow \texttt{P | } \textbf{False} \texttt{ | } \textbf{True} \\
\texttt{E := T = T | T} \leq \texttt{T | T < T | T} \neq \texttt{T | T} \geq \texttt{T | T > T} \\
\texttt{T := T + T | T - T | T * T | } n \texttt{ | } v
\end{array}
$$

où $v$ une variable numérique et $X$ une variable propositionnelle et $n$ une constante entière.

La première opération consiste à se ramener à un but où la conclusion est **False**. Si la conclusion initiale $C$ est décidable, il suffit de la remplacer par une hypothèse $\neg C$ :

$$\frac{\neg C \rightarrow H_1 \rightarrow H_2 \rightarrow \ldots \rightarrow \textbf{False}}{H_1 \rightarrow H_2 \rightarrow \ldots \rightarrow C} \quad C \text{ décidable}$$

Ainsi dans l'exemple, le but est transformé en :

$$\neg x > 0 \Rightarrow y < 0 \vee z = 3 \Rightarrow (x = 2 \wedge z \geq 0) \vee x > 4 \Rightarrow False$$

La deuxième opération consiste à extraire les équations des hypothèses (seuls quelques exemples de règles sont indiqués) :

$$\frac{...P_1 \rightarrow ... \rightarrow \textbf{False} \qquad ...P_2 \rightarrow ... \rightarrow \textbf{False}}{....(P_1 \vee P_2) \rightarrow ... \rightarrow \textbf{False}} \qquad \frac{...P_1 \rightarrow P_2 \rightarrow ... \rightarrow \textbf{False}}{....P_1 \wedge P_2 ... \rightarrow \textbf{False}}$$

Les négations sont repoussées vers les propositions élémentaires en utilisant les propriétés de décidabilité :

$$\frac{...\neg P_1 \rightarrow ... \rightarrow \textbf{False} \qquad ...\neg P_2 \rightarrow ... \rightarrow \textbf{False}}{....\neg(P_1 \wedge P_2) \rightarrow ... \rightarrow \textbf{False}} \qquad \frac{...P \rightarrow ... \rightarrow \textbf{False}}{....\neg\neg P ... \rightarrow \textbf{False}}$$

Pour l'exemple, la normalisation de la structure du but donne l'ensemble suivant d'équations :

$$
\begin{array}{lc}
(S1) & y < 0 \Rightarrow x = 2 \Rightarrow z \geq 0 \Rightarrow x \leq 0 \Rightarrow False \\
(S2) & z = 3 \Rightarrow x = 2 \Rightarrow z \geq 0 \Rightarrow x \leq 0 \Rightarrow False \\
(S3) & y < 0 \Rightarrow x > 4 \Rightarrow x \leq 0 \Rightarrow False \\
(S4) & z = 3 \Rightarrow x > 4 \Rightarrow x \leq 0 \Rightarrow False
\end{array}
$$

**Normalisation des équations** Enfin les équations sont normalisées pour que les seuls opérateurs de comparaison soient $\leq$, $=$ ou $\neq$ et que le membre gauche soit nul. Les exemples de règles suivantes

**CONSTANT_NOT_NUL(i)** $E(i) \equiv b = 0$ où $b \neq 0$

**CONSTANT_NEG(i)** $E(i) \equiv b \geq 0$ où $b < 0$

**NOT_EXACT_DIVIDE(i,m)** $E(i) \equiv \Sigma_i a_i x_i + b = 0$ où $\forall i.a_i = m\ a'_i$ et $b = mb' + d$ avec $0 < d < m$

**CONSTANT_NUL(i)** $E(i) \equiv 0 \neq 0$

**NEGATE_CONTRADICT(i,j)** $E(i) \equiv t = 0$, $E(j) \equiv t \neq 0$

**CONTRADICTION(i,j)** $E(i) \equiv \Sigma_l a_l x_l + b_i \geq 0$ , $E(j) \equiv \Sigma_l - a_l x_l + b_j \geq 0$ et $b_i + b_j < 0$

TAB. 1 – Étapes de la trace de $\Omega$ générant une contradiction

**DIV_APPROX(i,m,k)** $E' = E[k \mapsto \Sigma_j a'_j x_j + b' \geq 0]$ où
$E(i) \equiv \Sigma_j a_j x_j + b \geq 0$, $\forall j.a_j = m\ a'_j$ et $b = m\ b' + d$ avec $0 \leq d < m$

**EXACT_DIVIDE(i,m,k)** $E' = E[k \mapsto \Sigma_i a'_i x_i + b' = 0]$ où
$E(i) \equiv \Sigma_j a_j x_j + b = 0$, $\forall j.a_j = m\ a'_j$ et $b = mb'$

**SUM(k,(a,i),(b,j))** $E' = E[k \mapsto e]$ où
si $E(i) \equiv t_i = 0$ et $E(j) \equiv t_j = 0$ alors $e \equiv a\ t_i + b\ t_j = 0$,
sinon si $E(i) \equiv t_i \geq 0$ et $E(j) \equiv t_j = 0$ alors $e \equiv a\ t_i + b\ t_j \geq 0$
sinon si $E(i) \equiv t_i \geq 0$ et $E(j) \equiv t_j \geq 0$ alors $e \equiv a\ t_i + b\ t_j \geq 0$

**MERGE_EQ(i,j,k)** $E' = E[k \mapsto t = 0]$ si $E(i) \equiv t \geq 0$ et $E(j) \equiv -t \geq 0$

**STATE(i,k,m,v,t')** $E' = E[k \mapsto E(i) + m(t' - v)]$ si $v$ nouvelle variable définie par $v = t'$

TAB. 2 – Étapes de la trace de $\Omega$ générant une nouvelle équation

permettent d'abord d'éliminer les négations, puis de restreindre les opérateurs utilisés et enfin d'avoir un membre gauche nul.

$$\frac{...0 \neq T_2 - T_1 ... \to \textbf{False}}{...\neg T_1 = T_2 ... \to \textbf{False}} \qquad \frac{...0 \leq T_1 - T_2 - 1 ... \to \textbf{False}}{...\neg(T_1 \leq T_2) ... \to \textbf{False}}$$

$$\frac{...0 \leq T_2 - T_1 - 1 ... \to \textbf{False}}{...T_1 < T_2 ... \to \textbf{False}} \qquad \frac{...0 \leq T_2 - T_1 ... \to \textbf{False}}{...T_1 \leq T_2 ... \to \textbf{False}}$$

Alors chaque membre droit est normalisé en utilisant les règles de réécriture des anneaux commutatifs et en donnant un poids $w(t)$ à chaque terme $t$ pour ordonner le résultat (la procédure suivie est assez proche de la tactique Ring [Bou97]) . Le résultat est sous la forme $\Sigma_i a_i t_i + b$ où $a_i$ sont des constantes entières et $t_i$ sont des termes produits tels que $w(t_{i+1}) > w(t_i)$. Pour la suite chaque $t_i$ est remplacé par une variable $x_i$ en maximisant le partage entre celles ci. [1]

Pour l'exemple, le système normalisé de (S1) est :

$$(1): \ -y - 1 \geq 0, \qquad (2): \ x - 2 = 0, \qquad (3): \ z \geq 0 \qquad (4): \ -x \geq 0$$

**La résolution des systèmes** Après normalisation des buts, on peut extraire des hypothèses un système d'équation $E$. Le noyau de la tactique (baptisé $\Omega$) résout ces systèmes en construisant de nouvelles équations qui aboutissent à une contradiction évidente. Il s'agit d'une extension de la méthode du simplexe : l'intuition géométrique de la tactique est de prouver que le polyèdre décrit

[1] $t_i$ n'est pas nécessairement une variable au niveau Coq, ni au niveau réifié, il peut s'agir d'un terme complexe.

par les équations est vide. L'extension consiste à vérifier que même s'il n'est pas complètement vide, il ne contient pas de point aux coordonnées entières.

L'étape principale est le 'test de Banerjee' qui permet de générer à partir d'un ensemble d'égalité, un nouvel ensemble d'égalités dont les coefficients constants sont plus petits que dans le système initial. Ce changement de coordonnées introduit une nouvelle variable qui se substitue à une existante. En itérant cette étape, on atteint un point où l'une des équations peut s'écrire $x_i = \Sigma_{j \neq i} a_j x_J + b$. Il est alors possible d'éliminer $x_i$ sans perdre de précision sur la résolution du système.

Pour repérer les équations dans $E$, un indice leur est associé. On note $E(j)$ l'équation d'indice $j$ et $E[j \mapsto e]$, l'ajout de l'équation $e$ avec l'indice (toujours nouveau) $j$. Chaque étape élémentaire type est décrite dans la table 2 qui décrit comment transformer un système $E$ en un système $E'$ par adjonction d'une nouvelle équation et dans la table 1 qui liste les contradictions évidentes.

Une preuve de $E$ est une suite d'opération $[t_0 \dots t_n]$ telle que si $E_0 = E$, $t_i$ est une opération transformant $E_i$ en $E_{i+1}$ et $t_n$ est une contradiction sur $E_n$.

Pour la première équation de l'exemple (simple) présenté ci dessus, la trace est :

$$[\mathbf{SUM}(E5, (1, E4), (1, E2)); \mathbf{CONSTANT\_NEG}(E5)]$$

**Traitement optimisé pour $\neq$** La procédure originale de Pugh décompose $t \neq 0$ en deux équations $t < 0$ et $t > 0$ donnant deux systèmes à résoudre. Nous avons choisi de préserver les inégalités jusqu'à ce qu'il soit impossible de transformer le système sans utiliser l'inégalité (auquel cas le système est éclaté en deux systèmes) ou jusqu'à trouver une autre équation $t = 0$ (étape **NEGATE_CONTRADICT**).

**SPLIT_INEQ(i,j)** $E', E''$, $E' = E[j \mapsto t + 1 \leq 0]$ et $E'' = E[j \mapsto -t + 1 \leq 0]$ si $E(i) \equiv t \neq 0$

Il faut noter que la trace n'est plus une liste mais un arbre puisque **SPLIT_INEQ** génère deux systèmes à résoudre.

## 3.2. Optimisation de la trace

Les cas pratiques d'utilisation de la procédure se distinguent des exemples d'école par la présence d'hypothèses inutiles. S'il est impossible de les ignorer durant la recherche d'une solution, il n'est pas normal que leur présence ralentisse le rejeu d'une solution simple. Le but principal de l'optimisation de la trace est donc d'éviter que l'ajout d'hypothèses inutiles augmente la complexité de la solution. Il faut donc ne faire aucune action sur les hypothèses inutiles, ne pas réifier des termes non utilisés et ne pas mettre dans l'environnement des variables non accédées et ne pas décomposer le but plus que nécessaire et en particulier ne pas éclater des disjonctions quand on dispose de solution permettant de ne pas les faire intervenir dans la solution.

Dans la procédure précédemment décrite, de nombreuses étapes sont inutiles car elles ne contribuent pas à l'établissement des contradictions. Nous utilisons les règles suivantes pour diminuer la taille de la trace et remplir les exigences qui ont été ennoncées :

1. Pour chaque système $E$ on isole les étapes de la trace contribuant directement aux contradictions finales. Toutes les étapes construisant des équations non utilisées sont éliminées. On en déduit les équations du système initial réellement utiles (notées $utile(E)$) ;
2. un ensemble minimal de systèmes permettant de résoudre le but initial est extrait ;
3. seules les parties de la normalisation du but utiles à la génération des équations utilisées dans le système minimal sont effectuées ;
4. seules les équations de l'ensemble minimal de systèmes sont normalisées.

L'étape 2 est la seule réellement complexe. On utilise l'heuristique décrite ci dessous.

**Heuristique de minimalisation du nombre de systèmes à résoudre** Chaque connecteur logique du but initial est étiqueté. On dit qu'un connecteur est branchant si son élimination crée deux buts. On définit la fonction $Br$ qui, pour chaque équation, associe la liste des étiquettes des connecteurs branchant et la branche du connecteur (L ou R) contenant l'équation.

Par exemple, si on a une hypothèse de la forme : $H : e_0 \vee_a ((e_1 \wedge_b e_3) \vee_c e_2)$, la fonction $Br$ est définie pour les équations $e_0$ ... $e_3$ :

$$Br(0) = \{(a, L)\} \quad Br(1) = Br(3) = \{(a, R); (c, L)\} \quad Br(2) = \{(a, R); (c, R)\}$$

$Br$ est généralisé à un système $E$ par $Br(E) = \bigcup_{e \in utile(E)} Br(e)$. On appelle *descripteur de branche* un tel ensemble de paires. Enfin $CoBr(e)$ est l'ensemble des ensembles de paires qui caractérisent les branches créées pour atteindre l'équation $e$

$$CoBr(0) = \{\{(a, R)\}\} \quad CoBr(2) = \{\{(a, L)\}, \{(a, R); (c, L)\}\}$$

$$CoBr(1) = CoBr(3) = \{\{(a, L)\}, \{(a, R); (c, R)\}\}$$

On pose $\overline{L} = R$ et $\overline{R} = L$ et $\overline{X} = \{(x, \overline{D})/(x, D) \in X\}$. $Y$ est compatible avec $X$ si $\overline{Y} \cap X = \emptyset$. Dans la suite, $A$ est l'ensemble des descripteurs de branche pour pour lesquelles il faut chercher une solution. Au départ, on cherche une solution sans éclater la moindre équation donc correspondant à l'élément $\emptyset$. Dans $B$, on conserve la liste des branches déjà traitées et $S$ contient les systèmes qui résolvent les branches décrites dans $B$.

L'heuristique est la suivante :

1. on pose $A := \{\emptyset\}$, $B := \emptyset$, $S := \emptyset$ ;
2. soit $a \in A$, $A := A - \{a\}$ et $B := B \cup \{a\}$ ;
3. soit $E$ tel que $Br(E)$ soit minimal et $Br(E)$ compatible avec $a$, $S := S \cup \{E\}$ ;
4. $A = A \cup \bigcup_{e \in E} CoBr(e) \backslash B$ ;
5. tant que $A \neq \emptyset$, aller à l'étape 2 ;
6. retourner $S$ comme ensemble minimal de système utile.

Notez que $A \cap B = \emptyset$, que $B$ croit à chaque itération et que le nombre de descripteurs de branche est fini. Cette procédure appliquée à l'exemple annoté :

$$\forall x, y, z : Z.y < 0 \vee_{(a)} z = 3 \Rightarrow (x = 2 \wedge z \geq 0) \vee_{(b)} x > 4 \Rightarrow x > 0$$

montre que $(a)$ ne contribue pas à la solution. En effet $Br(S1) = \{(b, L)\}$ donc $CoBr(S1) = \{\{(b, R)\}\}$ et le système S3 qui vérifie $Br(S3) = \{(b, R)\}$, permet de résoudre complètement le but.

# 4. Traduction en tactique réflexive

## 4.1. Niveaux de représentation

Dans cette section, nous détaillons la mise en œuvre de la procédure décrite dans la section précédente en utilisant les principes de la bibliothèque pour la réflexion. La tactique Omega utilise quatre niveaux différents de représentation du but :

**le codage initial du but dans CIC :** il s'agit du but à résoudre dans la logique de Coq

**une version réifiée du but dans CIC :** il s'agit de termes construit principalement avec des types inductifs de CIC et dont l'interprétation redonne un but équivalent au but initial

**une version très optimisée pour la procédure de décision :** elle permet de représenter les systèmes d'équations et de leur appliquer de manière efficace la procédure de décision $\Omega$

**une traduction en structure ML du but :** elle sert d'intermédiaire entre les versions précédentes et est utilisé par la partie en ML pour optimiser la trace et calculer les arguments auxiliaires qui simplifient la tâche de la fonction réflexive en CIC qui joue le rôle de tactique.

A chaque niveau est associé une représentation de termes abstraits (variables) pour ce niveau.

## 4.2. Vision globale de la procédure de décision

La procédure Omega est organisée en quatre parties :
- un ensemble de fonctions assurant l'extraction et la réification de la partie intéressante du but puis la construction de la trace en Coq,
- une procédure de décision qui cherche une solution et rend une trace complète de ses actions,
- une procédure d'optimisation de la trace,
- un moteur de rejeu `do_omega`.

La procédure complète Omega transforme le but initial $G$ en une version intermédiaire en ML $G_{ML}$ et un environnement $\rho$ pour interpréter $G_{ML}$. De cette version on extrait un ensemble de systèmes $(S_i^{ML})i \in I$ qui sont traduits en des systèmes optimisés $(S_i^{\Omega})_{i\in I}$ résolus par le noyau $\Omega$ de la procédure de décision qui rend pour chacun une trace $t_i$. De ces traces, un ensemble de systèmes utiles $U \subset I$ est extrait ainsi qu'une trace optimisée combinant les traces utiles $t$.

On en déduit une sous partie $G'_{ML}$ de $G_{ML}$ (correspondant à l'omission des hypothèses inutiles) nécessaire pour la preuve finale, sa traduction en syntaxe Coq classique $G'$ et réifiée $\lceil G' \rceil$. On limite $\rho$ à la partie nécessaire $\rho'$ pour $G'$ Finalement, on traduit la trace $t$ en syntaxe Coq $T$.

En Coq, on utilise la généralisation pour remplacer la conclusion de $G$ par $G'$ puis par (`interp` $\rho'$ $G'$). On utilise alors le théorème de correction de la procédure de décision en Coq pour le remplacer par (`interp` $\rho'$ (`do_omega` $T$ $G'$)). Enfin la réduction de `do_omega` donne (`interp` $\rho'$ **True**) qui se réduit en `True` qui est trivialement vrai.

Par rapport à la procédure précédemment décrite, la tactique réflexive ajoute ses propres optimisations :

1. Seules les hypothèses réellement utilisées sont sélectionnées pour le but à résoudre, seules les équations utiles sont réifiées, les autres sont remplacées par des variables propositionnelles,
2. Seules les variables numériques utilisées sont introduites dans l'environnement de réification.

Il faut noter qu'une variable Coq (variable entière ou proposition) doit avoir un codage propre à chaque environnement. La gestion des divers environnements propres à chacun des codage est la principale source de difficulté dans le codage de la tactique ML.

La construction de la trace $T$ optimisée et augmentée pour `do_omega` et la construction de `do_omega` et sa preuve de correction sont détaillées dans la suite.

## 4.3. La construction de la trace

La trace donnée à la tactique réflexive regroupe :
- la normalisation des équations (liste d'éléments de `hyp_trace`),
- la déstructuration des buts en systèmes d'inéquations,
- la résolution des systèmes.

Les deux derniers points sont regroupés dans une seule trace de type `trace_sol`.

```
Inductive trace : Set :=  Trace : (list hyp_trace) -> trace_sol -> trace.
```

**Trace pour la normalisation des systèmes** La normalisation des équations utiles est effectuée avant l'éclatement du but en systèmes, car une même équation peut être partagée par plusieurs systèmes. La trace indique le numéro de l'hypothèse, le chemin dans les connecteurs logiques jusqu'à l'équation, le besoin d'introduire ou non une double négation qui s'annulera lors de l'éclatement du but et enfin la conversion normalisant l'équation.

```
Inductive eq_trace : Set :=
  P_LEFT: eq_trace -> eq_trace | P_RIGHT:  eq_trace -> eq_trace   (* déplacement *)
| P_TRACE: conv -> eq_trac  (* applique une conversion normalisant l'équation *)
...
(* Identification de l'hypothèse et du chemin vers une équation à normaliser *)
Inductive hyp_trace : Set := HTRACE : nat -> eq_trace -> hyp_trace.
```

Les conversions sont décrites par un type inductif ayant un constructeur par théorème d'égalité utilisé et des constructeurs pour décrire comment se déplacer dans le terme pour trouver les points d'application des réécritures.

```
Inductive conv : Set :=
 | C_DO_BOTH: conv -> conv -> conv (* applique un pas à chaque branche d'un terme *)
 | C_SEQ: conv -> conv -> conv (* applique consécutivement deux opérations *)
 ...
 (* codage de chacun des théorèmes de réécriture *)
 | C_OPP_PLUS: conv (* - (x + y) = -x + -y *)
 | C_OPP_OPP: conv (* - -x = x *)
 ...
```

**Trace pour la construction et la résolution des systèmes** L'éclatement des buts se décompose en trois types d'opérations : éliminer les connecteurs de branchement donnant à chaque fois deux buts à résoudre, extraire les équations (déjà normalisées) et enfin résoudre les systèmes avec les traces d'$\Omega$.

```
(* Codage pour retrouver une équation ou un opérateur *)
Inductive dir : Set := D_left : dir | D_right : dir | D_mono : dir.

Inductive om_trace : Set :=
   (* Elimination d'un constructeur logique branchant *)
   E_SPLIT: nat -> (list dir) -> om_trace -> om_trace -> om_trace
 | E_EXTRACT: nat -> (list dir) -> om_trace -> om_trace (* Extraction d'équation *)
 | E_SOLVE: t_omega -> om_trace.  (* Résolution d'un but avec la trace d'Omega *)
```

Le codage des pas de la trace de $\Omega$ est fait par un type inductif assez semblable à celui des conversions :

```
(* Codage des pas de la tactique *)
Inductive t_omega : Set :=
 | O_CONSTANT_NOT_NUL: nat -> t_omega
 | O_CONSTANT_NEG: nat -> t_omega
 ...
```

La trace porte un certain nombre d'informations auxiliaires qui pourraient être recalculées par la tactique réflexive au prix de longs calculs en CIC. C'est le cas par exemple des fonctions de renormalisation des équations sommées qui sont très détaillées. Ces informations auxiliaires sont précalculées par la tactique ML.

En particulier, CIC ne permet pas la récursion non bornée. Plutôt que d'utiliser une preuve de terminaison des fonctions récursives utilisées, il est souvent plus simple de passer un entier bornant le nombre d'itérations nécessaires.

### 4.4. La fonction de rejeu réflexive do_omega

La fonction de rejeu de la trace enchaîne trois opérations : la normalisation des hypothèses, l'éclatement/résolution du but et enfin la simplification des buts résolus :

```
Definition omega_tactic [trace : trace ; c:proposition; l:hyps] := Cases trace of
  (TRACE t1 t2) =>
    (reduce (decompose_solve t2 (normalize_hyps t1 (cons (concl_to_hyp c) l)))) end.
```

concl_to_hyp vérifie que c est décidable et si oui le rajoute après négation aux hypothèses.

**Réécriture des arguments** Chaque réécriture est codée comme une analyse de la structure du terme qui donne le terme réécrit si l'argument était correct ou rend l'argument tel quel sinon.

```
Definition Topp_plus [t: term] :=
  Cases t of (Topp (Tplus x y)) => (Tplus (Topp x) (Topp y)) | _ => t end.
```

La correction de chaque réécriture est capturée par la définition term_stable. Ces théorèmes sont prouvés par une tactique spécifique pour la définition du type term qui propage la fonction $[\![-]\!]$ jusqu'à faire apparaître le théorème d'égalité. La tactique est écrite dans le langage $\mathcal{L}$tac [Del00].[2]

```
Definition term_stable [rewriting: term -> term] :=
 (e: env; t:term) (interp_term e t) = (interp_term e (rewriting t)).

Theorem Topp_plus_stable : (term_stable Topp_plus).
```

Ces fonctions élémentaires sont assemblées dans rewrite qui parcourt la trace des conversions et applique les conversions sur chaque terme ou sous terme.

```
Definition apply_both [f,g: term -> term; t : term]:= Cases t of
 | (Tplus x y) => (Tplus (f x) (g y)) | (Tmult x y) => (Tmult (f x) (g y))
 | x => x  end.

Fixpoint rewrite [s: step] : term -> term :=
  Cases s of
   | (C_DO_BOTH s1 s2) => (apply_both (rewrite s1) (rewrite s2))
   | (C_SEQ s1 s2) => [t: term] (rewrite s2 (rewrite s1 t))
   | C_OPP_PLUS => Topp_plus
   ...
```

Le code de normalize_hyps applique chaque normalisation à la bonne hypothèse. Le code de rewrite_hyp qui opère sur des comparaison atomiques est bâti sur le même principe que celui de rewrite qui opère sur les termes.

```
Fixpoint normalize_hyps [steplist: (list (prod nat p_step)] : hyps -> hyps :=
  [hyplist:hyps] Cases steplist of
    nil => hyplist
  | (cons (pos, s) remainder) =>
      (normalize_hyps remainder (apply_oper_1 pos (rewrite_hyp s) hyplist)) end.
```

[2] Malheureusement $\mathcal{L}$tac ne dispose pas des mécanismes de réflexion sur la syntaxe de Coq permettant de généraliser en une tactique utilisable dans toute syntaxe.

**Éclatement des buts** On utilise principalement la fonction `(extract_hyp_pos dl i)` pour extraire l'équation ou le constructeur à éclater sous le chemin *dl* de la $i^{ème}$ hypothèse puis on passe le système obtenu à `execute_omega`.

```
Fixpoint decompose_solve [s: e_step] : hyps -> lhyps :=  [h:hyps]
  Cases s of
   (E_SPLIT i dl s1 s2) =>
        (Cases (extract_hyp_pos dl i h) of
           (Tor x y) => (app (decompose_solve s1 (cons x h))
                             (decompose_solve s2 (cons y h)))
           ...
        | _ => (cons h (nil hyps))
        end)
 | (E_EXTRACT i dl s1) =>
     (decompose_solve s1 (cons (extract_hyp_pos dl i h) h))
 | (E_SOLVE t) => (execute_omega t h) end.
```

**Résolution des systèmes** Les opérations de base de la trace d'Omega sont implémentées séparément comme des fonctions vérifiant les hypothèses passées en argument et rendant une nouvelle hypothèse (en particulier [**False**] s'il s'agit d'une contradiction). En cas d'erreur l'hypothèse est rendue telle quelle.

```
Definition constant_nul [i:nat; h: hyps] :=
  Cases (nth_hyps i h) of (NeqTerm (Tint ZERO) (Tint ZERO)) => absurd | _ => h end.

Theorem constant_nul_valid : (i:nat) (valid_hyps (constant_nul i)).
```

Chaque pas est assez proche d'un théorème de réécriture mais il peut être complété par des tests plus complexes sur la structure des termes utilisés. Par exemple, le test de décidabilité est exprimé sous forme d'une fonction qui parcourt les prédicats réifiés. Un théorème de validité est associé à cette fonction qui la relie à la notion classique de décidabililité en Coq.

```
Fixpoint decidability [p:proposition] : bool := Cases p of
    (EqTerm _ _) => true
  | (Tprop _) => false
  | (Tand t1 t2) => (andb (decidability t1) (decidability t2))
  ... end.

Theorem decidable_correct :
  (ep: PropList) (e: (list Z)) (p:proposition)
    (decidability p)=true -> (decidable (interp_proposition ep e p)).
```

Les opérations sont assemblées par `execute_omega` qui lit récursivement la trace suivant les principes de 2.4 et exécute les opérations sur le but.

```
Fixpoint execute_omega [t: t_omega] : hyps -> lhyps :=
  [l : hyps] Cases t of
   | (O_CONSTANT_NUL i) => (singleton (constant_nul i l))
   | (O_EXACT_DIVIDE k body t cont n) =>
       (execute_omega cont (apply_oper_1 n (exact_divide k body t) l))
   | (O_SPLIT_INEQ t i cont1 cont2) =>
       (split_ineq i t (execute_omega cont1) (execute_omega cont2) l)
   ... end.
```

```
(A) ∀x,y,z:Z. 1+x=z+1∨2+y=x+2 ⇒
        x=y+4∧x=x ∧ (y=y∧x=z+2∨z=x+3) ⇒ False
(B) ∀x,y,z:Z. x ≤ y ⇒ y ≤ z ⇒ x ≤ z
(C) ∀x,y,t:Z. x ≤ y ⇒ y ≤ z ⇒ (x ≤ z) ∧ (x ≤ y)
(D) ∀m,n,N,y,z:nat.
      0 < x ⇒ 0 < y ⇒ 0<z ⇒
      (z ≤ y ∧ z > N ∧ y=m ∧ ¬ y < m ⇒ (2 * z) ≤ y ⇒
        (2 * z) ≤ y ∧ (2 * z) ≥ N ∧ y=m ∨ y < m.
(E) ∀y,y0,y1,y2,y3,y4:Z.
    y < y1 ∨ (y = y1 ∧ y0 < y2) ⇒ y1 < y3 ∨ (y1 = y3 ∧ y2 < y4)
    ⇒ y < y3 ∨ (y = y3 ∧ y0 < y4)
```

TAB. 3 – Code des problèmes utilisés

## 5. Résultats expérimentaux

Les exemples présentés dans la table 3 ont été exécutés :

- avec la procédure Omega classique qui expande directement la trace en tactiques Coq et n'optimise pas le nombre de systèmes à résoudre (c'est la version disponible dans la distribution standard de Coq),
- une version de la tactique qui utilise la réflexion sur le noyau de la procédure de décision elle même mais pas sur le traitement des connecteurs logiques (donc qui traite un but à la fois mais ne sait pas gérer des sous buts multiples),
- enfin la procédure Omega qui utilise toutes les optimisations décrites.

Même sur de tels exemples pour lesquels la phase de traduction est importante et la résolution courte, on observe une amélioration réelle des résultats avec la réflexion et en particulier lorsque la réflexion est utilisée sur toute la procédure de décision.

L'amélioration de la taille des preuves est aussi très sensible. Cela a aussi une influence immédiate sur le temps de vérification des preuves (type-checking).

| | Temps | | | | | Taille | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | classique | refl. naïve | | refl. totale | | classique | refl. naïve | | refl. totale | |
| | t1 (s) | t2 (s) | t1/t2 | t3 (s) | t1/t3 | s1 (ko) | s2 (ko) | s1/s2 | s3 (ko) | s1/s3 |
| A | 2.07 | 2.04 | 1.0 | 1.12 | 1.8 | 92 | 56.7 | 1.6 | 20.6 | 4.5 |
| B | 0.61 | 0.44 | 1.4 | 0.45 | 1.4 | 27 | 12.7 | 2.1 | 9.8 | 2.8 |
| C | 0.89 | 0.76 | 1.2 | 0.60 | 1.5 | 39.3 | 20.9 | 1.9 | 12.6 | 3.1 |
| D | 1.83 | 1.81 | 1.0 | 1.28 | 1.4 | 82.9 | 57.3 | 1.4 | 22.8 | 3.6 |
| E | 5.60 | 4.55 | 1.2 | 1.84 | 3.0 | 231,6 | 113,4 | 2.0 | 55.2 | 4.2 |

Enfin, à titre d'information, voici quelques éléments sur la taille de l'implémentation. Le noyau de la tactique Omega représente 650 lignes de code ML pour la procédure de décision et la simplification de sa trace et 300 lignes de code COQ correspondant à des règles de réécriture arithmétiques utilisés par la procédure de décision.

La version classique est complétée par 1900 lignes de code ML calculant la tactique à partir de la trace. Pour la version réflexive, on a à la place 1550 lignes de code ML pour le lien entre la procédure de décision et d'optimisation des traces (en particulier l'heuristique pour limiter le nombre de systèmes résolus), et 2500 lignes de code COQ pour le codage du moteur réflexif.

# 6. Conclusion

Il existe de nombreuses implantations de procédures de décision pour l'arithmétique ([Sho77] pour une des plus anciennes). Dans [Nor03] M. Norrish a présenté deux méthodes complètes en HOL dont l'une est Omega. Par rapport, à la version en COQ, la version de Norrish réalise en plus les calculs permettant de contrôler l'absence de solution entière dans certains cas où le polyèdre des solutions réelles n'est pas vide mais est très réduit. Nous avons choisi de faire l'impasse sur cette étape car les cas d'utilisation sont rares et la complexité calculatoire importante.

L'utilisation efficace de la réflexion est récente et J. Harrison justifiait cette inefficacité dans [Har95]. En fait son argumentation est valable pour HOL où la réduction n'est pas une opération primitive contrairement à la réécriture. Après les résultats de D. Howe en NuPrL [How88] et de Boyer et Moore [BM81], Samuel Boutin [Bou97] a été le premier à implémenter une tactique réflexive montrant un gain important sur les tactiques classiques.

[RGL01] présente deux implantations d'une procédure de décision sur les automates, l'une sur une technique légère (la procédure elle même est externe), l'autre totale (la procédure de décision elle même est codée en Coq) et argumente en faveur de la seconde. Dans le cas d'Omega, la recherche d'une preuve est longue alors que la vérification est rapide, dans ce cas si la différence de vitesse d'exécution entre le méta-langage de tactique et l'interpréteur de la logique est importante (ce qui est le cas), la réflexion partielle est préférable.

La réécriture en Coq étant particulièrement lente, elle a concentré de nombreux travaux. H. Barendregt, G. Barthe et M. Ruys [BRB96] ont présenté une méthodologie pour le raisonnement équationnel basée sur la réflexion. Dans [Alv02], C. Alvarado a développé le premier cadre réellement générique pour la réécriture dans une algèbre multi-sortée. Il repose sur un codage complexe des termes utilisant les types dépendants et les performances ne sont pas encore très satisfaisantes.

La particularité de ce travail est de présenter l'application de la réflexion à une tactique assez complexe et de mélanger des optimisations sur la trace de réécriture faites dans le méta-langage à l'utilisation d'une tactique réflexive construite à partir d'une bibliothèque permettant de coder un ensemble de tactiques primitives pour un assistant de preuve sur une logique du premier ordre.

Il présente un traitement moins générique mais plus pragmatique de la réécriture que celui de [Alv02] et ne se limite pas à celle-ci. Néanmoins, la solution présentée ici a le défaut d'être difficilement extensible : il faut en effet ajouter de nouveaux types inductifs et modifier la représentation des environnements et la solution est probablement dans l'amélioration du codage de C. Alvarado pour «oublier» les types.

Les enseignements principaux sont les suivants :

- la bibliothèque pour construire les tactiques réflexives présentée ici n'est qu'une base qui doit être enrichie en particulier pour le traitement des quantificateurs ; néanmoins elle est déjà suffisante pour de nombreuses tactiques et le codage en Coq du moteur de rejeu de la trace bien que fastidieux est relativement immédiat ;
- l'optimisation des traces est ad'hoc, demande par contre des heuristiques complexes et l'existence de différents niveaux de représentation avec des objets équivalents mais pas forcément similaires (les variables n'ont pas le même sens dans $\Omega$ et Omega) introduisent des risques de confusion et donc de violation des invariants qui leur sont associés[3] ;
- le besoin de bibliothèques auxiliaires pour simplifier la rédaction des procédures :
  - pour la création et la traduction des structures de données entre Coq, ML et Coq réifié [4],
  - des outils de debugging pour tracer l'exécution de fonctions Coq,
  - un langage de tactique utilisable à la fois en mode interactif comme $\mathcal{L}$tac et traduisible en une

[3]Une erreur dans le code de la tactique n'introduit pas de contradiction dans le système. Simplement elle échouera sur des théorèmes qu'elle aurait du résoudre.

[4]La tactique `Quote` répond partiellement à la question. Elle n'est néanmoins pas adaptée pour le traitement des variables.

fonction Coq réflexive.

## Références


[Alv02] Cuihtlauac Alvarado. *Réflexion pour la réécriture dans le calcul des constructions inductives*. PhD thesis, Université Paris 11, december 2002.

[Bar98] Bruno Barras. Efficient reduction functions in coq. available at `http ://pauillac.inria.fr/ barras/reductionq`, 1998.

[BM81] R. Boyer and J S. Moore. Metafunctions : Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*. Academic Press, 1981.

[Bou92] Richard J. Boulton. On efficiency in theorem provers which fully expand proofs into primitive inferences. Technical report, University of Cambridge, 1992.

[Bou95] Samuel Boutin. Simple rewriting tactics in Coq. Dans le répertoire contrib de Coq, 1995.

[Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *TACS*, pages 515–529, 1997.

[BRB96] Gilles Barthe, Mark Ruys, and Henk Barendregt. A two-level approach towards lean proof-checking. In Sterano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop, Types 95*, volume 1158 of *LNCS*, pages 16–40. Springer Verlag, june 1996.

[Del00] David Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955, pages 85–95. Springer-Verlag LNCS/LNAI, November 2000.

[DM01] David Delahaye and Micaela Mayero. `Field` : une procédure de décision pour les nombres réels en Coq. In *Journées Francophones des Langages Applicatifs*. INRIA, 2001.

[Har95] John Harrison. Metatheory and reflection in theorem proving : A survey and critique. Technical Report CRC-053, SRI International, Cambridge, 1995.

[How88] Douglas Howe. Computational metatheory in nuprl. In *Automated Deduction, 5th International Conference (CADE'88)*, volume 310 of *LNCS*, pages 238–257. Springer Verlag, may 1988.

[Nor03] Michael Norrish. Complete integer decision procedures as derived rules in hol. In *Theorem Proving in Higher Order Logics (TPHOL 03)*, volume 2758, pages 71–86. Springer, 2003.

[Pug92] William Pugh. The omega test : a fast and practical integer programming algorithm for dependence analysis. *CACM*, (8) :102–114, august 1992.

[RGL01] Xavier Rival and Jean Goubault-Larrecq. Experiments with finite tree automata in coq. In *Theorem Proving in Higher Order Logics (TPHOL 01)*, volume 2152, pages 362–377. Springer Verlag, 2001.

[Sho77] Robert E. Shostak. On the SUP-INF method for proving presburger formulas. *JACM*, 24(4) :529–543, 1977.

[VGLPK00] Kumar Neeraj Verma, Jean Goubault-Larrecq, S. Prasad, and S. Arun Kumar. Reflecting bdds in coq. In *6th Asian Computing Science Conference (ASIAN'2000)*, volume 1961 of *LNCS*, Penang, Malaysia, november 2000.