Practical Genericity: Writing Image Processing Algorithms Both Reusable and Efficient

Roland Levillain[1], Thierry Géraud[1], Laurent Najman[2], and Edwin Carlinet[1,2]

[1] EPITA Research and Development Laboratory (LRDE)
14–16, Rue Voltaire, FR-94276 Le Kremlin-Bicêtre, France
thierry.geraud@lrde.epita.fr
[2] Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge, Équipe A3SI
ESIEE Paris, Cité Descartes, BP 99, FR-93162 Noisy-le-Grand, France
laurent.najman@esiee.fr

Abstract. An important topic for the image processing and pattern recognition community is the construction of open source and efficient libraries. An increasing number of software frameworks are said to be generic: they allow users to write reusable algorithms compatible with many input image types. However, this design choice is often made at the expense of performance. We present an approach to preserve efficiency in a generic image processing framework, by leveraging data types features. Variants of generic algorithms taking advantage of image types properties can be defined, offering an adjustable trade-off between genericity and efficiency. Our experiments show that these generic optimizations can match dedicated code in terms of execution times, and even sometimes perform better than routines optimized by hand.

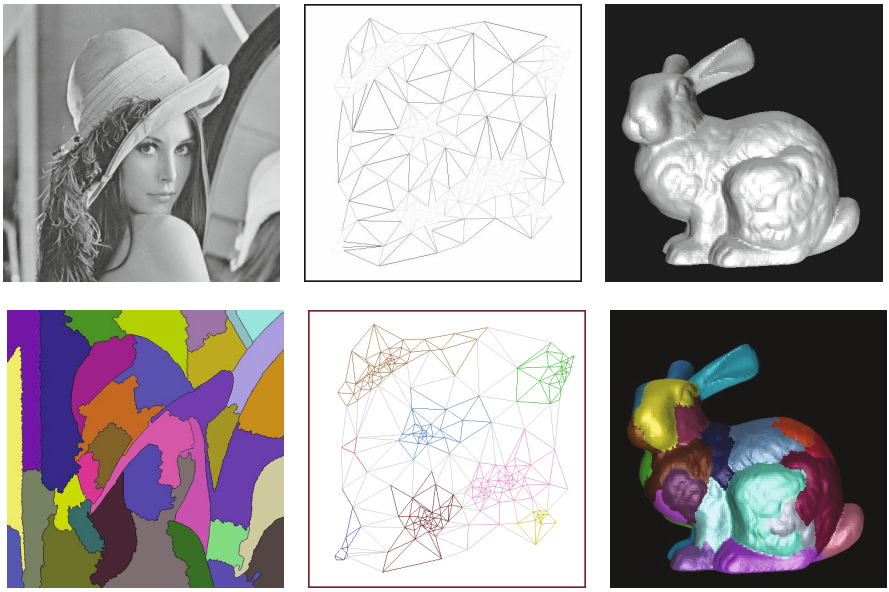
Keywords: Image Processing, Software Reusability, Computational Efficiency, Generic Programming, Reproducible Research.

1 Introduction

Like many numerical computing fields of computer science, Image Processing (IP) faces two kinds of issues which are hard to solve at the same time. On the one hand, the spectrum of data to process is very wide: IP methods can be used with "classical" 2-dimensional images with square pixels arranged along a regular grid, containing binary, gray-level, color, vectorial or even tensorial values; the dimension may also vary: 1D (signals), 3D (volumes), 2D+t (sequences of images) 3D+t (sequences of volumes); the domain of the image itself may be non-regular: images can be defined on mathematical structures such as graphs, simplicial complexes or topological maps.

The ability to handle so many data structures depends on the *genericity* of the theoretical framework and on the corresponding software tools as far as implementation is concerned. A *generic algorithm* is an algorithm that can be applied to a variety of inputs [10], as opposed to a *specific algorithm*, which applies to a unique data type [6]. Generic Programming (GP) is a field of computer

E. Bayro-Corrochano and E. Hancock (Eds.): CIARP 2014, LNCS 8827, pp. 70–79, 2014.



(a) On a "classical" regular grid.

(b) On an edged-valued graph.

(c) On a 3D surface mesh (implemented as a simplicial complex).

Fig. 1. Results of the *same* morphological segmentation processing chain code applied to different input image types (a 2D square-pixel image, a graph, and a 3D surface); this particular example of a generic processing chain is detailed in [22]

science (and a programming paradigm) which is concerned with issues of genericity in software[1]. Several software projects dedicated to IP rely on GP: Vigra [18,19], ITK [13], Morph-M [3], GIL [1]. Other noteworthy applications of GP to scientific computing include computational geometry [7] and graph theory [25].

One of the incentives behind GP is the *reusability* of software, e.g. minimizing the cost of using existing algorithms with new data structures and vice versa. Such a cost may lie in bad software engineering practice such as duplicating code to process various types of inputs; or weakening typing (and most likely impact run-time efficiency) through generalization (e.g. using `double` as input type to also allow the use of `bool`s, `int`s, or other types of values). Even well-engineered solutions, such as object-oriented ones, may harbour costs: dynamic dispatch (virtual methods) can become a prohibitively expensive feature w.r.t. execution times [12].

On the other hand, many IP issues and applications involve large data sets (either numerous or voluminous) or make use of complex techniques requiring

[1] In this paper, by "generic" we mean "generic with respect to programming", not "generic with respect to the approach used to solve an IP issue". See for instance the definition of GP by Jazayeri *et al.* [16].

intensive computations. In both cases, practical software solutions have to meet *efficiency* constraints, especially regarding processing speed. Unfortunately, genericity and efficiency are often conflicting: efficient IP software is mostly dedicated to specific image types, methods or domains (and is therefore not generic). Conversely, most generic frameworks do not provide performances as good as specific ones.

In this paper, we investigate the issue of genericity versus efficiency with respect to IP, from the algorithmic point of view (i.e., we do not cover hardware-based or assembly-level optimizations). We first explain how GP can be applied to IP, and what are the benefits of this approach (Section 2). Performance considerations in a generic context are addressed in Section 3. We study the causes of the opposition between genericity and efficiency and propose a trade-off as an answer to this problem, *generic optimizations*. The idea of generic optimization is carried further in Section 4, by not only acting on algorithm implementations, but by also working on input data types, so as to significantly increase performances at the expense of some genericity. Results of numerical experiments are presented and discussed in Section 5. The proposal of this paper is illustrated with an example from the field of Mathematical Morphology. The underlying ideas, are however very general, independent from the context (platform, compiler, etc.), and applicable to virtually any IP algorithm.

2 Genericity in Image Processing

A *generic* IP framework provides algorithms and data structures that have a *single* implementation and that can be used together to virtually implement any (valid) combination. This approach avoids duplication of similar code and enables a true reusability of algorithms on any compatible data structure (e.g., image type) without suffering from the combinatorial explosion. For example, the result images (right-hand column) presented in Figure 1 have been obtained using the *same* segmentation code on three images of very different types.

The simplest (though very limited) form of genericity in IP consists in parameterizing the type of values contained in images [24,17], similarly to C++ standard containers [14]. However, genuine genericity is based on the GP paradigm. The key idea of this approach is to design the target framework using *concepts*, representing abstract entities of the domain (here, image processing) [21]. Concepts define relationships between the corresponding entity (e.g. an image type) and other elements (e.g. image point type, image value type), as well as the minimal set of provided services (e.g. obtaining the value associated to a point). Generic algorithms are then written using concepts instead of specific concrete data types. As they expose no detail on the manipulated data types, such algorithms are generic implementations not tied to a specific input type. We have successfully applied this approach to Mathematical Morphology (MM) [5,20], discrete geometry [22], as well as other fields of IP. In particular, one of the benefits of the generic approach is to enable user to try and experiment with uncommon and new data structures such as the *Tree of Shapes* [9,4,26,27], easily and rapidly.

```
image dilation(const image& input) {
  image output(input.nrows(), input.ncols());
  for (unsigned r = 0; r < input.nrows(); ++r)
    for (unsigned c = 0; c < input.ncols(); ++c) {
      unsigned char sup = input(r,c);
      if (r != 0 && input(r-1,c) > sup)
        sup = input(r-1,c);
      if (r != input.nrows()-1 && input(r+1,c) > sup)
        sup = input(r+1,c);
      if (c != 0 && input(r,c-1) > sup)
        sup = input(r,c-1);
      if (c != input.ncols()-1 && input(r,c+1) > sup)
        sup = input(r,c+1);
      output(r, c) = sup;
    }
  return output;
}
```

Algorithm 1.1. Non-generic dilation implementation

```
template <typename I, typename W>
I dilation(const I& input, const W& win) {
  I output; initialize(output, input);
  // Iterator on sites of the domain of 'input'.
  mln_piter(I) p(input.domain());
  // Iterator on the neighbors of 'p' w.r.t. 'win'.
  mln_qiter(W) q(win, p);
  for_all(p) {
    // Accumulator computing the supremum on 'win'.
    accu::supremum<mln_value(I)> sup;
    for_all(q) if (input.has(q))
      sup.take(input(q));
    output(p) = sup.to_result();
  }
  return output;
}
```

Algorithm 1.2. Generic dilation implementation [20]

Let us illustrate the topic using a simple MM algorithm: a morphological dilation using a flat structuring element [11]. Algorithm 1.1 shows a simple C++ implementation of this filter. It however includes several implementation details which bounds the routine to specific inputs (2D image on a regular grid, having scalar values compatible with `unsigned char`). Moreover the (4-connected) structuring element cannot be changed. Therefore, we cannot use this algorithm to process, e.g., a 3D image composed of RGB color values and using 6-connectivity.

On the other hand, Algorithm 1.2 proposes a generic version of the previous morphological dilation. Input (resp. image and structuring element) types are now parameters of the algorithm (resp. `I`, an image type, and `W`, a *window* type); loops on vertical and horizontal ranges have been replaced by a single object `p` traversing the domain of the image `input`, called a *site iterator* on `input`'s domain; likewise, members of the (previously hard-coded) structuring element w.r.t. `p` are no longer explicitly mentioned, as a another iterator `q` on the window (a *qiter*) is used for this purpose; and instead of a hand-made computation of

the maximum value, an *accumulator* object is used to iteratively compute the supremum from the values within the sliding window.

3 Efficiency vs Genericity Trade-off

Following the ideas expressed the previous section, we have designed and implemented a generic C++ IP library, *Milena*[2], which is part of the Free Software IP platform *Olena* [23]. In addition to providing generic algorithms and data structures, Milena offers an extensible mechanism to implement efficient *and* reusable variants of some routines. In this section, we show how to implement efficient algorithms displaying a generic nature and how to wholly integrate them in the generic framework so that their use can be made transparent.

3.1 The Cost of Abstraction

Figures from Table 1 exhibit an important run time overhead in the generic case (Algorithm 1.2), which is about ten times longer to execute than the non-generic one (Algorithm 1.1). This is not a consequence of the GP paradigm per se. It is actually because of the highly abstract style of Algorithm 1.2, which in return makes the routine very versatile with respect to the context of use. The non-generic version is faster than the generic one because it takes advantage of known features of the input types. For instance the structuring element is "built in the function", whereas it is an object taken as a generic input in Algorithm 1.2. Therefore its contents and size are constant and known at compile-time. Such implementation traits convey useful *static* information that compilers can leverage to optimize code. Hence, what prevents a code from being generic seems to be the condition to generate fast code: implementation details.

3.2 Generic Optimizations

The trade-off between genericity and efficiency admittedly depends on the level of details, but these two aspects are not entirely antagonistic: by carefully choosing the amount of specific traits used in an algorithm, one can create intermediate variants showing good run-time performance while keeping many generic features.

For instance, a means to speed up Algorithm 1.2 is to avoid using site iterators to browse the domain common to the input and output images. In Milena, site iterators can be automatically converted into sites (points), that is, locations in the domain of one (or several) image(s). Such location information is not tied to a given image: in the case of a regular 2D image, a site `point2d(row, column)` is compatible with every regular, 2D, integer coordinate-based domain of the library (including toric spaces, non-rectangular 2D subspaces of $\mathbb{Z}^2$, etc.). This is why iterator `p` is used to refer to the same location in both `input` and `output` in Algorithm 1.2.

[2] Our library is available online from `http://olena.lrde.epita.fr`

```
template <typename I, typename W>
I dilation(const I& input, const W& win) {
  I output; initialize(output, input);
  // Iterator on the pixels of 'input'.
  mln_pixter(const I) pi(input);
  // Iterator on the pixels of 'output'.
  mln_pixter(I) po(output);
  // Iterator on the neighbor pixels of 'pi'.
  mln_qixter(const I, W) q(pi, win);
  for_all_2(pi, po) {
    accu::supremum<mln_value(I)> sup;
    for_all(q)
      sup.take(q.val());
    po.val() = sup.to_result();
  }
  return output;
}
```

Algorithm 1.3. Partially generic optimized dilation

The price to pay for such a general expression is usually a run-time overhead: computations have to be performed each time a site iterator is used to access data from an image. However, this flexibility is not always needed when the data to process exhibit certain properties. For instance, an image whose values are stored in a contiguous, linear memory space, can be browsed using a pointer, directly accessing values in a sequential manner using their memory addresses, instead of computing a location at each access. In Milena, such pointers are encapsulated in small objects called pixel iterators or *pixters* where a *pixel* refers to an image's (site, value) pair. A pixter is bound to one image and cannot be used to iterate any other image. Pixters can also be used to browse spatially-invariant structuring elements (windows) as long as the underlying image domain is regular.

Algorithm 1.3 shows a reimplementation of Algorithm 1.2 where site iterators have been replaced by pixel iterators. The code is very similar, except that images `input` and `output` are now browsed with two different pixel iterators, each of them holding a pointer to the corresponding data. Such an implementation of the morphological dilation is less generic than the one of Algorithm 1.2. Even so, it can still be used with a wide variety of image types, as long as their data present a regular organization, which comprises any-dimension classical images using a single linear buffer to store their values. Besides, it is compatible with any spatially-invariant structuring element (or in other words, any constant window). Thus it remains much more generic than Algorithm 1.1. As for efficiency, Algorithm 1.3 matches almost Algorithm 1.1 in terms of speed (see Table 1), so it is a good alternative to the generic dilation, when the trade-off between genericity and efficiency can be shifted towards the latter.

The approach presented here can be applied to other algorithms of the IP literature for which optimized implementations have been proposed. These optimizations are in practice compatible with a large set of input types, so their implementations can be considered as *generic optimizations* since they are not tied to a specific type.

4 Extra Generic Optimizations

The approach exposed in this paper can be carried further to improve the efficiency of generic optimizations. The idea is to involve data structures in the optimization effort: instead of acting only on algorithms, we can implement new optimized variants by working on their inputs as well.

For instance, in place of a window containing a dynamic array of vectors – the size of which is known at *run time* – we can implement and use a *static* window containing a static array carrying the same data, but whose size and contents are known at *compile time*. Modern compilers make use of this additional information to perform efficient optimizations (e.g, replace the loop over the elements of the window by an equivalent unrolled code). In this particular case, the implementation requires only the creation of two new, simple data types (static window, static pixel iterator). No additional implementation of the dilation is needed: Algorithm 1.3 is already compatible with this new window type. The resulting code delivers run times which are not only faster than the non-generic version of Algorithm 1.1, but which may also be faster than a hand-made, pointer-based optimized (hence even less generic) version of the dilation, as shown in the next section.

5 Results

Table 1 shows execution times of various implementations of the morphological dilation with a 4-connected structuring element (window) applied to images of growing sizes (512×512, 1024×1024 and 2048×2048 pixels) . Times shown correspond to 10 iterative invocations. Tests were conducted on a PC running Debian GNU/Linux, featuring an Intel Pentium 4 CPU running at 3.4 GHz with 2 GB of RAM clocked at 400 MHz, using the C++ compiler `g++` (GCC) version 4.4.5 with optimization options '`-O3`' and '`-DNDEBUG`'.

In addition to the implementations shown in this paper, an additional non-generic version using pointer-based optimizations has been added to the test suite, so as to further compare non-generic code – mostly optimized by hand – and generic code – mostly optimized by the compiler.

The overhead of the most generic algorithm is important: about ten times longer than Algorithm 1.1. The highly adaptable code of Algorithm 1.2 is free of implementation detail that the compiler could use to generate fast code (image values access with no indirection, statically-known structuring element). Algorithm 1.3 proposes a trade-off between genericity and efficiency: it is about 30% times slower than Algorithm 1.1, but is generic enough to work on many regular image types (as a matter of fact, the most common ones). The case of the dilation with a static window is even more interesting: reusing the same code (Algorithm 1.3) with a less generic input (a static window representing a fixed and spatially-invariant structuring element) makes the code twice faster, to the point that it outperforms the manually optimized pointer-based implementation. Therefore, having several implementations (namely Algorithms 1.2 and 1.3) is useful when flexibility and efficiency are sought.

Table 1. Execution times of various dilation implementations

Implementation	Time (s) per image (px)		
	512^2	1024^2	2048^2
Non generic (Alg. 1.1)	0.10	0.39	1.53
Non generic, pointer-based[3]	0.07	0.33	1.27
Generic (Alg. 1.2)	0.99	4.07	16.23
Fast, partly generic (Alg. 1.3)	0.13	0.54	1.95
Alg. 1.3 with a static window	0.06	0.28	1.03

6 Conclusion

This paper proposes an approach to reconcile genericity and efficiency in IP software. The key idea relies on generic optimizations expressed as algorithm specializations of the general case for a subspace of the acceptable input types.

The addition of less generic but more efficient versions of an algorithm should not alter the motivation for designing an IP framework as generic as possible. We believe the most generic version of an algorithm should always be defined first, and then complemented by faster implementations. Firstly, having a generic version of an algorithm means having (at least) one implementation for each valid input type. Secondly, generic implementations are usually simpler, shorter and faster to write, provided the framework features entities supporting a generic programming style. Finally, generic implementations constitute a good basis to implement specializations, as their codes often share a similar structure.

The results presented in this paper are *representative* of the general outcomes of our proposal and are essentially independent from the compiler or platform used. In addition, we have already applied this strategy and observed the same conclusions regarding many other (and also more complex) algorithms that the one shown in this paper. Finally, we are not aware of any similar initiative regarding the efficiency of algorithms in generic IP libraries.

The Milena library, used to implement this paper's examples, is available in the Olena platform, a Free Software project released under the GNU General Public License that can be downloaded from our Web site [23], as part of a *reproducible research* effort [2,8,15]. This library is also a proof of concept of the work presented in this paper. It features a collection of different image types along with many generic and efficient algorithms.

References

1. Adobe: Generic Image Library (GIL), `http://opensource.adobe.com/gil`
2. Buckheit, J., Donoho, D.: WaveLab and reproducible research. Tech. Rep. 474, Stanford University, Stanford CA 94305, USA (1995)

[3] This 26-line implementation is not shown in this paper for space reasons.

3. Centre for Mathematical Morphology: Morph-M: Image processing software specialized in mathematical morphology, http://cmm.ensmp.fr/Morph-M/
4. Crozet, S., Géraud, T.: A first parallel algorithm to compute the morphological tree of shapes of nD images. In: Proceedings of the 21st IEEE International Conference on Image Processing, ICIP (2014)
5. Darbon, J., Géraud, T., Duret-Lutz, A.: Generic implementation of morphological image operators. In: Proceedings of the International Symposium on Mathematical Morphology (ISMM), pp. 175–184. Sciro Publishing (2002)
6. Dobie, M., Lewis, P.: Data structures for image processing in C. Pattern Recognition Letters 12(8), 457–466 (1991)
7. Fabri, A., Giezeman, G.J., Kettner, L., Schirra, S., Schönherr, S.: On the design of CGAL a computational geometry algorithms library. Software - Practice and Experience 30(11), 1167–1202 (2000)
8. Fomel, S., Claerbout, J.: Guest editors' introduction: Reproducible research. Computing in Science and Engineering 11(1), 5–7 (2009)
9. Géraud, T., Carlinet, E., Crozet, S., Najman, L.: A quasi-linear algorithm to compute the tree of shapes of nD images. In: Hendriks, C.L.L., Borgefors, G., Strand, R. (eds.) ISMM 2013. LNCS, vol. 7883, pp. 98–110. Springer, Heidelberg (2013)
10. Géraud, T., Fabre, Y., Duret-Lutz, A., Papadopoulos-Orfanos, D., Mangin, J.F.: Obtaining genericity for image processing and pattern recognition algorithms. In: Proceedings of the 15th International Conference on Pattern Recognition (ICPR), vol. 4, pp. 816–819 (2000)
11. Géraud, T., Talbot, H., Van Droogenbroeck, M.: Algorithms for Mathematical Morphology. In: Mathematical Morphology—From Theory to Applications, ch. 12, pp. 323–353. ISTE & Wiley (2010)
12. Géraud, T., Fabre, Y., Duret-Lutz, A.: Applying generic programming to image processing. In: Proceedings of the IASTED International Conference on Applied Informatics (AI)—Symposium on Advances in Computer Applications, Innsbruck, Austria, pp. 577–581 (2001)
13. Ibáñez, L., Schroeder, W., Ng, L., Cates, J.: The ITK Software Guide. Kitware, Inc. (2005)
14. ISO/IEC: ISO/IEC 14882:2003 (e). Programming languages — C++ (2003)
15. Limare, N., Morel, J.-M.: The IPOL initiative: Publishing and testing algorithms on line for reproducible research in image processing. Procedia Computer Science 4, 716–725 (2011)
16. Jazayeri, M., Loos, R., Musser, D., Stepanov, A.: Report of the Dagstuhl seminar (98061) on generic programming (April 1998), http://www.dagstuhl.de/98171
17. Kohl, C., Mundy, J.: The development of the Image Understanding Environment. In: Proceedings of the International Conference on Computer Vision and Pattern Recognition, pp. 443–447 (1994)
18. Köthe, U.: Reusable software in computer vision. In: Jähne, B., Haussecker, H., Geißler, P. (eds.) Handbook of Computer Vision and Applications, vol. 3. Academic Press (1999)
19. Köthe, U.: STL-style generic programming with images. C++ Report 12(1), 24–30 (2000)
20. Levillain, R., Géraud, T., Najman, L.: Milena: Write generic morphological algorithms once, run on many kinds of images. In: Wilkinson, M.H.F., Roerdink, J.B.T.M. (eds.) ISMM 2009. LNCS, vol. 5720, pp. 295–306. Springer, Heidelberg (2009)

21. Levillain, R., Géraud, T., Najman, L.: Why and how to design a generic and efficient image processing framework: The case of the Milena library. In: Proceedings of the IEEE International Conference on Image Processing (ICIP), pp. 1941–1944 (2010)
22. Levillain, R., Géraud, T., Najman, L.: Writing reusable digital topology algorithms in a generic image processing framework. In: Köthe, U., Montanvert, A., Soille, P. (eds.) WADGMM 2010. LNCS, vol. 7346, pp. 140–153. Springer, Heidelberg (2012)
23. LRDE: The Olena image processing platform, `http://olena.lrde.epita.fr`
24. Ritter, G., Wilson, J., Davidson, J.: Image Algebra: an overview. Computer Vision, Graphics, and Image Processing 49(3), 297–331 (1990)
25. Siek, J., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library. Addison Wesley Professional (2001)
26. Xu, Y., Géraud, T., Najman, L.: Context-based energy estimator: Application to object segmentation on the tree of shapes. In: Proceedings of the 19th IEEE International Conference on Image Processing (ICIP), Orlando, Florida, USA, pp. 1577–1580 (October 2012)
27. Xu, Y., Géraud, T., Najman, L.: Salient level lines selection using the Mumford-Shah functional. In: Proceedings of the 20th IEEE International Conference on Image Processing, ICIP (2013)