Composants génériques de calcul scientifique

T. Géraud et A. Duret-Lutz

RAPPORT TECHNIQUE 9901
MARS 1999

Laboratoire de Recherche et Développement d'EPITA
14-16, rue Voltaire – 94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 44 08 01 01 – Fax +33 1 44 08 01 99

Document

Composants génériques de calcul scientifique
T. Géraud et A. Duret-Lutz
Rapport technique 9901
Mars 1999

Laboratoire de Recherche et Développement d'EPITA
14-16, rue Voltaire – 94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 44 08 01 01 – Fax +33 1 44 08 01 99

Résumé

Dans le cadre de l'écriture en C++ d'une bibliothèque de traitements d'images et d'un atelier de programmation par composants d'une chaîne de traitements, nous nous sommes fixés deux objectifs principaux : rendre les traitements génériques vis-à-vis du type de ses entrées sans entraîner de surcoût significatif à l'exécution, et pouvoir exécuter un traitement lorsqu'il a été introduit ultérieurement à la compilation de l'atelier.

Le premier objectif est atteint à l'aide de programmation générique et de la traduction en polymorphisme statique de certains idiomes (*design patterns*) définis pour le polymorphisme dynamique. La problématique du second objectif est double. Tout d'abord, nous devions réaliser la mise en correspondance d'un traitement dont les entrées-sorties sont des types abstraits et de la routine générique, chargée du traitement, dont les paramètres sont des types concrets ; ensuite, nous devions pouvoir compiler et lier de nouveaux traitements à la volée, lors de l'exécution de l'atelier. Pour atteindre ce double objectif, nous utilisons de la programmation générative et nous pratiquons l'instanciation paresseuse (*lazy*) de code générique.

Les solutions que nous apportons permettent la gestion de composants réutilisables de calcul scientifique, ce qui, à notre connaissance, est original.

Mots-clefs

Calcul numérique, programmation générique, programmation par composants.

1 Introduction

Dans le domaine du traitement d'images, ou plus généralement de la vision par ordinateur, aucune bibliothèque n'a été à ce jour suffisamment fédératrice pour être adoptée par de nombreux laboratoires. Plusieurs efforts ont été menés dans ce sens mais tous se sont heurtés à une même difficulté : la grande diversité des problèmes abordés dans ce domaine nécessite une généricité totale des outils proposés.

Nous devons nous affranchir de deux facteurs : le type des structures de données et le type-même des données. Suivant les domaines, les traitements doivent s'appliquer à des images 2D ou 3D, isotropes ou non, à des séquences temporelles d'images, à des pyramides multi-échelles ou multi-résolutions d'images, à des régions ou des graphes d'adjacences de régions, etc. Quant aux données, elles peuvent être scalaires ou booléennes, entières ou flottantes, complexes, vectorielles, ou encore composées (comme dans le cas de la couleur) et chaque composante peut avoir son propre codage, etc.

Un véritable enjeu est donc de proposer un jeu d'outils qui soit générique vis-à-vis du type des structures de données et du type des données, mais qui soit également performant et de taille raisonnable. En effet, l'obtention de la généricité ne doit pas se traduire par un surcoût significatif lors de l'exécution d'un traitement ; de plus, il n'est pas question de compiler l'ensemble des triplets *traitement* $\times$ *structure de données* $\times$ *données* car sa cardinalité est trop élevée et, du point de vue d'un groupe d'utilisateurs, seul un petit sous-ensemble est intéressant.

Dans cet article, nous présentons les solutions logicielles que nous avons développées. La section 2 décrit l'obtention de routines de calcul par programmation générique et montre comment le polymorphisme dynamique de certains idiomes est traduit en polymorphisme statique. La section 3 décrit deux mécanismes : celui qui permet de mettre en correspondance un composant dont les entrées et sorties sont de type abstrait avec la routine de calcul dont les paramètres sont de types concrets, et celui qui permet de compiler à la volée une routine lorsqu'elle n'a pas encore été instanciée.

Les solutions que nous avançons ne sont pas spécifiques à la problématique du traitement d'images ; elles sont directement généralisables au domaine non exploré du calcul scientifique générique par composants.

NOTA BENE : LE CODE PRÉSENTÉ DANS CE DOCUMENT EST VOLONTAIREMENT SIMPLIFIÉ POUR UNE PLUS GRANDE CLARTÉ DES EXEMPLES.

2 Généricité des routines de calcul

Un traitement se traduit par une routine de calcul qui nécessite un certain nombre d'outils. Les types des entrées et sorties du traitement conditionne non seulement la routine mais également les outils.

2.1 Exemple de traitement

Pour illustrer nos propos, considérons un traitement d'images classique, le filtre moyenneur. Sa description, donnée d'ores et déjà sous une forme abstraite, peut s'énoncer comme suit :

pour chaque site d'un agrégat d'entrée,
on calcule la moyenne des valeurs sur le voisinage de ce site
puis on affecte cette valeur au site correspondant de l'agrégat de sortie.

Dans les bibliothèques de traitement d'images, le paradigme le plus avancé que l'on puisse trouver est la généricité vis-à-vis des types de données [1] ; l'écriture de la routine de calcul, dans le cas d'images bidimensionnelles, est alors similaire à celui qui suit [6, 8].

```
template< typename T >
void mean( Image2D<T>& imageIn, Image2D<T>& imageOut )
{
   for ( int iRow = 0; iRow < imageIn.nRows; ++iRow )
      for ( int iColumn = 0; iColumn < imageIn.nColumns; ++iColumn )
      {
         T sum = 0;
         for ( iNeighbor = 0; iNeighbor < 4; ++iNeighbor )
         {
            sum += imageIn.data[ iRow    + C4[iNeighbor].deltaRow ]
                                 [ iColumn + C4[iNeighbor].deltaColumn ];
         }
         imageOut.data[iRow][iColumn] = sum / 4;
      }
}
```

Afin d'obtenir en plus une généricité vis-à-vis des structures de données, nous pouvons introduire des idiomes [5].

2.2 Ecriture d'une routine de calcul par idiomes

Afin de prendre en charge le balayage des sites d'un agrégat et d'abstraire les détails d'implantation liés à chaque type de structures, nous pouvons utiliser des itérateurs. Ces derniers sont ici les outils de la routine de calcul :

- un itérateur exhaustif pour parcourir tous les sites d'un agrégat,
- un itérateur de voisinage qui parcourt dans un agrégat les sites voisins du site courant d'un itérateur de référence,
- et un itérateur suiveur qui se positionne dans un agrégat de façon équivalente à un itérateur de référence défini sur un autre agrégat.

Deux premières classes abstraites sont définies : `Aggregate<T>`, qui représente un agrégat d'éléments de type `T`, et `Iterator<T>`, qui représente un itérateur sur cet agrégat et déclare un jeu de méthodes abstraites pour une manipulation uniforme de tout itérateur. De la classe `Iterator<T>` dérivent les classes abstraites `ExhaustiveIterator<T>` et `NeighborhoodIterator<T>` ainsi que la classe concrète `FollowerIterator<T>`.

La classe `Aggregate<T>` joue le rôle d'une usine abstraite pour produire les itérateurs à l'aide de méthodes *ad hoc*. Ainsi, par spécialisation, la classe `Image2D<T>` produit des outils `ExhaustiveIterator_Image2D<T>` et `NeighborhoodIterator_Image2D<T>`.

Avec cette modélisation, l'écriture du traitement ne fait pas intervenir de types particuliers à un type concret d'agrégat :

approche impérative	polymorphisme dynamique	polymorphisme statique
3.5 s	11.4 s	3.9 s

TAB. 1: Performance de différentes techniques de programmation.

```
template< typename T >
void mean( Aggregate<T>& inputAggregate, Aggregate<T>& outputAggre-
gate )
{
   Iterator<T>& input    = inputAggregate.createExhaustiveIterator(),
   Iterator<T>& neighbor = input.createNeighborIterator();
   FollowerIterator<T> output( outputAggregate, input );

   for_each( input )
   {
      T sum = 0;
      for_each( neighbor )
      {
         sum += neighbor.getValue();
      }
      output.setValue( sum / neighbor.getCard() );
   }
}
```

Les solutions fondées sur le polymorphisme dynamique présentent cependant quatre inconvénients énumérés dans [4] : ce paradigme renforce le couplage à travers la relation d'héritage [7], il induit un surcoût en mémoire pour chaque objet (pointeur vers la table de fonctions virtuelles [3]), il ne permet pas de vérifications très sévères du typage fort à la compilation, et enfin, il ajoute une indirection lors de chaque appel à une fonction polymorphe.

Pour une problématique de calculs intensifs, ce dernier inconvénient est rédhibitoire : la répétitivité des indirections peut être très élevée ce qui se traduit par un surcoût important en temps d'exécution par rapport à l'écriture impérative de la section 2.1. La table 1 donne, pour notre exemple, les temps d'exécution obtenus avec un PC à 333MHz sous Linux, l'image traitée possédant huit millions de points.

L'approche "classique" du paradigme objet étant inadaptée au cadre du calcul scientifique, nous avons utilisé un autre paradigme, apparu récemment [2], et nous allons voir qu'il ne s'agit que de traduire l'écriture des idiomes dans ce nouveau paradigme.

2.3 Traduction statique des idiomes

Avec l'adoption en 1994 par le comité de normalisation du C++ de la *Standard Template Library* (STL) [9], bibliothèque de conteneurs et d'algorithmes associés, une nouvelle approche de la programmation d'algorithmes génériques s'est révélée : le polymorphisme statique ; l'idée clef est de paramétrer les algorithmes non pas par le type des données mais par le type des structures. Deux bibliothèques récentes utilisent cette technique, CGAL [4] et Blitz++ [10], respectivement pour du calcul géométrique et pour du calcul algébrique.

La traduction statique d'une usine abstraite utilise la déduction de types. En effet, les types des outils d'une routine algorithmique se déduisent du type des struc-

tures sur lesquelles la routine s'applique. Ces déductions sont réalisées par des défini-tions de types dans les classes d'agrégats :

```
template< typename T >
class Image2D : public Aggregate<T>
{
public:
   typedef T                                  DataType;
   typedef ExhaustiveIterator_Image2D<T>    ExhaustiveIterator;
   typedef NeighborhoodIterator_Image2D<T>  NeighborIterator;
   // ...
};
```

Lorsqu'un module n'est pas polymorphe, il doit avoir autant de paramètres qu'il manipulerait de classes abstraites avec une implantation traditionnelle. C'est le cas de l'itérateur suiveur ; ainsi, il est paramétré par le type de l'agrégat sur lequel il itère et par le type de l'itérateur qu'il suit :

```
template< typename Aggr, typename Iter >
class FollowerIterator
{
   // ...
};
```

Au final, la routine de calcul a pour écriture statique :

```
template< typename Aggr >
void mean( Aggr& inputAggregate, Aggr& outputAggregate )
{
   typename Aggr::ExhaustiveIterator  input( inputAggregate );
   typename Aggr::NeighborIterator    neighbor( input );
   FollowerIterator< Aggr, Aggr::ExhaustiveIterator >
                                      output( outputAggregate, input );
   for_each( input )
   {
      typename Aggr::DataType::CumulType sum = 0;
      for_each( neighbor )
      {
         sum += neighbor.getValue();
      }
      output.setValue( sum / neighbor.getCard() );
   }
}
```

Grâce à la traduction statique, le coût de l'écriture dynamique de l'idiome itérateur disparaît en majeure partie ; le temps d'exécution est alors seulement légèrement supérieure à celui d'une écriture impérative (voir table 1).

Nous sommes actuellement en train de formaliser les règles d'écriture de logiciels en programmation générique avec polymorphisme statique, et en particulier, d'établir les règles d'écriture d'un certain nombre d'idiomes dans ce nouveau paradigme.

3 Mise en correspondance des composants et des routines génériques de calcul

Posons tout d'abord la problématique d'une plateforme basée sur des composants de calcul numérique.

3.1 Problématique des composants de calcul scientifique

Dans la plateforme de traitements d'images que nous sommes en train de réaliser, les fonctionnalités que nous souhaitons offrir à l'utilisateur sont :

- l'appel d'un traitement à partir d'un programme écrit en C++,
- l'exécution d'un traitement à partir d'une ligne de commande textuelle,
- l'exécution d'un traitement à partir d'une interface graphique (commande graphique),
- la programmation visuelle d'une chaîne de traitements,
- l'augmentation de la bibliothèque (par tout nouveau type de données, type de structures de données ou routine de traitement).

Un traitement se traduit par un composant qui peut être soit composite (c'est-à-dire un ensemble de composants) soit atomique. Dans un composite, les entrées et sorties des composants qu'il comporte sont raccordés par des liens. Les liens sont le support du flux des données et peuvent être considérés comme les arcs typés du graphe d'exécution, les nœuds du graphe correspondant aux composants. Une chaîne de traitements est donc un composite, ce qui permet à l'utilisateur de l'insérer au sein d'autres chaînes de traitements.

La description d'un composant, donnée par un fichier texte, permet de définir ses entrées et sorties, ses éventuelles contraintes ou déductions concernant le typage des entrées et sorties, et l'appel à la routine de calcul. Cette description permet d'interpréter la ligne de commande texte, de définir l'interface de la commande graphique, de gérer l'affichage du composant dans l'atelier, et de typer un graphe d'exécution.

Les entrées et sorties des composants sont du type statique abstrait `Data` (la classe `Aggregate<T>` qui apparaît dans le code de la section 2.3 dérive de la classe `Data`). Afin de pouvoir résoudre à l'exécution l'appel à des routines de calcul numérique à partir d'un composant, il faut donc être capable de connaître explicitement les types dynamiques concrets des entrées et sorties.

3.2 Résolution des types

A partir de la description d'un traitement, nous générons automatiquement le code des procédures qui traduisent les types statiques des entrées et sorties en types dynamiques, et qui se chargent alors d'appeler la routine générique de calcul. Quant au nom d'une de ces procédures, il est formé avec le nom du traitement et le nom des types paramètres. La génération de code (*generative programming* [11]) est donc totalement factorisable.

Pour reprendre notre exemple, lorsque l'entrée et la sortie sont de type `Image2D<Float>`, la procédure suivante aura été générée ou devra l'être :

```
void mean__Image2D_Float_( Component& component )
{
   Image2D<Float>& input
      = dynamic_cast< Image2D<Float>& >( component.entry( "input" ) );

   Image2D<Float>& output
      = dynamic_cast< Image2D<Float>& >( component.entry( "output" ) );

   mean( input, output );
}
```

L'appel d'un composant à une procédure ne fait plus intervenir, du point de vue du langage, les types concrets des entrées et sorties.

Afin d'effectuer dynamiquement l'appel à la procédure *ad hoc*, nous maintenons un tableau associatif qui associe au nom du traitement et à la chaîne de caractères caractéristique des types dynamiques des paramètres de la routine l'adresse de la procédure. Nous aurons ainsi en mémoire l'association :

```
Call::func["mean"]["Image2D_Float_"]  ==  & mean__Image2D_Float_
```

L'exécution du code de calcul numérique d'un composant est alors possible car chaque donnée sait renvoyer son nom de type dynamique (`Data` déclare une mé-thode polymorphe pour cela) et car la description d'un composant comporte la règle de nommage de sa routine.

Le dernier problème que nous avons résolu est l'instanciation à la volée d'une routine générique.

3.3 Instanciation paresseuse

Dans une bibliothèque générique, les triplets *traitement* $\times$ *structure de données* $\times$ *données*, comme par exemple `( mean, Image2D, Float )`, peuvent atteindre rapidement un nombre considérable. Pour un groupe d'utilisateurs qui partagent essentiellement les mêmes besoins, il est donc inutile de forcer l'instanciation de l'ensemble des routines.

Aussi, nous avons adopté une attitude paresseuse pour l'instanciation des routines ; la plateforme peut ainsi être totalement utilisable sans qu'un seul traitement ne soit compilé.

Lorsqu'un composant de traitement est appelé par l'utilisateur et lorsque les entrées du composant sont renseignées, la vérification d'éventuelles contraintes de typage, fournies par la description du composant, est effectuée. Ensuite, un test réalisé sur le tableau associatif permet de savoir si la procédure *ad hoc* est disponible. Si elle est disponible en mémoire, elle est exécutée ; si elle est disponible sous forme d'objet partagé (fichier `mean__Image2D_Float_.so` pour la routine du même nom), elle est chargée auparavant et le tableau associatif est mis à jour ; si la procédure n'est pas disponible, son code est généré puis compilé ce qui nous replace dans le cas précédent.

De plus, l'ajout d'une entité, que ce soit un traitement, un type de structure de données ou un type de données, est réalisé dynamiquement dans l'atelier afin d'accéder immédiatement à de nouvelles fonctionnalités par simple réutilisation de l'existant. Enfin, l'utilisateur qui connaît ses besoins peut demander explicitement la compilation de triplets.

4 Conclusion

Depuis quelques années, l'utilisation de la généricité ne se limite plus à des classes utilitaires mais constitue un nouveau paradigme de programmation. Du fait du regain d'intérêt récent vis-à-vis de la programmation générique, aucun ensemble de règles d'écriture n'a véritablement été formalisé à ce jour. Nous avons mis en évidence deux premières règles (soulignées en section 2.3) dont la première est

une traduction statique de l'idiome fabrique abstraite ; néanmoins, il reste encore à effectuer un important travail de formalisation.

Nous avons aussi montré comment mettre en correspondance les notions de composant et de généricité. Si notre travail s'inscrit dans le cadre de la réalisation d'une plateforme de calcul scientifique, il est applicable aux multiples domaines que la programmation générique promet de couvrir. Un composant peut véritablement être générique.

Références

[1] M.R. Dobie and P.H. Lewis, *Data structures for image processing in C*, Pattern Recognition Letters, vol. 12, n°8, pp. 457-466, 1991.

[2] G. Dos Reis, *Vers une approche du calcul scientifique en C++*, Rapport de recherche n°3362, INRIA, 1998.

[3] M.A. Ellis et B. Stroustrup, *The annotated C++ reference manual*, Addison-Wesley, 1990.

[4] A. Fabri *et al.*, *On the design of CGAL, the computational geometry algorithms library*, Rapport de recherche n°3407, INRIA, 1998.

[5] E. Gamma *et al.*, *Design Patterns – Elements of reusable object-oriented software*, Addison-Wesley, 1994.

[6] C. Kohl and J. Mundy, *The development of the Image Understanding Environment*, in proc. Int. Conf. on Computer Vision and Pattern Recognition, pp. 443-447, 1994.

[7] J. Lakos, *Large scale C++ software design*, Addison-Wesley, 1996.

[8] G.X. Ritter, J.N. Wilson and J.L. Davidson, *Image Algebra : an overview*, Computer Vision, Graphics, and Image Processing, vol. 49, n°3, pp. 297-331, 1990.

[9] A. Stepanov et M. Lee, *The standard template library*, Rapport, Hewlett-Packard, 1994.

[10] T.L. Veldhuizen, *Blitz++*, `http://monet.uwaterloo.ca/blitz/`, 1996.

[11] T.L. Veldhuizen et D. Gannon, *Active Libraries : rethinking the roles of compilers and libraries*, in Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, Yorktown Heights, New York, 1998.