Généricité dynamique pour des algorithmes morphologiques

Baptiste Esteban, Edwin Carlinet, Guillaume Tochon, Didier Verna

Laboratoire de Recherche et Développement de l'EPITA (LRDE)
14-16 rue Voltaire, 94270 Le Kremlin-Bicêtre , France
baptiste.esteban@lrde.epita.fr

Résumé – La généricité est un paradigme puissant dont l'usage permet d'implémenter un unique algorithme et de l'exécuter sur différents types de données. De ce fait, il est très utilisé lors du développement d'une bibliothèque scientifique, notamment en traitement d'images où les algorithmes peuvent s'appliquer à différents types d'images. Le langage C++ est un langage de choix pour ce genre de bibliothèque. Il supporte ce paradigme et ses applications sont performantes compte tenu de sa nature compilée. Néanmoins, contrairement à des langages dynamiques tels que Python ou Julia, ses capacités en matière d'interactivité, utiles lors des étapes de prototypage d'algorithmes, sont limitées en raison de sa nature statique. Nous proposons donc dans cet article une revue des différentes techniques qui permettent d'utiliser à la fois le polymorphisme statique et dynamique, puis nous évaluons le coût du transfert d'information statique vers des informations connues à l'exécution. En particulier, nous montrons que certaines informations d'une image sont plus importantes que d'autres en matière de performance, et que le surcoût dépend aussi de l'algorithme utilisé.

Abstract – Genericity is a powerful paradigm that enables a single implementation of an algorithm to be executed on different kinds of data structure. Thus, it is largely used for scientific library development, especially for image processing, where algorithms may be applied to several image types. The C++ language is a language of choice for this kind of library. It supports this paradigm and its applications are efficient thanks to its compiled nature. However, unlike dynamic languages such as Python or Julia, its interactivity capabilities, useful for algorithm prototyping, are limited considering its static nature. In this article, we review the different techniques that enable using both static and dynamic polymorphism and evaluate the cost of transferring compile-time information to run-time information. In particular, we highlight the fact that different kinds of information from an image affects performance in different ways, and that the runtime overhead also depends on the algorithm.

1 Introduction

En traitement d'images, les outils sont des éléments importants dans le processus de recherche. On attend généralement de ces derniers 3 propriétés : généricité, performance et utilisabilité. La généricité, telle que définie par Musser *et al.* [8], consiste à rajouter une couche d'abstraction sur un algorithme de manière à pouvoir l'utiliser avec différentes structures de données, améliorant ainsi sa réutilisabilité. La performance est un prérequis dans de nombreux cas d'utilisation, tels que les applications en temps réel ou encore la manipulation d'image en haute résolution. Enfin, l'utilisabilité d'un outil se manifeste au travers d'une forme d'interactivité permettant à l'utilisateur de le manipuler sans avoir à le modifier à l'exécution.

Le langage C++ est souvent utilisé dans le développement d'outils de traitement d'images. Étant compilé, il est axé sur de la programmation performante. De plus, il supporte une forme de généricité, basé sur les *templates*, permettant de traiter diverses structures d'image à l'aide d'un unique méta-algorithme [6] servant de modèle aux différentes spécialisations de template. Ces dernières ne générant du code à la compilation que lorsqu'elles sont utilisées, il est difficile de s'en servir dans des environnements où tous les cas de figure ne sont pas connus en amont tels que les environnements interactifs fournis par des langages dynamiques comme le Python. À cette fin, plusieurs stratégies sont employées dans les bibliothèques existantes pour exposer les algorithmes au langage Python. Higra [9] les instancie à la compilation avec un large ensemble de type de données, résultant en une explosion combinatoire. Vigra [5] utilise un procédé similaire mais limite les combinaisons à un ensemble minimal de type, nécessitant de convertir au préalable les valeurs des images avant de les utiliser. Enfin, OpenCV [3] présente une interface similaire en C++ et en Python, avec une fonction utilisant l'algorithme spécialisé au type dynamique contenu dans les structures de données de la bibliothèque.

Notre objectif est de fournir une interface entre les implémentations de nos algorithmes en C++ et le langage Python tout en restant générique mais en minimisant son coût lié à la génération de code pour chaque instanciation de template. À cette fin, nous effectuons une revue de 4 types d'images ayant la même interface mais dont le typage des valeurs et de leur mode d'accès à ces valeurs changent en fonction de l'implémentation. Nous évaluons ensuite les performances de ces différentes implémentations appliquées à différents schémas algorithmiques.

Cet article est structuré de la manière suivante : nous rappelons premièrement le concept d'algorithmes génériques ainsi que les avantages qu'ils présentent appliqués au traitement d'images en section 2. Ensuite, nous présentons différentes implémentations d'images ainsi que leur utilisation dans un algorithme générique en C++ en section 3. Nous comparons leurs performances sur différents schémas algorithmiques en section 4. Finalement, nous concluons en section 5.

```
void map(image2d input, image2d output,
↪  function<void(uint8_t,uint8_t&)> f)
{
  for (int y = 0; y < input.height(), y++)
    for (int x = 0; x < input.width(), x++)
      f(input(x, y), output(x, y));
}

void map(mesh input, mesh output,
↪  function<void(float,float&)> f)
{
  for (int n = 0; n < input.nedges(); n++)
    f(input(i), output(i));
}
```

Listing 1 – Exemple de fonction non générique

2 Généricité statique en C++

La généricité en C++ est un processus statique. Elle est basée sur les *templates*, des listes de paramètres dont les valeurs, évaluées à la compilation, sont des types ou des valeurs primitives. Pour chaque instanciation de template, le code machine dépendant est généré avec les paramètres de template correspondant. Ce type de généricité présente ainsi l'avantage d'avoir un code spécialisé, optimisé par le compilateur pour la combinaison de paramètres dans le template. Néanmoins, cela nécessite de connaitre ces valeurs à la compilation. Gérer le cas où le type dépend du programme à l'exécution demande d'instancier chaque combinaison de paramètre, résultant en une explosion combinatoire, et générant une quantité considérable de code machine, phénomène usuellement connu sous le nom de *code bloat*.

Appliquée au traitement d'images, la généricité permet, pour un algorithme, d'utiliser plusieurs types d'image définies par $I : \Omega \to \mathcal{V}$ avec Ω le domaine de définition de I et $\mathcal{V}$ son espace de valeurs. Ω peut représenter différents types de coordonnées tels que la position d'un pixel dans un domaine rectangulaire ou encore l'indice d'une arête d'un maillage et l'espace $\mathcal{V}$ est composé des valeurs en relation avec les valeurs de Ω. Le listing 1 expose une fonction `map` non générique. Dans cet exemple, `image2d` est une image définie sur un domaine rectangulaire dont les valeurs sont encodées en **`uint8_t`** et `mesh` représente un maillage dont les valeurs pondérant les arêtes sont encodées en **`float`**. Le corps des deux fonctions suit un schéma similaire : il applique à tous les éléments de `input` la fonction `f` et stocke le résultat dans `output`.

En rajoutant une couche d'abstraction, il est possible de donner une seule implémentation de `map` sous la forme d'une *fonction template*, tel qu'illustrée dans le listing 2. Dans cet exemple, les images `input` et `output` peuvent être n'importe quel type d'image, sous réserve que leur domaine de définition soit identique. Ainsi, l'implémentation générique d'un algorithme dépend d'une interface et non d'un type.

Ce type de généricité est parfait pour des applications où le type des images est connu à la compilation. Néanmoins, l'inconvénient majeur de ce système est le manque de dynamisme. Dans le cas où un outil donne la possibilité à l'utilisateur de choisir le type d'image à l'exécution, la fonction `map` doit être spécialisée pour tous les cas possibles, résultant en l'explosion combinatoire exposée dans la Figure 1.

```
template <typename I, typename O, typename F>
void map(I input, O output, F f)
{
  for (auto p : input.domain())
    f(input(p), output(p))
}
```

Listing 2 – Version générique de `map`

```
              template <typename I,
                        typename O,
                        typename F>
              void map(I, O, F)
                  ↙       ↓      ↘
map<image2d<uint8_t>,    ...    map<mesh<uint8_t>,
    image2d<double>,                mesh<rgb8>,
    inv>                            lut>
```

FIGURE 1 – Illustration de l'explosion combinatoire

3 Vers une généricité dynamique

3.1 Dynamisme des structures de données

Un objet manipulé par un algorithme générique doit respecter une certaine interface, peu importe son implémentation. La classe `buffer2d<T>`, illustrée dans la Figure 2, est une image définie sur un domaine rectangulaire en dimension 2 dont les valeurs, statiquement typées, sont stockées dans un tableau linéaire. Cette classe respecte une interface bien précise avec un accès au domaine de définition et un opérateur d'accès à une valeur de l'image reliée à un point de ce domaine. Néanmoins, comme expliqué dans la section 2, elle souffre de la généricité statique du type des valeurs lié au template dans le cas où elle est utilisée dans un contexte dynamique.

Un moyen concret de pallier ce problème repose sur un patron de conception utilisé en C++, *l'effacement de type*. Ce patron est utilisé dans plusieurs applications en C++ tel que `std::any` [4], un objet non typé qui enveloppe un objet, peu importe son type statique. Son fonctionnement repose sur le fait qu'il stocke les valeurs de l'objet statiquement typé dans un espace mémoire qui n'est pas typé, et converti à l'utilisation l'objet dans le type statique correspondant à celui de l'objet. Le cas de `std::any` ne dépend d'un paramètre de template que lors de la création de l'objet et de sa conversion. Au lieu d'utiliser une information statique, `std::any` stocke une information dynamique qui lui permet de s'assurer que la conversion est valide.

Appliqué à l'implémentation d'une image, l'effacement de type supprime l'information statique du type des valeurs, permettant ainsi de se passer du template dans l'implémentation de la classe. Il devient néanmoins nécessaire de garder dynamiquement l'information du type d'un élément pour pouvoir accéder à la valeur désirée. De plus, l'opérateur d'accès ne retourne plus une valeur mais une adresse mémoire pointant vers la position du premier octet de la valeur du pixel. Ainsi, afin de manipuler la valeur contenue au sein de cette adresse, il faut au préalable la convertir dans le type statique.

```
template <typename T>
struct buffer2d
{
  T& operator()(point2d p);
  domain2d domain;
- - - - - - - - - - - - - - - - - - - - - - - - -
  T* data;
};
```

Interface

Détails d'implémentation

FIGURE 2 – Image statique sous forme de tableau linéaire

```
struct buffer2d_any
{
  void* operator()(point2d p);
  domain2d domain;
- - - - - - - - - - - - - - - - - - - - - - - - -
  void* data;
  /*Implementation-defined*/ type_info;
};
```

Interface

Détails d'implémentation

FIGURE 3 – Image dynamique sous forme de tableau linéaire

La classe `buffer2d_any` représentant une image non typée est illustrée dans la Figure 3. On remarque que malgré le changement d'implémentation, l'interface reste la même que celle de `buffer2d<T>`.

Le modèle présenté précédemment rajoute du dynamisme sur les valeurs des images, mais est limité aux images encodées sous la forme d'un tableau de données contiguës. Or, les algorithmes génériques doivent avoir la possibilité de prendre diverses structures sur lesquelles il est possible d'effectuer des traitements tels que des graphes ou des matrices creuses. Pour gérer ce cas, un nouveau modèle d'image est ajouté, respectant la même interface que les deux types d'images présentés précédemment mais accédant aux données de manière indirecte. Ainsi, ces nouvelles structures, `indirect2d<T>` pour la version statique et `indirect2d_any` pour la version dynamique, ont une fonction interne qui gère l'accès aux données par indirection. Les propriétés des 4 types d'images présentées sont résumées dans la Table 1.

3.2 Adaptation aux algorithmes génériques

L'utilisation des images dynamiques dans les algorithmes génériques suit un schéma développé dans la bibliothèque standard du langage C, en particulier pour la fonction `qsort`, dont l'objectif est de trier un tableau dont le type des valeurs est effacé. Cette fonction prend 4 arguments : un tableau non typé, le nombre d'éléments de ce tableau, la taille de chaque élément et une fonction de comparaison dont le type des arguments est effacé. Cette dernière fonction est un *projecteur* au sein duquel une comparaison est effectuée après avoir déréférencé l'adresse de l'élément pointé en une valeur typée.

L'adaptation des algorithmes génériques aux images dynamiques suit la même logique que la fonction `qsort`. Ces algorithmes prennent en argument des images dynamiques et des projecteurs qui convertissent les valeurs avant d'effectuer une opération. Dans le listing 3, deux spécialisations de la fonction `map` sont proposées. Le code de cette fonction est écrit une fois et celle-ci est instanciée pour chaque niveau de dynamisme. La version statique, dénotée par `map_st`, n'a que peu de différences avec la version générique, cela étant qu'elle

	Typage des valeurs	*Mode d'accès*
`buffer2d<T>`	✓	Direct au buffer
`buffer2d_any`	✗	Direct au buffer
`indirect2d<T>`	✓	Indirect
`indirect2d_any`	✗	Indirect

TABLE 1 – Résumé des propriétés connues statiquement par les images présentées

```
template <typename I, typename O, typename F>
void map_st(buffer2d<I> in, buffer2d<O> out, F f)
{
  map(in, out, f);
}

void map_dyn(buffer2d_any in, buffer2d_any out,
↪    function<void(void*, void*)> f)
{
  map(in, out, f);
}
```

Listing 3 – Spécialisation de `map` pour des images statiques (`map_st`) et des images dynamiques (`map_dyn`)

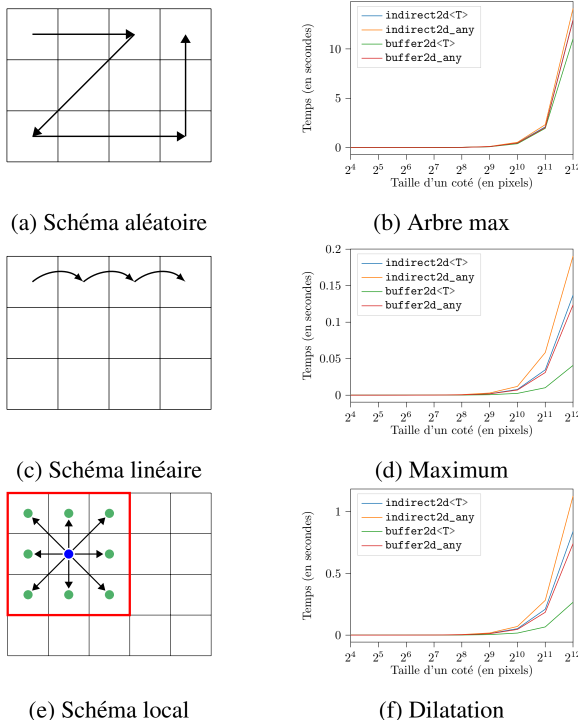
est spécialisée pour la classe `buffer2d<T>`. Néanmoins, la version dynamique, dénotée par `map_dyn`, présente plusieurs différences. Premièrement, elle ne dépend d'aucun paramètre de template. De ce fait, il n'y a plus de problème d'explosion combinatoire, le code n'étant généré qu'une seule fois. Ensuite, cette fonction a en argument un projecteur, qui convertit les valeurs de l'image en valeur typée et y applique une opération.

Enfin, grâce à l'interface requise par les algorithmes génériques, leur adaptation aux images à accès indirect n'a aucune différence avec l'adaptation pour des images à accès direct. Ainsi, les représentations dynamiques sont facilement utilisables par des algorithmes génériques, mais aussi exposable dans le langage Python en un seul point d'entrée, ce qui n'est pas le cas pour la version typée des images, qui doit être instanciée à la compilation, les modules Python écrits en C++ étant des bibliothèques compilées.

4 Résultats

L'utilisation d'images dynamiques simplifie l'exposition des fonctionnalités C++ en Python mais cela a un coût : certaines optimisations effectuées par le compilateur ne sont pas applicables à cause du manque d'information de type. Dans [7], Meyers explique que l'utilisation de la fonction `std::sort` est 670% plus rapide que la fonction `qsort` sur un vecteur d'un million de valeurs typées en **`double`**. Cette différence se justifie par le fait que les appels à la fonction de comparaison dans `std::sort` sont en ligne. Dans cette section, nous évaluons le coût de l'effacement de type en utilisant différents schémas algorithmiques.

Les schémas algorithmiques testés sont courants dans les algorithmes de morphologie mathématique [11]. Le premier, illustré dans la Figure 4a, parcourt l'image sans ordre prédéfini. Il est testé en utilisant l'algorithme de Berger *et al.* [2] pour le calcul de l'arbre max [10]. Le second schéma, illustré dans la Figure 4c, parcourt l'image dans l'ordre où les valeurs



(a) Schéma aléatoire (b) Arbre max

(c) Schéma linéaire (d) Maximum

(e) Schéma local (f) Dilatation

FIGURE 4 – Évaluation des différents types d'images sur différents schémas algorithmiques

sont stockées en mémoire. Il est utilisé pour calculer l'image maximum de deux images. Enfin, le dernier schéma, illustré dans la Figure 4e, est local et effectue des comparaisons avec ses pixels voisins. Il est utilisé pour calculer une dilatation.

L'évaluation des performances, effectuée avec Google Benchmark dans des programmes compilés avec GCC 10.2.1 avec les options de compilation *-03, -ftree-vectorize, -mavx* et *-unroll-loops*, est exécutée sur une machine Linux Debian 11 équipée d'un processeur Intel i7-3770, 3,40GHz. Les résultats de cette évaluation sont affichés dans les Figures 4b, 4d et 4f. On observe que l'arbre max a des performances similaires entre les différentes images. Néanmoins, pour le schéma local et le schéma linéaire, les performances de l'image statique sans indirections sont bien meilleures qu'en utilisant les 3 autres types d'images. Ces différences de performances s'expliquent par plusieurs raisons. Premièrement, le parcours linéaire direct d'un tableau typé est vectorisé par le compilateur, permettant ainsi d'appliquer la même opération sur plusieurs valeurs simultanément. Ensuite, l'utilisation de `std::function` pour les indirections entraîne une incapacité par le compilateur de mettre en ligne l'appel de la fonction, celle-ci n'étant connue qu'à l'exécution par `std::function`. De plus, le parcours d'un bloc linéaire de mémoire sans indirection permet une meilleure analyse d'alias par le compilateur. Enfin, les opérations de dilatation et de maximisation ont un pourcentage moyen d'échec de cache inférieur à 0,5%, à comparer avec les 3% de l'algorithme de l'arbre max.

Ces résultats montrent que le coût du passage d'une information statique à une information dynamique dépend de l'information mais aussi de l'algorithme employé. Dans le cas d'un algorithme dont le travail annexe est conséquent, ce surcoût est moindre, montrant ainsi que le spécialiser pour une multitude de type n'est pas rentable dans le cadre d'une exposition vers Python. En revanche, un traitement particulier doit être apporté dans le cas où l'optimisation de l'algorithme s'appuie sur les propriétés d'implémentation d'une image tel que l'encodage sous la forme d'un tableau de données contiguës.

5 Conclusion

Dans cet article, nous avons présenté une revue de différents niveaux de dynamisme pour l'implémentation d'une image et nous avons appliqué ces différentes représentations aux algorithmes génériques grâce à l'interface commune de ces images. Nous avons ensuite observé que les performances de ces différents niveaux de dynamisme appliqués à ces algorithmes dépendaient du schéma algorithmique utilisé. Pour confirmer cette observation, nous prévoyons d'évaluer ces performances sur un ensemble d'algorithmes plus large. De plus, pour réduire le coût du dynamisme dans un schéma dont l'optimisation dépend de la propriété contiguë d'un tableau, nous étudierons l'utilisation d'un processus de compilation à la volée tel qu'AsmJit [1] ou de modules externes précompilé qui seront chargés à l'exécution par le programme.

Références

[1] *AsmJit Project, Machine Code Generation for C++*. URL : `https://asmjit.com/`.

[2] C BERGER et al. "Effective component tree computation with application to pattern recognition in astronomical imaging". In : *IEEE International Conference in Image Processing*. T. 4. 2007, p. IV-41.

[3] G BRADSKI et A KAEHLER. "OpenCV". In : *Dr. Dobb's journal of software tools* 3 (2000), p. 2.

[4] B DAWES et A MEREDITH. *P0220R1 : Adopt Library Fundamentals V1 TS Components for C++17 (R1).*

[5] U KÖTHE. "Reusable software in computer vision". In : *Handbook of computer vision and applications* 3 (1999), p. 103-132.

[6] R LEVILLAIN, T GÉRAUD et L NAJMAN. "Milena : Write generic morphological algorithms once, run on many kinds of images". In : *International Symposium on Mathematical Morphology*. Springer. 2009, p. 295-306.

[7] S MEYERS. *Effective STL : 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison Wesley, 2001.

[8] D MUSSER et A STEPANOV. "Generic programming". In : *International Symposium on Symbolic and Algebraic Computation*. Springer. 1988, p. 13-25.

[9] B PERRET et al. "Higra : Hierarchical graph analysis". In : *SoftwareX* 10 (2019), p. 100335.

[10] P SALEMBIER, A OLIVERAS et L GARRIDO. "Antiextensive connected operators for image and sequence processing". In : *Transactions on Image Processing* 7.4 (1998), p. 555-570.

[11] J SERRA. *Image Analysis and Mathematical Morphology*. Academic press, 1982.