In the Proceedings of the 15th International Conference on Pattern Recognition (ICPR'2000),
IEEE Computer Society, vol. 4, pages 816-819, Barcelona, Spain, September 2000.

Obtaining Genericity for Image Processing and Pattern Recognition Algorithms

Thierry Géraud, Yoann Fabre, Alexandre Duret-Lutz
EPITA Research and Development Laboratory
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre cedex, France
thierry.geraud@lrde.epita.fr

Dimitri Papadopoulos-Orfanos, Jean-François Mangin
Service Hospitalier Frédéric Joliot, CEA
4 place du Général Leclerc, F-91401 Orsay cedex, France
papadopo@shfj.cea.fr

Abstract

Algorithm libraries dedicated to image processing and pattern recognition are not reusable; to run an algorithm on particular data, one usually has either to rewrite the algorithm or to manually "copy, paste, and modify". This is due to the lack of genericity of the programming paradigm used to implement the libraries. In this paper, we present a recent paradigm that allows algorithms to be written once and for all and to accept input of various types. Moreover, this total reusability can be obtained with a very comprehensive writing and without significant cost at execution, compared to a dedicated algorithm. This new paradigm is called "generic programming" and is fully supported by the C^{++} language. We show how this paradigm can be applied to image processing and pattern recognition routines. The perspective of our work is the creation of a generic library.

1 Introduction

Great effort has gone into building image processing and pattern recognition libraries that can be used and augmented by different research centers. However, the main difficulty that is systematically encountered and that remains unsolved is how to manage the large number of input types used in this domain. An algorithm developed for a particular input can rarely be reused. As a consequence, no one library has succeeded in being unanimously adopted by the scientific community.

An ideal library should be generic, i.e. supply generic algorithms. A generic image processing algorithm is written once, and indistinctly accepts 2D and 3D images (isotropic or not), regions, region adjacency graphs, image and graph pyramids, sequences, collections and so forth; the types of data contained in these structures is scalar (Boolean, integer or float), complex, composed (e.g. RGB). Existing libraries are usually dedicated to a particular data structure (mostly 2D images) and their algorithms are restricted to few data types (mostly unsigned 8 bit integers). Ideal pattern recognition algorithms also have this problem: for instance, a given primitive such as a contour can have different representations.

Most algorithm data can have different forms (i.e., different types) and should be used as the input of algorithms in a transparent way. In this paper, we show that recent advances in C^{++} programming allow this genericity with a very comprehensive syntax and without leading to a significant extra cost of execution time as compared to dedicated algorithms. This new paradigm is called "generic programming". We have successfully applied it both to low level image processing routines and to high level pattern recognition algorithms in a library that we are currently developing.

In section 2, we present the generic programming paradigm and its benefits compared to usual paradigms. Then, in section 3, we explain how to design a generic algorithm and we show with a simple example algorithm how this paradigm can be applied to the field of image processing, and how we can obtain maximal genericity. Lastly, in section 4, we conclude and give future perspectives of our work.

2 The generic programming paradigm

To emphasize the benefits of generic programming applied to image processing and computer vision, we will first point out some drawbacks of existing libraries.

2.1 Current libraries

In current *C* libraries, an algorithm has to be written as many times as there are input types [1]; see figure 1. For instance, a simple addition of a constant to image elements leads to four routines if we want this algorithm to deal with 2D and 3D images with Boolean or floating elements. Since the combination "algorithms $\times$ structure types $\times$ data types" can be enormous, many libraries limit the number of structure types and data types handled by each algorithm. The *C*-like programming paradigm has two main drawbacks: the capabilities of such libraries are limited, and introducing a new structure type or data type is a tedious task.

Some object languages such as C^{++} offer genericity, which means that classes and procedures can be parametrized. A common use of genericity is to parameterize the definitions of data structures and routines by the data type of their elements. As a consequence, reusability is enhanced but, although some libraries use genericity in this way [6, 5], the reusability is far from being total: routines still have to be written for each structure type. In fact, existing libraries do not rely on the generic programming paradigm presented below.

The limits of reusability of image processing algorithms induced by different programming paradigms are explained with further details in [3].

2.2 The novelty

Generic programming is a new paradigm to write fully generic algorithms without leading to significant overheads at run-time as compared to dedicated code. This paradigm is very attractive for scientific numerical programming and is used in some recent libraries: CGAL [2] and Blitz++ [7], respectively dedicated to geometric and algebraic calculi.

The generic programming paradigm is based on two key ideas:

- an algorithm is parametrized by its input types (in contrast to data parameterization, see section 2.1),
- the tools, helper objects needed by the algorithm, are deduced from its input (like the iterators presented in section 3.2).

When an algorithm is used, the compiler generates the appropriate machine code for the particular input types; see figure 2. Moreover, each method call in this code can be replaced by its implementation, so the cost of method calls is avoided. Therefore, the executable code is similar to that of a routine written for the particular input types and generic procedures are roughly as fast as dedicated procedures. Generic programming makes the compiler do the work that the programmer has to do in usual programming.

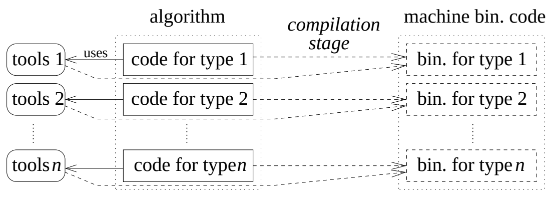

Figure 1. Usual mechanism in *C*.

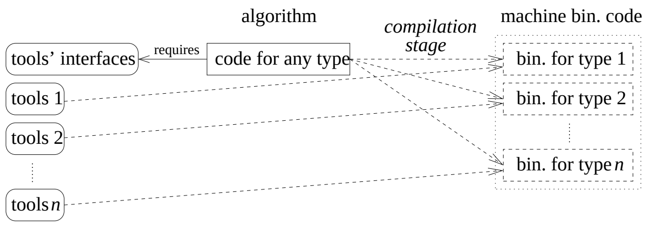

Figure 2. Generic mechanism in C^{++}.

C^{++} is a language that offers a good compromise between efficiency and generic capabilities [4], so all existing generic libraries are implemented in C^{++}. In the next section, we present the syntactic bases which are of prime importance to understand the generic programming paradigm.

2.3 Syntax

The sample code below gives a part of the definition of a generic 2D image class in C^{++}:

```
template< typename T >  // parameterization
class Image2D {
  public:
    typedef T value_type;  // a type alias
    size_t size() const;   // a method
    //...
};
```

This code contains two main syntactical elements.

Firstly, the class definition is parametrized (keyword `template`) and a single parameter is defined, `T`, whose nature is a type (keyword `typename`). This parameter represents the type of the image elements. For instance, the programmer can then use the type `Image2D<float>` to manipulate 2D images containing floating values. The parameterization thus allows the definition of only one class per structure type in the library whatever the data type.

Lastly, one can define a type alias (keyword `typedef`) within a class; for instance, the parametric class `Image2D` contains the alias `value_type`, which gives the element type, `T`. Such an alias is used as follows:

`typename class_name::alias_name` .

Let us consider a procedure that is parametrized by the type `I` of an input aggregate; this aggregate is the argument `input` of the procedure. In the procedure body, the programmer can use type aliases such as `I::value_type`, and methods of `I` such as in `input.size()`; see for instance `proc` in the code below:

```
template< typename I >  // parametrization
void proc( I& input ) {
  size_t  a_size = input.size();
  typename I::value_type  a_value;
  //...
}

int main() {
  Image2D<float>  ima;
  proc( ima );
}
```

When `proc` is called, the type `I` is known, and the dedicated procedure is compiled (if it has not already been compiled). The compiler then checks that the type effectively contains a method `size()` and an alias `value_type` (these two points represent a required interface). Since it is the case for the type `Image2D<float>`, in `main()`, the variable `value` is of type `float` in the compiled routine.

3 Generic algorithm design

The generic programming paradigm being quite recent, algorithm design does not follow a standard process. So, we first present the method that we have established.

3.1 Design method

The design method is sequential and composed of several steps.

1. Express the algorithm in mathematical language while paying attention to remaining as general as possible. This step ensures that we will not provide an algorithm that depends upon particular considerations (for instance, an algorithm linked to a given data structure).
2. Identify the objects that are involved in the expression obtained at the previous step and describe their role and their required behavior.
3. Point out each algorithm option that should be set by the user, and give each option a default value if it is pertinent. If needed, return to step 2 to take into account these options.
4. Analyze the dependencies between the types of the objects used in the algorithm, and deduce its parameters.
5. Finally, write the generic algorithm.

3.2 A simple example

To illustrate how to obtain the highest genericity for an image processing or pattern recognition algorithm, let us study a very simple example: the addition of a constant to the elements of an aggregate.

Step 1 For the sake of genericity, let us first generalize this example and consider that the algorithm has to be designed for any operator similar to `val += cst`. We can then formulate this algorithm as follows : “for each element of an input aggregate, apply the operator to the element value”. Please note that this description conceals all implementation details related to the particular types of the objects that could be involved in the algorithm.

Step 2 The objects involved in the algorithm are: an input aggregate (argument `input`), an iterator (internal variable `iter`), a constant value (argument `cst`).

To translate this description into a generic algorithm, the introduction of an object which iterates over the elements of an aggregate is required; such an object, called an *iterator*, is a common tool in software design. One iterator class is defined per structure type and all iterators provide the same interface (i.e., the same subset of methods and aliases) in order to be uniformly manipulated. In this way, iterating over graph vertices leads to the same syntax as iterating over image points.

Step 3 The options of the algorithm are:

- an operator (for instance, a saturated addition),
- a predicate to restrict the addition to certain elements of the aggregate (by default, the predicate returns always true),
- an accessor to specify, if the element type is structured, which field is concerned by the addition (by default, the accessor is the identity; its name is `get_value<>`),

Each option is translated into one extra object in the algorithm.

Step 4 Let us denote by `I` the type of the input aggregate, by `C` the type of the constant value, by `O` the type of the operator, by `P` the type of the predicate, by `get_A` the type of the accessor (when accessing field `A` of `I`), and by `T` the type of the iterator. Then, we have the following statements:

- `C` is given by `get_A::output_type`,
- `T` is given by `I::iterator_type`,
- `get_A<>::input_type` is given by `I::value_type` and `get_A<>::output_type` is given by `I::value_type::A_type`,
- `O::args_type` is given by `get_A::output_type`.

The parameters of the algorithm are: `I`, `O`, `P`, and `get_A`. The first parameter is known when the algorithm is called and the last two parameters have default values. The only parameter that the user has to set is `O`.

Step 5 The core of the algorithm is transformed into the following description. Define `iter` on `input`; then, for each iteration handled by `iter`, conditioned by `pred`, apply `oper` with `cst` on the value given by `iter` through `access`. Finally, this algorithm is ready to be translated into the generic C^{++} code[1]:

```
template< typename O,
          template< class U > class get_A = get_value,
          typename P = Pred_true >
struct op
{
   template< typename I > static
   void on( I& input,
            const get_A< I::value_type >::output_type& cst,
            P pred = P() )
   {
      O oper;    get_A< I::value_type > access;
      I::iterator_type  iter( input );
      for ( iter.first(); ! iter.isDone(); iter.next() )
         if ( pred( access( iter() ) ) )
            oper( access( iter() ), cst );
   }
};
```

This algorithm accepts various structure and data types. Moreover, the user can parameterize its behavior (for instance, the user can subtract a constant value from the red component of some elements of a 3D image). Since the user is not required to set all parameters, the simplest call of the algorithm (arithmetical plus) is:

```
        op<plus>::on( input, cst );
```

4 Conclusion

In this paper, we have presented the generic programming paradigm that enables the implementation of generic algorithms. Then, we have demonstrated with an example how to apply this paradigm to the field of image processing and pattern recognition and how to obtain the highest genericity.

The main difficulty of building a generic library lies in correct designing of algorithms and tools. For that, the closer the code is to the theory, the better the genericity is. Although the example given in this paper is very simple, the design process is rigorously equivalent for high-level routines.

This paradigm has five major advantages that we set out below.

Reusability Since algorithms are generic, their reusability is maximal. Each algorithm is programmed only once and accepts data of various types. When one wants to introduce a new data type in the library, one only has to conform to few requirements in order to benefit from the existing algorithms.

Functionalities From the viewpoint of the user, such a library is no more complicated than current libraries. Algorithms can be called very simply because they define their own default settings, while it remains possible for the user to be more specific.

Development The development cost of a generic algorithm is dramatically reduced as compared to that of current libraries (see figures 1 and 2). Consequently, maintenance and reliability are significantly improved.

Efficiency Generic algorithms are roughly as fast as dedicated algorithms (the compiler expands generic code and makes it similar to dedicated code).

Federative A generic library is able to federate tools and algorithms developed by different research centers. We believe that this point is of prime importance for the community because such a library enhances the capitalization of knowledge.

We are currently developing such a library whose first version will be soon freely available. We do not aim at providing a wide range of algorithms and tools but the most usual ones to facilitate algorithm programming.

[1]The full implementation of the example is available at the URL `http://www.lrde.epita.fr/download/`

References

[1] M. Dobie and P. Lewis. Data structures for image processing in C. *Pattern Recognition Letters*, 12(8):457–466, 1991.

[2] A. Fabri, G. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the computational geometry algorithms library. Technical Report 3407, INRIA, 1998.

[3] T. Géraud, Y. Fabre, D. Papadopoulos-Orfanos, and J.-F. Mangin. Vers une réutilisabilité totale des algorithmes de traitement d'images. In *17th Symposium on Signal and Image Processing (GRETSI'99)*, volume 2, pages 331–334, Vannes, France, September 1999. In French; available in English as a technical report at `http://www.lrde.epita.fr/publications`

[4] S. Haney and J. Crotinger. How templates enable high-performance scientific computing in C++. *IEEE Computing in Science and Engineering*, 1(4), 1999.

[5] C. Kohl and J. Mundy. The development of the Image Understanding Environment. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition*, pages 443–447, 1994.

[6] G. Ritter, J. Wilson, and J. Davidson. Image Algebra: an overview. *Computer Vision, Graphics, and Image Processing*, 49(3):297–331, 1990.

[7] T. Veldhuizen. Arrays in Blitz++. In *Proc. of the 2nd Intl. Conf. in Object-Oriented Parallel Environments (ISCOPE'98)*, number 1505 in Lectures Notes in Computer Science, pages 223–230. Springer Verlag, 1998.