An Image Processing Library in Modern C++: Getting Simplicity and Efficiency with Generic Programming

Michaël Roynard(✉), Edwin Carlinet, and Thierry Géraud

EPITA Research and Development Laboratory, Le Kremlin-Bicêtre, France
`{michael.roynard,edwin.carlinet,thierry.geraud}@lrde.epita.fr`

Abstract. As there are as many clients as many usages of an Image Processing library, each one may expect different services from it. Some clients may look for efficient and production-quality algorithms, some may look for a large tool set, while others may look for extensibility and genericity to inter-operate with their own code base... but in most cases, they want a simple-to-use and stable product. For a C++ Image Processing library designer, it is difficult to conciliate genericity, efficiency and simplicity *at the same time*. Modern C++ (post 2011) brings new features for library developers that will help designing a software solution combining those three points. In this paper, we develop a method using these facilities to *abstract* the library components and augment the *genericity* of the algorithms. Furthermore, this method is not specific to image processing; it can be applied to any C++ scientific library.

Keywords: Image processing · Modern C++ · Generic programming · Efficiency · Simplicity · Concepts

1 Introduction

As many other numerical fields of computer science, *Computer Vision* and *Image Processing (IP)* have to face the constantly varying form of the input data. The data are becoming bigger and comes from a wider range of input devices: the current issue is generally not about acquiring data, but rather about handling and processing it (in a short time if possible...). In image processing, the two-dimensional RGB model has become too restrictive to handle the whole variety of *kinds* of image that comes with the variety of images fields. A non-exhaustive list of them includes: *remote sensing* (satellites may produce hyperspectral images with some thousands of bands), *medical imaging*, (scans may provide 3D and 3D+t volumes with several modalities), *virtual reality* (RGB-D cameras used for motion capture provide 2D/3D images with an extra 16-bits *depth* channel), *computational photography* (some high-dynamic-range sensors produce 32-bits images to preserve details in all ranges of the luminance)...

These examples already highlight the need for versatility, but some more domain-oriented applications attempt to broaden further the definition of

B. Kerautret et al. (Eds.): RRPR 2018, LNCS 11455, pp. 121–137, 2019.
https://doi.org/10.1007/978-3-030-23987-9_12

images. For example, in digital geometry, one would define images over non-regular domains as graphs, meshes or hexagonal grids. The increase of image type should not require to write several implementation of the algorithm. A single version should be able to work on several image types. The Fig. 1 illustrates this idea with the same watershed implementation applied on an image 2D, a graph as well as a mesh.

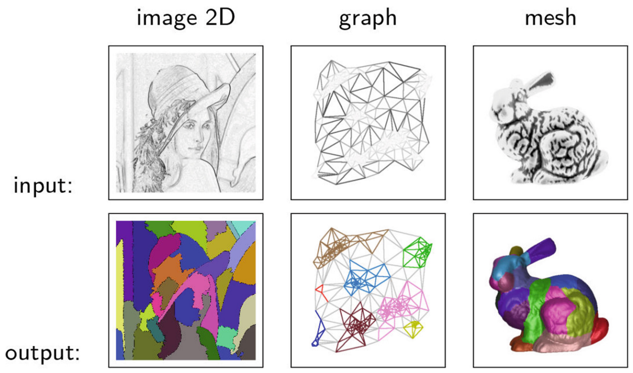


Fig. 1. Watershed algorithm applied to three images having different types.

Tools able to handle many data representations are said to be *generic*. In the particular case of a *library* providing a set of *routines*, *genericity* means that the *routines* can be applied to a variety of inputs (as opposed to *specific* routines that support inputs of unique *predefined* type). As an example, consider the morphological *dilation* that takes two inputs: an image and a flat structuring element (SE). Then, the set of some possible inputs is depicted in Fig. 2. Note that in this example, the image is already a type product between the underlying *structure kind* and the *value kind*. Let s be the number of structures, v the number of types of values, and k the number of structuring elements. With no *generalization*, one would have to write $s \times v \times k$ *dilation* routines.

Many IP libraries have emerged, developed in many programming languages. They all faced this problem and tried to bring solutions, some of which are reviewed in Sect. 2. Among these solutions, we see that *generic programming* is good starting point [15] to design a *generic* library but still has many problem. In particular, we focus on the case of MILENA [16,22], a generic pre-modern C++ IP libray and its shortcomings that led to the design of PYLENA [5]. The work presented in this paper contrasts with the previous works on obtaining genericity for mathematical morphology operators [8,21] and digital topology operators [23].

In Sect. 3, we present this new generic design, that emerged with the evolution of the Modern C++ and allowed solving some MILENA's shortcomings. Not only does this new design re-conciliate *simplicity* and *performance*, but it

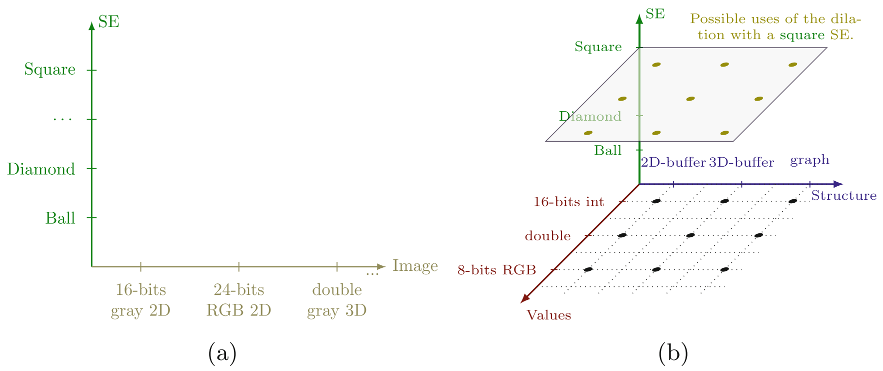


Fig. 2. The space of possible implementation of the *dilation(image, se)* routine. The image axis shown in (a) is in-fact multidimensional and should be considered 2D as in (b).

also promotes *extensibility* as it enables easily creating custom image types as those shown in Sect. 3.5.

2 Bringing Genericity in IP Libraries: Solutions and Problems

Generic programming aims at providing more flexibility to programs. It is itself a very *generic* term that means different things to different people. It may refer to *parametric polymorphism* (which is the common sense in C++), but it may also refer to *data abstraction* and *reflection /meta-programming* [17]. The accordance on a strict definition of *generic programming* is not our objective, but we can observe a manifestation of the *generic programming*: a parametrization of the routines to augment flexibility.

To tackle the problem of a *generic* dilation from the introduction, several programming techniques have been reviewed by Levillain et al. [24], Géraud [13]. We discuss these techniques w.r.t. some criteria: *usability* (simplicity from the end-user), *maintainability* (simplicity from the library developer stand-point), *run-time availability* (running routines on images whose *kind* is unknown until run-time), *efficiency* (speed and binary size tradeoff).

Ad-Hoc Polymorphism and Exhaustivity. The straightforward brute-force solution is the one that enumerates every combination of parameters. This means one implementation for each couple *('image 'kind, 'structuring element 'kind)* and involves much code duplication. Both run-time or compile-time selection of the implementation are possible depending on the parametrization. If the *kind* of parameters are known at compile time (through their *type*), routine *overload* selection is automatically done by the compiler. On the contrary, if the *kind* of parameters are known at run-time (or if the language does not support *overloading*, the user has to select the right implementation by hand (a manual *dispatch*) as illustrated below:

```
// Static parametrization
auto dilate(image2d<uint8>, square) -> image2d<uint8>;
...
auto dilate(image_graph<rgb8>, ball) -> image_graph<rgb8>;

// Dynamic parametrization
auto dilate(any_image img, any_se se)
{
  switch ( (img.structure_kind, img.value_kind, se.se_kind) )
    {
    case (BUFFER2D, UINT8, SQUARE): return dilate( (image2d<uint8>)img, (square) se);
    ...
    case (GRAPH, RGB8, BALL): return dilate( (image_graph<rgb8>)img, (rgb8) se);
    }
}
```

Such a strategy is simple and efficient as the best implementation is written for each case. However it cannot scale, as any addition of a new kind (either a SE, a structure or a value type) would require duplicating many routines and lead to maintenance issues that is why no IP library has chosen such a strategy.

Generalization. A second approach is to generalize to the greatest common type (*a type to rule them all*) instead of augmenting their number. For example, one can consider that all *value types* are `double` since `uint8`, `int16`, ... can roughly be represented by `double` as in MegaWave [10]. Even, if a library supports different *value kinds* for images, it is also common to use an *adapter* that performs a value conversion before calling a specific routine with the most general type. OpenCV [4] uses such an approach where one has to adapt his value types from one routine to another, which makes it painful to extend due to the wide variety of types and algorithms that this library has to offer. Dynamic data analysis framework as Matlab, Octave, Numpy/SciPy have many routines implemented for a single value type and convert their input if it does not fit the required type. The major drawback to this approach is a performance penalty due to conversions and processing on larger type.

Structures can also be generalized to a certain extent. In the image processing library CImg, all images are 3-dimensional with an arbitrary number of channels. It leads to an interface where users can write `ima(x,y,z,channel)` even if the image has a single dimension. There are three drawbacks to this approach: the number of dimensions is bounded (cannot handle 3D+t for example), the interface allows the image to be used incorrectly (weak type-safety), every algorithm has to be written following a 4D pattern even if the image is only 2D. Moreover, *generalization* of *structures* is not trivial when they are really different (e.g. finding the common type between a 3D-buffer encoded image and an image over a graph).

Inclusion & Parametric Polymorphism. A common conception of *generic programming* relates the definitions of *abstractions* and *template methods*.

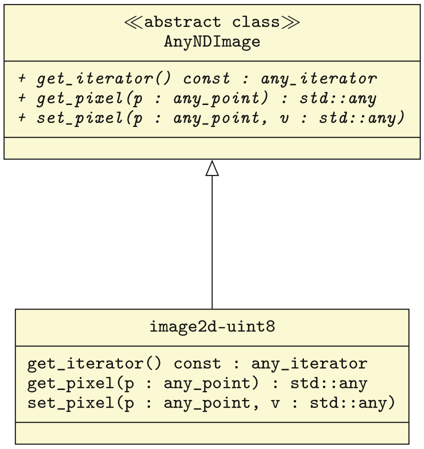
A first programming paradigm that enables such a distinction is *object oriented programming (OOP)*. In this paradigm, *template methods*, as defined by [11], are general routines agnostic to the implementation details and specific properties of a given type. A *template method* defines the skeleton of an algorithm with customization points (calls can be redefined to our own handlers) or we can plug our own types. Hence, *template methods* are *polymorphic*. They rely on the abality to *abstract* the behavior of objects we handle. The *abstraction* thus declares an *interface*: a set of services (generally *abstract methods*) which are common to all the *kinds*. The *concrete* types have then to define *abstract methods* with implementation details.

On the other hand, *generic programming*, in the sense of **(author?)** [25] provides another way of creating *abstraction* and *template methods*. In this paradigm, *the abstraction* is replaced by *a concept* that defines the set of operations a type must provide. *OOP template methods* are commonly refered as *template functions* and implement the algorithm in terms of the *concepts*.

While similar in terms of idea, the two paradigms should not be confused. On one hand, OOP relies on the *inclusion* polymorphism. A single routine (implementation) exists and supports any sub-type that inherits from the *abstract type* (which is the way of defining an *interface* in C++). Also, the *kind* of entities is not known until *run-time*, types are *dynamic* and so is the selection of the right method. This has to be compared to *generic programming* that relies on the *parametric* polymorphism, which is *static*. The *kinds* of entities have to be known at *compile time* and a version of the *template function* is created for each input types. In Fig. 3, we illustrate these differences through the implementation of the simple routine `copy` (`dilate` would require a more advanced abstraction of an image that will be detailed in Sect. 3). Basically, *copy* just has to traverse an input image and stores the values in an output image. The way of *abstracting* the traversal is done with *iterators*.

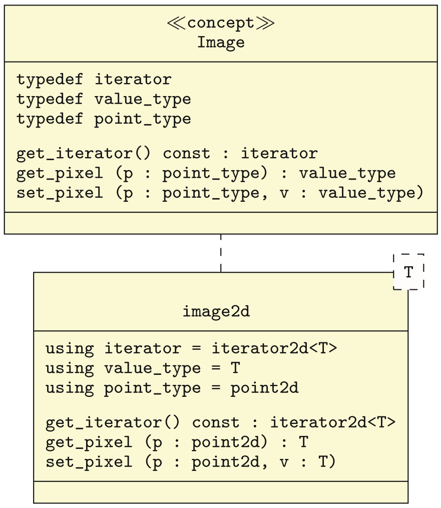
Run-time polymorphism offers a greater flexibility in *dynamic* environment, where the types of the image to handle are not known until the execution of the program. For example, *scipy.ndimage*, a python image processsing library for interactive environments, uses a C-stylished version of the iterator abstraction [28] and value abstraction given above (*C-style* stands for an hand-made *switch* dispatch instead of *virtual methods*). GEGL [1], used in GIMP, is also written in C ans uses C-style run-time polymorphism to achieve abstraction over colors and data buffers (e.g. to model graphs).

Nevertheless, this flexibility comes at the cost of degraded performances due to the *dynamic dispatches*. On the other hand, *static* polymorphism provides better performances because *concrete types* are known at compile time and there is no need to resolve *methods* at run-time. As there is never no free lunch, performance comes at the cost of less run-time flexibility. Moreover, since *parametric polymorphism* is implemented through *templates* in C++, many instanciations of the same code occur for different input types and may lead to *code bloat* and large executables.


≪abstract class≫
AnyNDImage
+ get_iterator() const : any_iterator
+ get_pixel(p : any_point) : std::any
+ set_pixel(p : any_point, v : std::any)
image2d-uint8
get_iterator() const : any_iterator
get_pixel(p : any_point) : std::any
set_pixel(p : any_point, v : std::any)

```
void copy(const AnyNDImage& in, AnyNDImage& out)
{
  for (any_point p : in.get_iterator())
    out.set_pixel(p, in.get_pixel(p));
}
```

(a) Dynamic, object-oriented polymorphism


≪concept≫
Image
typedef iterator
typedef value_type
typedef point_type
get_iterator() const : iterator
get_pixel (p : point_type) : value_type
set_pixel (p : point_type, v : value_type)
T
image2d
using iterator = iterator2d<T>
using value_type = T
using point_type = point2d
get_iterator() const : iterator2d<T>
get_pixel (p : point2d) : T
set_pixel (p : point2d, v : T)

```
template <class InImage, class OutImage>
void copy(const InImage& in, OutImage& out)
{
  for (auto p : in.get_iterator())
    out.set_pixel(p, in.get_pixel(p));
}
```

(b) Static, parametric polymorphism

Fig. 3. Comparison of the implementations of a polymorphic routine with the *object-oriented programming* and *generic programming* paradigms

Parametric Polymorphism in C++ Image Processing Libraries. Parametric polymorphism is common in many C++ IP libraries to a certain extent. Many of them, (e.g. CImg [34], Video++ [12], ITK [19]) provide *value type* genericity (e.g. `image2d<T>`) while a few provide a full structural genericity (DGTal [7], GrAL [2], Milena [24], VIGRA [20], Boost.GIL [3]). To reach such a level a genericity, these libraries have been written in a complex C++ which remains visible from the user standpoint. *Erich Gamma* [11] notice that *dynamic, highly parameterized software is harder to understand than more static software.* In particular, errors in highly templated code is hard to read and to debug because they show up very deep in the compiler error trace.

Also, they have not been written with the modern user-friendly features that the new C++ offers. Worst, in the case of Milena, some design choices made in pre-C++ 11, makes the library not compatible with the current standard and prevents usage of these new features.

Additionally, there exists other, non-library approach, such as designing a whole new DSL (Domain Specific Language) to reach a specific goal. For instance, Halide [29] chose this approach to fully focus on the underlying generated code to optimize it for vectorization, parallelism and data locality.

Unfortunately this implies trade-offs on genericity and interoperability as we are not dealing with native C++ anymore.

3 C++ Generic Programming and Concepts

C++ is a multi-paradigm language that enables the developer to write code that can be *object oriented*, *procedural*, *functional* and *generic*. However, there were limitations that were mostly due to the backward compatibility constraint as well as the zero-cost abstraction principle. In particular the *generic programming* paradigm is provided by the *template metaprogramming* machinery which can be rather obscure and error-prone. Furthermore, when the code is incorrect, due to the nature of templates (and the way they are specified) it is extremely difficult for a compiler to provide a clear and useful error message. To solve this issue, a new facility named *concepts* was brought to the language. It enables the developer to constraint types: we say that the type *models* the *concept(s)*. For instance, to compare two images, a function *compare* would restrict its input image types to the ones whose value type provides the *comparison operator ==*.

In spite of the history behind the *concept checking* facilities being very turbulent [30,32,33], it will finally appear in the next standard [35] (C++20).

3.1 From Algorithms to Concepts

The C++ *Standard Template Library* (STL) is a collection of algorithms and data structures that allow the developer to code with generic facilities. For instance, there is a standard way to *reduce* a collection of elements: `std::accumulate` that is agnostic to the underlying collection type. The collection just needs to provide a facility so that it can work. This facility is called *iterator*. All STL algorithms behave this way: the type is a template parameter so it can be anything. What is important is how this type behaves. Some collection requires you to define a `hash` functions (`std::map`), some requires you to set an *order* on your elements (`std::set`) etc. This emphasis the power of genericity. The most important point to remember here (and explained very early in 1988 [25]) is the answer to: "*What is a generic algorithm?*". The answer is: "*An algorithm is generic when it is expressed in the most abstract way possible*".

Later, in his book [31], Stepanov explained the design decision behind those algorithms as well as an important notion born in the early 2000s: the concepts. The most important point about concepts is that it constraints the behavior. Henceforth: "*It is not the types that define the concepts: it is the algorithms*".

The *Image Processing* and *Computer Vision* fields are facing this issue because there are a lot of algorithms, a lot of different kind of images and a lot of different kind of requirements/properties for those algorithms to work. In fact, when analyzing the algorithms, you can always extract those requirements in the form of one or several *concepts*.

3.2 Rewriting an Algorithm to Extract a Concept

Gamma Correction. Let us take the gamma correction algorithm as an example. The naive way to write this algorithm can be:

```
1   template <class Image>
2   void gamma_correction(Image& ima, double gamma)
3   {
4     const auto gamma_corr = 1 / gamma;
5
6     for (int x = 0; x < ima.width(); ++x)
7       for (int y = 0; y < ima.height(); ++y)
8       {
9         ima(x, y).r = std::pow((255 * ima(x, y).r) / 255, gamma_corr);
10        ima(x, y).g = std::pow((255 * ima(x, y).g) / 255, gamma_corr);
11        ima(x, y).b = std::pow((255 * ima(x, y).b) / 255, gamma_corr);
12      }
13  }
```

This algorithm here does the job but it also makse a lot of hypothesis. Firstly, we suppose that we can write in the image via the `=` operator (l.9–11): it may not be true if the image is sourced from a generator function. Secondly, we suppose that we have a 2D image via the double loop (l.6–7). Finally, we suppose we are operating on 8bits range (0–255) RGB via `'.r'`, `'.g'`, `'.b'` (l.9–11). Those hypothesis are unjustified. Intrinsically, all we want to say is "*For each value of* `ima`*, apply a gamma correction on it.*". Let us proceed to make this algorithm the most generic possible by lifting those unjustified constraints one by one.

Lifting RGB constraint: First, we get rid of the 8bits color range (0–255) RGB format requirement. The loops become:

```
using value_t = typename Image::value_type;

const auto gamma_corr = 1 / gamma;
const auto max_val = std::numeric_limits<value_t>::max();

for(int x = 0; x < ima.width(); ++x)
  for(int y = 0; y < ima.height(); ++y)
    ima(x, y) = std::pow((max_val * ima(x, y)) / max_val, gamma_corr);
```

By lifting this constraint, we now require the type Image to define a nested type `Image::value_type` (returned by `ima(x, y)`) on which `std::numeric_limits` and `std::pow` are defined. This way the compiler will be able to check the types at compile-time and emit warning and/or errors in case it detects incompatibilities. We are also able to detect it beforehand using a `static_assert` for instance.

Lifting bi-dimensional constraint: Here we need to introduce a new abstraction layer, the *pixel*. A *pixel* is a couple $(point, value)$. The double loop then becomes:

```
for (auto&& pix : ima.pixels())
  pix.value() = std::pow((max_val * pix.value()) / max_val, gamma_corr);
```

This led to us requiring that the type *Image* requires to provide a method `Image::pixels()` that returns *something* we can iterate on with a range-for loop: this *something* is a *Range* of *Pixel*. This *Range* is required to behave like an *iterable*: it is an abstraction that provides a way to browse all the elements one by one. The *Pixel* is required to provide a method `Pixel::value()` that returns a *Value* which is *Regular* (see Sect. 3.3). Here, we use `auto&&` instead of `auto&` to allow the existence of proxy iterator (think of `vector<bool>`). Indeed, we may be iterating over a lazy-computed view Sect. 3.5.

Lifting writability constraint: Finally, the most subtle one is the requirement about the *writability* of the image. This requirement can be expressed directly via the new C++20 syntax for *concepts*. All we need to do is changing the template declaration by:

```
template <WritableImage Image>
```

In practice the C++ keyword `const` is not enough to express the *constness* or the *mutability* of an image. Indeed, we can have an image whose pixel values are returned by computing $cos(x + y)$ (for a 2D point). Such an image type can be instantiated as *non-const* in C++ but the values will not be *mutable*: this type will not model the *WritableImage* concept.

Final version

```
template <WritableImage Image>
void gamma_correction(Image& ima, double gamma)
{
  using value_t = typename Image::value_type;

  const auto gamma_corr = 1 / gamma;
  const auto max_val = numeric_limits<value_t>::max();

  for (auto&& pix : ima.pixels())
    pix.value() = std::pow((max_val * pix.value()) / max_val, gamma_corr);
}
```

When re-writing a lot of algorithms this way: lifting constraints by requiring behavior instead, we are able to deduce what our *concepts* needs to be. The real question for a *concept* is: "*what behavior should be required?*"

Dilation Algorithm. To show the versatility of this approach, we will now attempt to deduces the requirements necessary to write a classical *dilate* algorithm. First let us start with a naive implementation:

```
template <class InputImage, class OutputImage>
void dilate(const InputImage& input_ima, OutputImage& output_ima)
{
  assert(input_ima.height() == output_ima.height()
    && input_ima.width() == output_ima.width());

  for (int x = 2; x < input_ima.width() - 2; ++x)
    for (int y = 2; y < input_ima.height() - 2; ++y)
    {
      output_ima(x, y) = input_ima(x, y)
      for (int i = x - 2; i <= x + 2; ++i)
        for (int j = y - 2; j <= y + 2; ++j)
          output_ima(x, y) = std::max(output_ima(x, y), input_ima(i, j));
    }
}
```

Here we are falling into the same pitfall as for the *gamma correction* example: there are a lot of unjustified hypothesis. We suppose that we have a 2D image (l.7–8), that we can write in the `output_image` (l.10, 13). We also require that the input image does not handle borders, (cf. loop index arithmetic l.7-8, 11-12). Additionally, the *structuring element* is restricted to a 5×5 window (l.11-12) whereas we may need to dilate via, for instance, a 11×15 window, or a sphere. Finally, the algorithm does not exploit any potential properties such as the *decomposability* (l.11-12) to improve its efficiency. Those hypothesis are, once again, unjustified. Intrinsically, all we want to say is "For each value of `input_ima`, take the maximum of the $X \times X$ window around and then write it in `output_ima`".

To lift those constraints, we need a way to know which kind of *structuring element* matches a specific algorithm. Thus, we will pass it as a parameter. Additionally, we are going to lift the first two constraints the same way we did for *gamma correction*:

```
template <Image InputImage, WritableImage OutputImage, StructuringElement SE>
void dilate(const InputImage& input_ima, OutputImage& output_ima, const SE& se)
{
  assert(input_ima.size() == output_ima.size());

  for(auto&& [ipix, opix] : zip(input_ima.pixels(), output_ima.pixels())
  {
    opix.value() = ipix.value();
    for (const auto& nx : se(ipix))
      opix.value() = std::max(nx.value(), opix.value());
  }
}
```

We now do not require anything except that the *structuring element* returns the neighbors of a pixel. The returned value must be an *iterable*. In addition, this code uses the `zip` utility which allows us to iterate over two ranges at the same time. Finally, this way of writing the algorithm allows us to delegate the issue about the border handling to the neighborhood machinery. Henceforth, we will not address this specific point deeper in this paper.

3.3 Concept Definition

The more algorithms we analyze to extract their requirements, the clearer the *concepts* become. They are slowly appearing. Let us now attempt to formalize them. The formalization of the *concept Image* from the information and requirements we have now is shown in Table 1 for the required type definitions and valid expressions.

The *concept Image* does not provide a facility to write inside it. To do so, we have refined a second *concept* named *WritableImage* that provides the necessary facilities to write inside it. We say "*WritableImage* refines *Image*".

The *sub-concept ForwardRange* can be seen as a requirement on the underlying type. We need to be able to browse all the pixels in a forward way. Its *concept* will not be detailed here as it is very similar to *concept* of the same name [26,27] (soon in the STL). Also, in practice, the *concepts* described here

Table 1. Formalization of concepts.

Let *Ima* be a type that models the concept *Image*. Let *WIma* be a type that models the concept *WritableImage*. Then *WIma* inherits all types defined for *Image*. Let *SE* be a type that models the concept *StructuringElement* . Let *DSE* be a type that models the concept *Decomposable*. Then *DSE* inherits all types defined for *StructuringElement*. Let *Pix* be a type that models the concept *Pixel*. Then we can define:

	Definition	Description	Requirement
Image	`Ima::const_pixel_range`	type of the range to iterate over all the constant pixels	models the concept *ForwardRange*
	`Ima::pixel_type`	type of a pixel	models the concept *Pixel*
	`Ima::value_type`	type of a value	models the concept *Regular*
Writable Image	`WIma::pixel_range`	type of the range to iterate over all the non-constant pixels	models the concept *ForwardRange*

Let *cima* be an instance of *const Ima*. Let *wima* be an instance of *WIma*. Then all the valid expressions defined for *Image* are valid for *WIma*. Let *cse* be an instance of *const SE*. Let *cdse* be an instance of *const DSE*. Then all the valid expressions defined for *StructuringElement* are valid for *const DSE* Let *cpix* be an instance of *const Pix*. Then we have the following valid expressions:

	Expression	Return Type	Description
Image	`cima.pixels()`	`Ima::const_pixel_range`	returns a range of constant pixels to iterate over it
Writable Image	`wima.pixels()`	`WIma::pixel_range`	returns a range of pixels to iterate over it
Structuring Element	`cse(cpix)`	`WIma::pixel_range`	returns a range of the neighboring pixels to iterate over it
Decomposable	`cdse.decompose()`	`implementation defined`	returns a range of structuring elements to iterate over it

are incomplete. We would need to analyze several other algorithms to deduce all the requirements so that our *concepts* are the most complete possible. One thing important to note here is that to define a simple *Image concept*, there are already a large amount of prerequisites: *Regular*, *Pixel* and *ForwardRange*. Those *concepts* are basic but are also tightly linked to the *concept* in the STL [6]. We refer to the STL *concepts* as *fundamental concepts*. *Fundamentals concepts* are the basic building blocks on which we work to build our own *concepts*. We show the C++20 code implementing those *concepts* in the code below.

```
template <class Ima>
concept Image = requires {
    typename Ima::value_type;
    typename Ima::pixel_type;
    typename Ima::const_pixel_range;
  } && Regular<Ima::value_type>
  && ForwardRange<Ima::const_pixel_range>
  && requires(const Ima& cima) {
    { cima.pixels() }
      -> Ima::const_pixel_range;
  };
```

```
template <class WIma>
concept WritableImage = requires Image<WIma>
  && requires {
    typename WIma::pixel_range;
  } && ForwardRange<WIma::pixel_range>
  && ForwardRange<WIma::pixel_range,
      WIma::pixel_type>
  && requires(WIma& wima) {
    { wima.pixels() } -> WIma::pixel_range;
  };
```

```
template <class I>
using pixel_t = typename I::pixel_type;
template <class SE, class Ima>
concept StructuringElement = Image<Ima>
  && requires(const SE& cse,
       const pixel_t<Ima> cpix){
    { se(cpix) } -> Ima::const_pixel_range;
  };
```

```
template <class DSE, class Ima>
concept Decomposable =
  StructuringElement<DSE, Ima>
  && requires(const DSE& cdse) {
    { cdse.decompose() }
      -> /*impl. defined*/;
  };
```

3.4 Specialization Vs. Properties

Another advantage of *concepts* are that they allow a *best match* machinery over requirement(s) met by a type. We call this mechanic the *property specialization*. It allows to select the correct overload (*best match* machinery) when the given template parameter satisfies the requirement(s) expressed via the concept(s). Historically we used the template specialization mechanism to achieve the same thing (via inheritance of specialized types and other tricks) but it came with lot of disadvantages. Those are the cost of the abstraction and indirection, the difficulty to extend as well as to inject new type or behavior for a new user, being tied to a type and finally, each new type needs its own new implementation. Switching to a property-based approach with an automatic *best match* machinery is much more efficient and user-friendly.

This machinery could be emulated pre-C++20 via cryptic template metaprogramming tricks (i.e. type-traits, SFINAE and enable_if). However, C++20 brings a way to remove the need of these need, making it widely accessible. The code in Fig. 4 shows this difference in action.

```
template <class Image, class Value,
  std::enable_if_t<
    is_writable_image_v<Image>
    && is_value_v<Value>
    && !is_image_v<Image>>* = nullptr>
Image operator+(Image ima, Value v)
{
  for (auto&& pix : ima.pixels())
    pix.value() += v;
  return ima;
}

template <class ImLhs, class ImRhs,
  std::enable_if_t<
    is_writable_image_v<ImLhs>
    && !is_value_v<ImRhs>
    && is_image_v<ImRhs>>* = nullptr>
ImLhs operator+(ImLhs lhs, const ImRhs& rhs)
{
  for (auto&& [p_lhs, p_rhs] : zip(lhs, rhs))
    p_lhs.value() += p_rhs.value();
  return lhs;
}
```

```
template <WritableImage Ima, Value V>
Ima operator+(Ima ima, V v)
{
  for (auto&& pix : ima.pixels())
    pix.value() += v;
  return ima;
}

template <WritableImage ImLhs, Image Rhs>
ImLhs operator+(ImLhs lhs, const Rhs& rhs)
{
  for (auto&& [p_lhs, p_rhs] : zip(lhs, rhs))
    p_lhs.value() += p_rhs.value();
  return lhs;
}
```

Fig. 4. C++17 SFINAE trick vs. C++20 Concepts.

The result about which code is clearer, easier to read and less error-prone should be obvious. The first advantage is that the compiler do the *type-checking* pass early when instantiating the template instead of waiting until the *overload resolution* pass (the improper functions candidate are removed from *overload resolution* thanks to *SFINAE*). This directly *enhances the error messages* emitted by the compiler. Instead of having a long error trace one needs to scroll down to find the error within, the compiler will now emits the error first at the top with the incorrect *behavior requirement* that does not match the *concept* for a given instantiated type.

Also, in the C++17 code, with heavy *metaprograming trick*, informations about function prototypes such as *return type*, *parametric types* and *input types* are fuzzy and not very clear. It needs carefully designed and written user documentation to be usable by a tier. Furthermore, this documentation is often difficult to generate and documentation generators do not help because they have a very limited understanding of templated code. However, we can see in the C++20 code that, with *concepts*, we just have two different *overloads* with a single piece of information changing: the 2^{nd} input parameter. The information is clear and can be easily understood by documentation generators.

In addition, as *concepts* are working with a *best-match* machinery, we can notice that it is not the case with the SFINAE tricks version. Each time you add a new variant, every possibilities, incompatibilities and ambiguities between all the overloads have to be manually lifted. Not doing so will lead to multiple overloads ambiguities (or none selected at all). Also, the compiler will issue a non-friendly error message difficult to address.

In *Image Processing* we are able to make use of this machinery, in particular with a property on *structuring elements*: the *decomposability.* For reminder a multi-dimensional *structuring element* is *decomposable* if it can be rewritten as many simpler *structuring elements.*

Indeed when the *structuring element* window is tiny, it makes little sense to exploit this property for efficiency. If instead of browsing the image once while selecting 4 neighbors for each pixel, then we browse the image twice while selecting 2 neighbors for each pixel, the difference is not relevant. However, the more the *structuring element* window grows, the more neighboring pixels are selected for each pixel. With a multi-dimensional *structuring element* the growth is quadratic whereas it is linear if the *structuring element* is *decomposed.*

Henceforth, bringing the *property best-match* machinery with *concepts* as well as this *decomposable* property lead us to this dilate algorithm version:

```
template <Image I,  WritableImage O, StructuringElement SE> requires Decomposable<SE>
void dilate(const I& input, O& output, const SE& large_se)
{
  auto tmp = copy(input);
  for (auto&& small_se : large_se.decompose())
  {
    for (auto&& [ipix, opix] : zip(tmp.pixels(), output.pixels())
    {
      opix.val() = ipix.val();
      for (auto&& nbx : small_se(ipix))
        opix.val() = std::max(opix.val(), nbx.val());
    }
    std::swap(tmp, output);
  }
  std::swap(tmp, output);
}
```

It is much more efficient as it reduces the complexity dramatically when the structuring element has a large selection window.

3.5 Practical Genericity for Efficiency: The Views

Let us introduce another key point enabled by genericity and concepts: the *Views*. A *View* is defined by a non-owning lightweight image, inspired by the design introduced in *Ranges for the Standard Library* [9] proposal for *non-owning collections*. A similar design is also called *Morphers* in Milena [13,21]. *Views* feature the following properties: *cheap to copy*, *non-owner* (does not *own* any data buffer), *lazy evaluation* (accessing the value of a pixel may require computations) and *composition*. When chained, the compiler builds a *tree of expressions* (or *expression template* as used in many scientific computing libraries such as Eigen [18]), thus it knows at compile-time the type of the composition and ensures a 0-overhead at evaluation.

There are four fundamental kind of views, inspired by functional programming paradigm: `transform(input, f)` applies the transformation f on each pixel of the image *input*, `filter(input, pred)` keeps the pixels of *input* that satisfy the predicate *pred*, `subimage(input, domain)` keeps the pixels of *input* that are in the domain *domain*, `zip`($input_1$, $input_2$, ..., $input_n$) allows to pack several pixel of several image to iterate on them all at the same time.

Lazy-evaluation combined with the view *chaining* allows the user to write clear and very efficient code whose evaluation is delayed till very last moment as shown in the code below (see [14] for additional examples). Neither memory allocation nor computation are performed; the image i has just recorded all the operations required to compute its values.

```
image2d<rgb8>   ima1 = /* ... */;
image2d<uint8_t> ima2 = /* ... */;

// Projection: project the red channel value
auto f = view::transform(ima, [](auto v) {
  return v.r;
});

// Lazy-evaluation of the element-wise
// minimum
auto g = view::transform(view::zip(f, ima2),
   [](auto value) {
     return std::min(std::get<0>(value),
                std::get<1>(value));
});
```

```
// Lazy-Filtering: keep pixels whose value
// is below < 128
auto h = view::filter(g, [] (auto value) {
  return value < 128;
}));

// Lazy-evaluation of a gamma correction
using value_t = typename Image::value_type;
constexpr float gamma = 2.2f;
constexpr auto max_val =
  std::numeric_limits<value_t>::max();
auto i = view::transform(h,
  [gamma_corr = 1 / gamma] (auto value) {
    return std::pow(value / max_val,
              gamma_corr) * max_val;
});
```

4 Conclusion and Perspectives

Through a simple example, we have shown a step-by-step methodology to make an algorithm *generic* with zero overhead[1]. To reach such a level of genericity and be able to write versatile algorithms, we had to *abstract* and *define* the most simple and fundamental elements of the libray (e.g. *image*, *pixel*, *structuring element*). We have shown that some tools of the Modern C++, such as *concepts*, greatly facilitate the definition and the usage of such abstractions. These tools

[1] The zero-cost abstraction of our approach is not argued here but will be discussed in an incoming paper with a comparison with the state of the art libraries.

enable the library designer to focus on the abstraction of the library components and on the user-visible development. The complex *template meta-programming* layer that used to be a large part of *C++ generic programming* is no more inevitable. In this context, it is worth pointing out the approach is not limited Image Processing libraries but works for any library that wants to be modernized to augment its productivity.

As one may have noticed, the solution presented in this paper is mostly dedicated to C++ developer and C++ end-user. Unlike dynamic environments (such as Python), C++ is not the most appropriate language when one has to prototype or experiment an IP solution. As a future work, we will study the conciliation of the *static genericity* from C++ (where types have to be known at compile time) with a *dynamic* language (with a run-time polymorphism) to allows the interactive usage of a C++ generic library.

References

1. Generic Graphic Library
2. Berti, G.: GrAL-the grid algorithms library. Future Gener. Comput. Syst. **22**(1–2), 110–122 (2006)
3. Bourdev, L., Jin, H.: Boost generic image library. In: Adobe Stlab (2006). Available at https://stlab.adobe.com/gil/index.html
4. Bradski, G.: The OpenCV Library. Dr. Dobb's J. Softw. Tools **120**, 122–125 (2000)
5. Carlinet, E., et al.: Pylena: a modern C++ image processing generic library. EPITA Research and Developement Laboratory (2018). Available at https://gitlab.lrde.epita.fr/olena/pylene
6. Carter, C., Niebler, E.: Standard library concepts June 2018. https://wg21.link/p0898r3
7. Coeurjolly, D., Lachaud, J.O.: DGtal: Digital geometry tools and algorithms library. http://dgtal.org
8. Darbon, J., Géraud, T., Duret-Lutz, A.: Generic implementation of morphological image operators. In: Proceedings of the 6th International Symposium of Mathematical Morphology (ISMM), pp. 175–184. Sydney, Australia (2002)
9. Eric N., Sean Parent, A.S.: Ranges for the standard library: Revision 1, October 2014. https://ericniebler.github.io/std/wg21/D4128.html
10. Froment, J.: MegaWave. In: IPOL 2012 Meeting on Image Processing Libraries (2012)
11. Gamma, E.: Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education India, Chennai (1995)
12. Garrigues, M., Manzanera, A.: Video++, a modern image and video processing C++ framework. In: Conference on Design and Architectures for Signal and Image Processing (DASIP), pp. 1–6. IEEE (2014)
13. Géraud, T.: Outil logiciel pour le traitement d'images: Bibliothèque, paradigmes, types et algorithmes. Habilitation thesis, Université Paris-Est (2012). (in French)
14. Géraud, T., Carlinet, E.: A modern C++ library for generic and efficient image processing. Journée du Groupe de Travail de Géométrie Discrète et Morphologie Mathématique, Lyon, France, June 2018. https://www.lrde.epita.fr/theo/talks/geraud.2018.gtgdmm_talk.pdf

15. 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. Barcelona, Spain (2000)
16. Géraud, T., Levillain, R., Lazzara, G.: The Milena image processing library. IPOL meeting, ENS Cachan, France, June 2012. https://www.lrde.epita.fr/~theo/talks/geraud.2012.ipol_talk.pdf
17. Gibbons, J.: Datatype-generic programming. In: Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J. (eds.) SSDGP 2006. LNCS, vol. 4719, pp. 1–71. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76786-2_1
18. Guennebaud, G., Jacob, B., et al.: Eigen v3. http://eigen.tuxfamily.org (2010). Available at http://eigen.tuxfamily.org
19. Ibanez, L., Schroeder, W., Ng, L., Cates, J.: The ITK software guide (2005)
20. Köthe, U.: Generic Programming for Computer Vision: The VIGRA Computer Vision Library. University of Hamburg, Cognitive Systems Group, Hamburg (2003)
21. 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). https://doi.org/10.1007/978-3-642-03613-2_27
22. 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. Hong Kong September 2010 (2010)
23. 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). https://doi.org/10.1007/978-3-642-32313-3_10
24. Levillain, R., Géraud, T., Najman, L., Carlinet, E.: Practical genericity: writing image processing algorithms both reusable and efficient. In: Proceedings of the 19th Iberoamerican Congress on Pattern Recognition (CIARP). LNCS, vol. 8827, pp. 70–79. Puerto Vallarta, Mexico (2014)
25. Backhouse, R., Gibbons, J. (eds.): Generic Programming. LNCS, vol. 2793. Springer, Heidelberg (2003). https://doi.org/10.1007/b12027
26. Niebler, E., Carter, C.: Deep integration of the ranges TS, May 2018. https://wg21.link/p1037r0
27. Niebler, E., Carter, C.: Merging the ranges TS, May 2018. https://wg21.link/p0896r1
28. Oliphant, T.E.: Multidimensional iterators in NumPy. In: Oram, A., Wilson, G. (eds.) Beautiful Code, vol. 19. O'reilly, Sebastopol (2007)
29. Ragan-kelley, J., Barnes, C., Adams, A., Durand, F., Amarasinghe, S., et al.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: PLDI **2013**, (2013)
30. Seymour, B.: LWG papers to re-merge into C++0x after removing concepts July 2009. https://wg21.link/n2929
31. Stepanov, A., McJones, P.: Elements of Programming. Addison-Wesley Professional, Boston (2009)
32. Stroustrup, B., Reis, G.D.: Concepts - Design choices for template argument checking, October 2003. https://wg21.link/n1522

33. Sutton, A.: Working draft, C++ extensions for concepts, June 2017. https://wg21.link/n4674
34. Tschumperlé, D.: The CImg library. IPOL 2012 Meeting on Image Processing Libraries (2012)
35. Voutilainen, V.: Merge the concepts TS working draft into the C++20 working draft, June 2017. https://wg21.link/p0724r0