Modèles commutatifs pour l'optimisation

Laurent THIRY[1]

[1]Laboratoire MIPS, ENSISA
12, rue des frères Lumière. 68093 Mulhouse, France.
laurent.thiry@uha.fr

Résumé— **Cet article s'intéresse aux modèles P, M tels que $P \circ M \cong M \circ P$. Par exemple, un polynôme P dont les coefficients sont des matrices M peut être réécrit comme une matrice de polynômes, ou réciproquement. Si les deux représentations sont équivalentes, la seconde présente l'avantage de pouvoir être simplifiée - ce qui conduit à moins de calculs. L'article montre alors comment utiliser le concept de "foncteur" pour décrire (P, M) et établir des isomorphismes $\phi : P \circ M \to M \circ P$ qui soient des optimisations. Un algorithme permettant de réduire certaines expressions symboliques est proposé comme application des éléments présentés.**

Mots-clés— **Théorie des catégories, Optimisation, Calcul formel**

I. Introduction

Un algorithme de commande peut être dégradé par son implémentation. Inversement, le choix d'une implémentation particulière doit permettre de garder des performances optimales. En particulier, une implémentation nécessitant moins de calculs sera plus rapide et la réduction du nombre d'opérations nécessaires sera une optimisation.

La mise en oeuvre d'une optimisation nécessite un cadre formel pour montrer que deux modèles M et M' sont les "mêmes" mais que M' est "mieux" que M. Parmi les théories possibles, celle des Catégories [2], [5] a permis de proposer différentes approches pour raisonner et éprouver les logiciels en général avec notamment les concepts de "program calculation" [11] ou de "point-free programming" [12]. Plus précisément, les programmes sont modélisés par des fonctions [6], [9] combinées à l'aide d'opérateurs particuliers dont les propriétés algébriques sont établies. Ces propriétés sont exploitées alors pour inférer des équivalences entre deux modèles ou pour mesurer la complexité d'un programme [3]. Cela dit, les travaux utilisant les catégories pour formaliser et étudier plus particulièrement les logiciels de commande restent encore aujourd'hui peu nombreux. Parmi ceux-ci, nos travaux ont déjà permis de montrer que les modèles de systèmes dynamiques pouvaient être formalisés à l'aide de fonctions particulières (qui sont des foncteurs) utilisables alors pour simuler ou piloter ces systèmes [13], [14]. Cet article présente une continuation de ces éléments avec la prise en compte de nouveaux objets mathématiques (expressions symboliques, polynômes et matrices), et l'utilisation du support formel apporté par les catégories pour établir des relations entre ces objets. S'il existe déjà des descriptions de ces derniers avec, par exemple, [4], [7], [8], [10], la particularité du travail présenté est qu'il s'intéresse avant tout à leur composition. Ainsi, le terme de "commutation" exprime la propriété de deux modèles à pouvoir être composés dans deux sens différents sans perte d'information. $P \circ M \cong M \circ P$ a alors le même sens que $A \times B = B \times A$ en arithmétique. Dans le cas des logiciels de commande, caractérisés notamment par des calculs intensifs, cette propriété permet de choisir une forme plus simple pour un modèle mathématique et nécessitant moins de calculs. L'article profite alors de la théorie des catégories pour prouver ces équivalences et pour équiper les modèles d'une métrique permettant de les comparer.

Cet article se compose de trois parties. La première partie présente la théorie des catégories et plus précisément les concepts de *foncteurs*, de *transformations naturelles* et *d'adjonction*. Elle montre comment les foncteurs peuvent être utilisés pour représenter à la fois des ensembles structurés (appelés aussi *algèbres*) et les traitements sur ces structures. Elle montre ensuite comment les transformations naturelles et les adjonctions permettent d'établir des équivalences entre des foncteurs. Le concept de *morphisme* est aussi introduit pour représenter les changements de représentation. La seconde partie propose trois exemples de foncteurs utilisables pour décrire des systèmes avec des expressions symboliques $E(A)$, des polynômes $P(A)$ et des matrices $M(A)$; A représente le type des éléments contenus dans une expression, un polynôme ou une matrice. Les foncteurs sont utilisés aussi pour établir un ensemble d'opérateurs pour chaque type, i.e. pour calculer, par exemple, l'addition ou la multiplication de deux polynômes ou de deux matrices. Ces opérateurs ont la particularité d'être décrits de façon compacte et formelle ; ce qui permet alors de comparer plus simplement et plus précisément les relations existant entre différentes représentations. Cette seconde partie propose ensuite un (iso)morphisme ϕ tel que si r est un modèle composé de polynômes de matrices et si $|r|$ est le nombre d'opérations élémentaires $(+, \times)$ pour évaluer le modèle alors $|\phi(r)| \leq |r|$. La troisième partie reprend les éléments importants de cet article et présente les perspectives envisagées.

II. Théorie des catégories

Cette première partie définit à la fois le concept de "modèle" utilisé dans l'article, et les relations possibles entre deux modèles. La définition proposée s'appuie sur la théorie des catégorie car celle-ci permet notamment de décrire de façon uniforme des structures (de données) et les opérations sur ces structures. Ainsi, le langage utilisé dans la proposition (partie III) se résumera à la grammaire suivante : `M=MO | (MxM) | (M+M) | M*`. Un modèle `M`, correspondant alors à un élément d'information ou de transformation, sera défini soit par un élément de base (`MO`), soit par une paire d'éléments (`x`) ou une alternative entre deux éléments (`+`), soit par une collection d'éléments (`*`). Dans le

cadre des catégories, les constructions précédentes correspondent à des foncteurs (partie II-A) et la connaissance de ces derniers permet d'établir certaines propriétés telle que "l'équivalence" de deux foncteurs et donc de deux modèles (partie II-B).

A. Catégories et foncteurs

Une *catégorie* est une structure mathématique composée de deux ensembles appelés "objets" O et "morphismes" M [2], [5]. L'article s'appuie alors sur la catégorie $\mathcal{E}ns$ dont les objets sont des ensembles E_i (et aussi des types de données) et les morphismes sont des fonctions $f_j : E_k \to E_l$ (utilisées, par exemple, par les langages fonctionnels). Plus précisément, pour avoir une catégorie, il doit y avoir un morphisme identité pour chaque objet $1_O : O \to M$, et un opérateur de composition $(\circ) : M \times M \to M$. Ces éléments doivent vérifier un ensemble de propriétés telles que si (A, B, C, D) sont des objets et (f, g, h) des morphismes alors $1_A \circ f = f = f \circ 1_B$ et $(f \circ g) \circ h = f \circ (g \circ h)$ qui dit simplement que $(\circ)$ est commutatif. Pour que ces équations soient vérifiées, il faut en plus $f : A \to B$, $g : B \to C$ et $h : C \to D$. A partir de là, les catégories peuvent être reliées par des foncteurs.

Un *foncteur* est une transformation qui s'applique à la fois aux objets et aux morphismes, tout en préservant l'identité et la composition. Un foncteur peut être vu comme une paire $(F_0 : O \to O', F_1 : M \to M')$ vérifiant les propriétés : $F_1(1_A) = 1_{F_0(A)}$ et $F_1(f \circ g) = F_1(f) \circ F_1(g)$. De plus, si $f : A \to B$ est un morphisme alors $F_1(f) : F_0(A) \to F_0(B)$. Un exemple de foncteur est $(*) : \mathcal{E}ns \to \mathcal{E}ns$ tel que si A est un ensemble et f une fonction $A \to B$ alors A^* est l'ensemble des suites d'éléments de A et f^* est une fonction qui applique f à tous les éléments d'une suite. Cette fonction bien connue des langages fonctionnels[1] est utilisée, par exemple, pour décrire la sémantique des modèles flots de données [15]. Ainsi, si $\mathbb{R}^*$ représente un signal alors un bloc de type "gain k" sera représenté par $(\times k)^* : \mathbb{R}^* \to \mathbb{R}^*$. La Figure 1 présente un autre exemple de foncteur entre deux automates : les objets sont ici les états et les morphismes les transitions [1]. F_0 correspond aux flèches en pointillés fins et F_1 aux pointillés aux pointillés épais. Cet exemple donne ainsi une autre application du concept de foncteur dans le domaine des Systèmes à Evénements Discrets (SED) ; F montre alors que A' est une abstraction de A.

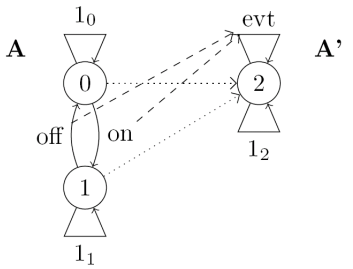

Fig. 1. Exemple de foncteur.

Deux autres exemples de foncteurs fondamentaux sont donnés par le produit et le somme. Dans le *produit* $\mathcal{E}ns \times \mathcal{E}ns$, les objets sont donnés par le produit cartésien $A \times B = \{(a, b) \bullet a \in A, b \in B\}$ et les morphismes sont définis par $(f \times f')(a, b) = (f(a), f'(b))$. Dans la *somme* $\mathcal{E}ns + \mathcal{E}ns$, les objets sont donnés par l'union disjointe $A + B = \{(0, a) \bullet a \in A\} \cup \{(1, b) \bullet b \in B\}$ et les morphismes sont définis par $(f + f')(0, a) = (0, f(a))$ et $(f + f')(1, b) = (1, f'(b))$. Ces foncteurs sont utilisés notamment pour décrire d'autres foncteurs ou pour *spécifier des types de données* [4], [11]. Par exemple, les listes A^* citées plus haut peuvent être spécifiées par deux fonctions permettant de créer une liste vide $nil : 1 \to A^*$ (avec "1" un type/ensemble à un élément) et pour ajouter un élément à une liste $cons : A \times A^* \to A^*$. Ces deux fonctions peuvent être regroupées en $1 + A \times A^* \to A^*$, ce qui peut aussi s'écrire $A^* = F(A)$ avec $F(A) = 1 + A \times A^*$ le foncteur décrivant la structure de liste. $F(f)$ est alors égal à $1 + f \times F(f)$. Maintenant, produit et somme sont appelés des "limites" en théorie des catégories. Ceci signifie que si $f : P \to A$ et $g : P \to B$ sont deux morphismes alors il existe un unique morphisme $< f, g > : P \to A \times B$ tel que $\pi_1 \circ < f, g >= f$ et $\pi_2 \circ < f, g >= g$. (π_1, π_2) sont appelées "projections" et ont pour type $\pi_1 : A \times B \to A$ et $\pi_2 : A \times B \to B$. Pour la somme, si $f : A \to S$ et $g : B \to S$ sont deux morphismes alors il existe un unique morphisme $[f, g] : A + B \to S$ tel que $[f, g] \circ i_1 = f$ et $[f, g] \circ i_2 = g$. (i_1, i_2) sont appelées "injections" et ont pour type $i_1 : A \to A + B$ et $i_2 : B \to A + B$. Ainsi, il est possible de remarquer que les fonctions $[nil, cons]$ définies plus haut sont équivalentes aux fonctions $[i_1, i_2]$ et donc toutes les fonctions sur les listes, du type $A^* \to B$, pourront s'exprimer à l'aide d'une *fonction générique* $g(v, h) = [v, h \times g(v, h)]$ avec $v : 1 \to B$ et $g : A \times B \to B$. Par exemple, f^* sera égal à $g(nil, f \times f^*)$, la concaténation de deux listes à $l \oplus l' = g(l', cons)(l)$, la concaténation d'une liste de listes à $g(nil, \oplus)$, l'inverse d'une liste par $l^{-1} = g(nil, \lambda(x, l).l \oplus cons(x, nil))$[2], etc. Cette dernière expression montre ainsi que les listes et les opérateurs présentés permettent de généraliser des opérateurs binaires. En particulier, Σ_i sera égal à $g(0, +)$ et Π_i à $g(1, \times)$. La partie III utilisera alors cette fonction générique pour simplifier la description de certaines fonctions.

Définitions. A partir des éléments précédents, un modèle sera défini comme un élément M de l'ensemble S généré en prenant des éléments de base M_0 (pouvant être un type primitif comme $\mathbb{R}$ ou $\mathbb{N}$, ou des fonctions prédéfinies), et en les composants avec les foncteurs présentés, i.e. $M \in S = \{M_0, M_0 \times M_0, M_0^*, (M_0 \times M_0)^*, ...\}$. La partie suivante définit alors le concept d'équivalence (i.e. $M \cong M'$) en utilisant celui de transformation naturelle, et d'isomorphisme, entre deux foncteurs (i.e. deux modèles sont équivalents s'il existe un isomorphisme $\phi : M \to M'$) - "iso" signifiant que ϕ^{-1} existe.

B. Transformations naturelles et adjonctions

Une *transformation naturelle* est une relation entre deux foncteurs et donc aussi entre deux structures (de données) ou deux transformations. Plus précisément, une transformation naturelle entre deux foncteurs (F, G) est définie par une famille de morphismes $\eta_A : F(A) \to G(A)$ (A étant un objet quelconque) telle que pour tout morphisme

[1] f^* est généralement appelée `map(f)`.

[2] L'expression $\lambda x.e$ dénote la fonction f telle que $f(x) = e$.

$f : A \rightarrow B$, $G(f) \circ \eta_A = \eta_B \circ F(f)$. Plus simplement, si (F, G) sont deux structures de données telles que de F il est possible de passer à G par η alors il est possible 1) d'appliquer une transformation f puis de changer de représentation ou 2) de changer de représentation avant d'appliquer la transformation ; le résultat sera le même. Un exemple de transformation naturelle est $\eta =< \pi_2, \pi_1 >$ qui permet de changer l'ordre des éléments dans une paire, i.e. $\eta : A \times B \rightarrow B \times A$. Cette transformation vérifie la propriété $\eta \circ \eta = 1_{A \times B}$ donc elle est inversible et $\eta^{-1} = \eta$. Lorsqu'une telle propriété est vérifiée, les foncteurs et les modèles représentés sont dits *équivalents* ou isomorphes avec ici $A \times B \cong B \times A$. Cette propriété établit la commutativité du produit. De la même manière, il est possible d'établir des transformations naturelles (inversibles) pour l'associativité $(A \times B) \times C \cong A \times (B \times C)$ ou la distributivité $A \times (B + C) \cong (A \times B) + (A \times C)$. La dernière équivalence est donnée alors par $\eta(a, i_1(b)) = i_1(a, b)$ et $\eta(a, i_2(c)) = i_2(a, c)$. Les transformations naturelles peuvent s'appliquer à d'autres foncteurs. En particulier, il est possible de montrer l'équivalence $(A \times B)^* \cong A^* \times B^*$ donnée par les fonctions `zip/unzip` bien connues en programmation fonctionnelle. Comme exemple pratique d'utilisation dans le cadre des flots de données, un bloc de type sommateur pourra se représenter par $(+)^* \circ zip$ dont le type est $\mathbb{R}^* \times \mathbb{R}^* \rightarrow \mathbb{R}^*$. Pour finir, le concept de transformation naturelle est lié à celui d'adjonction. Une *adjonction* est une paire de foncteurs (F, G) telle que si $M[A, B]$ représente l'ensemble des morphismes $A \rightarrow B$ alors $M[A, F(A)] \cong M[G(B), B]$. Un exemple d'adjonction donné par la théorie des catégories et repris dans la suite de l'article est $M[A \times B, C] \cong M[A, C^B]$ avec l'exponentielle C^B représentant l'ensemble des fonctions $B \rightarrow C$[3]. L'équation est bien connues des langages fonctionnels avec les formes curryfiées de fonctions binaires, i.e. une fonction à deux paramètres est équivalente à une fonction d'un paramètre retournant une autre fonction d'un paramètre. L'intérêt de la curryfication est de pouvoir évaluer partiellement une fonction et d'offrir un moyen de générer une famille de fonctions. Par exemple, la fonction binaire $plus(x, y)$ s'écrira $plus(x)(y)$ sous forme currifiée, et $plus(x)$ représentera simplement une fonction unaire dont l'effet est d'ajouter x.

III. Commutation de modèles et optimisation

Cette partie propose à la fois trois modèles formels dédiés au calcul, et un modèle de transformation pour réduire le nombre d'opérations nécessaires pour réaliser un calcul. Elle montre ainsi comment les concepts définis dans la partie précédente permettent de décrire de façon compacte et formelle des modèles, ou comment préciser le concept "d'optimisation".

A. Trois modèles pour le calcul

Le rôle fondamental d'un logiciel de commande est de *calculer* les valeurs à appliquer à un système pour qu'il ait le comportement souhaité. Ainsi, la dynamique d'un système est souvent représentée par un couple d'équations $x_{n+1} = f(x_n, u_n)$ et $y_n = g(x_n, u_n)$ dans lesquelles x_n représente l'état interne à l'instant n, u_n correspond à l'entrée et y_n à la sortie du système. (f, g) sont alors des expressions prenant généralement la forme de polynômes (ex. $a.x+b$) dans lesquels (a, b) sont des matrices/vecteurs. L'évaluation de ces expressions doit se faire plusieurs fois, i.e. pour un instant n allant de 0 à N (N pouvant être important), et le temps nécessaire à l'évaluation dépend de la forme choisie pour exprimer (f, g). Par exemple, si l'expression $x^2 + a.x$ est égale à $x.(x + a)$, l'avantage de la seconde expression est d'avoir une multiplication en moins. Sur un interval de taille N, le temps nécessaire pour évaluer la seconde expression devient donc moins important que le temps nécessaire pour évaluer la première. La seconde expression sera alors dite "meilleure" que la première, et le passage de la première représentation à la seconde sera considérée comme une "optimisation". Cette partie montre alors comment représenter les expressions (et les modèles) utilisées à l'aide de foncteurs, et comment définir des transformations naturelles pour optimiser certaines expressions. Cette présentation est composée de 3+1 étapes.

La première étape consiste à définir un *modèle d'expressions* $E(A)$ composées de valeurs de type A, de sommes ou de produits de deux expressions, et d'une variable X. Le modèle utilisé est spécifié par quatre fonctions $x : 1 \rightarrow E(A)$, $val : A \rightarrow E(A)$ et $plus, mult : E(A) \times E(A) \rightarrow E(A)$. Ainsi, l'expression $1+(2\times3)$, dont le type est $E(\mathbb{N})$, sera représentée par $e = plus(val(1), mult(val(2), val(3)))$. Le foncteur pour ce type de données est $E(A) = 1 + A + E(A) \times E(A) + E(A) \times E(A)$; E est associé alors à la fonction générique donnée par $g_E(f_0, f_1, f_2, f_3) : E(A) \rightarrow B$, avec $g_E(f_i)(x) = f_0$, $g_E(f_i)(val(v)) = f_1(v)$, $g_E(f_i)(plus(g, d)) = f_2(g_E(f_i)(g), g_E(f_i)(d))$, etc. Avec cette fonction, l'évaluation d'une expression sera donnée par $eval = g_E(0, id, (+), (\times))$, et $eval(e)$ se réduira à $(+)(1, (\times)(2, 3)) = 7$. De même, pour obtenir le nombre d'opérations $(+, \times)$ qui composent une expression, il suffit de prendre $t = g_E(f_i)$ avec $f_0(x) = f_1(v) = 0$, $f_{2,3}(n, m) = n + m + 1$; (n, m) représentent le nombre d'opérateurs à gauche et à droite de l'opérateur $(+, \times)$. La Figure 2 illustre le comportement de la fonction générique g_E associée au foncteur $E(A)$. Elle présente aussi un exemple d'utilisation de cette fonction pour calculer le degré d'un polynôme représenté par une expression (i.e. avec $g_E(1, \lambda x.0, max, (+))$).

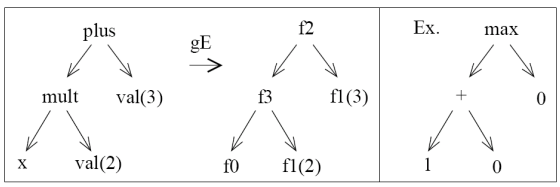

Fig. 2. Effet de la fonction $g_E : E(A) \rightarrow B$ et exemple.

La seconde étape consiste à définir un *modèle de polynômes* $P(A)$ dont les coefficients sont de type A. Un polynôme $P = \sum_{i=1}^{n} a_i.X^i$ peut se représenter par une liste de coefficients $(a_i)_{i \in I}$, et donc $P(A) \cong A^{\mathbb{N}}$ (ou encore A^*). Ainsi, en représentant plus simplement la liste vide nil par $()$ et l'ajout d'un élément $cons(a, l)$ par (a, l), le polynôme $p = a_0 + a_1.x + a_2.x^2$ sera représenté de façon unique par $p = (a_0, (a_1, (a_2, ())))$ (ou, plus simplement,

[3]Avec cette notation, une liste A^* pourra se représenter aussi par $A^{\mathbb{N}}$, i.e. $A^* \cong A^{\mathbb{N}}$ comme expliqué en note 5.

$p = (a_0, a_1, a_2)$). Il est possible d'utiliser alors la fonction générique sur les listes g (présentée dans la partie II) pour définir les opérations sur les polynômes. En particulier, le degré du polynôme p sera obtenu avec $degre(p) = g(0, (1+) \circ \pi_1)(p) - 1$. Dans le cas du polynôme p ci-dessus, l'expression se réduit à $(1+(1+(1+0)))-1 = 2$. De même, avec la fonction $map(h) = h^*$, la multiplication d'un polynôme p par une constante v s'écrira $map(\times v)(p)$. Avec la fonction zip, la somme de deux polynômes à coefficients entiers $P(\mathbb{N})$ s'écrira $p+q = zip(+)(p,q)$ et le produit terme à terme s'écrira $p \otimes q = zip(\times)(p,q)$. Si les coefficients sont des expressions $P(E)$ alors ces expressions deviennent $p+q = zip(plus)(p,q)$ et $p \otimes q = zip(mult)(p,q)$. Il faut noter ensuite que la création ou la manipulation des listes ne se limitent pas aux fonctions présentées ici. En particulier, en posant $h(v,f)(0) = v$ et $h(v,f)(n) = f(h(v,f)(n-1))$, $range(n) = h((), \lambda l.cons(n,l))(n)$ va retourner l'intervalle $(0, 1, ..., n)$, et $repete(n,x) = h((), \lambda l.cons(x,l))$ va retourner une liste de taille n dont tous les éléments sont égaux à x. De même, il est possible de définir par exemple les fonctions $take(n)$ pour extraire les n premiers éléments d'une liste, et $drop(n)$ pour supprimer ces n premiers éléments. Avec ces remarques et en reprenant les fonctions déjà présentées, le produit de deux polynômes sera donné par $p \times q = map(\lambda n.(repete(n,0) \oplus p) \otimes q^{-1})(range(degre(p) + degre(q)))$. Ainsi, le modèle de polynômes est complété par les opérateurs $(+, \times)$ et il est possible de définir des polynômes d'expressions $(P(E(A)))$ ou des expressions de polynômes $(E(P(A)))$[4]. Enfin, il faut noter que des transformations peuvent être établis entre ces différents modèles comme l'explique la remarque finale de cette section.

La troisième étape consiste à définir un *modèle de matrices* $M(A)$. Le modèle de listes utilisé pour décrire les polynômes peut-être repris pour décrire des matrices, avec $M(A) \cong (A^{\mathbb{N}})^{\mathbb{N}}$. Le morphisme générique g et les fonctions définies plus haut permettent alors de spécifier l'addition et la multiplication de matrices. En particulier, l'addition de deux matrices s'écrira simplement $m + m' = zip(+)(m,m')$ où le second $(+)$ correspond à l'addition terme à terme de deux listes. Pour définir la multiplication de deux matrices, il faut définir la fonction $dot(v,w) = \Sigma_i(v \otimes w)$ qui retourne le produit scalaire de v et w, puis la fonction $tran(m) = map(\lambda i.map(\lambda v.v(i))(m))(range(degre(\pi_1(l))))$ qui retourne la transposée d'une matrice. Ceci permet d'établir le produit par $m \times m' = map(\lambda v.map(\lambda w.dot(v,w))(m))(tran(m'))$. Ainsi, comme pour les polynômes, $M(A)$ est complété par les opérateurs $(+, \times)$ et il est possible de définir des matrices de valeurs $M(\mathbb{N})$, de termes $M \circ E$, de polynômes $M \circ P$, des expressions de matrices $E \circ M$, des polynômes dont les coefficients sont des matrices $P \circ M$, etc. La seule contrainte étant toujours que A soit muni des opérateurs internes $(+, \times)$. A partir de là, la dernière étape consiste à établir des relations entre ces différents modèles et plus précisément entre les modèles $P(M(E))$ et $M(P(E))$.

Remarque. Il existe différentes relations possibles entre les trois foncteurs (E, P, M) présentés. Par exemple, une expression comme $2x + 1$, qui est du type $E(\mathbb{N})$, peut être représentée par le polynôme $(1, 2)$ (de type $P(\mathbb{N})$) et réciproquement ; ceci établit, de façon informelle, que $E \cong P$. Il est facile de vérifier que l'équivalence précédente peut être généralisée à des listes d'expressions et de polynômes, i.e. $E^* \cong P^*$. Par exemple, $(2x + 1, x + 3) \cong ((1, 2), (3, 1))$, ce qui correspond aussi à une matrice $M(\mathbb{N})$ (donc $M \cong E^* \cong P^*$). Maintenant, il est possible de vérifier qu'une expression d'expressions peut être réduite à une expression, i.e. $E \circ E \cong E$. Par exemple, $plus(val(plus(val(1), val(2))), val(val(3)))$ qui est du type $E(E(\mathbb{N}))$ peut être ramenée à $plus(plus(val(1), val(2)), val(3))$. A partir de là, d'autres équivalences peuvent être établies comme, par exemple, $E \circ P \cong P \circ E$, etc. La partie suivante explique alors comment établir une équivalence en donnant la transformation naturelle (inversible/isomorphe) pour passer d'un modèle à un autre.

B. Optimisation par commutation de modèles

La définition d'une transformation naturelle $\eta_A : P(M(A)) \rightarrow M(P(A))$, ce qui se note plus simplement $\eta : P \circ M \rightarrow M \circ P$, permet d'établir l'équivalence entre des polynômes dont les coefficients sont des matrices (i.e. $P \circ M$) et les matrices dont les coefficients sont des polynômes (i.e. $M \circ P$). L'équivalence $P \circ M \cong M \circ P$ correspond aussi à la commutativité de $(\circ)$ pour (M, P). A partir de là, chacune de ces représentations est caractérisée par un nombre t d'opérateurs $(+, \times)$, et une représentation r sera dite "meilleure" qu'une autre représentation r' si $r \cong r' \wedge t(r) < t(r')$. A partir de ces définitions, cette partie montre comment, à partir de η et d'un ensemble de simplifications sur les expressions $s : E \rightarrow E$, construire une métrique t et un morphisme ϕ qui soit une optimisation (i.e. tel que $\phi(r)$ nécessite moins de calculs que r).

Dans le cas présent, la transformation η consiste à développer un polynôme p dont les coefficients sont des matrices comme l'illustre la Figure 3. Sur cette figure, p est du type $P(M(\mathbb{N}))$ et $m = \eta(p)$ est du type $M(P(\mathbb{N}))$. η est définie de la façon suivante : si $p = \sum_k a_k.x^k$, avec a_k une matrice, alors les coefficients de la matrice résultat m seront donnés par $(m)_{ij} = \sum_k (a_k)_{ij}.x^k$ - (i, j) étant respectivement la ligne et la colonne considérées. Il est possible de vérifier que η est inversible ce qui établit l'équivalence $P \circ M \cong M \circ P$. Maintenant, si k est le degré de p et si chaque matrice a_k a $n \times n$ éléments alors l'évaluation en x de p et de $m = \phi(p)$ nécessitera $t = 2 \times k \times n^2$ multiplications et additions. Cela dit, la seconde représentation $M \circ P$ a l'avantage de pouvoir être simplifiée. En effet comme expliqué plus haut, un polynôme $P(\mathbb{N})$ est équivalent à une expression $E(\mathbb{N})$ et $p = (1, 0, 2) \cong 1 \times x \times x + 0 \times x + 1$ qui peut se ramener, lorsqu'il s'agit d'une expression, à $x \times x + 1$. Les règles utilisées alors sont simplement : $(r_1) : 0+x = x+0 = x$, qui permet d'éliminer une addition, et $(r_2) : 1 \times x = x \times 1 = x \wedge 0 \times x = x \times 0 = 0$, qui permet d'éliminer une multiplication ou un terme.

Les règles (r_1, r_2) peuvent être utilisées pour définir la simplification d'expressions $s = g_E(id, id, r_1, r_2) : E(\mathbb{N}) \rightarrow E(\mathbb{N})$. Cette fonction a alors la propriété $\forall e.t(s(e)) \leq t(e)$ avec $t : E \rightarrow \mathbb{N}$ la fonction retournant le nombre d'opérateurs $(+, \times)$ définie dans la partie précédente.

[4]L'opérateur de compopsition $(\circ)$ peut être utilisé pour simplifier la notation, et $(P \circ E)(A) = P(E(A))$.

L'égalité est obtenue si aucun coefficient n'est égal à 0 ou 1. Les fonctions proposées dans la partie III-A permettent alors de généraliser (s, t) aux polynômes et aux matrices. Par exemple, un polynôme d'expressions sera simplifié avec $s_P = s^*$ et le nombre d'opérateurs sera obtenu par $t_P(p) = \Sigma_i(t^*(p))$. Comme pour les expressions, il est possible alors d'établir l'équation $t_P(s_P(p)) \leq t_P(p)$. Le même raisonnement peut être appliqué aux matrices, ce qui conduit alors à une nouvelle inégalité : $t_M((s_P^* \circ \eta)(p)) \leq t_P(p)$. La fonction $\phi : P \circ M \to M \circ P$ définie par $\phi = s_P^* \circ \eta$ est donc une optimisation par rapport au critère t donnant le nombre d'opérations élémentaires $(+, \times)$ requis pour évaluer une expression.

L'illustration de la Figure 3 montre l'intérêt de ϕ. Le polynôme de matrices p est de taille 8, i.e. son évaluation requiert 4 multiplications pour $A.x$ et 4 additions pour $(A.x) + B$. La matrice de polynômes $m = \eta(p)$ est aussi de taille 8. La simplification de cette matrice $\tilde{m}$ est quant à elle de taille 3 : il n'y a plus que deux multiplications avec x et une addition ! Le nombre de calculs nécessaires pour évaluer $p(x)$ est donc considérablement réduit (d'autant plus si N évaluations sont nécessaires).

$$
\begin{array}{ccc}
p(x) = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}.x + \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix} & \xrightarrow{t} & 4+4=8 \\
\Big\downarrow \eta & & \Big\| \\
m(x) = \begin{pmatrix} 1.x+2 & 0.x+0 \\ 2.x+0 & 1.x+3 \end{pmatrix} & \xrightarrow{t} & 4+4=8 \\
\Big\downarrow s_p^* & & \\
\tilde{m}(x) = \begin{pmatrix} x+2 & 0 \\ 2.x & x+3 \end{pmatrix} & \xrightarrow{t} & 2+1=3
\end{array}
$$

Fig. 3. Exemple d'optimisation.

A partir de là, l'intérêt des modèles et des fonctions présentés est de pouvoir être *mis en oeuvre directement avec un langage fonctionnel*. En particulier, les modèles présentés ont été implémentés dans le langage fonctionnel Haskell en suivant la démarche proposée dans [14]. Le code ci-dessous donne un extrait de cette implémentation et de l'exemple donné par la Figure 3. Plus précisément, le foncteur pour les expressions est traduit par une définition de type `data E` (le foncteur + est alors représenté par `|` et le foncteur × par une mise en séquence) ; `gE` représente, pour rappel, la fonction générique sur les expressions. Deux applications de cette fonction sont données avec les fonctions `t` pour connaître la taille d'une expression, i.e. le nombre d'opérateurs $(+, \times)$, et `s` pour simplifier une expression. La fonction `e` permet ici de convertir un polynôme en une expression. Ces trois fonctions sont généralisées aux matrices avec t', s', e', et la fonction ϕ est donnée par "phi" [5]. Enfin, les derniers éléments montre les résultats retournés par l'interpréteur Haskell en appliquant les fonctions précédentes au polynôme p.

```
data E a = Val a | X | Plus (E a) (E a)
         | Mult (E a) (E a)

gE(f1,f1,f3,f4) (Val  v  )=f1(v)
gE(f1,f1,f3,f4) (X       )=f2
gE(f1,f1,f3,f4) (Plus x y)=f3(gE(...)(x)) (gE(...)(y))
gE(f1,f1,f3,f4) (Mult x y)=f4(gE(...)(x)) (gE(...)(y))


t = gE(f1,f2,f3,f4)
  where f1 x  = 0
        f2    = 0
        f3 x y= x+y+1
s = gE(Val,X,r1,r2)
  where r1 (Val 0) x = x
        r1 x (Val 0) = x
        r2 ...
e [a1,a0] = ((Val a1) `Mult` X) `Plus` (Val a0)

t' = sum . map (sum . map t)
s' = map (map s)
e' = map (map e)

phi= \p->map(\i->map(\j->map(
     \k->pm!!k!!i!!j)[0..d])[0..m])[0..n]
  where d=length pm -1;  m=length (pm!!0) -1;
        n=length (pm!!0!!1) -1

-------------------- Exemple ----------------------
p  = [ [[1,0],[2,1]],  [[2,0],[0,3]] ]
--e (phi p)        = [1,0],[2,1]].x+[[2,0],[0,3]]
--e'(phi p)        = [[1.x+2,0.x+0],[2.x+0,1.x+3]]
--t'(e'(phi p))    = 8
--s'(e'(phi p))    = [[x+2,0],[2.x,x+3]]
--t'(s'(e'(phi p)))= 3
-----------------------------------------------------
```

[5] La notation `x->e` correspond à la fonction anonyme $\lambda x.e$, et `l ! !n` correspond à la fonction permettant d'obtenir le n-ième élément d'une liste l. La forme curryfiée de cette dernière fonction est utilisable alors pour établir l'isomorphisme (!!) : $A^* \to (\mathbb{N} \to A)$ ou $A^* \to A^{\mathbb{N}}$.

L'exemple précédent montre ainsi comment l'utilisation conjointe de la théorie des catégories et d'un langage fonctionnel permet de combiner à la fois des aspects théories/formels (avec des modèles mathématiques, et les foncteurs, décrivant des expressions, des polynômes ou des matrices) avec des aspects pratiques/applicatifs (avec ici un outil Haskell dédié au calcul formel de certaines expressions symboliques).

IV. Conclusion

Cet article a présenté une approche formelle de certains modèles utilisés par les logiciels de commande. La formalisation s'appuie sur le théorie des catégories et trois foncteurs ont été proposés pour représenter et traiter des modèles d'expressions symboliques E, de polynômes P et de matrices M. Ces trois modèles (de base) peuvent être combinés pour former des modèles plus complexes en composant les foncteurs correspondant (avec des matrices de polynômes $M \circ P$, par exemple). L'article a montré alors comment relier ces foncteurs par des transformations naturelles, pour passer d'une représentation à une autre, et pour établir des équivalences entre différentes représentations. En particulier, l'article s'est intéressé à la transformation $\phi : P \circ M \to M \circ P$ utilisable pour réduire le nombre d'opérations, et donc aussi le temps, nécessaire à un calcul. Pour cela, l'article introduit une métrique $t : E \to \mathbb{N}$ retournant le nombre d'opérations $(+, \times)$ utilisées par un modèle et un opérateur de simplification $s : E \to E$. Ainsi, les équations établies permettent de montrer que si p est un polynôme dont les coefficients sont des matrices alors $t(\phi(p)) \leq t(p)$; ce qui montre que ϕ est une optimisation et il n'y a pas de perte d'information car ϕ est inversible. En particulier, l'exemple donné par la Figure 3 montre que, dans le cas où plusieurs coefficients sont égaux à 1 ou à 0, le nombre d'opérations nécessaires pour évaluer un modèle peut être considérablement réduit. A partir de là, l'avan-

tage des modèles proposés est de pouvoir être implémentés directement dans un langage fonctionnel comme expliqué dans [14] et *les équations établies correspondent aussi à du code*.

Les perspectives considérées maintenant vont consister à 1) compléter les règles d'optimisation et 2) étudier d'autres représentations. En particulier, le modèle d'expressions peut être complété par d'autres opérateurs comme $(-, /, exp, cos, ...)$, par exemple. Ceci conduit à d'autres règles de simplification pouvant être utilisées pour *rechercher une expression équivalente mais nécessitant moins de calculs*. Ensuite, l'étude d'autres représentations doit permettre d'intégrer de nouvelles optimisations dans les calculs. En particulier, la représentation des polynômes avec des listes peut se révéler inadaptée et une autre représentation possible consiste à prendre des listes de paires composées d'un coefficient et d'un entier représentant le degré associé à ce coefficient. Ainsi, le polynôme $2.x^3$ représenté par $(0, 0, 0, 2)$ dans la représentation actuelle serait représenté par $((2, 3))$ dans la nouvelle représentation (avec un nouveau foncteur $P(A) = (\mathbb{N} \times A)^{\mathbb{N}}$). Cette seconde représentation a l'avantage de nécessiter aussi *moins d'espace mémoire*, ce qui est un critère à considérer dans le cadre des systèmes embarqués. La taille mémoire d'une représentation fournit donc une autre métrique possible et à étudier.

Enfin, de façon plus générale, des travaux sont envisagés pour montrer comment relier les éléments proposés à des outils plus classiques comme Matlab, par exemple.

Références

[1] J. Adamek and V. Trnkova. *Automata and Algebras in Categories*, volume 37 of *Mathematics and its Applications*. Springer, 1989.

[2] M. Barr and C. Wells. Category Theory - Lecture Notes for ESSLLI. page 123, 1999.

[3] R. Bird and O. Moor. *Algebra of programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[4] J. Fokker. Explaining algebraic theory with functional programs. In *FPLE '95 : Proceedings of the First International Symposium on Functional Programming Languages in Education*, pages 139–158. Springer-Verlag, 1995.

[5] J. Goguen. A Categorical Manifesto. *Mathematical Structures in Computer Science*, 1(1) :49–67, 1991.

[6] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2) :98–107, 1989.

[7] R. Lammel. Programmable rewriting strategies in Haskell. In *Electronic Notes in Theoretical Computer Science*, pages 101–112. Elsevier, 2004.

[8] J.R. Malaquias and C.R. Lopes. Implementing a computer algebra system in Haskell. *Applied Mathematics and Computation*, 192(1) :120 – 134, 2007.

[9] D.A. Mathaikutty. *Functional Programming and Metamodeling frameworks for System Design*. PhD thesis, Faculty of Virginia Polytechnic Institute and State University, 2005.

[10] M.D. Mc Ilroy. Functional pearls : Power series, power serious. *Journal of Functional Programming*, 1(1) :1–13, 1998.

[11] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144. Springer-Verlag, 1991.

[12] J.N. Oliveira and C.J. Rodrigues. Pointfree factorization of operation refinement. In *FM 2006 : Formal Methods, 14th International Symposium on Formal Methods*, pages 236–251, 2006.

[13] L. Thiry and B. Thirion. Functional Metamodels for the Development of Control software. In *International Federation of Automatic Control, IFAC'08, Seoul*, 2008.

[14] L. Thiry and B. Thirion. Functional metamodels for systems and software. *Journal of Systems and Software, Elsevier*, 82 :1125–1136, 2009.

[15] T. Uustalu and V. Vene. The essence of dataflow programming. In Springer, editor, *Lecture Notes on Computer Science*, volume 4164, pages 135–167, 2006.