Generic Algorithmic Blocks dedicated to Image Processing

Jérôme Darbon[1,2] Thierry Géraud[1] Patrick Bellot[2]

[1]EPITA / LRDE
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre - France
{darbon, geraud}@lrde.epita.fr

[2] ENST / INFRES
46 rue Barrault, 75634 Paris Cedex 13 - France
{darbon,bellot}@enst.fr

Abstract

This paper deals with the implementation of algorithms in the specific domain of image processing. Although many image processing libraries are available, they generally lack genericity and flexibility. Many image processing algorithms can be expressed as compositions of elementary algorithmic operations referred to as blocks. Implementing these compositions is achieved using generic programming. Our solution is compared to previous ones and we demonstrate it on a class image processing algorithms.
Keywords: Generic Programming, Image Processing, C++.

1 Introduction

Writing a software for image processing is not an easy task. Due to the difficulty of implementing image processing algorithms, computer vision methods suffer from lack of comparison [1, 2]. It is particularly difficult when one aims at meeting implementation with theory in conjunction with efficiency. Many scientific libraries cover a specific domain and provide few data and image structure types. This last statement is a critical issue. Indeed, many image processing algorithms should accept one/two/three-dimensional images, graphs... which contains scalar, complex, composed values and so forth. Consequently routines which are able to work on different input types are required. Such routines are said to be "generic". Köthe in [3], D'Ornellas and van den Boomgaard in [4, 5] proposed to use *generic programming* to implement generic image processing algorithms.

The aim of this paper is to present another generic framework for both a confirmed programmer who wishes to implement image processing algorithms and a novice programmer who simply needs to program a method from existing algorithms. Motivations, expected features and the needs of these two kind of people are presented in section 2. After a discussion of classical generic paradigms in section 3 a description of our framework is presented in section 4. Finally, we draw some conclusions in section 5.

2 Motivation and Requirements

An image processing library should fulfills the following requirements:

- First of all, scientific computing requires **fast** routines [6]. They are generally provided by libraries.
- Moreover since most of the people involved in image processing are mathematicians, rather than computer scientists, algorithms should be **expressive** [7]. In other words, theory and implementation should meet.

- Besides implementation should be **generic** [3, 4, 5] in order to cope with combinatorial explosion if one has to write a specific routine for each data type.

- In addition code should be written in C or **C++** since most of people involved in image processing only know these languages [8]. However, it is well know that the C language cannot achieve as high a level of generic programming as in C++. Consequently, C++ is the only remaining choice since we do not want people to learn another language.

- **Type inference** should be used in order to make programming as bug-free as possible. Moreover, static type checking should be used in order to detect errors at compile-time.

- Finally **ease of code** is required for an image processing practitioner. And **ease of re-use** of code is needed for a developer. Libraries often do not provide this feature in practice. This is mainly due to the difficulty of implementing reusable algorithms without run-time overhead. Moreover, the library scaling problem should be avoided.

The last item highlights the fact that two different kinds of people are involved in image processing: practitioners and developers who are respectively assumed to be novice and confirmed programmers. We now precise their respective needs.

2.1 Practitioner's and Developer's Constraints

Let us illustrate the needs of both kind of programmers with the following example in pseudo-code:

```
Image1_type ima1;
Image2_type ima2 = algo2((algo1(ima1)));
save(ima2);
save(algo3(ima2));
```

This example is representative of what is needed in image processing. The significant point lies in line 2 where `Image2_type` is defined. Indeed, one generally needs to hold intermediate results to save them and to reuse them into other algorithms. Problem arises because one needs to know the type of the return image. This problem is crucial in C++. Of course, a functional language could immediately solve this problem, but recall it is not adequate for us. Consequently, a novice programmer should be able to deduce and easily write the return type of an algorithm.

Writing high level algorithms should be systematic and easy: for instance, composition of two algorithms (still line 2) should not be considered as a problem. This point is important since many image processing algorithms can be expressed using algorithmic patterns [5] which rely on composition of algorithms. We now explain this point.

2.2 Building Blocks

From a practical point of view, a scientific library consists of algorithms working on data structures. An algorithm simply takes some data as inputs and produces a result after processing them with a finite number of operations. From this definition, it seems natural that an algorithm should be a function. Moreover, since we need generic algorithms, it must be parameterized by input parameters. Return type is deduced by the compiler once the input type is provided by the user. The block diagram of this process is depicted figure 1.

However, most of image processing algorithms can be further decomposed into blocks. Let us give a simple example where three stages are involved:

```
initialization                // First stage
for each point in the image   // Second stage
   do some stuff              // Variation
finalization                  // Last stage
```

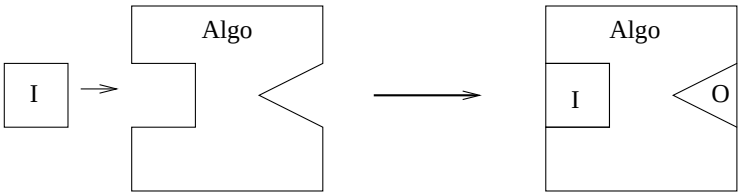

Figure 1: Block diagram of a generic algorithm. A user manipulate an "abstract" algorithm. Once input type I (square) is provided, the compiler infers return type O (triangle) and specializes the algorithm for this particular input.

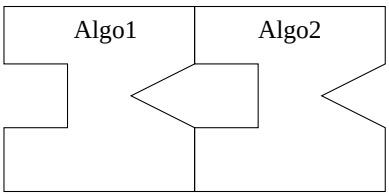

Figure 2: An example of an horizontal composition of 2 algorithms. Input type will be processed by `Algo1` then by `Algo2`. Note that Output type of `Algo1` is the input type of `Algo2`.

As one can see, an algorithm is the composition of many simple blocks (also algorithms) which are independent from each other. Combining blocks yields higher level algorithms. Basically, two types of combination can be distinguished:

1. Horizontal composition: this mainly corresponds to chain algorithms which are on the same level. It is depicted on figure 2. For instance, our last example on initialization/process/post-process fits into this framework.

2. Vertical composition: it corresponds to a refinement of a high level algorithm in order to cope with variations. Basically, it allows flexibility and variability of an algorithm without modifying it. It is depicted in figure 3. Note variations are also algorithms.

This way of building algorithms have great advantages. Scaling library problem is avoided and ease of reuse is achieved thanks to composition. Finally, code will be expressive since blocks composition exactly reflects the algorithm. However, one has to exhibit patterns and independent blocks for a given domain. We still have to provide a solution to compose blocks. Recall this composition must be easily implemented (practitioners of this domain are not computer scientists), and a kind of recipe should be provided. So far, type propagation within blocks has not been addressed. In other words, input and output types must be passed from one block to another statically in order to be safe and fast (avoiding run-time dispatch). Besides, assembling these blocks must be done at compile time in order to avoid run-time errors.

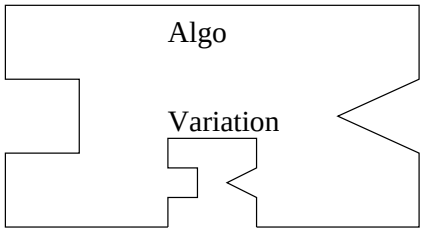

Figure 3: An example of a vertical composition of an algorithm with one variation. A high-level `Algo` is refined using `Variation`.

3 Overview of Classical Solutions

Classical solutions to compose blocks and related problems are presented in this section. Since we aim at composing building blocks which yield higher level algorithms, our running example is the composition of two functions. More precisely, for the sake of clarity only composition of two unary functions will be studied. The two functions will be referred to as `F` and `G`. We insist on the point that this example is not restrictive. Indeed, this simple example also provides solution to more complicated cases. All code example are in C++. Quoted text "..." refers to some omitted code for brevity and which is useless for the understanding of the paper. In what follows we suppose we have a system of *traits*. A trait refers to type characteristics. It can be used to infer type. For instance, output type of an algorithm is deduced from its input types. In the rest of this paper, such a deduction will be referred to as `Output_Algo_Trait`. Since such explanation of this system is out of the scope of this paper, we refer the reader to [9] and [10].

3.1 Function Templates

Remember that a natural way of representing an algorithm is a function. In order to be generic with respect to input data types, its function is parameterized by them. This leads to the following code:

```
template <class Input>
typename Output_Twice_Trait<Input>::RET twice(const Input & x) { return 2*x }
```

This solution is fine since a programmer do not explicitly precise parameters when she/he uses the function. The compiler statically gets input parameters and generates output type using a trait. However this solution has a severe limitation. Indeed, it is not possible to pass function templates to other function templates as an input parameter. In other words, higher order polymorphic functions cannot be expressed this way. Composition is as follows: `F((G(input)))`. The return type is given by:

```
typename Output_F_Trait< typename Output_G_Trait<Input>::RET>::RET
```

In other words, the return type of this composition is the return type of `F` where its input parameter is the result type of `G`; and the input parameter of `G` is `Input`. A user has to write this latter line every time she/he needs it, which is really cumbersome. Note the result of many compositions is almost intractable. Another solution makes use of functors also known as function objects.

3.2 STL-style Coding

C++ has the ability to create objects which behaves like functions. It is based on overloading the operator "()". They are called *function objects*. The C++ Standard Library inspired by the Standard Template Library (STL) [11], have greatly popularized this solution. Implementation is straightforward:

```
template <class Input, class Output>
struct Twice : public std::unary_function<Input, Output>{
   Typedef Output Result;
   Result operator()(const Input & x) { return 2*x; }
};
```

In this case, input and output parameters must be provided by the programmer before calling the algorithm. STL provides an operator for unary functions know as `compose1`. Consequently, composition is as follows:

```
typedef G<Input, Output_G_Trait<Input>::RET > G_T ;
compose1(F< G_T::Result, Output_F_Trait< G_T::Result> >(), G_T())
```

Once again, deduction type must be explicitly written by the user. And the return type of this function composition is the same as function templates' one. Note that providing input type of an algorithm is not a gig deal but a user do not want to cope with intermediate and result type. STL requires that a programmer explicitly writes the signature of a function. We now turn off to a much more elegant solution.

3.3 Polymorphic Direct Functoids

Recall that we are interested in composing functions in C++ which should be polymorphic, like in a functional language. McNamara and Smaragdakis present in [12, 13] a library called FC++ which provides a solution to this problem. They intend to emulate a functional mechanism using C++. The solution consists of nested template members. In order to enhance the semantics of a function, the method declares explicitly the signature of a function. This is achieved by parameterizing both `operator()` and output traits by input parameters. We present a modified version of their solution for simplicity. The code is as follows:

```
struct Twice {
  template <class Input>
  struct Sig
  {
     typedef Input FirstArgType;
     typedef typename OutputTwiceTrait<Input>::RET   ResultType;
  };
  template <class Input>
  typename Sig<Input>::ResultType operator()(const Input & x)
  {  return 2*x; }
} twice;
```

Encoding of signature is achieved by `Sig` member. It defines the type of input and output types which allows type inference like a functional language would do. Note `operator()` is defined like we did for function templates. Since the function embeds its own signature, we can pass it to a higher order function. Indeed the latter will be able to deduce its signature from signatures of input object function. Since a higher order function should be like a classical function, the same form of construction should be used. Recall that the return type should be a function. Here is the code for a composition of unary functions:

```
struct Compose
{
  ... // Definition of Compose's Sig

  template <class F, class G>
  Composer<F,G> operator()(const F &f, const G & g)
  { return Composer<F,G>(f,g); }
} mycompose;
```

The return type of composition is an instance of an object which is built from the input functions, f and g, and whose `operator()` performs $g(f(.))$. Its code is as follows:

```
template <class F, class G>
struct Composer
{
  Composer( const F& ff, const G& gg ) : f(ff), g(gg){}
  const F &f;
  const G &g;

  ... // Definition of Sig
```

```
  template <class X> typename Sig<X>::ResultType
  operator()( const X& x ) const { return f( g(x) ); }
};
```

Note that this way of building composition necessary involves two separate parts. Indeed, `Compose` is unable to hold both f and g since it does not know their types. It is really cumbersome for a developer.

Let us see how a practitioner can use these functions. For instance, a user respectively uses `twice` and `compose` as follows: `twice(3)`, `compose(twice, twice)(3)`. As one can see, this code is really nice for a practitioner. However, like previous solutions, she/he still needs to explicitly declare variable type to store a result although all intermediate types has been inferred. For a simple composition a user have to write the following code:

```
MyCompose::Sig<Twice, Twice>::ResultType::Sig<int>::ResultType i=
            mycompose(twice, twice)(3);
```

Note the code to write in order to get the output type for multiple compositions remains cumbersome. Imagine writing return type of `compose(compose(twice, twice),compose(twice, twice))`

We now present the solution.

4 Our Generic Framework For Image Processing

Our solution follows the work of McNamara and Smaragdakis. Indeed, we keep the solution for enhancing C++ signature of functions. Recall our goal is to achieve an easy writing code for both a developer and a practitioner. Please note that a user always knows the image she/he wants to process. Intermediate types should then be deduced automatically.

Writing and using our algorithms is of the same order of complexity as McNamara's library. It still makes use of nested member. Contrary to McNamara and Smaragdakis' method, a structure which defines both the output trait and the `operator()` is parameterized by input parameters. Then, we use FC++ method for type inference. Contrary to what is generally done, an algorithm is defined with two levels. The first one deals with the "abstract" functionality of an algorithm (like twice) as in FC++, without taking care of input types. The second one specializes an "abstract" algorithm for a particular input type. Of course, only algorithms of the second type can perform computations. The main advantage are the following: On the first hand, both types of algorithm are types; it means that a developer can refer to an algorithm using the keyword "typedef". On the second hand, once one has defined an algorithm for a specific input type, writing the output type is trivial: it is simply given by `Output`. Note this way of programming is a mix between STL programming and FC++.

We explain this way of programming on both simple and higher order functions, and show the difference between FC++'s and our paradigm.

4.1 Simple Algorithm

Let us begin with the `twice` example. Remember figure 1 where an algorithm is presented like a function which takes some inputs and deduces an output type once input type is known. Our solution reflects this diagram by defining a member `Output` inside a nested structure which is parameterized by input types.

```
struct twice {
  template <class Input>
  struct toReal {
    typedef ... Output;
    Output operator()(const Input & x)
    { return 2*x; }
  };
```

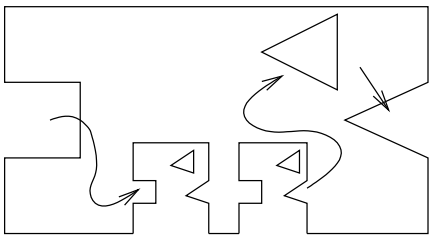

Figure 4: Block Diagram for compose using our solution. Return type which is depicted as a triangle deduced and stored in each algorithm. Lines with arrows represent type propagation.

```
};
```

As one can see, the only difference between this solution and FC++'s one is that the nested template defines both return type and `operator()`. This nested template is called `toReal` which refers to "to real type". Indeed, contrary to FC++, a user has to explicitly precise input type of the algorithm before using it. But the return type is easily obtained using `Output` trait. Here is an example:

```
typedef twice::toReal<int> algo;
std::cout << algo()(3) << std::endl;
algo::Output i = algo()(3);
std::cout << i << std::endl;
```

As one can see, this solution to call twice is not as convenient as FC++ neither function templates. However, an algorithm is now a type, and we will see advantages brought by this.

4.2 Composition and Algorithm with Variations

We now show how to deal with composition. Recall figure 3 which presents a vertical composition to deal with an algorithm which has a variation. This diagram is recursive (the variation has the same pattern as the whole algorithm itself). Block diagram for composition should also has this recursive pattern contrary to what is presented figure 2. Figure 4 block diagram for a composition with type inference. As one can see, a recursive pattern is introduced and type inference can be done. In other words, there is no difference (with respect to block diagram and so code) between vertical composition nor horizontal composition.

Let us cope with composition. Contrary to a simple function, a higher order function is directly parameterized by inner functions for composition:

```
template<class F, class G>
struct compose {...};
```

Writing the composition of two unary functions is a type and is easy to write: `compose<twice, twice>` for instance. This type is now used like a simple function(in fact, it is). Inner structure needs to define the real type for functions `F` and `G`, and the return type. Finally, `operator()` performs the computations. The code is as follows:

```
template<class Input>
struct toReal
{
  typedef typename G::toReal<Input>                  _G;
  typedef typename F::toReal<typename _G::Output>    _F;
  typedef typename _F::Output                        Output;
  Output operator()(const Input& x)
  { return _F()(_G()(x)); }
};
```

Contrary to FC++, we do not need to split the code of composition into two parts because algorithms are types. Instances of algorithms are created only when computations are performed. Since all algorithms are types, a practitioner can use the keyword `typedef` to define algorithms without taking care of input types.

```
typedef compose<twice, twice> Algo1_;
typedef compose<twice, twice> Algo2_;
typedef compose<Algo2, Algo1_> FinalAlgo_;
```

Note that is a convenient way for a novice programmer to write algorithms. Once algorithms are written, the programmer gives input types, and no matter how complex an algorithm is, the return type is always given by the trait `Output`. This leads to the following code:

```
typedef int Input;
typedef FinalAlgo_::toReal<Input> FinalAlgo;
FinalAlgo::Output i = FinalAlgo()(3);
```

So far, we succeeded in defining higher order functions. Return types are easily obtained. However input types have to be explicitly written.

4.3 Convenient Objects for Image Processing

Contrary to FC++, this solution is able to define objects which are able to store variables. For instance, consider one needs to find the minimal and maximal element of a sequence. The code for such an object function is straightforward:

```
struct findMinMax {
  template <Input>
  struct toReal {
    toReal(): count(0)
    void reset { count = 0;}
    void operator()(const Input & x)
    {
      if (count == 0) { min = max = x;}
      else {
        if (max < x) max = x;
        if (x < min) min = x;
      }
    }
    int count;
    Input min,max;
  };
};
```

Although it seems a little bit complicated to write such a code, do not forget that these function objects are written only once and work for any input types.

4.4 A Simple Real Example: Generalized Convolution Operations

Generalized convolution operations are a large class of image processing patterns. Computations involve pixel values in a neighborhood of a pixel. These values are combined with the values of a kernel. The result is put into the central pixel. Classical algorithms which fits into this pattern are classical linear convolution (like gaussian filter), and non-linear filters like erosions/dilations in mathematical morphology [7].

This pattern needs is made of two parts. The first one deals with initialization of the output image. The second one is the type of convolution used. Inputs are the image to process and the kernel used. The implementation is straightforward:

```
template <class Init, class Convol>
struct generalizedConvolution  {
  template <class Image, class Kernel>
  struct toReal
  {
     typedef ... Output; // Deduce output type from Image, Kernel and Convol
     ...                 // Define "real type" for Init and Convol

    Output operator()(const Image image, const Kernel & kernel)
      {
        Output output;
        init()(output, image, kernel);
        for_all(p in image)
           output[p] = convol()(image, p, kernel)
        return output;
      }
  };
};
```

5 Conclusion

We now review the characteristics of our system and draw some lines for future work.

- Routines are efficient since method dispatch is static and is combined with inlining. Note that this feature is not achieved using virtual methods. It leads to poor performance because of runtime over-head induced by dynamic dispatching.
- Code is type safe thanks to type inference and parametric polymorphism.
- Library scaling is achieved thanks to building blocks.
- Writing an image processing method is simple since algorithms are types. However, one has to explicitly write input types of these algorithms.
- In order to constraint template parameters, we should use static interfaces [14] [15].
- In order to constraint template parameters, we should use static interfaces [14] [15].
- Finally, we should introduce expression templates, introduced in[16], into our framework.

References

[1] Köthe, U.: Reusable implementations are necessary to characterize and compare vision algorithms. in DAGM-Workshop on Performance Characteristics and Quality of Computer Vision Algorithms (1997)

[2] Price, K.: Anything you can do, i can do better (no you can't).... Computer Vision, Graphics, and Image Processing **36** (1986) 387–391

[3] Köthe, U.: Chapter 3: Reusable Software in Computer Vision. In: Handbook on Computer Vision and Applications. Acadamic Press (1999)

[4] d'Ornellas, M.C., van den Boomgaard, R.: Generic algorithms for morphological image operators — a case study using watersheds. In Heijmans, H., Roerdink, J., eds.: Mathematical Morphology and its Applications to Image and Signal Processing. (1998) 323–330

[5] d'Ornellas, M.C.: Algorithmic Patterns for Morphological Image Processing. PhD thesis, University of Amsterdam (2001)

[6] Scientific Computing In Object-Oriented Languages: (2000) Web page. http://oonumerics.org/.

[7] Darbon, J., Géraud, T., Duret-Lutz, A.: Generic implementation of morphological image operators. In: Mathematical Morphology, Proceedings of the 6th International Symposium (ISMM), Csiro Publishing (2002) 175–184

[8] Pitas, I.: Digital Image Processing Algorithms and Applications. Wiley (2000)

[9] Czarnecki, K., Eisenecker, U.: Generative programming: Methods, Tools, and Applications. Addison-Wesley (2000)

[10] Myers, N.C.: Traits: a new and useful template technique. C++ Report **7** (1995) 32–35

[11] Stroustrup, B.: The C++ Programming Language. Addison-Wesley (1997)

[12] McNamara, B., Smaragdakis, Y.: Functional programming in C++. In: Proceedings of the International Conference on Functional Programming (ICFP), Montreal, Canada (2000)

[13] Smaragdakis, Y., McNamara, B.: Fc++: Functional tools for object-oriented tasks. Sotware: Practice and Experience **32** (2002) 1015–1033

[14] McNamara, B., Smaragdakis, Y.: Static interfaces in C++. In: Proceedings of First Workshop on C++ Template Programming, Erfurt, Germany. (2000)

[15] Burrus, N., Duret-Lutz, A., Geraud, T., Lesage, D., Poss, R.: A static c++ object-oriented programming (scoop) paradigm mixing benefits of traditional oop and generic programming. In: Workshop on multiple paradigm with OO languages. MPOOL'03. (2003)

[16] Veldhuizen, T.: Expression templates. C++ report (1995)