Vers une modélisation diagrammatique de la bibliothèque C++ d'algèbre linéaire LINBOX

Jean-Guillaume Dumas — Dominique Duval

Université Joseph Fourier, Laboratoire de Modélisation et Calcul,
50 av. des Mathématiques. B.P. 53 X, 38041 Grenoble, France.
{Jean-Guillaume.Dumas, Dominique.Duval}@imag.fr

RÉSUMÉ. *Nous proposons un nouveau langage de modélisation diagrammatique, DML. Le paradigme utilisé est celui de la théorie des catégories et en particulier des sommes amalgamées (ou "pushouts"). Nous montrons que la plupart des structures orientées objet peuvent être décrites avec ce langage et proposons de nombreux exemples en C++, de l'héritage et du polymorphisme à la généricité (par "templates"). Par ailleurs, la bibliothèque C++ d'algèbre linéaire* LINBOX *a été conçue pour allier efficacité et généricité. Elle requiert donc de nombreuses structures génériques et polymorphes. Grâce à DML, nous proposons une description simple des structures complexes de cette bibliothèque.*

ABSTRACT. *We propose a new diagrammatic modeling language, DML. The paradigm used is that of the category theory and in particular of the pushout. We show that most of the object-oriented structures can be described with this language and have many examples in C++, ranging from virtual inheritance and polymorphism to template genericity. With this powerful tool, we propose a quite simple description of the C++* LINBOX *library. This library has been designed for efficiency and genericity and therefore makes heavy usage of complex template and polymorphic mechanism. By reverse engineering, we are able to describe in a simple manner the complex structure of archetypes in* LINBOX.

MOTS-CLÉS : *Spécifications diagrammatiques, modélisation de C++ ; modélisation objet ; modélisation diagrammatique*

KEYWORDS: *Diagrammatic specifications ; C++ modeling ; object-oriented modeling ; diagrammatic modeling*

1. Introduction

La bibliothèque LINBOX est une bibliothèque C++ de calculs exacts en algèbre linéaire avec des matrices denses ou creuses/structurées (“boîte noire”), sur les entiers ou sur les corps finis. LINBOX fournit des implémentations très performantes des algorithmes les plus avancés en algèbre linéaire exacte. et un système extrêmement complexe de classes C++, souvent paramétrées (par “templates” [MUS 96]), est utilisé pour obtenir simultanément la performance et la généricité requises [DUM 02]. En particulier, les algorithmes de LINBOX sont génériques relativement au type des coefficients et relativement à la structure de donnée interne des matrices.

Nous proposons une modélisation par rétro-ingénierie de ce système afin d’éclaircir son mécanisme sous-jacent et de décrire ses fonctionnalités de façon unifiée. Le paradigme choisi est celui de la *modélisation diagrammatique* et des *catégories*. Notre principal outil catégorique est la notion de *pushout* (ou *somme amalgamée*), qui correspond à plusieurs constructions en C++. Les pushouts sont largement utilisés pour décrire la combinaison de deux spécifications partageant une partie commune, voir par exemple [GOG 73, SRI 95, BUR 77, ORI 00]. Cependant, les langages de modélisation diagrammatique comme UML [MUL 00] ne permettent pas de traduire ces constructions de pushouts. Par exemple, en UML, tout comme dans les “design patterns” [GAM 94], les diagrammes de classes et les diagrammes d’objets sont distincts, alors que nous proposons de les grouper en un unique type de diagrammes, où une instanciation (d’une classe par un objet) est au même niveau qu’une association entre deux classes ou un lien entre deux objets. De plus, au contraire d’UML, nous considérons la relation entre une classe générique et son paramètre comme une sorte d’association, ce qui nous permet de considérer le passage de paramètres comme une construction par pushout. Pour toutes ces raisons, nous proposons d’utiliser un nouveau langage de modélisation diagrammatique (appelé DML), différent de UML. En particulier, cela nous permet d’étudier les notions orientées objet de passage de paramètre classique ou “templatge”, d’héritage virtuel, d’instanciation, comme des constructions de pushout dans une catégorie.

Quelques notions de base sur les catégories sont rappelées section 2 ; on peut les trouver plus détaillées dans de nombreux livres, comme [Mac 97, BAR 90]. Ensuite à la section 3 nous présentons le nouveau langage de modélisation diagrammatique DML et son application à C++ (et aussi sur Java pour certains passages) ; mais le langage DML pourrait être adapté à d’autres langages orientés objet. Finalement, DML est utilisé pour analyser la structure de la bibliothèque LINBOX à la section 4.

2. Catégories et pushouts

2.1. *Catégories*

Une catégorie peut être vue comme un monoïde généralisé. C’est le cas, par exemple, si $\mathcal{F}$ dénote les fonctions sur les réels, c’est-à-dire les fonctions de $\mathbb{R}$ vers $\mathbb{R}$,

comme $\sin, \cos, \exp : \mathbb{R} \to \mathbb{R}$. Ces fonctions peuvent être composées, par exemple $\exp . \sin$ est définie par $\exp . \sin(x) = \exp(\sin(x))$. Cela fournit une structure de *monoïde* sur l'ensemble $\mathcal{F}$: cela signifie que la composition est associative, i.e., $(f.g).h = f.(g.h)$, qui est donc noté $f.g.h$, et qu'il y a une unité pour la concaténation, à savoir l'identité id, définie par $id(x) = x$, telle que $f.id = f$ et $id.f = f$.

Maintenant, $\mathcal{F}$ peut également dénoter les fonctions de X vers Y, où X et Y peuvent être $\mathbb{R}$ ou $\mathbb{C}$, par exemple $\mathbb{R} \to \mathbb{R}$ (fonction sinus), $\mathbb{C} \to \mathbb{C}$ (conjugué complexe), $\mathbb{C} \to \mathbb{R}$ (module) ou $\mathbb{R} \to \mathbb{C}$ (inclusion). Ces fonctions peuvent encore être composées, mais seulement si elles sont consécutives : si $f : X \to Y$ et $g : Y \to Z$, alors $g.f : X \to Z$. L'associativité est encore vérifiée, lorsqu'elle a un sens. Il y a maintenant deux identités, $id_R : \mathbb{R} \to \mathbb{R}$ et $id_C : \mathbb{C} \to \mathbb{C}$. Les axiomes deviennent : si $f : X \to Y$ alors $f.id_X = f$ et $id_Y.f = f$. Ces fonctions ne forment plus un monoïde, à cause des restrictions dues au typage, mais elles forment une *catégorie*, au sens défini ci-dessous.

Definition 1 Une *catégorie* $\mathcal{C}$ est formée de *points* X, Y,... et de *flèches* f, g,..., chaque flèche a une *source* et un *but* ($f : X \to Y$ ou $X \xrightarrow{f} Y$), chaque point X a une flèche *identité* $id_X : X \to X$, chaque couple de flèches consécutives $X \xrightarrow{f} Y \xrightarrow{g} Z$ a une flèche *composée* $X \xrightarrow{g.f} Z$, et les axiomes suivants sont vérifiés : $(h.g).f = h.(g.f)$, $f.id_X = f$ et $id_Y.f = f$, dès que cela a un sens.

2.2. *Un exemple, l'héritage*

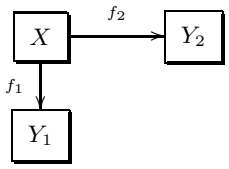
Une catégorie peut aussi être vue comme un ordre généralisé. Par exemple, considérons la relation d'héritage dans un langage orienté objet. La relation d'héritage définit un ordre partiel sur les classes : si Z hérite de Y, qui hérite de X, alors, par transitivité, Z hérite de X. Ajoutons une flèche $X \to Y$ à chaque fois que Y hérite de X[1]. Alors, la transitivité de la relation d'héritage correspond à la composition des flèches : s'il y a deux flèches consécutives $X \to Y \to Z$, alors il y a une flèche composée $X \to Z$. Lorsque l'héritage multiple est interdit, comme en Java, il n'est pas nécessaire de nommer les flèches, puisqu'il y a au plus une flèche de source et but données. Dans un langage orienté objet qui autorise l'héritage multiple, deux situations peuvent se produire, elles sont appelées respectivement l'*héritage ordinaire* et l'*héritage virtuel* en C++ [STR 97]. Si $X \to Y_1 \to Z$ et $X \to Y_2 \to Z$, dans la relation d'héritage ordinaire Z hérite de X de deux façons différentes, alors que dans la relation d'héritage virtuelle Z hérite de X d'une seule façon. Donnons un nom aux flèches d'héritage : $X \xrightarrow{f_1} Y_1 \xrightarrow{g_1} Z$ et $X \xrightarrow{f_2} Y_2 \xrightarrow{g_2} Z$. D'un point de vue catégorique, il y a deux flèches composées $X \xrightarrow{g_1.f_1} Z$ et $X \xrightarrow{g_2.f_2} Z$. Si rien de plus n'est dit, elles sont distinctes, cela correspond à l'héritage ordinaire. Mais si l'égalité $g_1.f_1 = g_2.f_2$ est ajoutée, alors cela correspond à l'héritage virtuel.

1. **Attention !** *Cette flèche va dans le sens opposé de la plupart des approches diagrammatiques, comme en UML ou dans [STR 97] ; les raisons de ce choix seront expliquées à la section 3.*

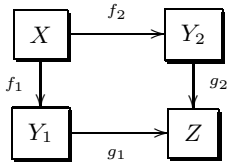
Désormais, selon la terminologie de C++, lorsque la classe A hérite de la classe B, nous disons que A est une classe *dérivée* (ou sous-classe) de B et que B est une classe de *base* (ou super-classe) pour A. La classe de base peut être une *interface*, au sens de Java, c'est-à-dire en C++ une classe abstraite dont *toutes* les méthodes sont virtuelles pures. L'idée est de construire une interface que les classes dérivées doivent respecter ("mandatory methods") ; ce type d'héritage est utilisé à la section 3.6. L'héritage peut aussi être une *extension*, où les classes dérivées ajoutent de nouvelles fonctionnalités ou de nouveaux membres à la classe de base, voir par exemple [TAI 96] pour plus de détails sur l'héritage. Dans les deux cas, la classe dérivée ajoute quelque chose à la classe de base ; dans le premier cas, elle ajoute seulement des implémentations.

2.3. *Pushouts*

Soit $\mathcal{C}$ une catégorie. Un *span* Sp dans $\mathcal{C}$ est formé de deux flèches de même source :

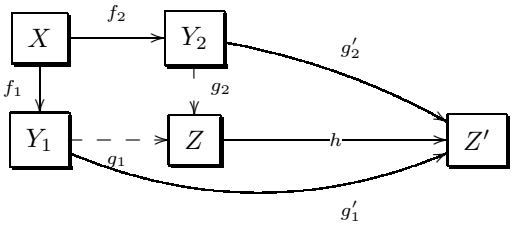


Un *cône* Co de *base* Sp dans $\mathcal{C}$ est formé d'un span Sp, d'un point Z, appelé le *sommet* de Co, et de deux flèches $g_1 : Y_1 \to Z$, $g_2 : Y_2 \to Z$, appelées les *coprojections* de Co, de façon que $g_1.f_1 = g_2.f_2$ (il s'agit donc d'un *diagramme commutatif*) :



Le pushout d'un span Sp est défini ci-dessous comme un cône de base Sp qui satisfait une condition d'*initialité* (il s'agit donc d'un diagramme commutatif "*minimal*"). Dans cet article, les coprojections d'un pushout sont représentées par des flèches tiretées.

Definition 2 Un *pushout* de base Sp est un cône Co de base Sp tel que, pour chaque chaque cône Co' de même base Sp, il y a une unique flèche $h : Z \to Z'$ telle que $h.g_1 = g'_1$ et $h.g_2 = g'_2$:



Un span Sp ne peut avoir qu'un seul pushout (à isomorphisme près), on l'appelle *le* pushout de base Sp. Le point X est appelé le *point de recollement* de Sp. Ce pushout signifie que Z est obtenu en "recollant Y_1 et Y_2 le long de l'image de X". La condition d'initialité affirme que tout ce qui figure dans Z provient de Y_1 ou (non exclusivement) de Y_2.

Supposons qu'il y a une notion "raisonnable" de *modèles*, (ou d'*interprétations*, ou d'*instances*) pour chaque objet de la catégorie $\mathcal{C}$, comme ce sera le cas par la suite. Alors, dans un tel pushout, les modèles de la spécification Z peuvent être décrits comme les couples de modèles de Y_1 et Y_2 qui coïncident sur leur interprétation du point de recollement X. En termes catégoriques, cela signifie que $\mathcal{M}od(Z)$ est le *pullback* (ou *produit fibré*) de $\mathcal{M}od(Y_1)$ et $\mathcal{M}od(Y_2)$ au-dessus de $\mathcal{M}od(X)$.

3. DML : un Langage de Modélisation Diagrammatique

3.1. *Une catégorie pour DML*

Dans le but de modéliser la structure d'un logiciel C++, une catégorie $\mathcal{C}_{dml}$ est décrite maintenant, de façon informelle ; une définition plus précise de la catégorie $\mathcal{C}_{dml}$ nécessiterait une étude plus détaillée. La catégorie $\mathcal{C}_{dml}$ fournit une vision synthétique d'une architecture C++, en ne faisant pas de distinction entre classe et instance, par exemple. C'est cette vision qui va permettre d'obtenir une description simple des mécanismes complexes mis en jeu dans le logiciel LINBOX. Il serait possible d'affiner cette approche, afin de distinguer les classes des instances etc, mais ce n'est pas le but de cet article.

Les points de la catégorie $\mathcal{C}_{dml}$ sont appelés les *spécifications*. Parmi les spécifications figurent tous les *types* de C++, aussi bien les types prédéfinis que les classes, et aussi les "typenames". Une spécification peut aussi être une valeur dans un type prédéfini ou une instance d'une classe. Cela peut aussi être une variable représentant un type, une valeur ou une instance. Une spécification A détermine un ensemble de *modèles* $\mathcal{M}od(A)$. Typiquement, si la spécification est une classe A, ses modèles sont les instances de la classe A, et si la spécification est une instance a, son unique modèle est lui-même. Ainsi, on peut voir une spécification soit du point de vue *syntaxique*, comme un morceau de code C++, soit du point de vue *sémantique*, comme un ensemble de modèles.

Les flèches de la catégorie $\mathcal{C}_{dml}$ sont appelés les *morphismes de spécifications*. Ces morphismes regroupent des relations de nature variée entre spécifications. Un morphisme $\varphi : A \to B$ permet de considérer le code de A comme une partie du code de B. Par exemple, un morphisme de spécifications peut être un *héritage*, entre deux classes. Quand B hérite de A, la classe B contient tous les membres (attributs et méthodes) de la classe A, plus quelques nouveaux, il s'agit donc d'un morphisme $\varphi : A \to B$. Ce morphisme induit une fonction $\mathcal{M}od(\varphi) : \mathcal{M}od(B) \to \mathcal{M}od(A)$, en oubliant l'interprétation des membres de B qui ne sont pas membres de A. Pour cette raison, il est souvent illustré par une flèche *de B vers A* : c'est le cas en UML, par exemple, mais dans cet article l'orientation $\varphi : A \to B$ est toujours choisie. Une *paramétrisation générique* (ou "*template*") est aussi un morphisme de spécifications. Quand une classe générique T est un paramètre générique pour une classe B, les membres de T peuvent être utilisés dans B, donc il y a un morphisme $T \to B$. Une *instanciation* est une autre espèce de morphisme de spécifications. Quand un objet a

est créé comme instance d'une classe A, les membres de A sont instanciés dans a, ce qui peut être vu comme un morphisme $A \to a$. Une *implémentation* d'une classe abstraite A par une classe B est aussi un morphisme $A \to B$.

Les spécifications peuvent être construites progressivement, par des constructions systématiques, grâce aux pushouts. Dans la section suivante, nous montrons que les pushouts de la catégorie $\mathcal{C}_{dml}$ correspondent à des constructions fondamentales en C++ : l'héritage virtuel est détaillé en section 3.2, le passage de paramètres en section 3.3, le passage de paramètres template en section 3.4, l'instanciation d'objets est décrite en section 3.5 et le polymorphisme en section 3.6. Plusieurs applications à la bibliothèque LINBOX sont données à la section 4.

3.2. *Héritage virtuel*

Dans le cas de l'héritage multiple de deux classes dérivant d'une même troisième classe mère, l'héritage virtuel donne lieu à des cônes, comme expliqué dans la section 2.2. Un tel cône est un pushout si et seulement si la classe doublement dérivée Z est "minimale", dans le sens où Z n'a aucun membre additionnel autre que ceux hérités de Y_1 et Y_2. Le code C++ correspondant est comme suit :

```
struct X {void m0(){...} };
struct Y1: public virtual X {void m1(){...} };
struct Y2: public virtual X {void m2(){...} };
struct Z: public virtual Y1, public virtual Y2 { };
```

Alors, les méthodes m_1, m_2 et une méthode m_0, sont héritées dans Z. Toutefois, quand une méthode m_i est un constructeur, elle doit apparaître explicitement dans la classe dérivée puisqu'elle n'est pas héritée en C++. Il s'agit toujours d'un pushout, en dépit de cet ajout.

3.3. *Passage de paramètre*

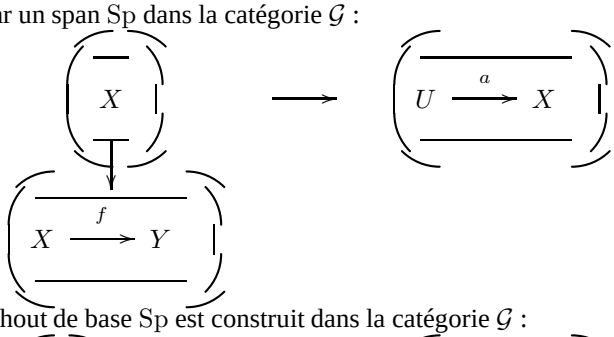
La formalisation de l'héritage multiple par un pushout, comme décrit ci-dessus, est un exemple d'utilisation symétrique des pushouts où les deux flèches du span sont de même nature. Dans cet article nous nous intéressons également aux différentes utilisations dissymétriques des pushouts [BAR 90]. Le paradigme des constructions des sections suivantes est le paradigme du passage de paramètre, comme décrit ci-dessous (voir aussi [EHR 85]).

Étant donnée une expression $f(x)$ et une valeur a pour x, le passage de paramètres construit l'expression $f(a)$. Ici $f : X \to Y$ est une fonction, $x : X$ est un symbole appelé le *paramètre formel*, et $a : X$ est une constante appelée le *paramètre réel*, de sorte que le résultat $f(a) = f.a : Y$ est une constante. Le paramètre et le résultat sont considérés comme des fonctions constantes $a : U \to X$ et $f(a) = f.a : U \to Y$, où U est le type *unité*, ou "void" en C++, qui est interprété comme un singleton. Ainsi,

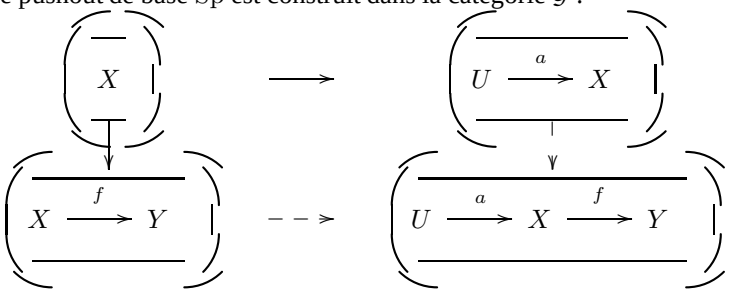
le passage de paramètre est vu comme une application de la règle de composition des flèches :

$$\frac{a : U \to X \qquad f : X \to Y}{f.a : U \to Y}$$

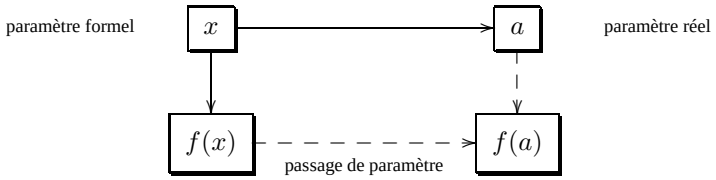
Le pushout est utilisé pour construire une occurrence des prémisses de cette règle. La catégorie où le pushout est formé est la catégorie $\mathcal{G}$ des (multi-)graphes (orientés). D'abord la donnée, faite de $f : X \to Y$ et $a : U \to X$, avec le même type X, est représentée par un span Sp dans la catégorie $\mathcal{G}$:



Ensuite le pushout de base Sp est construit dans la catégorie $\mathcal{G}$:

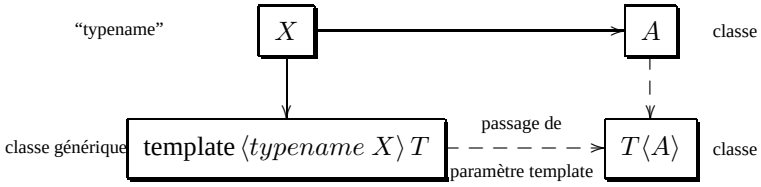


Le sommet de ce pushout est une instance des prémisses de la règle de composition. Ensuite, la constante $f.a$ est obtenue en appliquant cette règle. Ce point de vue sur les règles de déduction est détaillé dans [DUV 02, DUV 03]. Donc, schématiquement, le passage de paramètre correspond au pushout suivant :



3.4. *Généricité et types paramétrés*

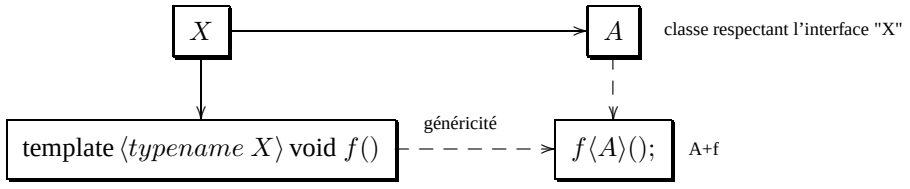
En C++, une classe peut jouer le rôle de paramètre pour construire une nouvelle classe, grâce au mécanisme de passage de paramètre *template*. Cela fonctionne comme le passage de paramètres classique, par conséquent le passage de paramètre template peut lui aussi être formalisé par un pushout de spécifications :



Ici, X est le nom d'un paramètre template formel, qui est utilisé dans la définition de la classe générique T. La classe A est le paramètre réel, et le sommet du pushout est la classe $T\langle A \rangle$ construite par ce mécanisme. Une fonction générique peut être construite de la même manière, avec une classe A respectant l'interface X :

```
template<typename X> void f() { X::g(); }
struct A { static void g() {} };
```

En C++, le pré-processeur gère automatiquement les classes ou fonctions paramétrées statiquement en recopiant le corps générique pour chaque paramètre template avant compilation. Cet automatisme est donc représentable par un pushout :

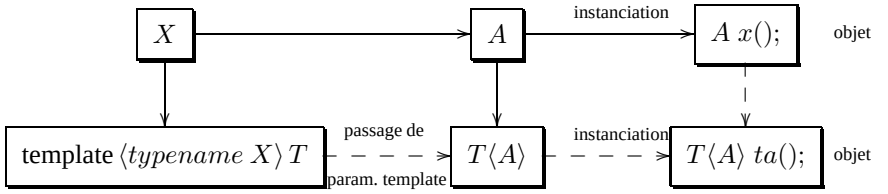


3.5. *Instanciation*

L'instanciation d'un objet peut être obtenue par un pushout de spécifications, de différentes façons. Prenons par exemple une classe paramétrée contenant un membre de sa classe paramètre :

```
template<typename X> struct T {
    X x;
    T () : x() {} // Le constructeur de T appelle celui de X
};
```

Ainsi, le morphisme vu en section 3.4, de la classe A vers la classe $T\langle A \rangle$, peut être utilisé pour construire une instance de $T\langle A \rangle$. Deux pushout sont alors composés. En effet la construction d'une instance `ta` fait alors appel au constructeur vide de la classe A, `A()` qui construit une instance de la classe A, comme suit :



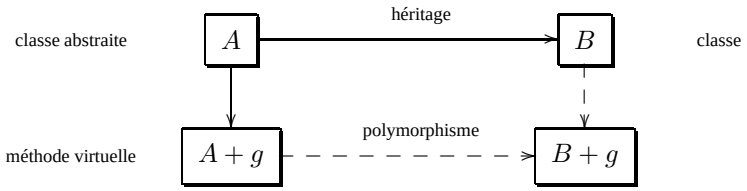
Ce pushout fournit une instance ta de $T\langle A\rangle$, en utilisant le constructeur vide de T, qui appelle le constructeur vide de A.

3.6. *Polymorphisme*

Dans cette section nous proposons de voir le polymorphisme d'un langage orienté objet (polymorphisme utilisant l'héritage ou polymorphisme d'inclusion) comme un pushout. Selon [STR 97, §12.2.6], le polymorphisme orienté objet fonctionne comme suit en C++ : les méthodes appelées doivent être *virtuelles* et les objets doivent être manipulés par l'intermédiaire de pointeurs ou de références. Ainsi, le polymorphisme est obtenu quand une classe dérivée est utilisée via un pointeur vers la classe de base, et quand cette classe de base contient des méthodes virtuelles, par exemple elle peut être une classe abstraite. L'idée est d'écrire les algorithmes sur la classe de base et de leur passer ensuite des valeurs de la classe dérivée. Voici un exemple en C++ :

```
#include <iostream>
struct A { virtual void f() = 0; }; // classe abstraite
void g(A * a) { a->f(); }           // fonction globale
// la classe dérivée ajoute une implémentation de la méthode f.
struct B : public A { void f() { std::cout << "f de B"; } };
int main() {
    B b; g( &b );  // un pushout est utilisé
    return 0;
}
```

Dans cet exemple, la classe A est une classe abstraite, avec une méthode virtuelle f ; d'un côté, la classe B hérite de A, et ajoute une implémentation de f ; de l'autre côté, la fonction g est implémentée à partir de la seule connaissance de l'interface de f, décrite dans A. Ensuite, un pointeur vers un objet b de type B peut être passé en argument à cette fonction g. Le pushout correspondant "recolle" A et g d'un côté, et la classe dérivée B de l'autre côté, avec la classe abstraite A comme point de recollement :



4. Application à la bibliothèque d'algèbre linéaire LINBOX

LINBOX est une bibliothèque générique C++ d'algèbre linéaire. Elle a été conçue pour travailler sur des matrices à coefficients dans différents domaines, en particulier sur des corps ou des anneaux finis et sur les entiers ou les rationnels. La performance et la généricité sont les deux buts de LINBOX. La généricité est obtenue par l'utilisation

d'un ensemble restreint d'interfaces. Les algorithmes sont implémentés en C++ avec des paramètres génériques (template), qui peuvent être instanciés par toute classe respectant l'interface spécifiée. Les performances sont atteintes par des spécialisations judicieuses (éventuellement partielles) des algorithmes génériques. Il est tout à fait dans l'esprit du projet d'introduire de nouvelles implémentations. Ainsi un utilisateur peut invoquer un algorithme LINBOX, par exemple le calcul du déterminant ou du rang d'une matrice, qui travaillera sur une matrice en boîte noire de sa composition, ou sur une implémentation de corps fini qui lui est propre. Inversement, les interfaces de corps LINBOX et leurs différentes représentations spécifiques peuvent être utilisées par ailleurs, en dehors de LINBOX. Afin de résoudre simultanément les problèmes de généricité et de performance, une structure relativement complexe a été créée pour l'arithmétique. Il s'agit d'un ensemble de cinq classes que chaque domaine de calcul LINBOX doit respecter. Ce système est extrêmement efficace [TUR 02, Table 5.1], mais son utilisation et sa description sont assez techniques. Rappelons que le but de cet article est de donner une vision abstraite de cette architecture, grâce à DML, pour obtenir une interface utilisateur simple. Nous nous intéressons plus particulièrement au cas des corps mais des structures similaires existent, pour des anneaux par exemple.

4.1. *Quatre classes pour l'archétype des corps*

4.1.1. *Une classe "Field" pour les algorithmes*

Les algorithmes de LINBOX doivent fonctionner sur une grande variété de domaines, et en particulier sur des corps ou des anneaux finis. Pour pouvoir calculer, un algorithme peut avoir besoin de paramètres supplémentaires, comme le module `p` (un nombre premier) pour travailler sur le corps des entiers modulo `p`. Pour cela, LINBOX requiert un objet pour chaque corps et une classe pour les éléments. L'objet corps définit ses méthodes (i.e. les opérateurs arithmétiques) qui contiennent le code de calcul en lui-même. Une instance `F` de la classe `Field` correspond donc à un corps et ses paramètres. Une classe interne supplémentaire `Field::Element` doit être fournie pour le stockage des éléments du corps : par exemple, l'appel `F.add(x,y,z)` ajoute les éléments `y` et `z` dans le corps `F` et stocke le résultat dans l'élément `x`. Ce type `Field::Element` peut être un `longint` de C++ pour le corps des entiers modulo un nombre premier de taille le mot machine, ou une structure plus complexe.

4.1.2. *Une classe abstraite "FieldAbstract" pour la généricité*

Les corps doivent respecter une interface commune. Ceci est réalisé par un héritage commun à une classe abstraite (virtuelle pure), `FieldAbstract`.

4.1.3. *Une classe archétype "FieldArchetype" pour contrôler l'explosion de code exécutable*

En C++, le compilateur gère les classes ou fonctions paramétrées statiquement en recopiant le corps générique pour chaque paramètre `template` avant compilation. La multiplicité des niveaux de généricité en LINBOX induit donc un besoin de contrôle

de l'explosion de code exécutable ("code bloat"). La solution développée dans LINBOX est de définir une classe non-générique, `FieldArchetype` [DUM 02] (voir par exemple [VEL 00] pour d'autres alternatives). Cette classe contient un pointeur vers un objet de type `FieldAbstract`. Si la quantité de code exécutable produit rend nécessaire une réduction de celui-ci, les archétypes sont compilés séparément et linéairement. Ainsi, ce n'est pas la classe abstraite seule qui joue le rôle d'interface. Ce prototype [TUR 02, KAL 05], joue donc exactement trois rôles dans la bibliothèque : il décrit l'interface commune, contrôle l'explosion de code exécutable et fournit une instance compilée de code exécutable. Cependant, bien que le polymorphisme puisse être utilisé directement, il induit un surcoût principalement dû au déréférencement des pointeurs. Ce surcoût est prohibitif quand il est appliqué à chacune des opérations arithmétiques de base. Afin de conserver les performances de LINBOX à ce niveau, il faut alors abandonner le contrôle de l'explosion de code. Séparer l'archétype de la classe abstraite, dans le cas des corps finis [DUM 02, Fig. 1], permet alors de choisir entre les meilleures performances sans les archétypes, ou un meilleur contrôle du code exécutable au prix d'une baisse des performances [TUR 02, Table 5.4].

4.1.4. *Une classe enveloppe générique "FieldEnvelope" pour les domaines externes*

Deux problèmes découlent de l'organisation qui précède. Tout d'abord, tous les corps, même provenant d'une bibliothèque externe par exemple, doivent hériter de la classe abstraite. Ensuite, les constructeurs ne peuvent pas être hérités en C++. Il est donc impossible de forcer la déclaration de constructeurs dans une classe dérivée par rapport à un constructeur qui serait virtuel pur dans une classe de base abstraite. Une solution est d'ajouter une méthode virtuelle `clone` qui remplit le rôle de constructeur dans la classe abstraite. Cela induit donc une différence d'interface entre la classe abstraite et l'archétype. Ces deux problèmes d'héritage et d'interface sont résolus par la création d'une classe additionnelle `FieldEnvelope` que nous décrivons précisément ici. Ainsi, pour tout corps, même externe, son enveloppe héritera de la classe abstraite.

Une enveloppe est un "wrapper" générique ("adaptor" [WIS 96]) respectant l'interface de l'objet qu'elle enveloppe. En C++, nous distinguons plusieurs variantes de cette conception. Toutes sont des structures template dépendant d'un paramètre template B, par exemple le corps enveloppé.

1) *Enveloppe sans héritage :* la classe enveloppe est reliée à B par le type d'un de ses membres soit directement, soit via un pointeur. Dans une *enveloppe à recopie*, le paramètre template B est le type d'un membre de l'enveloppe.

```
template <typename B> struct Env {
private:   B _b;
};
```

Dans une *enveloppe à pointeur* (sans copie), c'est $B*$, un pointeur sur l'objet d'origine, qui est stocké à la place de B.

```
template <typename B> struct Env {
private:   B* _b;
};
```

2) *Enveloppe à héritage :* l'enveloppe hérite en outre de son paramètre template. Toutes les caractéristiques du paramètre sont conservées par héritage, à l'exception des constructeurs et destructeurs. Le polymorphisme est donc également préservé.

```
template <typename B> struct Env : public B;
```

Une enveloppe ajoute deux fonctionnalités. Premièrement, une enveloppe donne un héritage interne à des classes externes inaltérables : avec l'héritage multiple, une classe externe associée à une enveloppe peut néanmoins utiliser du polymorphisme.

```
// Toute classe  Env<B> hérite d'une classe de base A.
template <typename B> struct Env : public B, public A;
```

Deuxièmement, une enveloppe permet l'abstraction de méthodes génériques. En effet, il est conceptuellement impossible de définir une classe abstraite avec des méthodes template : le mécanisme de table virtuelle ne peut pas résoudre un appel de méthode template virtuelle, puisque la méthode effective n'est pas connue à la création de l'objet. La structure d'enveloppe peut alors émuler une classe abstraite et forcer l'interface du paramètre template.

```
template <typename B> struct Env : public B {
   template <typename V> void mygenericmethod(V a) {
        // Cette méthode est requise pour tout B,
        // même si B est générique (sinon le code ne compile pas)
        B::mygenericmethod(a);
   }
};
```

Enfin, grâce à la généricité de C++, les enveloppes n'induisent aucun surcoût et garantissent l'efficacité originale de l'objet enveloppé [GIO 04, Table 2.6]

4.2. *Enveloppes en Java*

Ce formalisme d'enveloppes n'est pas restreint à C++. En Java, par exemple, il est possible d'avoir une construction similaire. D'abord, une classe externe est fournie, indépendante d'une bibliothèque donnée :

```
public class External {
        int a;
        External(int a) {a = a;}
        public void aMethod() { System.out.println("a: " + a);}
}
```

Ensuite, une classe abstraite est définie :

```
public interface Abstract{
        public void theMethod();
}
```

Et finalement, l'enveloppe `EnvelopeInherit` implémente `Abstract` et étend `External` :

```
public class EnvelopeInherit extends External implements Abstract{
```

```
        EnvelopeInherit(int a) {super(a);}
        public void theMethod() {super.aMethod();}
}
```

La classe `EnvelopeInherit` est donc une `External` qui implémente l'interface interne `Abstract`. Les enveloppes à recopie ou à pointeur peuvent se définir de la même manière (elles sont indifférenciées en Java puisque tout membre est un pointeur). Ici, l'enveloppe ne sert que pour un seul type de classe externe. En C++, grâce à la généricité, une seule enveloppe est nécessaire pour toutes les classes respectant une même interface implicite. Il est à noter que Java 1.5 introduit également une notion de généricité[BRA 98] ; la transposition des `template` C++ doit donc pouvoir se faire également en Java.

4.3. *Une description en DML de l'architecture des domaines de* LINBOX

Dans le but de visualiser les différentes classes qui sont utilisées par LINBOX pour représenter des corps, nous utilisons le Langage de Modélisation Diagrammatique DML de la section 3. Les spécifications utilisées par LINBOX pour calculer sur un corps (ici le corps à deux éléments), à l'exception de la classe `Field::Element`, sont représentées sur le diagramme de la figure 1.

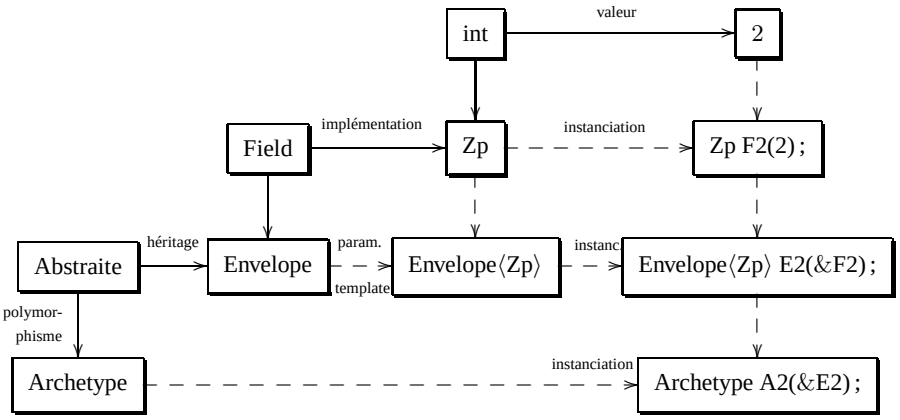


Figure 1. *Un diagramme DML pour l'architecture des corps dans* LINBOX*, avec une enveloppe à recopie*

Ce diagramme peut être lu de la gauche vers la droite, c'est-à-dire de l'abstraction vers les instances. À l'exception de la première ligne, les spécifications (les boîtes) de la colonne de droite sont des objets, et les autres sont des classes (`Zp` dénote une classe de corps finis premiers), et les flèches horizontales les plus à droite sont des instanciations. La première ligne est légèrement différente des autres : elle comporte un type prédéfini `int` à la place d'une classe, et une valeur `2` de type `int` au lieu d'une

instance. Les trois objets `F2`, `E2` et `A2` représentent tous les corps à deux éléments, de trois points de vue différents. Les trois pushouts à droite, de sommets `F2`, `E2` et `A2`, construisent des instanciations, comme à la section 3.5. Le pushout du milieu, de sommet `Envelope`⟨`Zp`⟩, correspond à un passage de paramètre template, comme à la section 3.4. La flèche horizontale de `Abstract` vers `Envelope`⟨`Zp`⟩ `E2(&F2);` est composée de trois morphismes de nature différente : d'abord un héritage, ensuite un passage de paramètre template, et finalement une instanciation. Dans ce diagramme, l'enveloppe est à recopie : la construction de l'enveloppe `E2` nécessite une copie du corps `F2`. Par contre le diagramme suivant, figure 2, comporte un enveloppe à héritage.

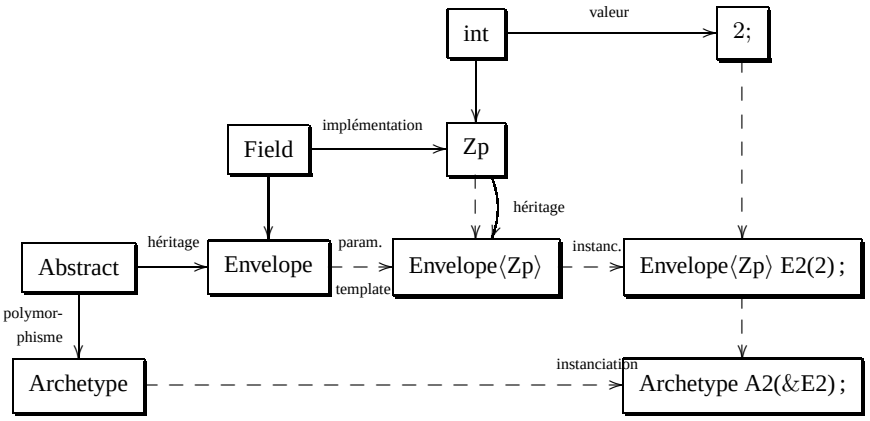


Figure 2. *Un diagramme DML pour l'architecture des corps dans* LINBOX*, avec une enveloppe à héritage*

Il apparaît clairement que la différence entre les deux diagrammes est l'instanciation du corps `E2`. En effet, la construction du corps `F2` n'est plus requise, elle est faite automatiquement pendant la construction de l'enveloppe. La figure 2 montre qu'il est encore possible d'instancier `Zp(2)` si c'est utile ailleurs, mais maintenant c'est fait automatiquement lors de la construction de l'enveloppe.

La construction des archétypes et des enveloppes en LINBOX découle de particularités de C++ et de la volonté de conserver une efficacité maximale. Néanmoins, cet effort de conciliation de généricité et d'efficacité semble mener pour tout language à un ensemble d'objets ou de modules interagissants de manière complexe. Un autre exemple est le logiciel de topologie algébrique de F. Sergeraert, écrit en Common Lisp [RUB 98, LAM 03].

5. Conclusion

Dans cet article nous avons proposé un nouveau langage de modélisation diagrammatique, DML. Le paradigme utilisé est la théorie des catégories et en particulier la

construction de pushout. Nous avons montré que de nombreuses structures orientées objet peuvent être décrites grâce à cet outil et nous avons décrit plusieurs exemples en C++, incluant aussi bien l'héritage virtuel et le polymorphisme que la généricité par template. La puissance de cet outil nous a permis de proposer une description simple de la bibliothèque LINBOX. Cette bibliothèque a été écrite dans un souci d'efficacité et de généricité, qui a nécessité l'utilisation de mécanismes complexes de "template" et de polymorphisme. Par une approche de rétro-ingénierie, nous avons réussi à décrire la structure fondamentale des archétypes de LINBOX. Cette structure contient plusieurs classes, génériques ou non, polymorphiques ou non, et notre description utilise quatre pushouts. Nous pensons que notre description par pushouts est suffisamment claire pour permettre une meilleure compréhension du comportement des archétypes. Il est probable que le succès de notre approche tient au fait que LINBOX est une bibliothèque de calcul formel, qui repose sur des bases mathématiques stables (l'algèbre linéaire) et une utilisation limitée des multiples possibilités sémantiques de C++. Cependant, d'autres mécanismes de la programmation orientée objet peuvent être appréhendés par des méthodes diagrammatiques, en particulier par des pushouts, et des extensions de DML sont envisageables. Ces extensions pourraient concerner, par exemple, l'étude de la surcharge des fonctions, en tenant compte du fait que la redéfinition des types d'arguments mélange les liaisons statique et dynamique. Ou encore une étude du mécanisme des exceptions [DUV 05], voire une analyse de l'exécution d'un programme [DUV 94], Par la suite, nous envisageons de construire des outils graphiques pour manipuler les diagrammes et pour engendrer des squelettes orientés objet. Des prototypes de ces outils ("Dessiner les Calculs" pour représenter des calculs par des diagrammes et "SketchUML" pour relier notre approche à UML) sont accessibles à partir de la page du projet InCa : `http://www-lmc.imag.fr/MOSAIC/InCa`.

Ce travail a été effectué avec le soutien de l'Institut d'Informatique et de Mathématiques Appliquées de Grenoble, projet InCa.

6. Bibliographie

[BAR 90] BARR M., WELLS C., *Category Theory for Computer Science*, International Series in Computer Science, Prentice Hall, 1990.

[BRA 98] BRACHA G., ODERSKY M., STOUTAMIRE D., WADLER P., « Making the Future Safe for the Past – Adding Genericity to the Java(TM) Programming Language », *Proceedings of OOPSLA'98, ACM SIGPLAN Notices*, Montreal, Canada, 1998, p. 183–200.

[BUR 77] BURSTALL R., GOGUEN J., « Putting theories together to make specifications », *Proc. 5th Internat. Joint Conf. on Artificial Intelligence*, 1977, p. 1045–1058.

[DUM 02] DUMAS J.-G., GAUTIER T., GIESBRECHT M., GIORGI P., HOVINEN B., KALTOFEN E., SAUNDERS B. D., TURNER W. J., VILLARD G., « LinBox : A Generic Library for Exact Linear Algebra », COHEN A. M., GAO X.-S., TAKAYAMA N., Eds., *Proceedings of the 2002 International Congress of Mathematical Software, Beijing, China*, World Scientific Pub, août 2002, p. 40–50.

[DUV 94] DUVAL D., REYNAUD J.-C., « Sketches and computation I : Basic Definitions and Static Evaluation », *Mathematical Structures in Computer Science*, vol. 4, n° 2, 1994,

p. 185-238.

[DUV 02] DUVAL D., LAIR C., « Diagrammatic Specifications », Rapport de recherche 1043 m, janvier 2002, IMAG-LMC.

[DUV 03] DUVAL D., « Diagrammatic Specifications », *Mathematical Structures in Computer Science*, vol. 13, n° 6, 2003, p. 857–890.

[DUV 05] DUVAL D., REYNAUD J.-C., « Diagrammatic logic and exceptions : an introduction », *Dagstuhl Seminar Proceedings – MAP05, Mathematics, Algorithms, Proofs*, 2005.

[EHR 85] EHRIG H., MAHR B., *Fundamentals of Algebraic Specifications 1 : Equations and Initial Semantics*, vol. 6, Springer, 1985.

[GAM 94] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison Wesley, Massachusetts, 1994.

[GIO 04] GIORGI P., « Arithmétique et algorithmique en algèbre linéaire exacte pour la bibliothèque LINBOX », PhD thesis, École normale supérieure de Lyon, décembre 2004.

[GOG 73] GOGUEN J., « Categorical foundations for general systems theory », *Advances in Cybernetics and System Research*, Transcripta Books, 1973, p. 121–130.

[KAL 05] KALTOFEN E., MOROZOV D., YUHASZ G., « Generic matrix multiplication and memory management in LinBox », KAUERS M., Ed., *Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation, Beijing, China*, ACM Press, New York, juillet 2005.

[LAM 03] LAMBÁN L., PASCUAL V., RUBIO J., « An Object-oriented Interpretation of the EAT System », *Applicable Algebra in Engineering, Communication and Computing*, vol. 14, n° 3, 2003, p. 187–215.

[Mac 97] MAC LANE S., *Categories for the Working Mathematician*, vol. 5 de *Graduate Texts in Mathematics*, Springer-Verlag, New York, 2nd édition, 1997, (1st ed., 1971).

[MUL 00] MULLER P.-A., GAERTNER N., *Modélisation objet avec UML*, Eyrolles, 2000.

[MUS 96] MUSSER D. R., SAINI A., *STL Tutorial and Reference Guide : C++ Programming with the Standard Template Library*, Addison-Wesley, Reading (MA), USA, 1996.

[ORI 00] ORIAT C., « Detecting equivalence of modular specifications with categorical diagrams », *TCS*, vol. 247, n° 1–2, 2000, p. 141–190.

[RUB 98] RUBIO J., SERGERAERT F., SIRET Y., « Overview of EAT », *Symbolic and Algebraic Computation (SAC) Newsletter*, vol. 3, 1998, p. 69–79.

[SRI 95] SRINIVAS Y., JÜLLIG R., « Specware Language Manual », 1995.

[STR 97] STROUSTRUP B., *The C++ Programming Language : Third Edition*, Addison-Wesley Publishing Co., Reading, Mass., 1997.

[TAI 96] TAIVALSAARI A., « On the Notion of Inheritance », *ACM Computing Surveys*, vol. 28, n° 3, 1996, p. 438–479.

[TUR 02] TURNER W. J., « Blackbox linear algebra with the LinBox library », PhD thesis, North Carolina State University, mai 2002.

[VEL 00] VELDHUIZEN T. L., « Five compilation models for C++ templates », *First Workshop on C++ Template Programming, Erfurt, Germany*, octobre 2000.

[WIS 96] WISE G. B., « An overview of the standard template library », *SIGPLAN Not.*, vol. 31, n° 4, 1996, p. 4–10, ACM Press.