Typage des graphes de décisions ternaires

Hervé Marchand, Michel Le Borgne

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Typage des graphes de décisions ternaires

HERVÉ MARCHAND, MICHEL LE BORGNE

N° 2185

24 mars 1994

PROGRAMME 2

Calcul symbolique,

programmation

et génie logiciel

ISSN 0249-6399

INRIA
RENNES

Typage des graphes de décisions ternaires

Hervé Marchand *, Michel Le Borgne **

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet EP-ATR

Rapport de recherche n ° 2185 — 24 mars 1994 — 34 pages

Résumé : Dans ce rapport, nous allons étudier le problème de la représentation des formes polynomiales sous forme de graphes de décisions ternaires typés : TDD typés.

Après avoir présenté brièvement les travaux antérieurs sur les TDD (ternary decision diagram), nous allons montrer comment il est possible de typer ces TDD tout en s'attachant à conserver la canonicité de ceux ci.
On proposera ensuite une étude comparative entre les deux modes de représentations. Cette étude permettra de montrer l'interêt du typage à l'aide de permutations pour les fonctions polynomiales de base. On verra par la suite différentes méthodes visant à améliorer la représentation en mémoire des TDD.

Mots-clé : SIGNAL, Graphes de décisions ternaires

(Abstract: pto)

*INRIA e-mail hmarchan@irisa.irisa.fr
**IRISA e-mail leborgne@irisa.irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

The marked ternary-decision diagrams

Abstract: In this report, we study the problem of the representation of the polynomial functions by the marked, reduced, ordered, ternary-decision diagrams : the marked TDD. After a short presentation of B. Dutertre's work on ordinary TDD, we show how it's possible to mark branches of TDD with permutations. Next, we propose a comparative study between the two representations of TDD. This shows the benefit of using the marked TDD for the basic polynomial functions . After that, we see different methods which can be used to improve the efficiency of the marked TDD's algorithms and to reduce the memory used by the TDD.

Key-words: SIGNAL, ternary-decision diagrams

Table des matières

1 Introduction

L'objectif du projet EP-ATR est de contribuer au développement de concepts, de méthodes et de logiciels pour la conception et la mise en œuvre d'applications dans le domaine du temps réel. Les travaux effectués s'inscrivent dans le contexte de la programmation synchrone. Les langages synchrones visent à rendre plus sûre et plus facile la programmation de systèmes qui interagissent continûment avec des environnements exigeants. C'est dans ce cadre qu'a été défini un langage de programmation temps réel synchrone, SIGNAL[6], destiné à la conception et à la mise en œuvre de systèmes temps réel.
Un programme SIGNAL définit implicitement un système de transition à partir d'un sytème d'équations dont les variables sont les identificateurs de signaux. Ces équations peuvent être organisées en sous systèmes (ou processus). Un signal est une suite de valeurs à laquelle est associée une horloge, qui permet de définir l'ensemble des instants où le signal possède une valeur. Une algèbre de systèmes dynamiques dans $\mathbf{Z}/_{3\mathbf{Z}}$ [1], le corps des entiers modulo 3, qui décrit complètement la synchronisation d'un processus, est définie sur le langage noyau. Ce calcul, appelé calcul d'horloges, permet la vérification de la correction d'un programme vis-à-vis des synchronisations et la synthèse d'un graphe de dépendance des calculs, conditionné par les horloges.
Le comportement d'un programme SIGNAL, du point de vue logique et temporel, peut être modélisé par un système dynamique dans $\mathbf{Z}/_{3\mathbf{Z}}$. C'est à dire la donnée

1. d'un ensemble de variables d'états $X = \{X_1, \ldots, X_n\}$,
2. d'un ensemble de variables d'événements $Y = \{Y_1, \ldots, Y_m\}$,
3. d'un ensemble de contraintes $Q(X, Y) = 0$,
4. d'une fonction d'évolution $P(X, Y)$ de $(\mathbf{Z}/_{3\mathbf{Z}})^{n+m}$ dans $(\mathbf{Z}/_{3\mathbf{Z}})^n$,
5. d'une contrainte d'initialisation $Q_0(X) = 0$.

Un système dynamique sera noté :

$$\begin{cases} Q(X,Y) &= 0 \\ X' &= P(X,Y) \\ Q_0(X) &= 0 \end{cases}$$

Les X proviennent de la traduction des signaux retardés, les Y représentent les signaux booléens et les horloges du programme. Le système dynamique décrit les suites de couples états / événements que peut produire le programme, c'est à dire les suites $(x_i, y_i)_{i \in \mathbb{N}}$ telles que $Q_0(x_0) = 0$ et pour tout $i \in I\!N$,

$$\begin{aligned} Q(x_i, y_i) &= 0 \\ P(x_i, y_i) &= x_{i+1} \end{aligned}$$

Ces suites sont appelées les *trajectoires* du système. Un couple x, y tel que $Q(x, y) = 0$ est dit *admissible* ; on dira aussi qu'un état x est admissible s'il existe un y tel que $Q(x, y) = 0$. Si x et x' sont deux états et qu'il existe un y tel que $Q(x, y) = 0$ et $x' = P(x, y)$, on dira que x' est un *successeur* de x et réciproquement que x est un *prédécesseur* de x'. L'équipe EP-ATR a développé SIGALI , un système de calcul formel interactif spécialisé dans les calculs algébriques sur l'anneau $\mathcal{F}_3[X]/ < X^3 - X >$ [5]. Il est destiné à la vérification des propriétés statiques et dynamiques de programmes SIGNAL et plus généralement de tout système dynamique polynomial dans $\mathbf{Z}/_{3\mathbf{Z}}$. SIGALI est un programme que l'on peut schématiser de la manière suivante (voir figure 1).

- **Interpréteur** :Le langage de l'interpréteur permet entre autres d'écrire des fonctions polynomiales. En particulier, il existe une commande permettant de charger le code $\mathbf{Z}/_{3\mathbf{Z}}$ généré par le compilateur.
- **module polynôme** Dans ce module sont codés les fonctions polynomiales sous forme de TDD (graphes de décisions ternaires). Il fait appel pour cela au module TDD.

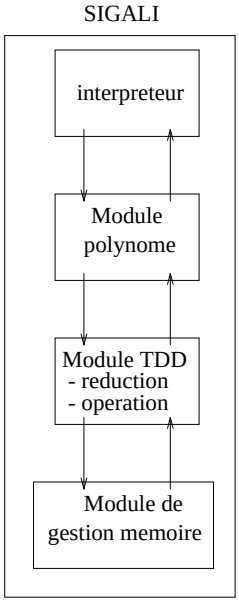

Figure 1 :

- **module TDD** Dans cette partie sont effectuées les opérations et les réductions sur les polynômes en provenance du module polynôme.
- **module gestion mémoire** SIGALI est un langage de calcul formel, ce qui nécessite une gestion dynamique de la mémoire.

Le programme SIGALI travaille dans $A = \mathcal{F}_3[X]/ < X^3 - X >$ avec $X = (X_1, X_2, .., X_n)$. On munit les variables d'un ordre complet $X_1 < X_2 < ... < X_n$. Le fait de se placer dans A réduit la preuve des propriétés des systèmes dynamiques à de simples calculs sur des polynômes à coefficients dans $\mathbf{Z}/_{3\mathbf{Z}}$ de degré au plus 2 en chaque variable (petit théorème de Fermat). Ceci a pour conséquence de réduire considérablement la taille des polynômes. Cependant, malgré cela, la taille est au pire exponentielle, chaque fonction polynomiale peut contenir dans le pire des cas 3^n monômes différents.
Il apparait donc indispensable de trouver une représentation de ces fonctions polynomiales qui soit la plus compacte possible. Dans l'algèbre de Boole , le problème a déjà été résolu, en utilisant un graphe pour représenter les fonctions polynomiales: les graphes de décision binaire ou BDD [2]. On va dans la suite adapter les BDD en graphes de décision ternaire ou TDD.
Dans une 1^{ere} partie, on définira les outils permettant de décrire les TDD. Puis dans une 2^{eme} partie, on expliquera comment améliorer l'efficacité des TDD en utilisant des permutations pour les typer. Enfin dans les 3^{eme} partie et 4^{eme} partie, on fera une comparaison entre ces deux méthodes d'implémentation des fonctions polynomiales ainsi que les améliorations à apporter aux méthodes déjà existantes.

2 Méthode de décomposition des fonctions polynomiales sous forme de TDD

2.1 Méthodes algébriques utilisées

A la base, le comportement logico-temporel d'un langage SIGNAL peut être modélisé par un système dynamique dont les variables sont dans $\mathbf{Z}/_{3\mathbf{Z}}$ [1]. C'est dans cette optique que l'on utilise une algèbre de

polynômes à coefficients dans $\mathbf{Z}/_{3\mathbf{Z}}$: $A = \mathcal{F}_3[X]/ < X^3 - X >$ avec $X = (X_1, X_2, .., X_n)$.
On se place dans A.
On pose $A_i = \mathcal{F}_3[X_i, .., X_n]/ < X_i^3 - X_i, .., X_n^3 - X >$.
où A_i est l'ensemble des fonctions qui ne dépendent que de $X_1, X_2, .., X_{i-1}$ et A_{n+1} l'ensemble des fonctions constantes. Les différents anneaux (non intègres) que l'on obtient de cette manière sont par conséquent ordonnés par inclusion :

$$A_1 \supset A_2 \supset ... \supset A_{n+1}$$

On pose $\forall i \in [1..n]$

$$\begin{aligned} e_i^1 &= -X_i - X_i^2 && (1)\\ e_i^2 &= X_i - X_i^2 && (2)\\ e_i^3 &= 1 - X_i^2 && (3) \end{aligned}$$

Une rapide vérification montre que:

$$\begin{aligned} e_i^\alpha . e_i^\alpha &= e_i^\alpha && (4)\\ e_i^\alpha . e_i^\beta &= 0 \text{pour } \alpha \neq \beta && (5)\\ e_i^1 + e_i^2 + e_i^3 &= 1 && (6) \end{aligned}$$

Remarque 1 *On obtient ainsi une famille d'idempotents orthogonaux. Ceci sera très important pour les opérations sur les TDD, comme nous le verrons plus tard.*

Pour obtenir un polynôme sous une forme compactée, on va représenter celui-ci sous la forme d'un arbre. On utilise pour cela la formule généralisée de Shannon .
La décomposition du polynôme en la variable i s'effectue de la manière suivante :

Définition 1 décomposition généralisée de Shannon
Soit f une fonction polynomiale appartenant à A_i et x_i la variable dominante de cette fonction polymômiale, alors

$$f = e_i^1 . f_1 + e_i^2 . f_2 + e_i^3 . f_3 \quad (7)$$

avec

$$\begin{aligned} f_1 &= f/_{x_i=1} && (8)\\ f_2 &= f/_{x_i=-1} && (9)\\ f_3 &= f/_{x_i=0} && (10) \end{aligned}$$

De cette décomposition, on tire une représentation du polynôme f appelée h-expression qui s'obtient récursivement :

- $h(f)$=constante si f est une constante
- sinon, on cherche le plus petit A_i contenant f et on décompose alors f par rapport à la variable x_i .

$$f = e_i^1 . f_1 + e_i^2 . f_2 + e_i^3 . f_3$$

$$h(f) = e_i^1 . h(f_1) + e_i^2 . h(f_2) + e_i^3 . h(f_3)$$

Cette expression peut être représentée par le graphe de la figure 2

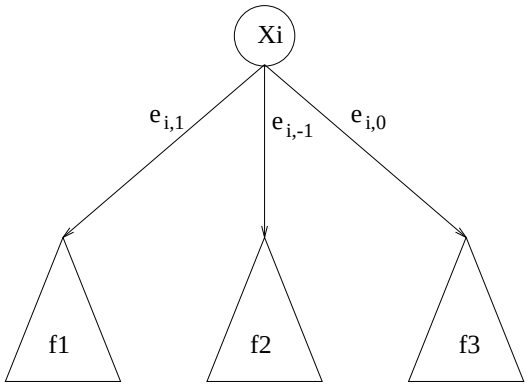

Figure 2 :

Exemple :

$$f = X_1X_2^2 + X_1X_3^2 - X_2^2$$

Par décomposition, on obtient :

$$\begin{aligned} f_1 &= X_3^2 \\ f_2 &= -2X_2^2 - X_3^2 \\ f_3 &= -X_2^2 \end{aligned}$$

On continue la décomposition de la même manière pour les polynômes f_1, f_2 et f_3.

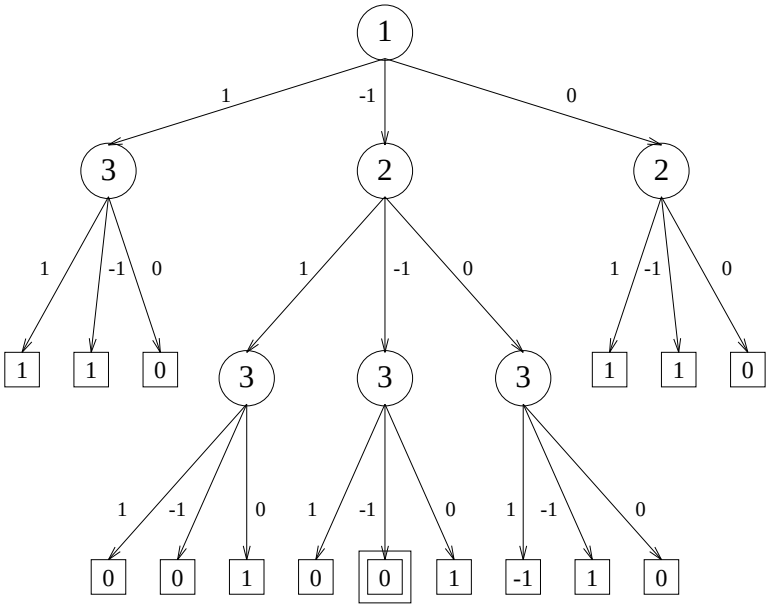

Figure 3 : exemple de décomposition sous forme de graphe

La décomposition finale donne l'équation suivante :

$$f = e_1^1(e_3^1 + e_3^2) + e_1^2(e_2^1e_3^3 + e_2^2e_3^3 + e_2^3(-e_3^1 - e_3^2)) + e_1^3(-e_2^1 - e_2^2)$$

Ce qui peut se représenter par l'arbre de la figure 3
Chaque feuille de l'arbre est une constante correspondant à la valeur prise successivement par le polynôme quand chacune des variables prend la valeur correspondant à l'unique chemin menant de la racine à cette feuille. Par exemple, dans la figure 3, la feuille 0 (entourée 2 fois) correspond à la valeur prise par le polynône quand $x_1 = -1$, $x_2 = -1$ et $x_3 = -1$.
Chaque sous arbre représente une sous h-expression de $h(f)$. On se rend compte dans le graphe de la figure 3 que plusieurs sous arbres sont identiques (Ils représentent la même sous h-expression de f). Il apparait donc possible de compacter encore plus cette représentation afin de diminuer les redondances qui se trouvent sur ce graphe. Ceci nous amène à introduire la notion de graphes de décision ternaires : TDD.

2.2 Les graphes de décision ternaires : TDD

La nécessité d'obtenir une représentation plus compacte pour représenter les fonctions polynomiales est primordiale pour des raisons de place mémoire. Ce chapitre présente la manière dont ce compactage a été réalisé à l'aide des graphes de décision ternaires. Une étude a déjà été réalisée sur ce sujet par B. Dutertre [5].

2.2.1 Définition

Un graphe de décision ternaire est un graphe orienté, connexe, sans circuit, possédant une racine, où :

- Chaque sommet terminal est une constante.
- Chaque sommet non terminal est étiqueté par une variable, et possède trois sommets successeurs.

Plus formellement:

Définition 2 : définition d'un TDD
on considère n variables $X_1, X_2, ..., X_n$ que l'on ordonne de façon arbitraire $X_1 < X_2 < ... < X_n$; on considère $G =< S, \delta, var, val >$ avec :

- *$S = T \bigcup N$ ensemble fini de sommets du graphe*
  T:ensemble des sommets terminaux
  N:ensemble des sommets non terminaux
- *$\delta : N \times \mathbf{Z}/_{3\mathbf{Z}} \rightarrow X$*
  tel que $(\delta(a, x) = b) \Rightarrow (a, b)$ forme un arc de G
- *$var : N \rightarrow X$*
  $val : T \rightarrow \mathbf{Z}/_{3\mathbf{Z}}$
  Ces deux fonctions définissent l'étiquetage des sommets.

On ajoute de plus la contrainte suivante

$$\forall a \in N, \forall b \in N, \forall x \in \mathbf{Z}/_{3\mathbf{Z}} \ , \ \delta(a, x) = b \ \Rightarrow var(a) < var(b) \tag{11}$$

Remarque 2 *Grâce à cette contrainte, le graphe ne peut avoir de circuit.*

On pose [G]= cardinal de S = taille du graphe et on note G_a le TDD tronqué ayant a comme racine, où a est normalement un nœud du TDD G.

Remarque 3 *Toute fonction polynomiale à coefficients dans $\mathbf{Z}/_{3\mathbf{Z}}$ admet une décomposition sous forme de TDD.*

2.2.2 Interprétation

A tout sommet a du graphe $G =< S, \delta, var, val >$, on associe un élément de $\mathcal{F}_3[X]/ < X^3 - X >$ que l'on appelle $\mathcal{P}(a)$ défini récursivement de la manière suivante :

- Si $a \in T$: $\mathcal{P}(a) = val(a)$
- Si $a \in N$ et $X_i = var(a)$, alors

$$\mathcal{P}(a) = e_i^1.\mathcal{P}(\delta(a,1)) + e_i^2.\mathcal{P}(\delta(a,-1)) + e_i^3.\mathcal{P}(\delta(a,0))$$

On peut également regarder un TDD comme une machine qui renvoie l'image d'un point $x = (x_1, x_2, ..., x_n)$,où x_i est la valeur prise par la variable X_i. La lecture d'un TDD se ramène donc à la lecture successive de $n+1$ sommets $a_0, ..., a_n$ que l'on sélectionne ainsi:

- Le sommet a_0 est la racine du TDD
- Si le sommet a_i est non terminal et $var(a_{i-1}) = X_i$ alors $a_i = \delta(a_{i-1}, x_i)$ sinon $a_i = a_{i-1}$
  Le sommet a_n est forcément un terminal et $f(x_1, x_2, ..., x_n) = val(a_n)$.

En fait, parmi tous les TDD qui représentent le même polynôme, il en existe un minimal et unique, à un isomorphisme près, qui sera appelé TDD réduit.

Remarque 4 *On obtient ainsi la canonicité : pour un polynôme donné, il n'existera qu'un unique TDD réduit.*

2.2.3 TDD réduit

Définition 3 TDD réduit *Un TDD $G < S, \delta, var, val >$ est dit réduit si et seulement si on ne peut trouver parmi les sommets du graphe deux sommets distincts a et b tels que $\mathcal{P}(a) = \mathcal{P}(b)$.*

2.2.4 Propriété

Propriété 1 *Un TDD $G < S, \delta, var, val >$ est réduit si et seulement si les trois conditions suivantes sont réalisées:*

1. *S ne possède pas 2 terminaux distincts a et b tels que $val(a) = val(b)$*
2. *Il n'existe pas parmi les non-terminaux de G deux sommets a et b tels que $var(a) = var(b)$ et $\forall x \in \mathbf{Z}/_{3\mathbf{Z}} \quad \delta(a,x) = \delta(b,x)$*
3. *Il n'existe pas de non-terminal a tel que $\delta(a,1) = \delta(a,0) = \delta(a,-1)$*

2.2.5 exemple de TDD réduit

(cf figure 4)

La réduction n'apparait pourtant pas encore tout à fait satisfaisante. En fait, la réduction ne regroupe que les nœuds qui représentent la même fonction polynomiale. Or si l'on considère une fonction polynomiale f dans $\mathcal{F}_3[X]/ < X^3 - X >$, et une permutation σ dans $\mathbf{Z}/_{3\mathbf{Z}}$, la fonction obtenue en composant f avec σ est encore une fonction polynomiale dans $\mathcal{F}_3[X]/ < X^3 - X >$. Le but de ce qui va suivre est de montrer que l'on peut regrouper en un seul nœud, les deux nœuds qui représentaient respectivement f et $\sigma(f)$.

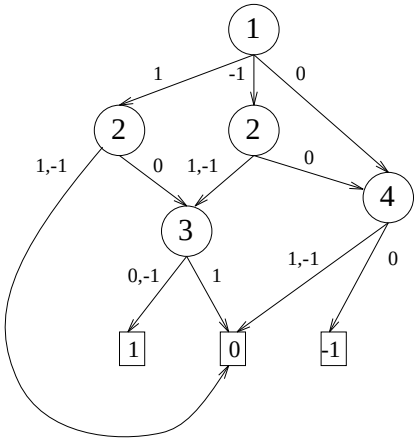

Figure 4 : TDD réduit

3 TDD typés

Soit σ une permutation de $\mathcal{S}_3$, où $\mathcal{S}_3$ représente l'ensemble des permutations d'un ensemble à 3 éléments (ici $\mathbf{Z}/_{3\mathbf{Z}}$). On définit une action du groupe $\mathcal{S}_3$ sur les fonctions polynomiales en posant : $\sigma(f)(x) = \sigma(f(x))$ Soit f une fonction polynomiale et G son TDD. Le TDD de $\sigma(f)$ est obtenu à partir de G, en effectuant une copie totale de ce dernier et en changeant dans les feuilles de l'arbre 0 par $\sigma(0)$, 1 par $\sigma(1)$ et -1 par $\sigma(-1)$. L'idée est de se servir de cette propriété pour obtenir une représentation des fonctions polynomiales encore plus compacte. En effet, intuitivement si l'on considère deux sous-graphes G_1 et G_2 représentant les fonctions polynomiales f_1 et f_2, si de plus il existe une permutation σ telleque $f_2 = \sigma(f_1)$, alors l'idée est de regrouper les deux sous-graphes, G_1 et G_2, en un seul, comme l'illustre la figure 5. On suppose que dans la figure 5,

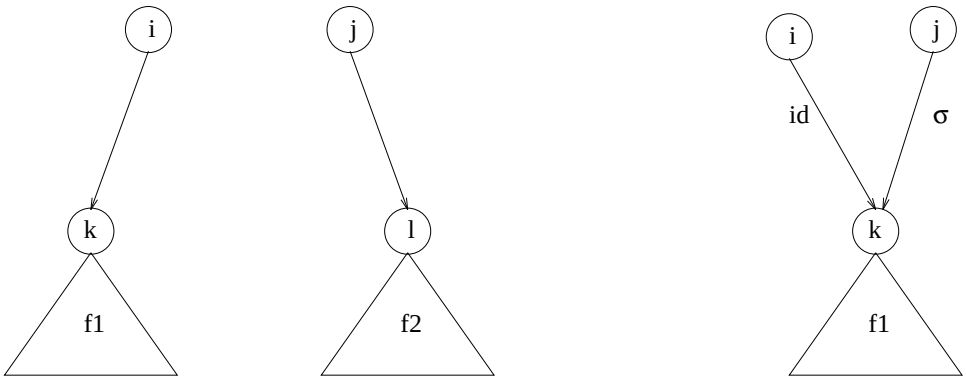

Figure 5 : réduction par typage

il existe une permutation σ tel que $\sigma(f_1) = f(2)$ pour les nœuds (k) et (l), on peut alors supprimer le nœud (l) et relier l'arc partant de (j) et allant sur (l) sur le nœud (k) en lui affectant la permutation σ.
On obtiendra de cette manière une représentation plus compacte: en effet, lorsque l'on réduisait les TDD sans les typer, on supprimait un nœud b si et seulement si seulement il existait déjà un noeud a tel que $\mathcal{P}(a) = \mathcal{P}(b)$; ici, on pourra supprimer un nœud b lorsque celui-ci sera équivalent, à une permutation près, à un autre nœud a, soit, lorsque $\sigma(\mathcal{P}(a)) = \mathcal{P}(b)$.

3.1 Définition des TDD typés

3.1.1 Définition

Définition 4 TDD typé
Un graphe de décision ternaire typé est un graphe orienté, connexe, sans circuit, possédant une racine, où :

- *Chaque sommet terminal est une constante*
- *Chaque sommet non terminal est étiqueté par une variable et possède trois sommets successeurs qui sont interprétés par trois permutations.*

Plus formellement:

Définition 5 *on considère n variables $X_1, X_2, ..., X_n$ que l'on ordonne de façon arbitraire $X_1 < X_2 < ... < X_n$*
On considère $G =< S, \delta, perm, var, val >$ avec :

- $S = T \bigcup N$ *ensemble fini de sommets du graphe*
  T: ensemble des sommets terminaux
  N: ensemble des sommets non terminaux
- $\delta : N \times \mathbf{Z}/_{3\mathbf{Z}} \rightarrow X$
  tel que $\delta(a, x) = b \Rightarrow (a, b)$ *forme un arc de G*
- $perm : N \times \mathbf{Z}/_{3\mathbf{Z}} \rightarrow \mathcal{S}_3 \quad perm(a, i) = \sigma_i$
  $var : N \rightarrow X$
  $val : T \rightarrow \mathbf{Z}/_{3\mathbf{Z}}$
  Ces trois fonctions définissent l'étiquetage des sommets et des arcs.

On ajoute de plus la contrainte suivante:

$$\forall a \in N, \forall b \in N, \forall x \in \mathbf{Z}/_{3\mathbf{Z}} \ , \ \delta(a, x) = b \ \Rightarrow var(a) < var(b)$$

3.1.2 Interprétation

Soit $G =< S, \delta, perm, var, val >$, un TDD. De manière similaire au cas des TDD non typés, on associe, à chaque sommet a de G , un élément de $\mathcal{F}_3[X]/ < X^3 - X >$ que l'on appelle $\mathcal{P}(a)$ défini récursivement de la manière suivante :

- Si a est un terminal : $\mathcal{P}(a) = val(a)$
- Si $a \in N$, $X_i = var(a)$ et $\sigma_i = perm(a, i)$

$$\mathcal{P}(a) = e_i^1.\sigma_1(\mathcal{P}(\delta(a, 1))) + e_i^2.\sigma_2(\mathcal{P}(\delta(a, -1))) + e_i^3.\sigma_3(\mathcal{P}(\delta(a, 0)))$$

Remarque 5 *La formule de Shannon donne, en effet, pour les TDD typés :*

$$f = e_i^1.\sigma_1(f_1) + e_i^2.\sigma_2(f_2) + e_i^3.\sigma_3(f_3)$$

Remarque 6 *On remarque facilement qu'un TDD non typé est en fait un TDD typé, où toutes les permutations sont égales à l'identité. On en déduit donc que toute fonction polynomiale admet une représentation sous forme de TDD typé.*

Le paragraphe qui suit définit les notations et donne la table de composition des permutations.

3.1.3 Permutations utilisées

$$\sigma_1 = \begin{cases} 0 \mapsto 0 \\ 1 \mapsto 1 \\ -1 \mapsto -1 \end{cases} \quad \sigma_3 = \begin{cases} 0 \mapsto 1 \\ 1 \mapsto 0 \\ -1 \mapsto -1 \end{cases} \quad \sigma_5 = \begin{cases} 0 \mapsto -1 \\ 1 \mapsto 0 \\ -1 \mapsto 1 \end{cases}$$

$$\sigma_2 = \begin{cases} 0 \mapsto 0 \\ 1 \mapsto -1 \\ -1 \mapsto 1 \end{cases} \quad \sigma_4 = \begin{cases} 0 \mapsto 1 \\ 1 \mapsto -1 \\ -1 \mapsto 0 \end{cases} \quad \sigma_6 = \begin{cases} 0 \mapsto -1 \\ 1 \mapsto 1 \\ -1 \mapsto 0 \end{cases}$$

Tableau de compositions des permutations :

o	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
σ_1	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
σ_2	σ_2	σ_1	σ_5	σ_6	σ_3	σ_4
σ_3	σ_3	σ_4	σ_1	σ_2	σ_6	σ_5
σ_4	σ_4	σ_3	σ_6	σ_5	σ_1	σ_2
σ_5	σ_5	σ_6	σ_2	σ_1	σ_4	σ_3
σ_6	σ_6	σ_5	σ_4	σ_3	σ_2	σ_1

$\forall x \in \mathcal{F}_3$, $\sigma_i \circ \sigma_j(x) = \sigma_i(\sigma_j(x))$

Tableau des inverses des permutations :

σ_1^{-1}	σ_2^{-1}	σ_3^{-1}	σ_4^{-1}	σ_5^{-1}	σ_6^{-1}
σ_1	σ_2	σ_3	σ_5	σ_4	σ_6

3.1.4 Lecture d'un TDD typé

Si l'on veut uniquement savoir la valeur de la fonction polynomiale en un seul point, la lecture d'un TDD se fait très simplement. Il suffit de descendre de la racine jusqu'à la feuille en composant les permutations. Par exemple, si l'on considère le TDD typé de la figure 6 :

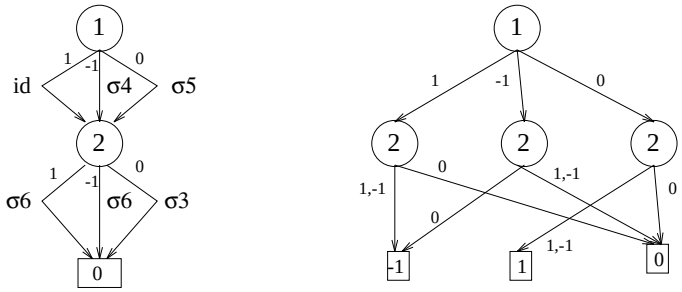

Figure 6 : TDD typé , le même TDD, mais non typé

$\mathcal{P}(-1,-1) = \sigma_4 \circ \sigma_6(0) = 0$ et$\mathcal{P}(0,1) = \sigma_5 \circ \sigma_6(0) = 1$
En revanche, si l'on veut retrouver l'expression d'un polynôme à partir d'un TDD typé, on reprend la même méthode que celle utilisée pour les TDD normaux, en utilisant les formules 1,2 et 3. Soit en exemple, le graphe de la figure 6

Le sommet étiqueté par X_2 représente:

$$p_1 = \sigma_6(0).e_2^1 + \sigma_6(0)e_2^2 + \sigma_3(0).e_2^3 = -e_2^1 - e_2^2 + e_2^3$$

et le sommet étiqueté par X_1 représente :

$$P = e_1^1(-e_2^1 - e_2^2 + e_2^3) + e_1^2(\sigma_4(-1).e_2^1 + \sigma_4(-1)e_2^2 + \sigma_4(1)e_2^3) + e_1^3(\sigma_3(-1).e_2^1 + \sigma_5(-1)e_2^2 + \sigma_5(1)e_2^3)$$

$$P = X_1 + X_2^2$$

Remarque 7 *Par la suite, par soucis de ne pas surcharger les figures, la branche de gauche d'un noeud représentera la valeur prise par la fonction polynomiale en 1, la branche du milieu la valeur prise par la fonction polynomiale en -1 et enfin la branche de droite représentera la valeur prise en 0.*

3.2 Équivalence entre nœuds

3.2.1 Définition

Définition 6 *On définit une relation d'équivalence entre 2 nœuds (i) et (j) :*
$(i) \approx (j)$ si et seulement si il existe une permutation σ telle que $\sigma(f_1) = f_2$, où f_1 (respectivement f_2) est le polynôme obtenu à partir du TDD initial en considérant (i) (respectivement (j)) comme racine.

On vérifie facilement qu'il s'agit d'une relation d'équivalence.

Remarque 8 *On remarque que $(i) \approx (j)$ implique forcément $var(i) = var(j)$*

3.2.2 Propriété fondamentale

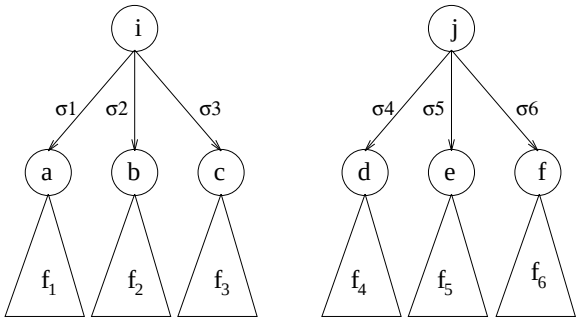

Figure 7 :

Propriété 2 *Dans le graphe de la figure 7 , pour que $(i) \approx (j)$ il faut et il suffit que :*

- $(a) \approx (d)$
  $(b) \approx (e)$
  $(c) \approx (f)$
- *et qu'il existe une permutation α telle que :*
  - *Pour tout fils k de i équivalent à une feuille, $\alpha \circ \alpha_k(c_k) = \alpha_{x+3}(c_k^{'})$ avec c_k et $c_k^{'}$ les valeurs prises respectivements par f_k et f_{k+3}*
  - *Pour tout autre fils k de i, $\alpha = \alpha_{k+3} \circ \alpha_k^{-1}$*

3.2.3 Démonstration

Nous allons seulement démontrer l'équivalence dans le sens $\Rightarrow$, l'autre sens étant évident.

- 1^{er} cas :*(i) et (j) sont des nœuds non terminaux.*
  D'après la remarque 8, (i) et (j) représentent la même variable, que nous appelerons X_g.
  On a donc, d'après la décomposition de Shannon généralisée aux TDD typés :

$$f_i = e_g^1.\alpha_1(f_a) + e_g^2.\alpha_2(f_b) + e_g^3.\alpha_3(f_c)$$

$$f_j = e_g^1.\alpha_4(f_d) + e_g^2.\alpha_5(f_e) + e_g^3.\alpha_6(f_f)$$

Comme $(i) \approx (j)$, il existe par définition une permutation $\alpha \in \{\sigma_1 \ldots \sigma_6\}$ telle que $\alpha(f_i) = f_j$.
De plus,

1. $f_i/_{X_g=1} = f_i(1, X_{g+1}, ..., X_n) = \alpha_1(f_1(X_{g+1}, ..., X_n))$
2. $f_i/_{X_g=-1} = f_i(-1, X_{g+1}, ..., X_n) = \alpha_2(f_2(X_{g+1}, ..., X_n))$
3. $f_i/_{X_g=0} = f_i(0, X_{g+1}, ..., X_n) = \alpha_3(f_3(X_{g+1}, ..., X_n))$

de la même manière :

1. $f_j/_{X_g=1} = f_j(1, X_{g+1}, ..., X_n) = \alpha_4(f_4(X_{g+1}, ..., X_n))$
2. $f_j/_{X_g=-1} = f_J(-1, X_{g+1}, ..., X_n) = \alpha_5(f_5(X_{g+1}, ..., X_n))$
3. $f_j/_{X_g=0} = f_J(0, X_{g+1}, ..., X_n) = \alpha_6(f_6(X_{g+1}, ..., X_n))$

donc

$$\alpha(f_i) = f_j \Leftrightarrow \left\{ \begin{array}{l} \alpha(f_i/_{X_g=1}) = f_j/_{X_g=1} = \alpha_4(f_4) \\ \alpha(f_i/_{X_g=-1}) = f_j/_{X_g=-1} = \alpha_5(f_5) \\ \alpha(f_i/_{X_g=0}) = f_j/_{X_g=0} = \alpha_6(f_6) \end{array} \right. \Leftrightarrow \left\{ \begin{array}{l} \alpha(\alpha_1(f_1)) = \alpha_4(f_4) \\ \alpha(\alpha_2(f_2)) = \alpha_5(f_5) \\ \alpha(\alpha_3(f_3)) = \alpha_6(f_6) \end{array} \right. \quad (12)$$

On en déduit donc que :

- $(a) \approx (d)$
- $(b) \approx (e)$
- $(c) \approx (f)$

Il se présente alors différents cas possibles :

1. Si les 3 fils de (i) et (j) ne sont pas équivalents à des feuilles. On déduit de la formule (12) que :
$$\alpha = \alpha_4 \circ \alpha_1^{-1} = \alpha_5 \circ \alpha_2^{-1} = \alpha_6 \circ \alpha_3^{-1}$$
2. Si un des fils de (i) est équivalent à une feuille (sans perte de généralité on peut considérer qu'il s'agit du nœud a).
Alors on déduit de la formule (12) : $\alpha = \alpha_5 \circ \alpha_2^{-1} = \alpha_6 \circ \alpha_3^{-1}$
Il faut de plus $\alpha \circ \alpha_1(c) = \alpha_4(c')$ avec c et c' les valeurs prises par f_1 et par f_4.
3. Si deux des fils de (i) sont équivalents à des feuilles (sans perte de généralité on peut considérer qu'il s'agit des nœuds a et b).
Alors il faut que $\alpha = \alpha_6 \circ \alpha_3^{-1}$, d'après 12
Il faut de plus que $\left\{ \begin{array}{l} \alpha \circ \alpha_1(c_1) = \alpha_4(c'_1) \\ \alpha \circ \alpha_2(c_2) = \alpha_5(c'_2) \end{array} \right.$
où c_1, c'_1, c_2, c'_2 sont définies de la même manière que c et c'.
4. Si les 3 fils de (i) et (j) sont équivalents à des feuilles alors il faut $\left\{ \begin{array}{l} \alpha \circ \alpha_1(c_1) = \alpha_4(c'_1) \\ \alpha \circ \alpha_2(c_2) = \alpha_5(c'_2) \\ \alpha \circ \alpha_3(c_3) = \alpha_6(c'_3) \end{array} \right.$
où c_1, c'_1, c_2, c'_2, c_3, c'_3 sont définie s de la même manière que c et c'.

- 2^{eme} cas :*(i) et (j) dont des nœuds terminaux.*
Il est alors évident que $(i) \approx (j)$. Ils seront représentés par la classe 0.

3.2.4 Algorithme de réduction sur des TDD typés

L'algorithme qui va suivre est inspiré de l'algorithme de Bryant sur les BDD pour la réduction [2], ainsi que du typage des BDD réalisé par Olivier Coudert [3]. Il fait bien entendu largement appel à la propriété 2.
Pour représenter un nœud, on utilise une structure déclaré comme suit :

typesommet
$gauche, droite, milieu$: *sommet*
$perm_1, perm_2, perm_3$: *permutation*
var : $[1..n+1]$
val : $(1, -1, 0, X)$
classe
fin

On représente de la même manière un nœud terminal et un nœud non terminal, mais les valeurs des champs pour un nœud v dépendent du type de ce nœud. L'algorithme de réduction s'effectue en deux phases :

1. Une phase de mise à jour des classes d'équivalence entre les différents nœuds du graphe ($init - classe$).
2. Une phase de réduction proprement dite du graphe ($reducetype$) durant laquelle le graphe réduit va être reconstruit à partir des informations récoltées par la fonction $init - classe$.

La première fonction est une fonction récursive, qui part des feuilles pour remonter jusqu'à la racine. Durant ce parcours, on attribue à chaque sommet sa classe d'équivalence ; on met le résultat dans le champ *classe*. C'est à dire, pour chaque sommet v du graphe est assignée une valeur $classe(v)$ telle que pour deux sommets u et v , $classe(u) = classe(v)$ si et seulement si les deux fonctions polynomiales qui sont représentées par u et v sont équivalentes à une permutation près .
L'algorithme qui suit explique comment est effectué ce classement.

```
fonction init − classe(vliste : tableau) : tableau
début
    nextclasse = −1;
    pour i de n+1 jusqu'à 1 pas -1 faire
        nextclasse := nextclasse + 1 ;
        pour chaque élement u de vlistes[i] faire
            si i = n + 1 alors faire
                u.classe = 0 ;
                pour chaque prédécesseur v de u faire
                    si u = 1 alors affecterperm(v, σ3);
                    si u = −1 alors affecterperm(v, σ6);
                    si u = 0 alors affecterperm(v, σ1);
                fpour
            sinon faire
                    affecter − classe(i, nextclasse, vliste)
            fsi
        fpour;
    fpour;
    retourne (vliste);
fin;
```

L'agorithme de la fonction *reducetype* effectue la véritable réduction du graphe. Il repose sur un certain nombre de principes de construction.
En premier lieu, les nœuds sont collectés dans des listes suivant les numéros de variables. On met ensuite à jour les classes d'équivalences entre les différents sommets, puis on traite alors chaque liste en partant de celle contenant les terminaux jusqu'à celle contenant la racine. Dans chaque liste, on prend le premier élément de chaque classe d'équivalence.Tous ces noeuds ainsi réunis représentent alors les nœuds du graphe réduit. On effectue finalement les nouveaux branchements et on affecte les bonnes permutations aux nœuds

pères. Ces nœuds sélectionnés vont former le nouvel arbre.

```
fonction reductype(v : noeud) : noeud;
var
    graphtransition :tableau[1..[G]] de noeuds ;
    vliste:tableau[1..n+1] de listes ;
    graphe :tableau de noeuds
debut
    classement-suivant-les-classes(v,vliste)      vliste = init − classe(vliste) ;
    nextid := 0 ;
    pour i de n + 1 jusqu'à 1 pas -1 faire
        pour u appartenant à vliste[i] faire
        u.classe=classe-prec
            si (u.classe <> classe − prec)
                classe − prec := classe − prec + 1; u.classe := classe − prec ;
                graphtransition[classe − prec] := u ;
                u.gauche := graphtransition[u.gauche.classe] ;
                u.droite := graphtransition[u.droite.classe] ;
                u.milieu := graphtransition[u.milieu.classe] ;
                compose − perm(u, perm) ;
            fsi
        fpour
    fpour
    retourner(graphtransition)
fin
```

3.2.5 Exemple de réduction sur un TDD typé

Ce paragraphe donne un exemple sur la manière dont est réduit un TDD.
Considérons le TDD de la figure 8, où nombre de sous-graphes sont égaux à une permutation près (Il faut donc les regrouper), ce qui nous permet d'obtenir, une fois la réduction effectuée, le graphe de la figure 9.

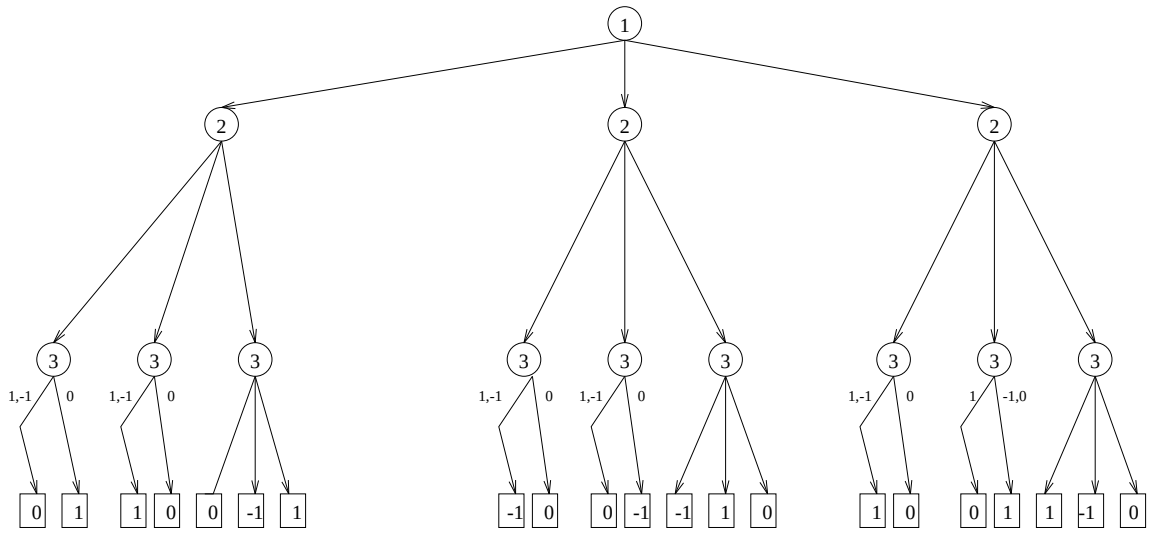

Figure 8 : TDD initial

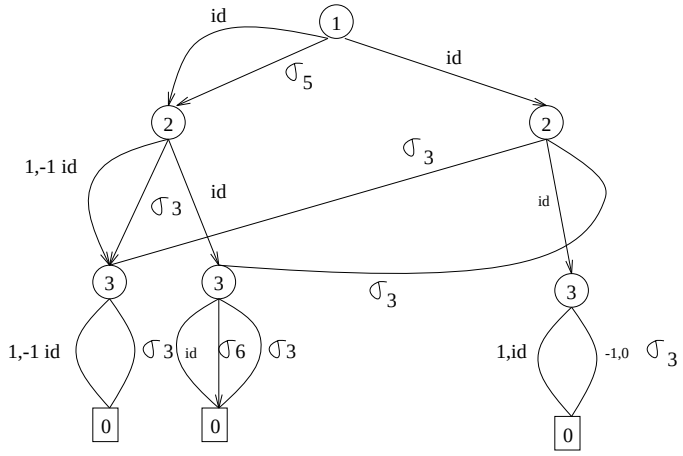

Figure 9 : TDD typé, réduit

Tout d'abord toutes les feuilles étant équivalentes, elles sont interprétées comme étant équivalentes à la classe 0. Puis, on commence à comparer les nœuds du niveau immédiatement supérieur. On s'aperçoit que parmi tous les nœuds ayant pour variable x_3,il existe 3 nœuds qui n'appartiennent pas à la même classe d'équivalence. On effectue donc le "regroupement" autour de ces trois représentants, et on recommence avec les niveaux supérieurs jusqu' à la racine.

3.3 Canonicite du TDD typé

Afin de faciliter la comparaison entre deux fonctions polynomiales, il est souhaitable d'avoir une représentation canonique de chaque fonction. En effet les fonctions polynomiales sur lesquelles nous travaillons peuvent avoir jusqu'à plusieurs centaines de variables différentes interdépendantes, ce qui conduit à des TDD possédant plusieurs milliers de sommets. L'égalité doit donc pouvoir se tester de manière rapide par une simple comparaison entre les deux graphes.
Hors, on a vu qu'il existait certains cas où la possibilité de choisir entre plusieurs permutations pouvait conduire à plusieurs TDD différents pour une même fonction polynomiale.
Une autre raison justifie l'impératif de la canonicité sur les TDD typés. En effet, la méthode retenue pour construire les TDD à partir des fonctions polynomiales utilise les opérations sur les graphes. Par exemple, pour engendrer le TDD de $p = X_1^2 + X_2$, on construit d'abord le graphe typé de X_1 que l'on réduit, puis par composition, on construit le graphe de X_1^2, que l'on réduit. On crée ensuite le graphe de X_2 puis enfin par addition on construit le graphe de $X_1^2 + X_2$. On voit bien qu'une condition nécessaire pour la réalisation d'un tel processus est la commutativité par rapport à l'addition et à la multiplication (on imagine mal que le TDD de $X_1^2 + X_2$ soit différent de celui de $X_2 + X_1^2$). Ces conditions imposent la canonicité des graphes. L'obtention de la canonicité implique, au niveau des feuilles, de faire des choix entre deux permutations différentes. En effet, on a toujours deux permutations qui envoient 0 sur -1 et 0 sur 1. En choisissant toujours la même permutation, on résoud ce problème. Par la suite lorsque l'on remonte dans le graphe pour le typage, les permutations sont alors fixées.
Cependant, cette condition nécessaire pour la canonicité, n'est cependant pas encore suffisante comme le montre l'exemple de la figure 10 dans lequel on a deux TDD typés qui représentent la même fonction polynomiale.
En fait, nous allons montrer que l'on peut obtenir la canonicité en forçant la permutation gauche à l'identité (σ_1). Pour la démonstration de la propriété qui va suivre nous allons nous appuyer sur le résultat suivant: si $f = e_i^1.\alpha_1(f_1) + e_i^2.\alpha_2(f_2) + e_i^3.\alpha_3(f_3)$ alors, on peut faire remonter la permutation α_1 comme le montre la figure 11. On obtient alors $\alpha_1^{-1}(f) = e_i^1.f_1 + e_i^2.\alpha_1^{-1} \circ \alpha_2(f_2) + e_i^3.\alpha_1^{-1} \circ \alpha_3(f_3)$.

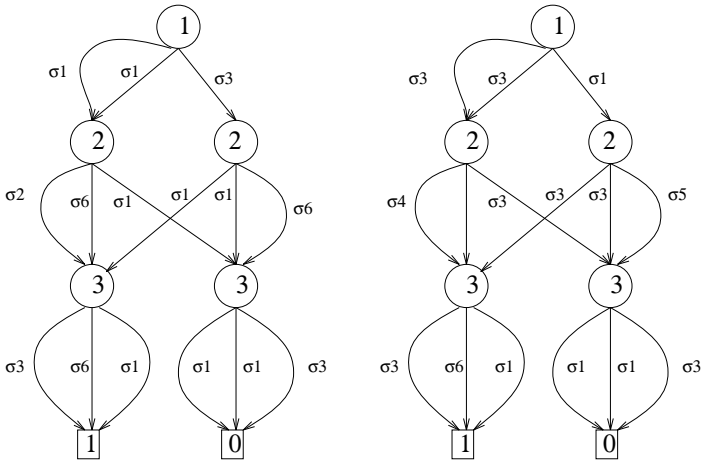

Figure 10 : Représentations différentes d'une même fonction polynomiale

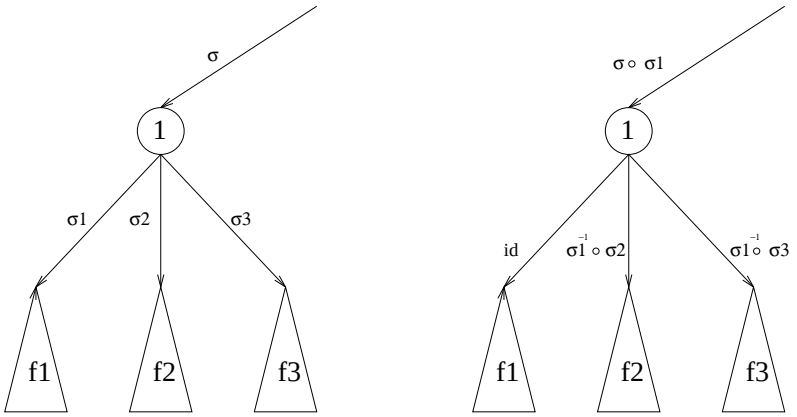

Figure 11 :

On part donc du principe que dans tous les nœuds différents stockés en mémoire toutes les permutations gauches sont égales à l'identité.

Remarque 9 *Étant donné que toutes les permutations gauches seront égales à l'*identité*, il n'est plus nécessaire de stocker cette information en mémoire. Un nœud sera donc représenté en mémoire avec les informations suivantes :* $(x_i,\ n_1,\ n_2,\ n_3,\ ,\alpha_2,\ \alpha_3)$

Propriété 3 *Soient deux nœuds* (i) *et* (j) *représentés respectivement par :* $(x_i,\ n_1,\ n_2,\ n_3,\ ,\alpha_2,\ \alpha_3)$ *et* $(x_i,\ n'_1,\ n'_2,\ n'_3,\ ,\alpha'_2,\ \alpha'_3)$*. Ces deux nœuds sont équivalents si et seulement si* $n_1 = n'_1$, $n_2 = n'_2$,$n_3 = n'_3$ *et si* $\alpha_2 = \alpha'_2$ *et* $\alpha_3 = \alpha'_3$

La démonstration découle de la propriété fondamentale en posant $\alpha_1 = \alpha'_1 =$identité.

Remarque 10 *Une fonction polynomiale* f *pourra donc être codée en mémoire par une permutation* α *et un nœud* (i) *où* α *représente la dernière permutation remontée et le nœud* (i) *le nœud correspondant au représentant de la classe choisie telle que* $f = \alpha(\mathcal{P}(i))$.

On obtient de cette manière une représentation canonique des TDD typés.

La fonction de réduction, qui découle de la nouvelle propriété, est au cœur de la représentation des fonctions polynomiales dans SIGALI. En effet, à chaque appel d'une fonction effectuant une opération sur un ou deux polynômes, celle-ci rend *a priori* un TDD qui n'est pas réduit. Il convient alors d'appeler la fonction de réduction afin de retrouver un TDD réduit. Le paragraphe suivant présente les différente opérations effectuées sur les TDD typés en adaptant ce qui avait été fait sur les TDD ordinaires [4].

3.4 Opérations sur les TDD typés

3.4.1 Principe des opérations élémentaires

On appelle opérations élémentaires, les opérations dites classiques sur les polynômes comme l'égalité, la somme, le produit ou encore la différence. Considèrons deux polynômes f et g dans A et ψ une opération élémentaire comme celle définie plus haut. On pose :

$$h = \psi(f,g) \Leftrightarrow h(x) = \psi(f(x), g(x))$$

On a donc $h(x)/_{X_i=a} = h(X_1, .., X_{i-1}, a, X_{i+1}, .., X_n)$
On en déduit :

$$\begin{aligned} h(x)/_{X_i=1} &= \psi(f(x)/_{X_i=1}, g(x)/_{X_i=1}) \\ h(x)/_{X_i=-1} &= \psi(f(x)/_{X_i=-1}, g(x)/_{X_i=-1}) \\ h(x)/_{X_i=0} &= \psi(f(x)/_{X_i=0}, g(x)/_{X_i=0}) \end{aligned}$$

De plus, on a :

$$\begin{aligned} f &= e_i^1.\alpha_1(f_1) + e_i^2.\theta_1(f_2) + e_i^3.\lambda_1(f_3) \\ g &= e_i^1.\alpha_2(g_1) + e_i^2.\theta_2(g_2) + e_i^3.\lambda_2(g_3) \end{aligned}$$

et finalement :

$$h = \psi(f,g) = e_i^1.\psi(\alpha_1(f_1), \alpha_2(g_1)) + e_i^2.\psi(\theta_1(f_2), \theta_2(g_2)) + e_i^3.\psi(\lambda_1(f_3), \lambda_2(g_3))$$

avec $\alpha_1(f_1) = e_i^1.\alpha_1 \circ \alpha_{11}(f_{11}) + e_i^2.\alpha_1 \circ \alpha_{12}(f_{12}) + e_i^3.\alpha_1 \circ \alpha_{13}(f_{13})$
On obtient les autres de la même manière par récurrence sur l'indice des variables.

3.4.2 Algorithme effectuant une opération élémentaire entre deux polynômes

Remarque 11 *Afin d'éviter une trop grande redondance des calculs intermédiaires, on stocke dans une table (implémentée en utilisant une fonction de hash-coding) tous les résultats intermédiaires qui ont déjà été calculés.*

$operation(a, b, \psi, perm_1, perm_2) =$
début
  si $present(a, b, perm_1, perm_2)$
    $y := memoire(a, b, perm_1, perm_2)$
  sinon
    si $a.var = n + 1$ et $b.var = n + 1$
    alors $nouvterm(\phi(a.val, b.val))$
    sinon
      si $a.var < b.var$
        $nouv(a.var, operation(a.perm_1(a.gauche), b, \psi, a.perm_1, perm_2),$
                $operation(a.perm_2(a.milieu), b, \psi a.perm_2, perm_2),$
                $operation(a.perm_3(a.droite), b, \psi, a.perm_3, perm_2))$
      sinon si $a.indice > b.indice$
          $nouv(b.var, operation(a, b.perm_1(b.gauche), \psi, perm_1, b.perm_1),$
                  $operation(a, b.perm_2(b.milieu), \psi, perm_1, b.perm_2),$
                  $operation(a, b.perm_3(b.droite), \psi, perm_1, b.perm_3))$
      sinon
      $nouv(a.var, operation(a.perm_1(a.gauche), b.perm_1(b.gauche), \psi, a.perm_1, b.perm_1),$
          $operation(a.perm_2(a.milieu), b.perm_2(b.milieu), \psi, a.perm_2, b.perm_2),$
          $operation(a.perm_3(a.droite), b.perm_3(b.droite), \psi, a.perm_3, b.perm_3))$
        fsi
      fsi
      $stocke(a, b, y, perm_1, perm_2)$
    fsi
    $resultat := y$
fin
On obtient alors le TDD réduit en appliquant la fonction *operat*
$operat(G_1, G_2, \psi)$:
  $raz()$
  $resultat := reduire(operation(G_1, G_2, \psi, id, id))$
fin
avec :

- La fonction $nouvterm(a)$ renvoie un nouveau sommet terminal de valeur a et indexé par $n + 1$.
- La fonction $nouv(i, a_1, a_2, a_3)$, où a_1, a_2 et a_3 sont des sommets et i un indice compris entre 1 et n, crée un nouveau sommet b tel que :

$$\begin{aligned} b.var &= i \\ b.gauche &= a_1 \\ b.milieu &= a_2 \\ b.droite &= a_3 \end{aligned}$$

  avec $a_j.var > i$ pour garder la connexité du graphe.
- $raz()$ est une fonction d'effacement de toutes les données inutiles stockées en mémoire.

- $stocke(a, b, y, perm_1, perm_2)$ enregistre le résultat.
- $present(a, b, perm_1, perm_2)$ teste la présence d'un enregistrement (si le calcul de $\psi(perm_1(a), perm_2(b))$ a déjà été effectué).
- $memoire(a, b, perm_1, perm_2)$ lit un enregistrement qui a déjà été effectué.

Remarque 12 *perm(a) signifie que l'on compose les trois permutations du nœud a par la permutation perm.*

Par exemple, on cherche à faire dans la figure 12, l'addition entre deux TDD quelconques.

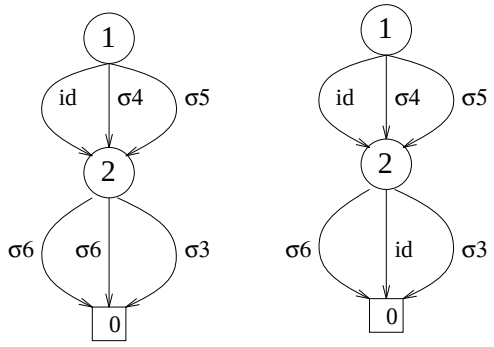

Figure 12 : addition entre deux fonctions polynomiales

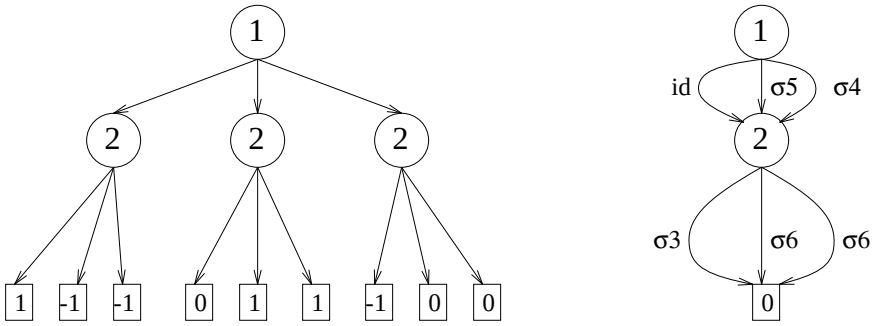

Figure 13 : résultat de l'addition + résultat typé et réduit

3.4.3 Développement d'un TDD typé

Une façon rapide de vérifier que les programmes servant à la réduction et aux opérations sur les TDD sont corrects est de développer le TDD obtenu sous une forme classique polynomiale afin de le comparer avec le polynôme initial.

Principe : soit f une fonction polynomiale, on peut décomposer f sous la forme suivante :

$$f = e_1^1.\sigma(f_1) + e_1^2.\theta(f_2) + e_1^3.\lambda(f_3)$$

on posera par la suite

$$\begin{aligned} u &= \sigma(f_1) \\ v &= \theta(f_2) \\ w &= \lambda(f_3) \end{aligned}$$

De plus, comme on travaille dans A=$\mathcal{F}_3[X]/ < X^3 - X >$, la fonction polynomiale f peut se mettre sous la forme suivante : $f = a.X_1^2 + b.X_1 + c$,où a, b et c sont des fonctions polynomiales de A_1. On en déduit donc que :

$$\begin{aligned} a &= -u - v - w \\ b &= -u + v \\ c &= w \end{aligned}$$

On calcule donc a, b et c et on recommence la décomposition sur ces trois fonctions polynomiales suivant la variable X_2.

Idée de l'algorithme: On part du TDD F qui représente f, on en extrait les sous graphes qui représentent $\sigma(f_1)$, $\theta(f_2)$ et $\lambda(f_3)$. On calcule les TDD représentant a, b et c et on recommence ainsi jusqu'à arriver aux feuilles.

3.4.4 Substitutions

Une substitution c'est le remplacement de certaines variables par des polynômes. Pour des raisons d'efficacité de mise en œuvre, trois sortes de substitutions ont été réalisées :

- la substitution de variables par des constantes : *l' évaluation*,
- la substitution de variables par d'autres variables : *le renommage*,
- la substitution générale : les variables sont remplacées par des polynômes quelconques.

Cette dernière peut être utilisée à la place des deux autres, mais elle est nettement plus coûteuse en temps de calcul. Pour cette raison, il est recommandé d'utiliser quand, on le peut, les deux formes spécialisées.

Evaluation: La substitution des variables d'un polynôme par des constantes est effectuée par la fonction **eval** : Par exemple **eval(p, [a,b,c], [0,1,-1]);** remplace dans **p**, les variables **a**, **b** et **c** par, respectivement, 0, 1 et -1. Les variables de **p** autres que **a**, **b** ou **c** sont inchangées.

Plus généralement, si **lvar** est une liste de variables et **lconst** une liste de constantes de même longueur que **lvar**, **eval(p, lvar, lconst)** substitue dans **p** la $i^{ème}$ variable de **lvar** par le $i^{ème}$ élément de **lconst**.

Renommage: Un renommage est une substitution de variables par d'autres variables. L'algorithme de renommage ne modifie pas la structure d'un T.D.D., il remplace simplement l' étiquette $var(x)$ de chacun des sommets non terminaux x. Pour que le T.D.D. résultant de cette opération soit cohérent, la substitution doit satisfaire la condition suivante : *pour tout couple de variables X_i, X_j tel que $X_i < X_j$, si X_i est renommée $X_{i'}$ et X_j est renommée $X_{j'}$ alors on doit avoir $X_{i'} < X_{j'}$.*

Par exemple, pour l'ordre $a < b < c < d$, la substitution de a par d et de b par c n'est pas un renommage valide. On peut, en revanche, remplacer a par c et b par d :

Si le changement de variables qu'on veut effectuer ne satisfait pas la contrainte, il faut impérativement utiliser la fonction de substitution générale.

Substitution: La fonction de substitution générale est notée **subst**.

Les règles sont toujours les mêmes : dans **subst(p, lvar, lpoly),**

- **lvar** est une liste de variables, **lpoly** est une liste de polynômes, les deux listes doivent être de même longueur.
- la $i^{ème}$ variable de **lvar** est remplacée par le $i^{ème}$ polynôme de **lpoly**

3.5 Codage interne des TDD typés

De manière interne un graphe de décision ternaire de taille n est codé par un tableau de n cellules. Chaque cellule représente un des nœuds du graphe ; chaque nœud est donc identifié par son indice dans le tableau. Les cellules sont des structures possédant 7 champs : le numéro de variable, l'indice dans le tableau des trois fils ainsi que les trois permutations avec lesquelles les fils doivent être interprétés.

Remarque 13 *Les nœuds sont triés suivant l'ordre décroissant des numéros de variables, et, à numéro identique, suivant l'ordre lexicographique sur les triplets constitués des indices des trois fils.*

Pour mettre en œuvre la réduction, on construit une structure intermédiaire qui représente le TDD non réduit. Les nœuds ne sont pas placés dans un tableau comme précédemment, ils sont identifiés non plus comme un indice mais par un pointeur. Les nœuds possédant le même numéro de variables sont placés au fur et à mesure de leur création dans des listes.

Remarque 14 *On ne construit jamais deux terminaux de même valeur.*
La fonction $nouv(i, a, b, c, \alpha, \beta, \gamma)$ *ne crée pas de nouveau nœud si* a, b *et* c *sont égaux et si* $\alpha = \beta = \gamma$

En fin de construction, on applique l'algorithme de réduction de Bryant. On connait alors le nombre de sommets du graphe réduit. Il ne reste plus qu'à allouer une table de cellules de la bonne taille et y recopier le TDD réduit.

4 Résultat : Comparaison entre les TDD et les TDD typés

Afin de comparer les TDD et les TDD typés, nous allons utiliser SIGALI, langage de calcul formel interactif spécialié dans les calculs algébriques sur l'anneau $\mathcal{F}_3[X]/ < X^3 - X >$, les fonctions polynomiales y étant codées sous forme de TDD.
Il convient pour bien comprendre de présenter rapidement le traitement des fonctions polynomiales pour SIGALI [5]. Nous verrons ensuite une comparaison entre l'ancienne technique de codage et la nouvelle, qui mettra en évidence les améliorations apportées dans certains cas, mais aussi ses limites.

4.1 Présentation de SIGALI

Lorsque l'on utilise SIGALI pour les calculs polynomiaux, on dispose au départ d'un certain nombre d'opérateurs permettant les calculs simples sur les polynômes.

- les opérateurs algébriques : +,-,*,^
- les opérateurs propres à SIGNAL : **default**, **when**, **event**
- les opérateurs booléens étendus :**and**, **not**, **or**

Ces opérateurs sont traduits par SIGALI dans $\mathbf{Z}/_{3\mathbf{Z}}$ de la manière suivante :

a `default` b	$a + (1 - a^2)b$
a `when` b	$a(-b - b^2)$
`when` a	$(-a - a^2)$
`event` a	a^2
a `and` b	$ab(ab - a - b - 1)$
a `or` b	$ab(1 - a - b - ab)$
`not` a	$-a$

Les résultats de ces évaluations sont affichés sous forme de TDD typés et réduits.

4.2 Affichage des TDD

Soit un polynôme P et $X_1, X_2, ..., X_n$ les variables de ce polynôme rangées dans cet ordre. Soit N le nombre de sommets du TTD G associé à ce polynôme. SIGALI affichera le polynôme de la manière suivante :

- A chaque sommet de G, SIGALI assigne un numéro compris entre 1 et $N-1$. La racine a pour numéro $N-1$; puis on trouve dans l'ordre décroissant les variables X_i jusqu'à la racine .

- Pour chaque non terminal, SIGALI affiche :

$$n \ : (X_i \ perm1 \ perm2 \ perm3 \ f1 \ f2 \ f3)$$

  avec n le numéro du sommet, X_i la variable, f_1, f_2, f_3 les numéros des trois sommets fils de X_i et $perm_1$, $perm_2$, $perm_3$ les 3 permutations avec lesquelles ils doivent être interprétés.
  Pour le sommet terminal on aura : n : 0 , où 0 représente la valeur du terminal.

Remarque 15 *l'ancienne version affiche, pour les TDD non typés :*

$$n \ : (X_i \ f1 \ f2 \ f3)$$

Par exemple considérons le polynôme
$P : a \ default(\ (b * c) \ when \ ((not \ d) \ default \ e^2)))$ avec $a < b < c < d < e$
P sera affiché de la manière suivante :

```
> declare(a,b,c,d,e);

>P: a default ((b*c) when ((not d) default e^2)));
P

>P;
  7 : (a pg 2 pm 5 pd 0 ng 0 nm 0 nd 6)
  6 : (b pg 0 pm 0 pd 0 ng 5 nm 4 nd 1)
  5 : (c pg 0 pm 0 pd 0 ng 3 nm 2 nd 1)
  4 : (c pg 0 pm 0 pd 0 ng 2 nm 3 nd 1)
  3 : (d pg 0 pm 2 pd 0 ng 1 nm 0 nd 1)
  2 : (d pg 0 pm 5 pd 0 ng 1 nm 0 nd 1)
  1 : (e pg 2 pm 2 pd 0 ng 0 nm 0 nd 0)
  0 :  0
```

4.3 Exemple de fonctions de SIGALI

Le développement

```
> declare(a,b,c,d);

> p: a when b;
p

> developpe(p);
- a * (b ^ 2 + b)
```

Substitution

```
> declare(a,b,c,d);

> q:a+b+c+d;
q

> eval(q,[a],[0]);
  3 : (b pg 4 pm 0 pd 3 ng 2 nm 2 nd 2)
  2 : (c pg 0 pm 3 pd 4 ng 1 nm 1 nd 1)
  1 : (d pg 2 pm 5 pd 0 ng 0 nm 0 nd 0)
  0 :  0

> developpe( eval(q,[a],[0]));
b + c + d
```

4.4 Comparaison entre les TDD et le TDD typés

Il est difficile de dire à *priori* par combien le typage des TDD divise le nombre de sommets d'un graphe par rapport à la version non typée. En effet certains polynômes sont déjà réduits quasiment au maximum lors d'une réduction sans typage ; il est alors évident que le typage de ceux ci ne permettra pas de gagner beaucoup de nœuds.
Dans le polynôme P vu précédemment, on ne gagne par exemple que deux nœuds entre les deux versions (ce qui représente le minimum possible de gains) (cf TDD numéro 1 de l'annexe 7 pour la version typée).

En revanche, d'autres polynômes, plus réguliers quant à la dépendance des variables les unes par rapport aux autres, donnent une réduction des nœuds significative entre les deux versions de réduction des TDD.
Par exemple, considérons le polynôme $P := a + b + c + d + e + f + g + h + i + j$ Par la simple réduction on obtient un graphe de 31 nœuds (cf TDD numéro 2 annexe 7). Alors que par la méthode de réduction par typage, on obtient un graphe de 11 nœuds ce qui représente le gain d'un facteur 3 par rapport à la version précédente.(cf TDD numéro 3 de l'annexe 7). On obtient, dans cet exemple, un nombre de nœuds minimal pour un graphe représentant un polynôme en 10 variables interdépendantes les unes des autres. D'autres essais sur des polynômes pris au hasard font apparaitre une réduction variant entre 30% et 50%. Il semblerait donc que, grâce aux permutations, on puisse obtenir de bonnes réductions sur les TDD.
Cependant l'utilisation de TDD typés sur des fonctions polynomiales provenant de véritables programmes SIGNAL s'est avérée décevante. L'exemple suivant illustre cette situation.

- TDD typé :

```
> chrono(true);

Utilisateur : 0.00, Systeme : 0.00, Clock = 0.00

>  read("VTMOUSE.z3z");

Utilisateur : 0.63, Systeme : 0.23, Clock = 0.85

>  Q:gen(contraintes);
Q
```

```
Utilisateur: 0.37, Systeme: 0.02, Clock = 0.38

> taille(Q);
280
```

- TDD non typé :

```
> chrono(true);

Utilisateur: 0.00, Systeme: 0.00, Clock = 0.00

>  read("VTMOUSE.z3z");

Utilisateur: 0.32, Systeme: 0.21, Clock = 0.52

> Q:gen(contraintes);
Q

Utilisateur: 0.06, Systeme: 0.02, Clock = 0.07

>  taille(Q);
282

Utilisateur: 0.00, Systeme: 0.00, Clock = 0.00
```

On constate que le rapport de réduction entre les deux méthodes utilisées est minimal en ce qui concerne le calcul du polynôme des contraintes et que, de plus, le temps de calcul est plus long, l'introduction des permutations augmentant le nombre de calculs. De même, si l'on teste la taille des TDD calculés lors de la lecture du fichier, on remarque qu'à de rares exceptions près, rencontrées sur les petits polynômes (facteur de réduction variant entre 1,5 et 2), les gros TDD ne sont pas plus réduits qu'avec l'ancienne méthode.
La raison principale qui tend à expliquer ces résultats semble être la faible densité des graphes avant la réduction. Par exemple, si l'on regarde plus attentivement la forme du polynôme représenté par le TDD Q (polynôme des contraintes), on remarque que les fils des différents nœuds de même niveau sont complètement différents, ce qui rend impossible une réduction efficace.
On note en particulier,en ce qui concerne les nœuds 20 et 19 qui représentent la même variable, le premier a pour fils les nœuds 16 , 12 et 0 tandis que le deuxième a respectivement pour fils les nœuds 17,15 et 0 (il en est de même pour tous les autres nœuds de même niveau), ce qui empêche toute réduction (on rappelle qu'une condition nécessaire pour que 2 nœuds soient équivalents est que le fils gauche de l'un possède le même numéro de variable que le fils gauche de l'autre (de même pour les autres fils))(cf TDD numéro 4 de l'annexe 7 pour avoir de ce TDD).

L' étude plus approfondie des polynômes intermédiaires qui permettent de calculer le polynôme Q des contraintes donne les mêmes résultats. Il convient donc de trouver une autre représentation des polynômes car avec les typage des TDD, non seulement, on ne gagne pas de place en mémoire, mais surtout, du fait de la plus grande complèxité des algorithmes, on perd du temps en calcul.
Une étude a donc été faite pour coder en mémoire de manière plus efficace les TDD (qu'ils soient typés par des permutations ou non, le choix restant à faire).

5 Perspectives

Une remarque que l'on peut faire tout de suite sur la manière dont sont codés en mémoire les TDD, est que deux polynômes égaux y sont implémentés deux fois. On perd donc une place énorme en mémoire pour peu que ces deux polynômes soient de taille importante.
L'idée est donc de créer un table contenant tous les nœuds des différents polynômes qui ont été calculés lors d'une session de SIGALI. On ne peut pas trouver dans cette table deux nœuds équivalents.
Une étude portant sur la réalisation d'une table de hachage pour le codage des BDD a déjà été réalisée par Brace, Rudell et Bryant [7] et donne des résultats plus qu'intéressants.
Nous allons montrer par la suite comment peuvent être réalisés les calculs sur les TDD typés lorsque l'on utilise une telle table.

5.1 Utilisation d'une table pour stocker les nœuds

5.1.1 Présentation de la table

On utilise comme table, implémentée à l'aide d'une fonction de hachage, une table similaire à celle utilisée précédemment pour la sauvegarde des résultats intermédiaires.
Un nœud est représenté par :

- son numéro de variable :v
- les permutations milieu et droite

  Remarque 16 *Il n'est pas nécessaire de stocker la permutation gauche, car on sait qu'elle est égale à l'*IDENTITÉ.

- les indices dans la table des trois fils h_1 , h_2 et h_3

On dispose au départ de deux fonctions :

- La fonction permettant d'aller voir si un nœud est équivalent à un nœud se trouvant déjà dans la table. Elle rend la permutation avec laquelle doit être composé le nouveau nœud pour être égal à celui se trouvant dans la table et rend le pointeur NULL sinon.
  $trouve - noeud(v, \sigma_2, \sigma_3, h_1, h_2, h_3)$
- La fonction permettant d'aller mettre en mémoire un nœud dans la table :
  $insere - noeud(v, \sigma_2, \sigma_3, h_1, h_2, h_3)$

5.1.2 Exemple

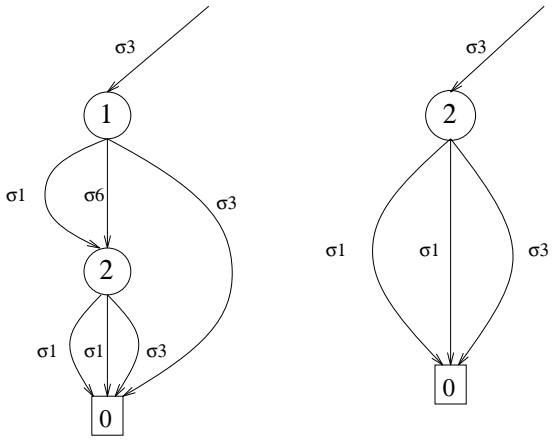

Figure 14 : opération sur les polynômes

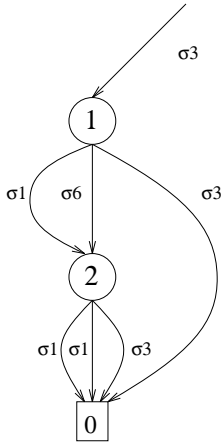

Figure 15 : résultat de la multiplication des deux polynômes

Ces trois polynômes sont représentés dans la table de la manière suivante (schéma simplifié) :

	num var	σ	ϕ	θ	h_1	h_2	h_3
0	1000	\	\	\	\	\	\
1	x_2	σ_3	σ_3	σ_1	0	0	0
2	x_1	σ_1	σ_2	σ_1	1	1	0

Les nœuds sont rangés au fur et à mesure de leur création dans la table à l'aide la fonction $insere - noeud(v, \sigma_2, \sigma_3, h_1, h_2, h_3)$, s'il n'y sont pas déjà. Ici on s'aperçoit que tous les nœuds créés lors de la multiplication entre les deux polynômes existaient déjà dans la table. Ils n'y sont donc pas remis. Dans cette exemple, il n'y a eu que 3 nœuds créés alors que normalement il y en aurait eu 9, ce qui représente un facteur d'amélioration assez important.
L'utilisation d'une table présente un autre avantage. En effet, avec l'implémentation précédente des TDD, on était obligé d'opérer, après chaque opération, une réduction du polynôme résultat (le TDD résultant d'une opération étant rarement réduit). Ici, par contre, du fait que plusieurs nœuds équivalents possèdent un seul représentant dans la table, la réduction s'effectue automatiquement lors de la création du polynôme et de ses nœuds.

5.2 Implémentation différente des opérations

5.2.1 Intérêt d'une nouvelle implémentation

L'idée de cette nouvelle implémentation est donnée dans l'article de Brace, Rudell et Bryant [7]. Tout comme l'ancienne version des opérations, elle consiste à créer une unique fonction dépendant de plusieurs paramètres, qui, lorsque que l'on fait varier ces paramètres, permet d'obtenir toutes les opérations possibles telles que WHEN, DEFAULT, EVENT, +, *, -, ...
L'intérêt d'une telle fonction réside dans l'utilisation simplifiée d'une mémoire cachée mémorisant des opérations déjà effectuées.

5.2.2 Présentation

On considère la fonction $pcase(F, G_1, G_2, G_3, \sigma, \phi, \theta)$, où F, G_1, G_2 et G_3 sont des fonctions polynomiales représentées par des TDD typés et les trois derniers paramètres des permutations dans $\mathbf{Z}/_{3\mathbf{Z}}$
on a :
$pcase(F, G_1, G_2, G_3, \sigma, \phi, \theta)(x) =$
si $F(x) = 1$ alors $\sigma(G_1(x))$
sinon
si $F(x) = -1$ alors $\phi(G_2(x))$
sinon $\theta(G_3(x))$
fin
De manière plus algébrique, on peut représenter cette fonction sous une forme polynomiale :

$$pcase(F, G_1, G_2, G_3, \sigma, \phi, \theta) = (-F - F^2).\sigma(G_1) + (F - F^2).\phi(G_2) + (1 - F^2).\theta(G_3)$$

Les différentes opérations se déduisent alors de la manière suivante :

- $$\begin{aligned} WHEN\ a &= -a - a^2 \\ &= pcase(a, 1, 0, 0, id, id, id) \end{aligned}$$

- $$\begin{aligned} EVENT\ a &= a^2 \\ &= pcase(a, 1, 1, 0, id, id, id) \end{aligned}$$

- $$\begin{aligned} a\ WHEN\ b &= a.(-b - b^2) \\ &= pcase(b, a, 0, 0, id, id, id) \end{aligned}$$

- $$\begin{aligned} a\ DEFAULT\ b &= a + (1 - a^2).b \\ &= pcase(a, -1, 1, b, id, id, id) \end{aligned}$$

- $$NOT\ a \ = \ -a$$

  C'est la seule fonction qui peut être codée différemment du fait de sa structure. Il suffit, pour coder $NOT\ a$, de composer les permutations interprétant les fils de la racine par la permutation qui envoie -1 sur 1 et 1 sur -1, ce qui est extrémement rapide.

- $$\begin{aligned} a\ AND\ b &= a.b.(a.b - a - b - 1) \\ &= pcase(a, b, NOT\ EVENT\ b, o, id, id, id) \\ &= pcase(a, b, pcase(b, -1, -1, 0, id, id, id), 0, id, id, id) \end{aligned}$$

- $$\begin{aligned} a\ OR\ b &= a.b.(1 - a - b - a.b) \\ &= pcase(a, EVENT\ b, NOT\ b, 0, id, id, id) \\ &= pcase(a, pcase(b, 1, 1, 0, id, id, id), pcase(b, 1, -1, 0, id, id, id), 0, id, id, id) \end{aligned}$$

- $$a \ + \ b \ = \ pcase(a, b, b, id, \sigma, \sigma^2)$$
  $$\text{avec } \sigma = \begin{cases} 0 \mapsto 1 \\ 1 \mapsto -1 \\ -1 \mapsto 0 \end{cases}$$

- $$a \ * \ b \ = \ pcase(a, b, NOT(b), 0)$$

Ces fonctions sont les principales fonctions utilisées par SIGALI. Elle peuvent donc toutes être codées à partir de la fonction *pcase* en choisissant les paramètres en fonction de l'opération choisie. Le problème maintenant est de trouver un algorithme suffisamment rapide pour que cette implémentation soit efficace.

5.2.3 Idée de l'algorithme

La démonstration qui va suivre est la clef de l'algorithme $pcase(F, G, H, K, \alpha, \beta, \gamma))$, où F, G, H et K sont représentés sous forme de TDD typés.
on a :
$F = e_v^1.\sigma_1(F_1) + e_v^2.\sigma_2(F_2) + e_v^3.\sigma_3(F_3)$
$G = e_v^1.\phi_1(G_1) + e_v^2.\phi_2(G_2) + e_v^3.\phi_3(G_3)$
$H = e_v^1.\theta_1(H_1 + e_v^2.\theta_2(H_2) + e_v^3.\theta_3(H_3)$
$K = e_v^1.\lambda_1(K_1 + e_v^2.\lambda_2(K_2) + e_v^3.\lambda_3(K_3)$
Considérons $Z = pcase(F, G, H, K, \alpha, \beta, \gamma)$ où v est la variable la plus petite de $F,\ G,\ H,\ K$ (au sens de l'ordonnancement des variables).
D'après Shannon, on peut écrire z sous la forme suivante :

$$Z \ = \ (-v - v^2).Z(v = 1) + (v - v^2).Z(v = -1) + (1 - v^2).Z(v = 0)$$

$$
\begin{array}{lcl}
Z & = & pcase(v, pcase(\sigma_1(F_1), \alpha(\phi_1(G_1)), \beta(\theta_1(H_1)), \gamma(\lambda_1(K_1))), id, id, id) \\
 & , & (pcase(\sigma_2(F_2), \alpha(\phi_2(G_2)), \beta(\theta_2(H_2)), \gamma(\lambda_2(K_2))), id, id, id) \\
 & , & pcase(\sigma_3(F_3), \alpha(\phi_3(G_3)), \beta(\theta_3(H_3)), \gamma(\lambda_3(K_3))), id, id, id), id, id, id)
\end{array}
$$

Les conditions d'arrêt sont les suivantes :
$pcase(1, G_1, G_2, G_3, \alpha, \beta, \gamma) = \alpha(G_1)$
$pcase(-1, G_1, G_2, G_3, \alpha, \beta, \gamma) = \beta(G_2)$
$pcase(0, G_1, G_2, G_3, \alpha, \beta, \gamma) = \gamma(G_3)$
$pcase(F, G, G, G, \alpha, \alpha, \alpha) = \alpha(G)$
Ce résultat nous permet donc d'écrire l'algorithme suivant :
$pcase(F, G, H, K, \alpha, \beta, \gamma)$
if $cas - terminal$ alors retourner $result$
sinon
si $present - table(F, G, H, K, \alpha, \beta, \gamma)$
retourner $resultat$
sinon
$v = variable$ minimale de F, G, H
$I = pcase(\sigma_1(F_1), \alpha(\phi_1(G_1)), \beta(\theta_1(H_1)), \gamma(\lambda_1(K_1)))$
$J = pcase(\sigma_2(F_2), \alpha(\phi_2(G_2)), \beta(\theta_2(H_2)), \gamma(\lambda_2(K_2)))$
$L = pcase(\sigma_3(F_3), \alpha(\phi_3(G_3)), \beta(\theta_3(H_3)), \gamma(\lambda_3(K_3)))$
si $I = J = L$ retourner I
inserer $R = (v, I, J, L, id, id, idd)$
retourner R
fin

6 Conclusion

L'objectif de ce rapport a été d'étudier le problème de la représentation dans SIGALI des formes polynomiales à partir de graphes de décisions ternaires typés : TDD typés.

Après un bref rappel des travaux déjà effectués par B. Dutertre sur les TDD, l'étude montre comment il est possible de typer ces TDD tout en s'attachant à conserver une structure interne identique à celle développée précédemment.

L'implémentation de la réduction et de certaines opérations ayant été réalisée, il a été possible d'effectuer une analyse comparative entre les deux modes de représentations. Cette étude a permis de montrer l'interêt du typage à l'aide de permutations pour des fonctions polynomiales de base. Mais elle a aussi montré le faible intrêt de ce typage en ce qui concerne les fonctions polynomiales provenant du codage de programme SIGNAL.

Le faible rendement des TDD typés par rapport aux TDD classiques, a permis de mettre en évidence les "faiblesses" de SIGALI, notamment en ce qui concerne la gestion en mémoire des TDD. Il a donc été essayé dans ce rapport d'apporter des débuts de réponse à ces problèmes. La création d'une table pour ranger les polynômes pourrait être envisagée par la suite.

7 Annexe

On présente dans cette annexe différents exemples de TDD, typés ou non, utilisés dans le paragraphe concernant la comparaison entre les TDD et les TDD typés.

TDD numéro 1 :

```
>P: a default b*c when not d default e^2;
P

>P:
  9 : (a 2 0 8)
  8 : (b 7 6 3)
  7 : (c 5 4 3)
  6 : (c 4 5 3)
  5 : (d 3 2 3)
  4 : (d 3 0 3)
  3 : (e 2 2 1)
  2 :  1
  1 :  0
  0 : -1
```

TDD numéro 2 :

```
>P:a+b+c+d+e+f+g+h+i+j;
P

>P;
 30 : (a 29 27 28)
 29 : (b 26 24 25)
 28 : (b 25 26 24)
 27 : (b 24 25 26)
 26 : (c 23 21 22)
 25 : (c 22 23 21)
 24 : (c 21 22 23)
 23 : (d 20 18 19)
 22 : (d 19 20 18)
 21 : (d 18 19 20)
 20 : (e 17 15 16)
 19 : (e 16 17 15)
 18 : (e 15 16 17)
 17 : (f 14 12 13)
 16 : (f 13 14 12)
 15 : (f 12 13 14)
 14 : (g 11 9 10)
 13 : (g 10 11 9)
 12 : (g 9 10 11)
 11 : (h 8 6 7)
 10 : (h 7 8 6)
  9 : (h 6 7 8)
  8 : (i 5 3 4)
  7 : (i 4 5 3)
  6 : (i 3 4 5)
  5 : (j 2 0 1)
  4 : (j 1 2 0)
  3 : (j 0 1 2)
  2 :  1
  1 :  0
  0 : -1
```

TDD numéro 3 :

```
>P:a+b+c+d+e+f+g+h+i+j;
P

>P;
 10 : (a pg 4 pm 0 pd 3 ng 9 nm 9 nd 9)
  9 : (b pg 0 pm 3 pd 4 ng 8 nm 8 nd 8)
  8 : (c pg 0 pm 3 pd 4 ng 7 nm 7 nd 7)
  7 : (d pg 0 pm 3 pd 4 ng 6 nm 6 nd 6)
  6 : (e pg 0 pm 3 pd 4 ng 5 nm 5 nd 5)
  5 : (f pg 0 pm 3 pd 4 ng 4 nm 4 nd 4)
  4 : (g pg 0 pm 3 pd 4 ng 3 nm 3 nd 3)
  3 : (h pg 0 pm 3 pd 4 ng 2 nm 2 nd 2)
  2 : (i pg 0 pm 3 pd 4 ng 1 nm 1 nd 1)
  1 : (j pg 5 pm 0 pd 2 ng 0 nm 0 nd 0)
  0 :  0
```

TDD numéro 4 : extrait de la représentation du polynôme des contriantes Q sous forme de TDD typé :

```
[20] : (cond_10 pg 0 pm 0 pd 2 ng 16 nm 12 nd 0)
[19] : (cond_10 pg 0 pm 0 pd 2 ng 17 nm 15 nd 0)
 18 : (cond_10 pg 2 pm 2 pd 0 ng 0 nm 0 nd 13)
 17 : (etat_3 pg 0 pm 0 pd 0 ng 14 nm 14 nd 11)
 16 : (etat_3 pg 0 pm 0 pd 0 ng 14 nm 11 nd 11)
 15 : (etat_3 pg 0 pm 0 pd 0 ng 11 nm 14 nd 11)
 14 : (cond_11 pg 0 pm 0 pd 2 ng 10 nm 10 nd 0)
 13 : (cond_11 pg 2 pm 2 pd 0 ng 0 nm 0 nd 9)
 12 : (cond_11 pg 2 pm 2 pd 0 ng 0 nm 0 nd 8)
 11 : (cond_11 pg 2 pm 2 pd 0 ng 0 nm 0 nd 7)
 10 : (cond_12 pg 0 pm 0 pd 2 ng 6 nm 6 nd 0)
  9 : (cond_12 pg 2 pm 2 pd 0 ng 0 nm 0 nd 4)
  8 : (cond_12 pg 2 pm 2 pd 0 ng 0 nm 0 nd 5)
  7 : (cond_12 pg 2 pm 2 pd 0 ng 0 nm 0 nd 6)
  6 : (cond_13 pg 0 pm 0 pd 2 ng 3 nm 3 nd 0)
  5 : (cond_13 pg 0 pm 0 pd 2 ng 2 nm 2 nd 0)
  4 : (cond_13 pg 2 pm 2 pd 0 ng 0 nm 0 nd 2)
  3 : (cond_14 pg 2 pm 2 pd 2 ng 1 nm 1 nd 0)
  2 : (cond_14 pg 2 pm 2 pd 0 ng 0 nm 0 nd 1)
  1 : (cond_15 pg 2 pm 2 pd 0 ng 0 nm 0 nd 0)
  0 :  0
```

References

[1] M. Le Borgne. *Systèmes dynamiques sur des corps finis.* PhD thesis, univeristé de Rennes, September 1993.

[2] R.E. Bryant. Graph-based algorithms for boolean function manipulations. *IEEE Transaction on Computers*, C-45(8):677–691, aug 1986.

[3] O. Coudert. Siam, une boite à outils pour la preuve formelles de systèmes séquentielles. *Thèse de ENS des télécommunications*, Octobre 1991.

[4] B. Dutertre. Sigali: un sytème de calcul formel pour la vèrification de programme signal. *Manuel d'utilisation*, Mai 1993.

[5] B. Dutertre. *Spécification et preuve de systèmes dynamiques: Application à* SIGNAL. PhD thesis, Université de Rennes, dec 1992.

[6] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. *Proceediing of the IEEE*, 9(79):1321–1336, September 1991.

[7] R. L. Rudell K. S. Brace and R. E. Bryant. Efficient implementation of a bdd package. *27th ACM/IEEE design automation conference*, 40–45, 1990.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399