In the Proceedings of the IASTED International Conference on Applied Informatics (AI'2001) – Symposium on Advances in Computer Applications, ACTA Press, pages 577-581, Innsbruck, Austria, February 2001.

APPLYING GENERIC PROGRAMMING TO IMAGE PROCESSING

T. Géraud, Y. Fabre, A. Duret-Lutz

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

Abstract

This paper presents the evolution of algorithms implementation in image processing libraries and discusses the limits of these implementations in terms of reusability. In particular, we show that in C++, an algorithm can have a general implementation; said differently, an implementation can be *generic*, i.e., independent of both the input aggregate type and the type of the data contained in the input aggregate. A total reusability of algorithms can therefore be obtained; moreover, a generic implementation is more natural and does not introduce a meaningful additional cost in execution time as compared to an implementation dedicated to a particular input type.

Keywords

Image processing, genericity, reusability, programming.

1 Introduction

Several times, great efforts have been performed in building an image processing library to provide a standard framework to laboratories. However, one of the main difficulties of such a project is taking into account the diversity of the problems tackled in image processing. In particular, the diversity of data aggregates (2D or 3D images, isotropic or not, sequences or pyramids of images, regions or adjacency graphs of regions, collections, etc.) and of data (Boolean, integer or floating formats, complex, structures like RGB, vectors, etc.)

In this paper, we focus on image processing algorithms programming techniques aiming to reuse these algorithms vis-à-vis of the data type and of the aggregate type that these algorithms can support. We are going to illustrate our remarks on two very simple processing examples: the addition of a constant to the value of the elements of an aggregate (for example the points of an image) and a mean filter on an aggregate. We will finally see that a completely general implementation of these algorithms is possible in C++ thanks to genericity and to type deduction; we will also see that this kind of implementation does not cause any significant additional cost in time of execution and that it is close to a natural language description of these algorithms.

The code given in this paper is deliberatly simplified in order to make easier the comprehension of the presented programming techniques. The entire code is available on the Internet at the location: `http://www.lrde.epita.fr/download/`.

2 C-Like Programming

In languages like the C (put differently, languages that are imperative, procedural and structured), an aggregate type is defined by a structure, where the field `data` that holds the values of the aggregate elements is untyped and where the field `type` gives information on their type. This style of definition is illustrated below:

```
typedef struct
{
  int     nRows;
  int     nColumns;
  type_t  type;
  void*   data;
}
  Image2D;
```

The advantage of this structure is to provide a unique definition for each aggregate type, and so whatever is the data type [1]; consequently, a processing algorithm on a given aggregate is translated into a unique function. For a function to support as input 2D images which data are unsigned 8 bit integers (`char` type in C), a routine specific to this type must be written in the function:

```
void add( Image2D* image, void* val )
/* function for a given aggregate type */
{
  int iRow, iColumn;
  switch ( image->type )
  {
    case CHAR:
    /* routine for a given data type */
      char** dataChar = toChar2D( image );
      char valChar = toChar( val );
      for ( iRow = 0;
          iRow < image->nRows;
          ++iRow )
        for ( iColumn = 0;
              iColumn < image->nColumns;
              ++iColumn )
          dataChar[iRow][iColumn] += valChar;
      break;
    /* other cases... */
    /* i.e., routines for other data types */
  }
}
```

T processing operators, S aggregate types and D data types lead to $T \times S \times D$ different routines. In order to reduce this combinative, many libraries limit the number of data types and aggregate types available for the user, and restrict the applicability of most processing operators to a small number of aggregates and data types. Thus, for aggregate types, most libraries manage 2D images, sometimes 3D images, and rarely adjacency region graphs. We find for data types, from the most classic to the less frequent, unsigned and signed integers on 8 and 16 bits, floating values, Booleans, RGB colors with unsigned integer components on 8 bit.

The traditional C-like programming drives to a nearly nil reusability: to introduce a new entity (a data type, an aggregate type, or a processing operator) imposes in such a library an implementation effort that results most the time in manual "copy-paste-modify".

3 Classic Object-Oriented Programming

3.1 Genericity

To avoid implementing a routine for each data type, some object-oriented languages such as C++ permits to define data aggregates with parameterized classes. This technique, used in the IAC++ [2] and IUE [3] libraries, leads to definitions below, where the parameter `Data` is the data type. The implementations of processing operators are parameterized in the same way.

```
template< class Data >
class Image2D
{
  int     nRows;
  int     nColumns;
  Data**  data;
  // ...
};

template< class Data >
void add( Image2D<Data>& image,
          Data val )
{
  int iRow, iColumn;

  for ( iRow = 0;
        iRow < image.nRows;
        ++iRow )
    for ( iColumn = 0;
          iColumn < image.nColumns;
          ++iColumn )
      image.data[iRow][iColumn] += val;
}
```

From this code, said *generic*, the compiler automatically constructs the necessary declensions, i.e., routines. Thus, the declaration and the call:

`Image2D<char> image; add( image, 3 );`

will force the compilation of the class `Image2D<char>` and of the function `add( Image2D<char>&, char );` the machine code of the routine dedicated to a particular type of data (the one of the section 2) is produced therefore automatically by the compiler.

Generic programming presents two advantages: typing is strong (the anonymous type `void*` disappears) and processing implementations become independent of data types. For the programmer, the number of functions to implement is then reduced to $T \times S$ (a function is a generic routine); but, each function is still dependent of the aggregate type (the function of our example only accepts 2D images).

3.2 Inheritance and Dynamic Polymorphism

In order to make a unique function handle various aggregate types, it is necessary to abstract details of implementation bound to every aggregate type. This is feasible while presenting a unique interface to manipulate these different aggregate types and while making intervene the notion of methods redefinition via inheritance. In [4], an object-oriented design of aggregates and iterators is given; here, we adapt it to our problematic.

Two abstract classes are defined: `Aggregate<Data>` and `Iterator<Data>`. The former class represents an aggregate of elements (the elements type is `Data`) and declares the abstract method `createIterator()` for the instantiation of an iterator over the aggregate. The latter class represents an iterator on this aggregate and declares a set of methods for the manipulation of an iterator:

- `void first()` to initialize the position of the iterator on the first element of the aggregate,
- `bool isDone()` to know if the iterator has finished to browse the aggregate,
- `Data& value()` to recover the value of the aggregate element corresponding to the current position of the iterator,
- `void next()` to move the iterator to the next aggregate element.

From these two abstract classes derives respectively the specialized concrete classes: `Image2D<Data>` and `Image2DIterator<Data>`. An iterator on the points of a

2D image has three attributes: the indices of raw and column of the current point and the reference to the targeted 2D image.

Finally, the implementation of the addition operator is:

```
template< class Data >
void add( Aggregate<Data>& aggregate,
          Data val )
{
  Iterator<Data>& iter
      = aggregate.createIterator();
  for ( iter.first();
        ! iter.isDone();
        iter.next() )
    iter.value() += val;
}
```

With this design, as shown below, only the abstract classes intervene in the implementation of the addition. A binding is performed at run-time to call the methods specific to the effective types (called dynamic types) of the objects `aggregate` and `iter`. Thus, with the declaration and call:

`Image2D<char> image; add( image, 3 );`

the method `createIterator()` that is executed is the one of the class `Image2D<char>`, the dynamic type of `iter` is `Image2DIterator<Data>`, and the executed iteration methods are indeed the ones of an iterator on a 2D image.

Let us consider now the example of a mean filter. We can define two new iterator types: a follower iterator and a neighbourhood iterator. An implementation using a classic object-oriented design is given below, with a macro `for_each` that simplifies the implementation of the loop over the aggregate elements. Here, `inputElt` iterates on the elements of the input aggregate, `neighborElt` iterates on the neighbourhood of the current element of the input aggregate, and `outputElt` iterates on the output aggregate in an equivalent way to the input aggregate iteration.

```
template< class Data >
void mean( Aggregate<Data>& inputAggregate,
           Aggregate<Data>& outputAggregate )
{
  Iterator<Data>& inputElt
      = inputAggregate.createIterator(),
  Iterator<Data>& neighborElt
      = inputElt.createNeighborIterator(),
  Iterator<Data>& outputElt
      = outputAggregate
        .createFollowerIterator( inputElt );

  for_each( inputElt )
  {
    Data sum = 0;
    for_each( neighborElt )
      sum += neighborElt.value();
    outputElt.value() = sum
                  / neighborElt.getCard();
  }
}
```

This implementation uses inheritance and dynamic polymorphism (more precisely, it involves both *inclusion polymorphism* and *operation polymorphism* [5]). In our examples, it permits to deduct from the type of the aggregate, input of processing operators, the adequate iterators. The functions are thus independent from both data type and aggregate type. The number of routines is therefore minimal: (T), one processing operator being programmed once for all. Besides, the implementation of a processing operator is nearly close to the natural language description of the operator algorithm.

3.3 Drawbacks

The solutions based on classic object-orientation present however four drawbacks, enumerated in [6]. The most penalizing in the field of numeric computing is the cost resulting from every call to a polymorphic method. Indeed, such a call results in two indirections: one for the access to the methods of the effective class of the targeted object and one for the access to the considered method.

In order to estimate the cost of these indirections, we have realised performance tests. The first test corresponds to the addition of a constant on a 2D image made of 8 millions points; the second test corresponds to a mean filter with a 6-neighborhood of a 3D image made of 8 millions points. The results are given in table 3.3 (the benchmarks have been performed on a 333 MHz PC with Linux and with the GNU C++ compiler). The use of classic object-orientation is prohibitive, the repetition of calls to the methods `first()`, `value()`, `isDone()` and `next()` being very expensive.

	C-like programming	dynamic polymorphism	static polymorphism
addition	0.5 s	1.5 s	0.8 s
mean	3.5 s	11.4 s	3.9 s

Table 1: Compared Performance From Different Programming Techniques.

Classic object-orientation is an interesting solution to make a unique routine handle various input types; however, in the context of image processing where the size of aggregates (for instance, the number of points of a 2D image) can be important, this "classic" approach is inadequate.

4 New Approach: Generic Programming

Accepted in 1994 by the ANSI/ISO C++ normalization committee as part of the C++ standard library, the *Standard Template Library* (STL) [7] provides a set of containers and algorithms. This library has helped the *generic programming* paradigm to become popular. This paradigm is used by some recent libraries [8] (see also the technical report [6]).

4.1 Type Aliases and Static Polymorphism

The key idea of generic programming is to parameterize the algorithms, by the aggregate type (Aggregate) rather than by the data type. The deduction of types from a aggregate type is achieved by type aliases defined in aggregate classes. The declaration of the class Image2D<Data> that allows the use of the type Image2D<Data>::Iterator, alias for Image2DIterator<Data>, is given below with the resulting implementation of the mean algorithm.

```
template<class Data>
class Image2D
{
public:
  typedef Data                    DataType;
  typedef Image2DIterator<Data>
                                  Iterator;
  typedef Image2DFollowerIterator<Data>
                                  FollowerIterator;
  typedef Image2DNeighborIterator<Data>
                                  NeighborIterator;
  // ...
};

template<class Aggregate>
void mean( Aggregate& inputAggregate,
           Aggregate& outputAggregate )
{
  typename Aggregate::Iterator
      inputElt( inputAggregate );
  typename Aggregate::NeighborIterator
      neighborElt( inputElt );
  typename Aggregate::FollowerIterator
      outputElt( outputAggregate, inputElt );

  for_each( inputElt )
  {
    typename Aggregate::DataType::CumulType
        sum = 0;
    for_each( neighborElt )
      sum += neighborElt.value();
    outputElt.value() = sum
                  / neighborElt.getCard();
  }
}
```

The definition of the abstract classes Aggregate<Data> and Iterator<Data> is not longer necessary and inheritance becomes obsolete. Every object targeted by method calls has here a concrete type, known at compile-time. The execution over-cost at run-time due to dynamic polymorphism in the classic object-oriented paradigm is therefore avoided.

Besides, the C++ language offers a mechanism that allows the compiler to replace a method call by the method code (when the method is not polymorph). If the function mean() given above is used for the aggregate type Image2D<IntegerS8>, the machine code generated by the compiler will be rigorously equivalent to the one of the section 2.

Finally, execution times of generic programming implementations of algorithms only present a light additional cost as compared to those of dedicated implementations, as testifies table 3.3. Generic programming is therefore a pertinent solution to the twofold problematic of implementing general algorithms (i.e, applicable on various aggregate types and data types) and having efficient processing functions. The intensive use of parameters and type aliases leads to *static polymorphism* (more formally, this polymorphism is also called *parametric polymorphism* [5]), in contrast to dynamic polymorphism.

4.2 Refinements

Generic programming permits to differentiate the particularities of certain aggregate and data types. Indeed, if an algorithm can or must be written differently for a specific type, all that is necessary is to overload the generic implementation by defining a function with the same signature but with an explicit parameter.

About data types, a same predefined storage type of a language can serve to data types of different semantics (for example, an unsigned integer coded on 8 bit can also represent a discrete fuzzy value); to make their distinction, which conducts to different implementations, it is necessary to define data types as classes. So, the class IntegerS8 replaces advantageously the type char; in particular, the type of the sum variable that serves to sum up the values is being deducted.

We currently develop a platform for image processing [9] that uses in an hybrid way, in function of the needed performances, classic and generic programming. Without increasing the simplicity of algorithm implementations, we also have obtained two additional forms of genericity that does not lead to a meaningful additional cost at the execution: every algorithm implementation can be restricted to the elements of an aggregate that verify a given predicate and can concern a field of data solely (for example, the red component of RGB). For that, we have translated classical *design patterns* to the generic programming paradigm [10].

5 Conclusion

In this paper, we showed that the evolution of programming techniques permits nowadays to have general and efficient implementations for image processing algorithms. These implementations are then completely reusable: they accept the introduction of new data types and aggregate types. Their mode of implementation, generic programming, of which we have presented the principle, is adapted to processing operators of various natures: filtering, Markovian relaxations, scale-space methods, and so on.

References

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

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

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

[4] E. Gamma, R. Helm, R. Johnson, & J. Vlissides, *Design patterns – Elements of reusable object-oriented software* (Professional Computing Series, Addison Wesley, 1995).

[5] L. Cardelli & P. Wegner, On understanding types, data abstraction, and polymorphism, *Computing Surveys*, 17(4), 1985, 471–522.

[6] A. Fabri, G.J. Giezeman, L. Kettner, S. Schirra, & S. Schönherr, *On the design of CGAL, the computational geometry algorithms library*, Technical Report 3407, INRIA, France, 1998.

[7] A. Stepanov & M. Lee, *The Standard Template Library*, Hewlett Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, February 1995.

[8] The object-oriented numerics page, `http://oonumerics.org/oon/`

[9] T. Géraud, Y. Fabre, A. Duret-Lutz, D. Papadopoulos-Orfanos, & J.-F. Mangin, Obtaining Genericity for Image Processing and Pattern Recognition Algorithms, *Proceedings of the 15th International Conference on Pattern Recognition (ICPR'2000)*, vol. 4, Barcelona, Spain, September 2000, 816–819.

[10] A. Duret-Lutz, T. Géraud, & A. Demaille, Generic Design Patterns in C++, *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001)*, San Antonio, TX, USA, January-February 2001. To appear.