Programming in modern C++ for image processing

Michaël Roynard

July 28, 2021

0.1 Acknowledgement

0.2 Abstract

TODO EN

TODO FR

0.3 Long abstract

Contents

List of Figures

List of Tables

Part I

Context

Chapter 1

Introduction

Outline

NOWADAYS *Computer Vision* and *Image Processing (IP)* are omnipresent in the day to day life of the people. It is present each time we pass by a CCTV camera, each time we go to the hospital do an MRI, each time we drive our car and pass in front of a speed camera and each time we use our computer, smartphone or tablet. We just cannot avoid it anymore. The systems using this technology are sometimes simple and, sometimes, more complex. Also the usage made of this technology has several different purposes: space observation, medical, quality of life improvement, surveillance, control, autonomous system, etc. Henceforth, Image processing has a wide range of research and despite having a mass of previous of work already contributed to, there are still a lot to explore.

Let us take the example of a modern smartphone application which provides facial recognition in order to recognize people whom are featuring inside a photo. To provide accurate result, this application will have to do a lot of different processing. Indeed, there are a lot of elements to handle. We can list (non exhaustively) the weather, the light exposition, the resolution, the orientation, the number of person, the localization of the person, the distinction between humans and objects/animals, etc. All of these is in order to finally recognizing the person(s) inside the photo. What the application does not tell you is the complexity of the image processing pipeline behind the scene that can not even be executed in its entirety on one's device (smartphone, tablet, ...). Indeed, image processing is costly in computing ressources and would not meet the time requirement desired by the user if the entire pipeline was executed on the device. Furthermore, for the final part which is "recognize the person on the photo", one needs to feed the pre-processed photo to a neural network trained beforehand through deep learning techniques in order to give an accurate response. There exists technologies able to embed neural network into mobile phone such as MobileNets [70] but it is still limited. It can detect a human being inside a photo but not give the answer about who this human being is for instance. That is why, accurate neural network system usually are abstracted away in cloud technologies making them available only via Internet. When uploading his image, the user does not imagine the amount of technologies and computing power that will be used to find who is on the photo.

We now understand that in order to build applications that interact with photos or videos nowadays, we need to be able to do accurate, fast and scalable image processing on a multitude of devices (smartphone, tablet, ...). In order to achieve this goal, image processing practitioners needs to have two kinds of tools at their disposal. One will be the prototyping environnement, a toolbox which allow the practitioner to develop, test and improve its application logic. The other is the production environnement which deploy the viable version of the application that was developed by the practitioner. Both environment may not have the same needs. On one hand, the prototyping environment usually requires to have a fast feedback loop for testing, an availability of state-of-the-art algorithms and existing software. This way the practitioner can easily build upon them and be fast enough in order not to keep waiting for results when testing many prototypes. On the other hand, the production environment must be stable, resilient, fast and scalable.

When looking at standards in the industry nowadays, we notice that Python is the main choice for prototyping. Also, Python may not be enough so that a viable prototype can be pushed in production with minimal changes afterwards. We find it non-ideal that the practitioner cannot take advantages of many optimisation opportunities, both in term of algorithm efficiency and better hardware usage, when proceeding this way. It would be much more efficient to have basic low level building blocks that can be adapted to fit as mush use cases as possible. This way, the practitioner can easily build upon them when designing its application. We distinguishes two kind of use cases. The first one is about the multiplicity of types or algorithms the practitioner is facing. The second one is about the diversity of hardware the practitioner may want to run his program. The goal is to have building blocks that can be intelligent enough to take advantage of many optimization opportunities, with regard to both input data types/algorithms and target hardware. Then the practitioner would have a huge performance improvement, by default, without specifically tweaking its application. As such, the concept of genericity was introduced. It aims at providing a common ground about how an image should behave when passed to basic algorithms needed for complex applications. This way, in theory, one only needs to write the algorithm once for it to work with any given kind of image.

Different data types and algorithms

In Image Processing, there exists a multitude of image types whose caracteristics can be vastly different from one another. This large specter is also resulting from the large domain of application of image processing. For instance, when considering photography we have 2D image whose values can vary from 8 bits grayscale to multiple band 32-bits color scheme storing informations about the non-visible specter of human eye. If we consider another domain of application, such as medical imaging, we now can consider sequence of images such as sequence of 3D image for an MRI for instance. More broadly there are two orthogonal constituent of an image: its topology (or structure) and its values. However, there are two more aspectes to consider here. Firstly, image processing provide plenty of algorithms that can or cannot operate over specific data types. There are also different kind of algorithms. Some will extract informations,(e.g. histogram) other will transform the image point-wise (e.g. thresholding), and some other will even combine several image to render a different kind of informations (e.g. background substraction). There are many simple algorithms and also many complex algorithms out there. Secondly, there are orbitting data around image types and algorithms that are also very diverse and necessary for their smooth operation. Indeed, a dilation algorithm will also need an additional information: the dilation disc. A thresholding algorithm may be given a threshold. A convolution filter requires a convolution matrix to operate. That is why, when considering both image types and algorithms, we need a 3D-chart (illustrated in fig. 1.1) to enumerate all possibilities, where one axis is the image topology, one axis is the color scheme and one axis enumerate the additional data that can be associated to an image.

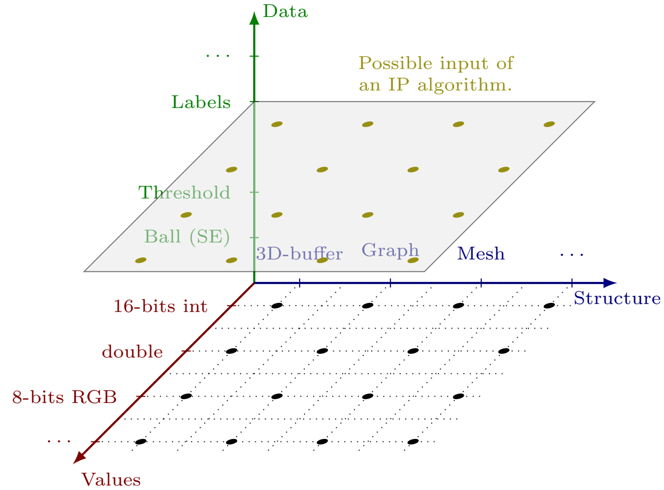


Figure 1.1: Illustration of the specter of the multitude of possibilities in the image processing world.

Different user profiles and their use cases

The end user is a non programmer user who wants to occasionally use image processing software through UI-rich interface, such as Adobe Photoshop [91] or The GIMP [84]. Its skills are non-relevant as the end user is using the software to get work done even though he does not fully understand the underlying principles. For instance, the end user will want to correct the bightness of an image, of remove some impurities from a face or a building. The end user does not want to build an application but wants to save time. The needs of the end user mainly revolves around a clean and intuitive software UI as well as a well as support for mainstream image types and operation a photograph may need to do.

The practitioner is what we are called when we first approach the image processing area. A practitioner is the end user of image processing libraries. Its skills mainly revolve around applied mathematics for image processing, prototyping and algorithms. A practitioner aims at leveraging the features the libraries can offer to build his application. For instance, a practitioner can be a researcher in medical imaging, an engineer build a facial recognition application, a data scientist labeling its image set, etc. The needs of practitioner are mainly revolving around a fast feedback loop. The developing environment must be easily accessible and installable. This way a practitioner can judge quickly wether one library will answer his needs. The documentation of the library must be exhaustive and didactic with examples. When prototyping, the library must provide fast feedback loops, as in a python notebook for instance. Finally it must be easily integrated in a standard ecosystem such as being able to work with NumPy's array natively without imposing its own types. To sum up, practitioner's programmatic skills do not need to be high as his main goal is to focus on algorithms and mathematics formulas.

The contributer is an advanced user of a library who is very comfortable with its inner working, philosophy, aims, strengths and potential shortcomings. As such, he is able to add new specific features to library, fix some shortcomings or bugs. Usually a contributer is able to add a feature needed for a practitioner to finish his application. Furthermore he can then contribute back his features to the main project via pull requests if it is relevant. This way, a maintainer will assess the pull request and review it. The two main points of a contributer are his deep knowledge of a library and his ability to write code in the same language as the source code of it. Also, a contributer must have knowledge of coding best practices such as writing unit tests which are mandatory when adding a feature to an existing library. To facilitate contribution, a library must provide clear guidelines about the way to contribute, be easy to bootstrap and compile without having heavy requirements on dependencies. The best case would be that the library is handled by standard packages managers such as system apt or python conan.

The maintainer is usually the creator, founder of the library or someone that took over the project when the founder stepped back. Also, when a library grows, it is not rare that regular contributers end up being maintainer as well to help the project. The maintainer is in charge of keeping alive the project by fulfilling several aspects: upgrade and release new features according to the user (practitioner) needs and the library philosophy. Also, a library may not evolve as fast as the user may want it because of lack of time from maintainers. A lot of open source projects are maintained by volunteers and lack of time is usually the main aspect slowing development progress. The maintainer is also in charge of reviewing all the contributers pull requests. He must check if they are relevant and completed enough, (for instance, presence of tests and documentation) to be integrated in the project. Indeed, merging a pull requests equals to accepting to take care of this code in the future too. It means that further upgrade, bug fix, refactoring of the project will consider this new code too. If the maintainer is not able to take care of this code then it should propably not be integrated in the project in the first place. Any project and library has its maintainers. A maintainer is someone very familiar with the inner working and architectural of the project. He is also someone that has some history in the project to understand why some decisions has been made, what choices has been made at some points and what the philosophy of the project is. It is important to be able to refuse a contribution that would go contrary to the philosophy of the project, even a very interesting one. Finally the profile of a maintainer is one of a developer that is used to the standard workflow in open source based on: forks, branches, merge/pull requests and continuous integration.

Different tools

Before stating the topic of the thesis, it is important to enumerate the different kind of tools the current market has to offer to know where we will be positioning ourself.

Graphic editors are what neophyte thinks about when they imagine what image processing is. Those are tools that allow a non expert user to apply a wide array of operation on an image from an intuitive GUI in a way the user does not have to understand the underlying logic behind each and every operation he is applying. Such tools are usually large complex software such as The GIMP [84] or Photoshop [91]. Their aim is to be usuable by end users while supporting a large set of popular image format.

Command line utilities are binaries that perform one operation or more invocable from a console interface or from a shell script through a command line interface (CLI). This CLI usually offers several options to pass data and/or information to the programs in order to have an processing happening. The informations can be, for instance, the input image path, the ouput image name and the name of a mathematical morphology algorithm to apply. Usually command line utilities come as projects such as ImageMagick [97], GraphicsMagick [86] or MegaWave [47, 27].

Visual programming environment are software that allow the user to graphically and intuitively link one or several image processing operations while interactively displaying the result. The processing can easily be modified and the results are updated accordingly. Those software are usually aimed at engineer or researchers doing prototyping work not exclusive to image processing. Mathcad [82] is a good example of such a software.

Integrated environment are feature-rich platforms for scientists oriented toward prototyping. Those platforms provides a fully functional programming language and a graphical interface allowing the user to run commands and scripts as well as viewing results and data (image, matrices, etc.). The most well-known integrated environnement are Matlab [88], Scilab [89], Octave [95], Mathematica [90] and Jupyter [67] notebooks.

Package for dynamic language has known a surge in development these last few years and a multitude of libraries has been brought to dynamic languages this way. For instance, let us consider the python programming language. There are two main package provider: PyPi [98] and Conda [85]. Both allow to install packages to enable the user to program his prototypes in Python very quickly. In image processing, there are packages such as Scipi [19], NumPy [32], scikit-image [63], Pillow [92] as well as binding for OpenCV [11].

Programming libraries is the most common tool available out there. They are a collection of routines, functions and structures providing features through a documentation and binaries. They require the user to be proficient with a certain programming language and also to be able to integrate a library into his project. For image processing we have: IPP [28], ITK [57], Boost.GIL [30], Vigra [16], GrAL [29], DGTal [66], OpenCV [11], CImg [53], Video++ [60], Generic Graphic Library [15] Milena [36, 39] and Olena [103, 45, 48, 62].

Domain Specific Languages (DSL) are tools developed when a library developer deem he is unable to express the concepts and abstraction layers he wants to express through publishing a library. In this case, the barrier is often the programming language itself and so the developer does think that another layer of abstraction above the programming language would be a good thing. It leads to the genesis of a new programming language in some cases like Halide [58] and SYCL [79, 78] but can also be a case of having the current programming language be "upgraded" to include another subset of features that are not natively included. This is often the case in C++ where we have in-language DSL like Eigen [38], Blaze [49, 50], Blitz++ [17]

or Armadillo [69]. They leverage a possibility of the C++ programming language (*expression templates* [7]) to achieve it.

Topic of thesis

In the end, (find the quote) it is often known that there is a rule of three about genericity, efficiency and ease of use. The rule states that one can only have two of those items by sacrificing the third one. If one wants to be generic and efficient, then the naive solution will be very complex to use with lot of parameters. If one wants a solution to be generic and easy to use, then it will be not very efficient by default. If one wants a solution to be easy to use and efficient then it will not be very generic. In this thesis, we chose to work on an image processing library though continuing the work on Pylene [74]. But only working at library level would restrict the usability of our work and thus its impact. That is why we aim to reach prototyping users through providing a package that can be used in dynamic language such as Python without sacrificing efficiency. In particular, we aim to be usable in a jupyter notebook. It is a very important goal for us to reach a usability able to permeate into the educational side which is a strength of Python. In this library, we demonstrate how to achieve genericity and efficiency while remaining easy to use all at the same time. The scope of this library would be to specialize in mathematical morphology as well as providing very versatile image types. We leverage the modern C++ language and its many new features related to genericity and performance to break this rule in the image processing area. Finally, we attempt, through a static/dynamic bridge, to bring low level tools and concepts from the static world to the high level and dynamic prototyping world for a better diffusion and ease of use.

With this philosophy in mind, this manuscript aims at presenting our thesis work related to the C++ language applied to the Image Processing domain. It is organized as followed:

Genericity 2 presents a state-of-the-art overview about the notion of genericity. We explain its origin, how it has evolved (especially within the C++ language), what issues it is solving, what issues it is creating. We explain why image processing and genericity work well together. Finally we tour around existing facilities that allows genericity (intrinsically restricted to compiled language) to exists in the dynamic world (with interpreted languages such as Python).

Images and Algorithms taxonomy 3 presents our first contribution which is a comprehensive work in the image processing area around the taxonomy of different images families as well of different algorithms families. This part explains, among others, the notion of concept and how it applies to the image processing domain. We explain how to extract a concept from existing code, how to leverage it to make code more efficient and readable. We finally offer our take about a collection of concepts related to image processing area.

Images Views 4 presents our second contribution which is a generalization of the concept of View (from the C++ language, the work on ranges [77]) to images. This allows the creation of lightweight, cheap-to-copy images. It also enable a much simpler way to design image processing pipeline by chaining operations directly in the code in an intuitive way. Ranges are the cement of news design to ease the use of image into algorithms which can further extend their generic behavior. Finally we discuss the concept of lazy evaluation and the impacts of views on performances.

Static dynamic bridge 5 presents our third contribution which is a way to grant access to the generic facilities of a compiled language (such as C++) to a dynamic language (such as Python) to ease the gap between the prototyping phase and the production phase. Indeed, it is really not obvious to be able to conciliate generic code from C++ whose genericity is resolved at compilation-time (we call this the "static world"), and dynamic code from Python which rely on pre-compiled package binaries to achieve an efficient communication between the dynamic code and the library (we call this the "dynamic world"). We also cannot ask of the user to provide a compiler each time he wants to use our library from Python. In this part, we discuss what are the existing solutions that can be considered as well as their pros and cons. We then discuss how we designed an hybrid solution to make a bridge between the static world and the dynamic world: a static-dynamic bridge.

Chapter 2

Genericity

In natural language we say that something is generic when it can fit several purpose at once while being decently efficient. For instance, a computer is generic tool that allows one to write documents, access emails, browse Internet, play video games, watch movies, read e-books etc. In programming, we will say that a tool is generic when it can fit several purposes. For instance, the gcc compiler can compile several programming languages (C, C++, Objective-C, Objective-C++, Fortran, Ada, D, Go, and BRIG (HSAIL)) as well as target several architectures (IA-32 (x86), x86-64, ARM, SPARC, etc.). Henceforth we can say that gcc is a generic compiler. At this point it is important to note that even though a tool is deemed generic, there is a scope on what the tool can do and what the tool cannot do. A compiler despite supporting many languages and architectures, will not be able to make a phone call or a coffee. As such it is important to note that genericity is an aspect that qualifies something. We will now study the generic aspect related to libraries and programming languages.

Genericity within libraries is described by the cardinality of how many use-cases it can handle. Very often a library provide data structures, to represent and give sens to the data the user wants to process, as well as algorithms to process those data and provide different type of results. A library will be then labeled as *generic* when (i) its data structure allows the user to express himself fully with no limitation and when (ii) its algorithm bank is large enough to do anything the user would want to do with its data. In reality such a library does not exists and there are always limitations. Studying those limitation and what reason motivates them is the key to understand how to surpass them in the future, by developing new hardware and/or software support to new feature allowing for more genericity.

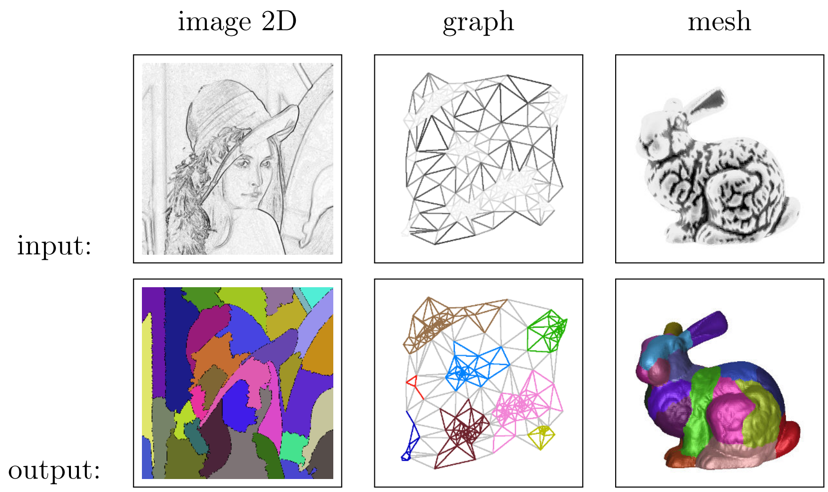
Genericity within programming language is described by the ability of the language to execute a code statement over a large amount of data structure, be they native (char, int, ...) or not (user defined). It is nowadays primordial for a programming language to be able to do so nowadays. Indeed, in a world where Information Technologies are everywhere, the amount of code written by software developers is staggering. And with it so are the amount of bugs and security vulnerabilities. Being able to natively have a programming language that enables to do *more* by writing *less* mathematically results in a reduced development and maintenance cost. Programming languages offers many ways to achieve genericity which is dependent of the language intrinsic specificities: compiled or interpreted, native or emulated, etc.

2.1 Genericity within libraries

Projecting the notion of genericity to Image Processing, we can deduce that we need two important aspects in order to be generic. First, we need to decorelate the data structures and its topology and underlying data from the algorithms. Indeed, we want our algorithms to support as much data structures as possible. Second, many algorithms share the same computational shape and can be factorized together.

Genericity can have two different meanings depending on the people you ask. For instance, some will argue that genericity is high level and qualifies a tool which is "generic enough" to handle all of his use-cases. Others will argue that genericity is about how the code is written: generic enough to handle all the use cases possibles. Neither is wrong. However, for the sake of comprehension we will use different words for each of these cases. A tool generic enough to handle a lot of use-case will be called *versatile*. Finally, for a tool whose aim is to provide a programming framework to handle code of any use-case we will use *generic*. In this paper, genericity will be about code. The figure 2.1 illustrates this result of the same generic watershed implementation applied on an image 2D, a graph as well as a mesh.

In image processing, there are 3 main axes around which genericity is working. The first axis is about the data type: grey level or RGB color (8-bits, 10-bits), decimal (double) and so on. The second axis, is about the structure of the image: a contiguous buffer (2D or 3D), a graph, a look-up table and so on. Finally, the third axis is about additional data that can be



The same code run on all these input.

Figure 2.1: Watershed algorithm applied to three different image types.

fed to image processing algorithms: structuring element (disc, ball, square, cube), labels (classification), maps, border information and so on. In the end, an image is just a point within this space of possibilities, illustrated in 2.2. Nowadays, it is not reasonable to have specific code for every existing possibility within this space. It is all the more true when one wants efficiency.

Figure 2.2: 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).

Genericity is not new and was first introduced in 1988 by Musser et al. [1] in 1988. The main point is to dissociate data structures and algorithms. The more your data structures and algorithms are tied together, the less you will be generic and will fail to handle multiple data structures in the same algorithm. Further work has been made about genericity in [4, 12]. Those works highlight the notion of abstraction to be able to turn an algorithm tied to a data structure into a generic algorithm. Notably in [37], Stepanov digs further and introduce the notion of *Concepts*, which are static requirements about the behavior of a type, by showing how to design a generic library and its algorithms. He highlights the importance of having the algorithms driving the behavior requirements, and not the opposite. These works are very suitable to be applied in the area of Image processing where we typically have a lot of algorithms (also called operators) that are required to work on a lot of different data structures (also called image types).

The authors explain in [83] how to capitalize on those works to turn a data-structure-specific image processing algorithm into a generic algorithm. We also explain how *concepts* can ease the implementation of generic algorithms. This approach is implemented in a library [74] which allows us to provide a proof of concept over the feasibility of having generic image processing operators running on multiple image types with near-native performances. Let us first explain briefly how we achieved this.

2.1.1 Different approaches to get genericity

First, let us 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 2.2. Without genericity, with s the number of image type, v the number of value type and k the number of structuring elements, one would have to write $s * v * k$ different *dilation* routine.

There are several ways to reach a high level of genericity. First there are the *code duplication* approach as well as the *generalization* approach. Finally, there is a way that consists in using expert, domain specific tools specifically engineered for this purpose and build upon them: those tools usually make heavy usage of *inclusion & parametric polymorphism*, also known as template metaprogramming in C++, to provide the basic bricks the user needs to build upon.

Code duplication approach consists in writing and optimizing the algorithm for a particular type in mind. Then, each time a new type is introduced, all the algorithm must be rewritten for this specific type. Additionally, each time a new algorithm is introduced, it must support all the existing types and thus be written multiple times. This approach does not scale well when the complexity of algorithms grows, and the number of data types increases. Neither it does allow the implementer to easily make use of optimization opportunities that can be offered by different data types having a common property. This translates into heavy switch/case statement in the code as show in 2.3 that illustrate how the *fill* algorithm needs to dispatch according to the input data type.

```
1  // image types parametrized by their
2  // underlying value type
3  template <ValueType V> struct image2d<V> { /* ... */ };
4  template <ValueType V> struct image_lut<V> {/* ... */};
5  // ...
6  void fill(any_image img, any_value v)
7  {
8    switch((img.structure_kind, img.value_kind))
9    {
10   case (BUFFER2D, UINT8):
11     fill_img2d_uint8( (image2d<uint8>) img,
12                       (uint8) any_value );
13   // ...
14   case (LUT, RGB8):
15     fill_lut_rgb8( (image_lut<rgb8>) img,
16                    (rgb8) any_value );
17   }
18 }
```

Figure 2.3: Fill algorithm skeleton with a switch/case dispatcher to ensure exhaustivity.

In addition, it is important to note that the exhaustivity aspect is only illustrated regarding the data structure types here. Indeed, the data structures are all already generic for their underlying data type (named *ValueType* in the code). When one write `image2d<uint8>` (l.10), it means *2D-image whose pixels' have a single channel 8-bits value*. This approach enables one to write an algorithm at maximum efficiency for a particular data type, however one can easily miss optimization opportunities if not knowledgeable enough too. This approach is best for early prototypes and trying to find common behaviors pattern among algorithms, or common properties across different data types. No IP library has chosen this approach due to the obvious maintenance issue induced.

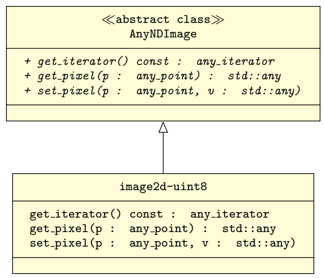
Generalization approach consists in finding a common denominator to all the image types. Once designed, this common denominator, also called supertype, will allow the library developer to write all the algorithms only once: for the supertype. The processing pipeline will then consist in three steps. First convert the input image type into the supertype, second process the supertype into the algorithm pipeline requested by the user, finally convert back the resulting image into the specific image type the user is expecting. This approach offers the advantage of being maintainable. Adding a new image type is just a matter of providing the two conversions facilities: to and from the supertype. Adding an algorithm is also just a matter of writing it once for the supertype. This mechanism is shown in 2.4. However, one must keep in mind that the conversion can be costly. Also, processing the supertype may induce a significant performance trade-off while processing the original type would be much faster. Furthermore, it is not always possible to find this common denominator when enumerating through some esoteric data types. Finally, the provided interface (from the supertype) may allow the image to be used incorrectly, such as a $2D$ image being processed into video $(3D + t)$ algorithm. Widely use libraries such as OpenCV [11], scikit-image [64] use this technique to handle as many image types as possible. There are other libraries, for instance CImg [53], MegaWave [47], that also use this approach however this paper will not address them.

```
struct any_image { /* ... */ }; // generalized type
// specific types w/ conversion routines
struct image2D { any_image to(); void from(any_image); };
struct imageGraph { any_image to(); void from(any_image); };
// ...
void fill(any_image img, any_value v) {
  for(auto p : img.pixels())
    p.val() = v;
}
```

Figure 2.4: Fill algorithm for a generalized supertype.

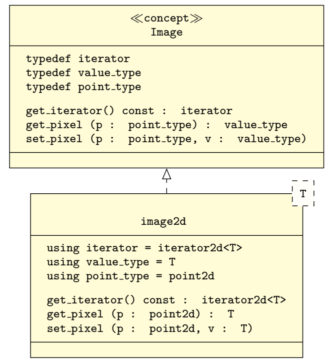
Inclusion & Parametric polymorphism approach consists in extracting behavior patterns from algorithms to group them into logical brick called *concepts* (for static parametric polymorphism), or *interface* (for dynamic inclusion polymorphism). Each algorithm will require a set of behavior pattern that the inputs need to satisfy. In C++, it can be done either by using inclusion polymorphism, or by using parametric polymorphism, as shown in 2.5. In [83], the authors leverage a new C++20 feature (the concept) to show how it is possible to turn an algorithm, specific to an image type, into a more abstract, generic one that does not induce any performance loss.

Multiple libraries exist and leverage this approach to try to achieve a high genericity degree as well as high performance by offering varied abstract facilities over image types and underlying data types. Those are IPP [28],




```
void fill(AnyNDImage& in,
            std::any val)
{
  for (any_point p :
        in.get_iterator())
      out.set_pixel(p, val);
}
```

(c)




```
template <class InImage>
void fill(InImage& in,
  InImage::value_type val)
{
  for (auto p :
        in.get_iterator())
      out.set_pixel(p, val);
}
```

(d)

Figure 2.5: Dynamic, object-oriented polymorphism (a) vs. static, parametric polymorphism (b).

ITK [57], Boost.GIL [30], Vigra [16], GrAL [29], DGTal [66], Milena [36, 39], Olena [103, 45, 48, 62] and Pylena [74]. Most of them have been written in complex C++ whose details remain visible from the user standpoint and thus are often difficult and complex to handle. It is also harder to debug because errors in highly templated code shows up very deep in compiler error trace.

Alternative approaches such as relying on a *Domain Specific Language* (DSL) exists and usually try to address heterogeneous computation, which can be considered as another kind of genericity related to the target architectures. Halide [58] and SYCL [79, 78] both provide their own DSL to that end. Others like Eigen [38], Blaze [49, 50], Blitz++ [17] or Armadillo [69] have their main goal set on performances and try to provide a generic way to address the issue of parallelization and/or vectorization leveraging lazy computing via a construct named *expression templates* [7]. They do not aim to be able to handle as many input types as possible, however, the lazy-computing techniques used generates new types on-the-fly. Henceforth, those libraries still need to have embed generic facilities. This paper address genericity at the input level rather than the target architecture level, henceforth, we will not broach this topic in this paper.

The table comparing all the pros. and cons. from the aforementioned approaches is presented in table 2.1.

Table 2.1: Genericity approaches: pros. & cons.

Paradigm	TC	CS	E	1IA	EA
Code Duplication	✓	✗	✓	✗	✗
Code Generalization	✗	≈	≈	✓	✗
Object-Orientation	≈	✓	✗	✓	✓
Generic Programming:					
with C++11	✓	≈	✓	✓	≈
with C++17	✓	✓	✓	✓	≈
with C++20	✓	✓	✓	✓	✓

TC: type checking; CS: code simplicity; E: efficiency
1IA: one implementation per algorithm; EA: explicit abstractions / constrained genericity

2.1.2 Unjustified limitations

Image processing community operates mostly with either Python or Matlab [21]. As such this paper will focus on those two technologies. Python offers access to two major libraries for image processing: OpenCV and scikit-image. Matlab has built-in support as well as toolboxes for more advanced features. When we intersect scikit-image and Matlab, we can notice that both are very similar both in feature and interface. As such, it is possible to regroup them both here for the sake of comprehension. As stated above, when considering a generic library, one must consider the three axes: underlying data type, domain structure and additional data. Let us compare how the mentioned library behave along those axes with a simple algorithm such as the morphological dilation.

Limitations regarding feasibility

Data type Dilating a grayscale or a binary image works fine as intended with all the libraries. However, when dilating a RGB colored image, usually the algorithm should be able to work if a supremum function is provided (or a defaulted one is automatically selected). Despite that fact, scikit-image does not allow one to dilate a colored RGB image and raises an error: it is required to convert the image beforehand.

OpenCV arbitrarily decides that the dilation consists in dilating each channel of the coloured image separately from one another, which most of the time is wrong because false colors may appear. Furthermore, it is not possible to provide a supremum function to the dilation algorithm.

Domain structure To perform a dilation, it is required to have a structuring element whose shape match the structure of the domain of the image. For instance, dilating a 2D-image requires the use of a structuring element whose shape may be a disc or a rectangle. To dilate a 3D-image, one would need to use a structuring element whose shape is of a ball or a cube. Scikit-image supports 3D-images as well as structuring element whose shape are compatible (ball, rectangle and octahedron). This naturally leads to having a support for the dilation of 3D-images. On the other hand, OpenCV does support 3D-images whereas its dilation algorithm cannot handle them. The algorithm exits with an error. Worse, when passing a wrong structuring element (a rectangle) to the dilation algorithm alongside the 3D-image, the algorithm works and produce a result which is false: it is different from the application of the 2D- structuring element on each slice of the image.

Limitations regarding optimizations

Each library has its own strategies to optimize its routines when implementing them.

Scikit-image Scikit-image, for instance, will check whether the structuring element is separable (only for rectangle shapes) so that it can dispatch on an optimized multi-pass 1D routine for each part separated which linearize the execution time and greatly improve performances for large structuring element.

Also, Scikit-image relies on SciPy internals which does not abstract the underlying data type for the algorithm implementer. As such, each algorithm must provide a switch/case dispatch for every supported type (floating points, 8-bit channel, 16-bit channel, RGB, etc.), and it must provide it in the middle of the algorithm implementation. If one type is not natively supported; an error occurs and the program halts. Henceforth, handling a new supported data type will requires to review every single written algorithm.

On the other hand, SciPy provides an abstraction layer over the dimensional aspect of the image by providing a tool named point iterator. This tool allows one to iterate over every point of the image, without being aware of the number of its dimension, and make the translation from the abstract iterator to the actual offset in the data buffer of the image. The implementer can then only worry about handling the underlying data type to provide a generic algorithm. This approach, sadly, is fully dynamic (that is, runtime) and does not allow the compiler to provide native optimization such as vectorization out of the box.

OpenCV & Matlab In OpenCV as well as in Matlab, the choice was made to systematically attempt to decompose big rectangular structuring elements into smaller $3x3$ structuring elements. This is not as effective as using multi-pass 1D algorithm but still allows for relatively stable performances.

Also, OpenCV let the implementer handle the cases he wants to support by himself. For instance, the dilation algorithm is written with a dispatch on the data type before the actual call to the algorithm. This enables compiler optimizations such as vectorization because all the required information is known at the right time. It also enables offloading the computation into GPU kernels when feasible. However, the downside is that few algorithms are written in a way to handle multidimensional images. Most are written to only handle specific subsets. As such, conversion from one subset to another may be unavoidable when writing an algorithm pipeline for a more complex application. For instance, it is currently not possible to dilate a 3D image with a 3D ball (as stated above).

Another point to note with OpenCV is the requirement to do temporary copies (to extract data or to have working copy) when writing an algorithm. For instance, it is currently not possible to write a blurring algorithm operating only on the green channel of an RGB image. One must first extract the green channel into a single channel temporary image, blur that image, to finally put the result back into the original image. Generally, in-place computation is poorly handled in OpenCV.

Benchmark When comparing performances of the simple dilation between Matlab and OpenCV, which is done in [52], shows that Matlab is very oriented toward prototyping and not toward production. The performance gap between the two libraries shows that performances may not a major concern for MatLab in this case. Opposite to this, OpenCV and scikit-image both have a C/C++ core to provide fast basic algorithms such as the dilation and erosion mathematical morphology.

As such, when comparing the performances of OpenCV, Scikit-image and Pylene in fig. 2.6, we can notice some interesting facts. Both scikit-image and Pylene have a very stable execution time even though the size of the structuring element grows by power of two. This corroborate the fact that the author did see code taking advantage of the structuring element's properties, such as the decomposability/separability. OpenCV has very good performances for a square because it has specific handwritten code for both vectorization and GPU offloading when possible: even if OpenCV decomposed its square into smaller sub square (and not periodic lines), it remains steady fast.

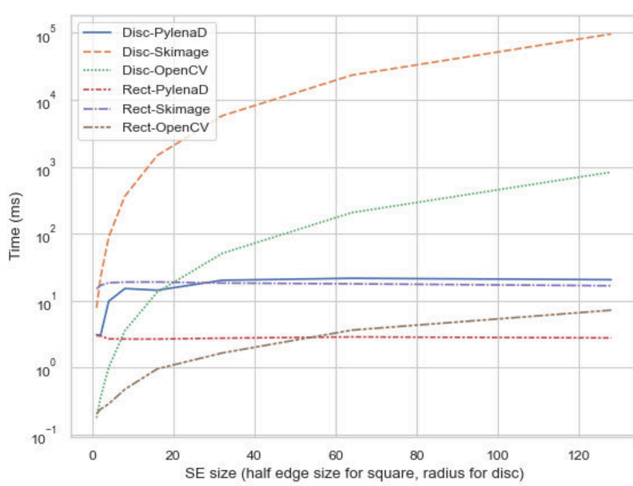


Figure 2.6: Benchmark: dilation of a 2D image (3128x3128 ≈10Mpix) with a 2D square and a 2D disc.

In the case of a structuring element shaped as a disc (also in fig. 2.6), we can observe that the execution time raises exponentially for both Scikit-image and OpenCV whereas Pylene remains regular and steady fast. These results show that Pylene's attempt to decompose each structuring element into periodic lines when possible may be a little bit slower for smaller structuring elements whereas it is much more regular and faster when the structuring element start to be of a certain size.

2.2 Genericity in pre C++11

Before C++11 [43] came out the genericity facilities offered by the C++ programming language were already turing complete [25]. However, it was lacking a certain amount of features the language now have that made writing generic code a real challenge at that time. For instance, when writing code with a variable number of type (nowadays designed as variadic templates) one had to write the generic code for each and every number of type supported. This meant that to implement `std::tuple`, one had to copy the implementation for every number of type supported by `std::tuple`. This limitation defeated the very first principle and motivation of generic programming which is to write *less* Code. To compensate, library implementers used tricks with macro not to have to rewrite code which made the initial code even harder to understand for outsiders.

2.2.1 SFINAE: Substitution-Failure-Is-Not-An-Error

In spite of all those limitations at the time, generic programming was well supported by the language and allowed the programmer to already design generic libraries, in particular thanks to the SFINAE [22] (substitution-failure-is-not-an-error) technique that leads to the popularisation of the usage of the `std::enable_if` meta-programming facility. The SFINAE technique relies on a feature of the C++ programming language. Indeed, when standardizing how the compiler should resolve and select function overloads, in a templated context, the standard committee chose to have the following behavior. When substituting the explicitly specified or deduced type for the template parameter fails, the specialization (function overload candidate) is discarded from the overload set (of matching functions) instead of causing an error.

This feature allows to write code that seem to be ill-formed, for instance in a function, trying to access to a class member type, variable or function that does not exist should be ill-formed. However, because it happens in the templated world, when the compiler tries to compile the function code with a given type, the compiler will just discard the function from the overload resolution at call-site instead of throwing a hard error. An error can occur only when the compiler tried all the overload it knows and still could not find an overload that was not ill-formed. If this happen, the compiler will then proceed to list all the overloads it tried, to list all the template substitution it tried and finally to list why it failed. This mechanism is the very reason of the unpopularity of this technique because it leads to situation where the compiler can output *several Mos* of error message for one single file. Error messages becomes incomprehensible very fast and programs are hard to debug. But still, it was the only technique we had to perform detections on types at compile time and process some kind of contraints on them. For instance, a code that detects wether a class provide a subtype named `custom_type` at that time is shown in fig. 2.7.

2.2.2 CRTP: Curiously Recurring Template Pattern

Another features that precede C++11 and was available in C++98 [9] and C++03 [24] is the curiously recurring template pattern (CRTP). This programming technique allowed a base class (in its specific code) to be aware of its derived class at compile type. This pattern is extremely useful to solve issues revolving around covariant return type polymorphism in C++ with pointers. Indeed, before smart pointers were standardized, they existed in libraries such as boost [93] and were used to solve memory leak issues. However, when implementing cloning facilities where lots of derived class were involved, one could not just return a smart pointer of the base class since no derived class derived from the capsule smart pointer class itself. This broke the covariant return type feature. CRTP is a tool that brings a solution: we were now able to construct a smart pointer capsule of the derived class inside the base class: there are now only one function in the base class and no ambiguities is detected by the compiler.

For instance, a cloning facility with no smart pointers was implemented with covariant return type in the code shown in fig. 2.8. The obvious disadvantage was to have to deal with naked pointers, `new`, `delete` and all the consequences that comes with the manual memory handling. When the programmer wants to switch to an implementation that uses smart pointers, he would naively write the code shown in fig. 2.9.

The solution is then found with the CRTP technique where we do all the creation inside the base class instead of forcing the derive class to implement its own cloning facility. The code in fig. 2.10 shows how it is done. In order for the user not to have to write `AbstractClass<Devived>` in his code, we have adopted another abstraction layer to hide this implementation detail in the form of a intermediary class `Base` that hides the CRTP complexity.

Thanks to these two techniques (SFINAE and CRTP), past work was done to achieve a design: SCOOP [23, 31, 35]. These design combined CRTP and SFINAE to build a machinery where it is possible to apply constraints via concepts (in the sense described in *Elements of Programming* by Stepanov and McJones). The library Olena [103, 48] was born and carried the work around this field of research applied to Image processing in [14, 13, 20, 26]. This design is described in details by Levillain among his work [36, 39, 40, 46, 45, 51, 62]. Finally, with the release of new C++ standards in 2011 [43], then 2014 [61], 2017 [71] where template metaprogramming facilities were greatly improved, it was necessary to review once more this design to improve the design in order to achieve genericity, performance and ease of use. This is the birth of a new library, Pylene [74]. In the end, it was C++20 [43] that marked the shift wanted by Stepanov [1, 4, 12, 37] and Stroustrup [5, 10, 100, 33] for years with the coming of Concepts, and all the new possibilities it brings to the programmer.

```cpp
// Step 1: write the detector using partial specialization
template <class TestedType, class Void = void>
struct has_nested_sub_type {
  typedef bool type;
  static const type value = false; // default value: not detected
};
// This class template specialization has value set as true
// when it detects that the nested type exists
template <class TestedType>
struct has_nested_sub_type<TestedType, typename TestedType::custom_type> {
  typedef bool type;
  static const type value = true;
};

// Step 2: declare the well- and ill-formed classes
class well_formed {
  typedef int custom_type;
};
class ill_formed {
  typedef int another_type;
};

// Step 3: implement the enable_if facility that will use our
// detector written at step 1
template<bool B, class T = void>
struct enable_if {
};
template<class T>
struct enable_if<true, T> {
  typedef T type;
};

// Step 4: write overloads the compiler will use
// overload #1
template <typename UserType>
void my_procedure(const UserType& ut) {}; // accept everything

// overload #2
template <typename UserType>
void my_procedure(const UserType& ut, // accept only constrained types
typename enable_if<has_nested_sub_type<UserType>::value>::type* = 0) {};

int main() {
  well_formed wlf;
  ill_formed ilf;

  my_procedure(wlf); // will call overload #1
  my_procedure(ilf); // no hard error, will call opverload #2

  // A hard error would occur only if overload #1 did not exist.
}
```

Figure 2.7: C++0x SFINAE detection of nested sub-type.

```cpp
#include <string>

class AbstractBase {
public:
    virtual ~AbstractBase()                         = default;
    virtual AbstractBase*       clone() const    = 0; // covariant return type
    virtual const std::string&  get_name() const = 0;
};

class Derived : public AbstractBase {
    std::string name_;
public:
    Derived(const std::string& name) : name_(name) {}
    Derived* clone() const /* override */ {
    return new Derived(name_);              // works thanks to covariance
    }
    const std::string& get_name() const {
    return name_;
    }
};

int main() {
    AbstractBase* objptr = new Derived("John");     // works
    AbstractBase* cloned_objptr = objptr->clone();  // also works
    objptr->get_name();        // "John"
    cloned_objptr->get_name(); // also "John"
    // Do not forget to delete to avoid memory leaks
    delete cloned_objptr;
    delete objptr;
}
```

Figure 2.8: C++0x cloning example with covariant return type.

```cpp
#include <string>
#include <smart_pointers>

class AbstractBase {
public:
  virtual ~AbstractBase()                           = default;
  virtual unique_ptr<AbstractBase> clone() const    = 0; // covariance is lost
  virtual const std::string&       get_name() const = 0;
};

class Derived : public AbstractBase {
  std::string name_;
public:
  Derived(const std::string& name) : name_(name) {}
  unique_ptr<Derived> clone() const /* override */ { // No covariance
    // does not work because Derived does not derive from unique_ptr
    return unique_ptr<Derived>(new Derived(name_));
  }
  const std::string& get_name() const{
    return name_;
  }
};

int main() {
  unique_ptr<AbstractBase> objptr =
      unique_ptr<AbstractBase>(new Derived("John"));        // works
  unique_ptr<AbstractBase> cloned_objptr = objptr->clone(); // does not work
  objptr->get_name();        // "John"
  cloned_objptr->get_name(); // also "John"
  // No delete needed
}
```

Figure 2.9: C++0x not-working cloning example with smart pointers.

```
#include <string>
#include <smart_pointers>

class AbstractBase {
public:
    virtual ~AbstractBase()                          = default;
    virtual unique_ptr<AbstractBase> clone()    const = 0;
    virtual const std::string&       get_name() const = 0;
};

template <class Derived>
class Base : public AbstractBase{
public:
    virtual unique_ptr<AbstractBase> clone() const /* override */ {
        // Covariance is kept by converting here
        return unique_ptr<Derived>(new Derived(get_name()));
    }
};

class Derived : public Base<Derived> {
    std::string name_;
public:
    Derived(const std::string& name) : name_(name) {}
    const std::string& get_name() const {
        return name_;
    }
};

int main() {
    unique_ptr<AbstractBase> objptr =
        unique_ptr<AbstractBase>(new Derived("John"));        // works
    unique_ptr<AbstractBase> cloned_objptr = objptr->clone(); // does work
    objptr->get_name();       // "John"
    cloned_objptr->get_name(); // also "John"
    // No delete needed
}
```

Figure 2.10: C++0x working cloning example with smart pointers.

2.3 Genericity in post C++11 (C++20 and Concepts)

```
1 template <Collection C, ValueType V>
2   requires Same<Collection::ValueType, ValueType>
3 void fill(C c, V v) {
4   for(auto e : c)
5     e = v;
6 }
```

Figure 2.11: Fill algorithm, generic implementation.

Most of the algorithms are *generic* by nature. What limits their genericity is the way they are implemented. This statement is justified by the work achieved in the Standard Template Library (STL) [12] in C++ whose algorithms are implemented and designed in a way where they work with all the built-in collections (linked list, vector, etc.). Let us take the example of the algorithm `fill(Collection c, Value v)` which set the same value for all the element of a collection (see fig. 2.11). There are three main requirements here that are not related to the underlying type of *Collection*. First, we check (l.2 2.11) that we are actually filling the collection with the correct type of value. Indeed, it would not make sense, for instance, to assign an RGB triplet color into a pixel from a grayscale image. Secondly, we need to be able to iterate over all the element of the collection (l.4 2.11). Finally, we need to be able to write a value into the collection (l.5 2.11). This requires the collection not to be read-only, or the collection's values not to be yielded on-the-fly. This allows us to deduce what is called a *concept*: a breakdown of all the requirement about the behavior of our collection. When writing down what a *concept* should require, one should always respect this rule: "*It is not the types that define the concepts: it is the algorithms*". Concepts in C++ are not new and there have been a long work to introduce them that goes back from 2003 [101, 100, 102] to finally appear in the 2020 standard [104] (referred as C++20 [43]). This allows us, as of today, to write code leveraging this facility.

2.3.1 Conceptification

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 [101, 100, 102], it will finally appear in the next standard [104] (C++20).

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 [1]) 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 [37], 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*.

Image processing algorithms, similarly, are *generic* by nature [2, 14, 20, 39, 62]. When writing an image processing algorithm, there is always a way to express it with a high level of genericity. For instance, if is possible to write a morphological dilation in a way that does not care about the underlying value type, the domain nor the structuring element specificities. The most abstract way to write a dilation is shown in 2.12.

This implementation introduces three concepts at line 1: *Image*, *WritableIm-*

```
1  template <Image I, WritableImage O,
2             StructuringElement SE>
3  void dilation(I input, O output, SE se) {
4    assert(input.domain() == output.domain());
5    for(auto pnt : input.points()) {
6      output(p) = input(p)
7      for (nx : se(p))
8        output(p) = max(input(nx), output(p))
9    }
10 }
```

Figure 2.12: Dilation algorithm, generic implementation.

age and *StructuringElement*. Following the behavior of each one of them into the algorithm, we can deduce a list of requirements for each one of them.

Image is the most basic representation of what an image should be. An image should (a) provide a way to access its domain (l.3 2.12) and (b) a way to iterate over its points (l.4 2.12). This then allows us later to (c) access to the value returned by the image at this point (l.5 2.12). To this point the value is only accessed in read-only. We can then write the following two concepts:

```
template <typename I>
concept Image = requires {
  typename I::point_range;            // needed for b
  typename I::point_type;             // needed for c
  typename I::value_type;             // needed for c
} && ForwardRange<I::point_range>     // needed for b
&& requires (I ima, I::point_type pnt) {
  { ima.domain() };                   // a
  { ima.points() } -> I::point_range  // b
  { ima(pnt) }     -> I::value_type   // c
};
```

In reality, more boilerplate code is needed to ensure, for instance that there is no type mismatch between the image's `point_type` and the `point_range`'s value type. For the sake of brevity this boilerplate code is omitted here.

WritableImage is a more specific concept based on the previous *Image* concept. It requires that the image's value can be (d) accessed to be modified: the user should be able to write into the image's value accessed by a specific point (l.6 2.12). We can then write the following two concepts:

```
template <typename WI>
concept WritableImage = Image<WI>
&& requires (WI wima, I::point_type pnt,
```

```
            I::value_type val) {
  { wima(pnt) = val };                 // d
};
```

StructuringElement is an additional input to the image defining the window around each point that will be considered during the dilation (also called the neighborhood). A structuring element should just provide a list of point when input with one (e). From this behavior we can deduce the following concept:

```
template <typename SE, typename I>
concept StructuringElement = Image<I>
&& requires (SE se,I::point_type pnt) {
  { se(pnt) } -> I::point_range;      // e
}
```

This new notion of concept is very important because it decorrelate the requirements on behavior required inside algorithms from the way the data structures are designed. One wan always wrap a specific data structure so that it can behave properly into an algorithm, without needing to rewrite that algorithm.

2.3.2 Simplifying code

The main advantage brought by using modern C++ as the implementation language for an image processing library is to be able to leverage what is called metaprogramming. Metaprogramming is a way to tell the compiler to make decision about which type, which code to generate. These decisions, made at compile time, and then absent from the resulting binary: only the fast and optimized code remains. This bring a new distinction between the static world (what is decided at compile time) and the dynamic world (what is decided at runtime). The more is decided at compile time the smaller, faster the binary will because there is work less to do at runtime. By following this principle, one can think of some properties that are known ahead of time (at compilation) when writing one's image processing algorithm. For instance, when considering the example of the dilation whose code is shown in 3.4, we can see that the property about the decomposability if the structuring element is linked to the type. This means that when the structuring element's type is of a disc, or a square, the compiler will know at compile time that it is decomposable. To tell the compiler to take advantage of a property at compile time, C++ has a language construct named if-constexpr. The resulting code then becomes:

```
template <Image Img, StructuringElement SE>
auto dilate(Img img, SE se) {
  if constexpr (se.is_decomposable()) {
    lst_small_se = se.decompose();
    for (auto small_se : lst_small_se)
      img = dilate(img, small_se) // Recursive call
    return img;
  } else if (is_pediodic_line(se))
    return fast_dilate1d(img, se) // Van Herk's algorithm;
  else
    return dilate_normal(img, se) // Classic algorithm;
}
```

There are other ways to achieve the same result with different language constructs in C++. There are two "legacy" language constructs which are tag dispatching (or overload) and SFINAE. With the release of C++17 came a new language construct presented above: if-constexpr. Finally, with C++20, it will be possible to use concepts to achieve the same result. To achieve the same result as above with tag dispatching, one would need to write the following code:

```
struct SE_decomp {};
struct SE_no_decomp {};

template <Image Img, StructuringElement SE>
auto dilate(Img img, SE se) {
  // either SE_decompo or SE_no_decomp
  return dilate_(img, se, typename SE::decomposable());
}

auto dilate_(Img img, SE se, SE_decomp) {
  lst_small_se = se.decompose();
  for (auto small_se : lst_small_se)
    // Recursive call
    img = dilate(img, small_se, SE_no_decomp)
  return img;
}
auto dilate_(Img img, SE se, SE_no_decomp) {
  if (is_pediodic_line(se))
    return fast_dilate1d(img, se) // Van Herk's algorithm;
  else
    return dilate_normal(img, se) // Classic algorithm;
}
```

To achieve the same result with SFINAE, one would need to write the following code:

```
// SFINAE helper
template <typename SE, typename = void>
struct is_decomposable : std::false_type {};
template <typename SE>
struct is_decomposable<SE,
```

```
    // Check wether the type provides the decompose() method
    std::void_t<decltype(std::declval<SE>().decompose())>
  > : std::true_type {};
  template <typename SE>
  constexpr bool is_decomposable_v =
                  is_decomposable<SE>::value;

  template <Image Img, StructuringElement SE,
    typename = std::enable_if_t<is_decomposable_v<SE>>>
  auto dilate(Img img, SE se) {
    lst_small_se = se.decompose();
    for (auto small_se : lst_small_se)
      img = dilate(img, small_se) // Recursive call
    return img;
  }

  template <Image Img, StructuringElement SE,
    typename = std::enable_if_t<not is_decomposable_v<SE>>>
  auto dilate(Img img, SE se) {
    if (is_pediodic_line(se))
      return fast_dilate1d(img, se) // Van Herk's algorithm;
    else
      return dilate_normal(img, se) // Classic algorithm;
  }
```

Comparing those two last ways of writing static code to the first one comes to an obvious conclusion: the if-constexpr facility is much more readable and maintainable than the two legacy ways of doing it. Finally, there is still another way to handle the issue and it is with C++20's concepts. The following code demonstrates how to leverage this language construct:

```
  template <typename SE>
  concept SE_decomposable = requires (SE se) {
    se.decompose(); // this method must exist
  };

  template <typename Img, typename SE>
  auto dilate(Img img, SE se) {
   if (is_pediodic_line(se))
      return fast_dilate1d(img, se) // Van Herk's algorithm;
    else
      return dilate_normal(img, se) // Classic algorithm;
  }

  template <typename Img, typename SE>
    requires SE_decomposable<SE>
  auto dilate(Img img, SE se) {
    lst_small_se = se.decompose();
    for (auto small_se : lst_small_se)
      img = dilate(img, small_se) // Recursive call
    return img;
  }
```

A best-match mechanic operates under the hood to select the function overload whose concept is the most specialized when possible.

2.3.3 Benchmarks

TODO: clean figures (titres, gradution, 2 par 2, labèles & axes)

In order to get a real feeling on how fast would the compile time of programs would be impacted, we wanted to do benchmarks. The aim is to mesure how much faster concepts will be in production code. To do so we used Metabench [94], a benchmarking facility to benchmark compilation time of C++ programs. We wrote programs (given in appendix C) to benchmark compilation time with three compilers Gcc-10, Clang-10 and Clang-11.

For Gcc-10, we can see the results in fig. 2.13. Those results show that the concept implementation is still very much slower than the SFINAE implementation. However, we were able to pinpoint the slowness of this implementation which is the implementation of the library trait `std::movable`. Indeed, when lightening this trait into some builtin intrinsic (that are almost equivalent), we can see the curve *concept_fast* in fig. 2.14 being faster than the SFINAE implementation.

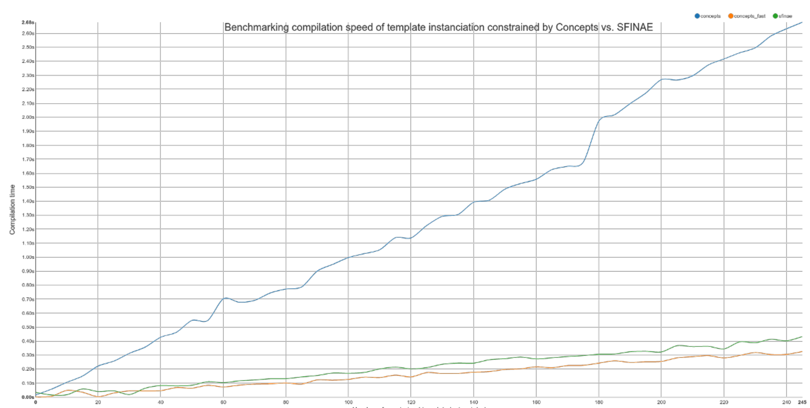


Figure 2.13: Benchmark GCC-10: Benchmarking compilation speed of template instanciation constrained by Concepts (vanilla) vs. Concepts (improved) vs. SFINAE.

For Clang-10, the concept implementation is globally slower than the SFINAE implementation (seen in fig. 2.15). Even not using the slow `std::movable` library trait does not do the trick as seen in fig. 2.16

Finally, for Clang-11, we can see that the builtin implementation of concepts has improved very much from its previous version (see fig. 2.17). How-

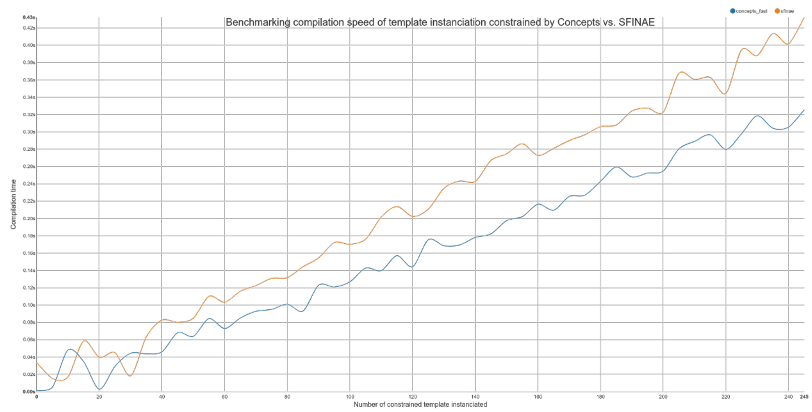


Figure 2.14: Benchmark GCC-10: Benchmarking compilation speed of template instanciation constrained by Concepts vs. SFINAE.

ever, we see that the library trait `std::movable` is still the source of a massive slowness (see fig. 2.18) that needs to be addressed in future versions of both Gcc and Clang.

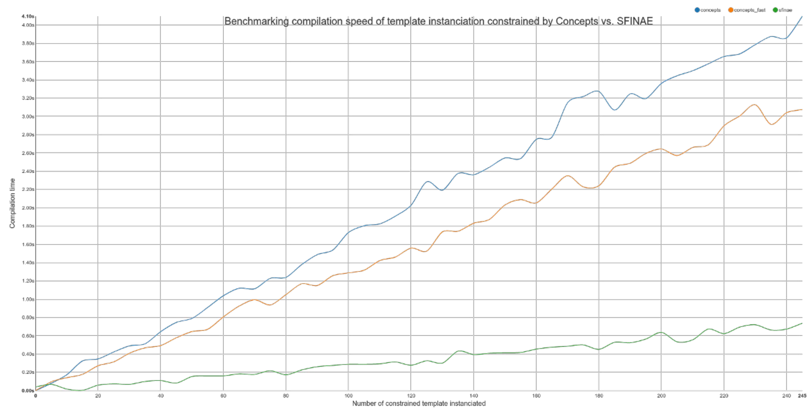


Figure 2.15: Benchmark Clang-10: Benchmarking compilation speed of template instanciation constrained by Concepts (vanilla) vs. Concepts (improved) vs. SFINAE.

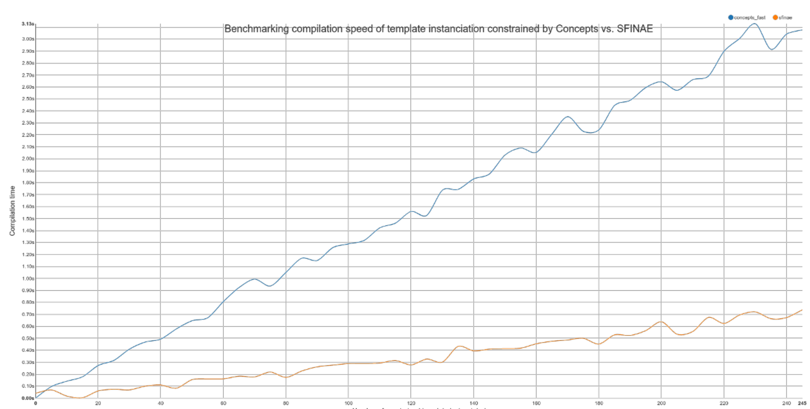


Figure 2.16: Benchmark Clang-10: Benchmarking compilation speed of template instanciation constrained by Concepts vs. SFINAE.

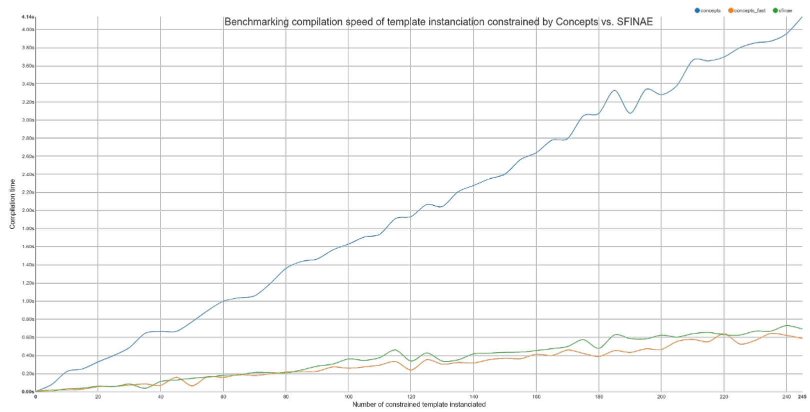


Figure 2.17: Benchmark Clang-11: Benchmarking compilation speed of template instanciation constrained by Concepts (vanilla) vs. Concepts (improved) vs. SFINAE.

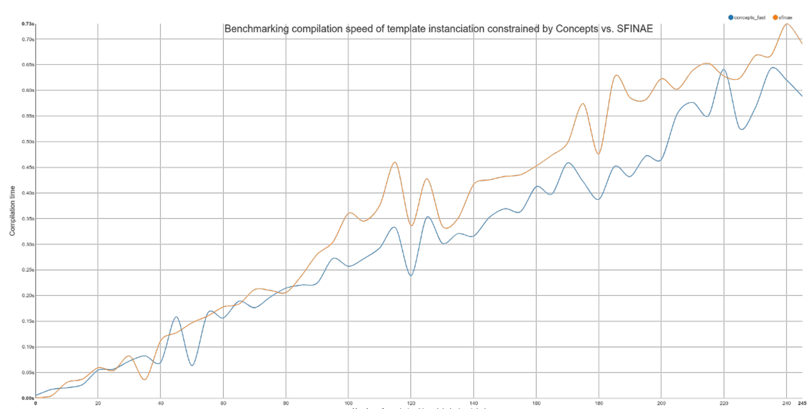


Figure 2.18: Benchmark Clang-11: Benchmarking compilation speed of template instanciation constrained by Concepts vs. SFINAE.

2.4 C++ templates in a dynamic world

There are two main categories of programming languages. First are the *compiled* programming languages which requires to feed the source code to a program (a compiler) that will output a binary. This binary will then produce the desired output once the user execute it. Some well known languages of this category are C, C++, Ada, Fortran. Secondly there are the interpreted programming language which requires to feed the source code to a program (an interpreter) that will directly produce the output as is a binary was executed. Some well known languages of this category are Javascript, Python, Matlab, Common Lisp. There is a third category that tries to combine the best of both world by compiling into bytecode which is an optimized intermediate language that will then be interpreted into a virtual machine. The most famous are indubitably Java and C#. Both categories have advantages as well as drawbacks.

Compiled languages are still widely spread and used as of today. They present a working pipeline which is very classic. First the programmer will write code, then the compiler will build a binary optimized for the target machine and finally the programmer can execute his binary to produce a result. Usually the compilation step is slow whereas executing the binary is fast. There is no additional step when it comes to the binary execution. This, however, has the effect of having a poor portability. Indeed, my binary optimized to use fast and recent SIMD AVX-512 instructions will not work on an old x86 machine that does not support those instructions. When distributing our program, multiple binaries must be produced for each supported CPU architectures. Furthermore, usually compiled languages have very poor support of dynamic language features such as reflection, code evaluation or dynamic typing. It tends to improve with time but solutions are limited to compile-time informations or need to ship a JIT-compiler into the final binary (such as cling [54] for C++) to generate new binary on-the-fly to be executed right after. This has two drawbacks: slowness when the case of needing compilation is presented and increase in the binary size.

Interpreted languages are also widely used, especially in the research area where a fast feedback loop between prototyping and getting results is needed. The compilation time is very fast and allows a program to be almost instantly executed. In fact, the compilation can be done just ahead of program execution not to compile unnecessary code. However the execution time will generally be slow. To the user it is invisible because both compilation step and execution step are blurred together. Furthermore, most of the time a same interpreted program is executed once. Then the programmer will modify it and continue its prototyping process. The real advantage of an interpreted language is the portability. As it is the responsibility of the client to install the correct language interpreter (for the correct version) before running the program from the source code, as long as an interpreter can be installed on a machine, the program can then be run. This also drastically slow distributed package for programs as only source code must be distributed instead of compiled binary. However, the source code is leaked with all the security implication this can have. Finally, interpreted language usually have better memory management (builtin garbage collectors), are easier to debug, have very rich support of dynamic typing, dynamic scoping, reflections facilities, on-the-fly evaluation from the source code or even more like modifying the Abstract Syntax Tree (AST) resulting from the first compilation pass on source code by the interpreter. This last one is implemented in Common Lisp in the name of macros. There are more to say about interpreted languages, especially about those that are compiled into bytecode and tend to get the best of both worlds without the drawbacks but this thesis will not discuss this matter any further.

The main point to understand here is that our main interest is set on C++, a compiled language with slow compilation time and very fast execution time. In C++ there are template metaprogramming to achieve genericity but *templates do not generate any binary code*. Why? Because when the compiler meet a templated type or a templated routine, it does not know which type it will be instantiated with when it is used. Therefore, it can not calculate stuff like type size, alignement, can not select which assembly instruction to select to do an addition or a division (fixed vs. floating point arithmetic). This is why the compiler does not generate any binary code when it first meet templates. The code is generated only it is used with a concrete and known type. This is a huge problem. Now, if a library implementer wants to distribute his generic library, he must distribute source code and have the user compile it. For a language like C++, with no standard dependency management, it can be a massive turn off. Furthermore, it may not be reasonable for the user to have C++ compiling facilities when the target client is embedded devices with limited storage space. Indeed, C++ intermediary compilation artefacts tend to use a lot of disk space before it is linked into a smaller binary. What solution do we have then?

SWILENA [8, 103] is a Python bindings wrapper using Swig for the Olena C++ generic library. This wrapper will enumerate all the common use cases and implement a binding for them. The compiler will then generate binary code from the templated generic code for each use case enumerated in the wrapper. This way, we have given access into the dynamic world (Python) to generic code (C++ template). But it is still limited to the supported types. Each type a new combinaison of type needs to be supported from Python, it needs to be explicitly declared and compiled in the wrapper. Other image processing libraries, such as VIGRA [16], chose this solution.

VCSN [55] is a novel solution that essentially take the same base as SWILENA but goes beyond the boundary to implement a handmade facility that do system compiler calls to compile and link needed code on-the-fly when the binding does not exist. It then leverage the code hotloading feature to plug new dynamic libraries (.dll on windows and .so on linux) into the wrapper to provide the user its needed bindings.

Cython [42] attempt to solve the issue of the Python inherent language slowness due to its interpreted nature by providing a facility able to transpile a Python program into a C program so that a genuine C compiler (with extensions) is able to compile it and to link it against the Python/C API in order to achieve a huge performance gain at the cost of near zero knowledge of the complex Python/C API for the user. This novel solution essentially bypass the work of a JIT compiler (that would be used by a programming language using bytecode such as Java or C#) and just offload it onto well known/proven solution: the machine's C compiler.

Autowig [75], Cppyy [68] and Xeus-cling [96] are all solutions aiming to generate automatically Python bindings on-the-fly using different solutions. Autowig has in-house code based on LLVM/Clang to parse C++ code in order to generate and compile a Swig Python binding using the Mako templating engine. Cppyy will generate Python bindings but can also directly interpret C++ code from Python code thanks to being base on LLVM/Cling, a Clang-base C++ interpreter. Finally, Xeus-cling is a Jupyter [67] kernel allowing to directly interpret C++ into a Jupyter notebook. Like Cppyy, it is based on LLVM/Cling. Those three projects are very promising and improve greatly the scope of possibilities for the future.

Part II

Contribution

Chapter 3

Taxonomy of Images and Algorithms

In this thesis, we researched how to apply all those new generic facilities from the C++ language into the Image processing area. This allows us to test them in a practical way on our predilection area while remembering our past work, both success and failures in this matter. However, as we saw in the previous chapter 2, birthing concepts from code is something that is done in an emerging way. Henceforth, the first work will be to do an inventory of all existing image algorithms as well as an inventory of all image processing algorithms (both basic and more complex) we can think of. This way, we will notice behavior patterns emerging from similar image types or similar algorithms. First we will study how to extract behavioral pattern from an algorithm in order to refine it into one or multiple concepts. We will then study the set theory behind images types, their conjunctions, disjunctions and the implications. We will then study behavioral pattern related to algorithms, as well as their specific problematics related to heterogeneous computation. Finally, we will propose show the result of this emerging inventory in the form of a taxonomy indexing a framework of concepts about image processing.

3.1 Rewriting an algorithm to extract a concept

3.1.1 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.1

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 section 3.1.3). 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 chapter 4.

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?*"

3.1.2 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:

```
1  template <class InputImage, class OutputImage>
2  void dilate(const InputImage& input_ima, OutputImage& output_ima)
3  {
4    assert(input_ima.height() == output_ima.height()
5      && input_ima.width() == output_ima.width());
6
7    for (int x = 2; x < input_ima.width() - 2; ++x)
8      for (int y = 2; y < input_ima.height() - 2; ++y)
9      {
10       output_ima(x, y) = input_ima(x, y)
11       for (int i = x - 2; i <= x + 2; ++i)
12         for (int j = y - 2; j <= y + 2; ++j)
13           output_ima(x, y) = std::max(output_ima(x, y), input_ima(i, j));
14     }
15 }
```

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.

3.1.3 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 3.1 for the required type definitions and in table 3.2 for the required valid expressions.

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*

Table 3.1: Concepts formalization: definitions

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

Table 3.2: Concepts formalization: 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 [105, 107] (soon in the STL). Also, in practice, the *concepts* described here 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 [106]. 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 3.1.

```
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*/;
  };
```

Figure 3.1: Concepts in C++20 codes

3.2 Images types viewed as Sets: version & specialization

Achieving true genericity in a satisfactory way is a complex problem that has components of different levels. The first goal is to natively support as many sets of image type as possible. Natively means that there is no need for a conversion from one type to a supertype under the hood. The second step is to support an abstraction layer above the underlying data type for each pixel. Indeed, the structure of an image is decorrelated from the underlying data type. The third step is to write image processing algorithms for each set of image type. Fourthly, the performance trade-off shall be negligible if not null. Finally, the final step is to provide a high degree of friendliness for the end user. Ease of use is always to be considered.

Considering the available options to achieve our goal, the parametric polymorphism approach is the way to go. This allows the implementer to design image types and algorithms with behavior in mind. To illustrate this remark, let us consider the set of supported set of image types shown in figure 3.2.

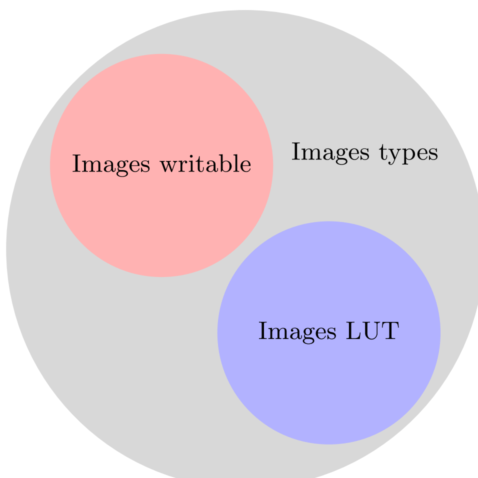


Figure 3.2: Set of supported image type.

To implement a basic image algorithm such as `fill` there really are two distinct ways of writing it. For the set of images type whose data type is encoded into each pixel, one must traverse the image and set each pixel's color to the new one. However, for the set of images type whose data type is encoded in a look-up table, one only has to traverse the look-up table to set each color to the new one. This translates into two distinct algorithms shown in fig. 3.3:

More generally, we consider that the set of image type is formed of several subset of image types. In the example there are two subsets: images whose pixel are writable and images whose data type are ordered in a look-up table.

$fill(I, v) \colon \forall p \in \mathcal{D}, I(p) = v$	$fill(I, v) \colon \forall i \in I.LUT, i = v$
(a) Writable image fill algorithm	(b) Image LUT fill algorithm

Figure 3.3: Comparison of implementation of the `fill` algorithm for two families of image type.

For each one of these subsets, if there is a way to implement an algorithm then we have a version *of this algorithm.*

Sometimes, it is possible to take advantage of a property on a particular image set, that may be correlated to an external data, to write the algorithm in a more efficient way. When those properties are linked to the types, this is call *specialization*. For instance, when considering a dilation algorithm, if the structuring element (typically the disc) is decomposable then we can branch on an algorithm taking advantage of this opportunity: decompose the dilation disc into small vectors and apply each one of them on the image through multiple passes. The speed-up comparing to a single pass with a large dilation disc is really significant (illustrated in 2.6). The code in 3.4 illustrate how an algorithm can be written to take advantage of the structuring element's decomposability property. The algorithm will first decompose the structuring element into smaller 1D periodic lines. It will then recursively call itself with those lines to do the multi-pass and thanks to known optimizations on periodic lines [3], it will be much faster.

```
template <Image Img, StructuringElement SE>
auto dilate(Img img, SE se) {
  if (se.is_decomposable()) {
    lst_small_se = se.decompose();
    for (auto small_se : lst_small_se)
      img = dilate(img, small_se) // Recursive call
    return img;
  } else if (is_pediodic_line(se))
    return fast_dilate1d(img, se) // Van Herk's algorithm;
  else
    return dilate_normal(img, se) // Classic algorithm;
}
```

Figure 3.4: Dilate algorithm with decomposable structuring element.

The figure 3.5 shows how an algorithm specialization may exists in a set of algorithms version. In this figure there exists a specialization of algorithms when it is known that the data buffer has the following property: its memory is contiguous. This implies that, for example, an algorithm like `fill` can be implemented using low level and fast primitives such as `memset` to increase its efficiency.

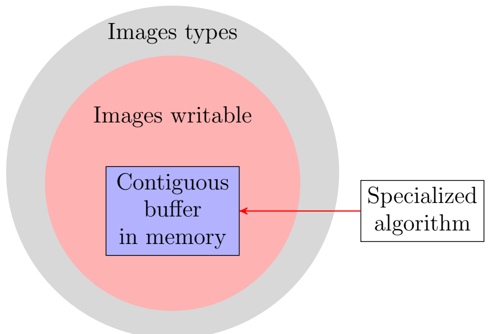


Figure 3.5: Algorithm specialization within a set.

There are more details that go in depth when considering the distinction between runtime dispatch (dynamic) and compile time dispatch (static) that we saw earlier in this manuscript in section 2.3.2.

3.3 Generic aspect of algorithm: canvas

Genericity is always referred to with this sentence "write once, work for every types, run everywhere". However, very quickly we learned that the *run* aspect can be a combination of:

- as fast as possible on a single CPU unit;
- as fast as possible on thanks to using many CPU units;
- as fast as possible on thanks to using many GPU units;
- as fast as possible on thanks to using many computers (cloud) and their CPU/GPU units.

How do we decide what is the most efficient way to do? There is no simple answer to this question but we can start by studying the morphology of algorithms with two goals in mind: (i) find what can be parallelized/distributed and (ii) find one or several algorithms abstraction. Indeed, in image processing there are a lot of common patterns when looking at algorithms, the most famous being *for all pixel of image, do something to pixel.* But there are other more high-level similarities that we can leverage to have more generic algorithm. Let us first study the different programmatic model there are to process an image.

The pipeline is the classic way of doing the work. It consists of an imbrication of different operators (algorithms) taking as input one or several image, maybe additional data as well (such as labels, adjacency map, etc.) in order to process the data “from left to right”. The result will show at the end of the pipeline and the optimization opportunities are located inside the smaller operators and in the form of correctly managing data (no useless copies, locality, etc.)

Kernels and tiling is the trendy way of the last decade. It consists in breaking the original image into small tiles in order to feed those tiles into a massively multicore GPU (via CUDA, Halide [58]). Processing will happen concurrently on those core, but it is costly to swap concents inside those GPU from the RAM memory. It's then preferred to design pipeline on those tiles directly to minimize the number of back and forth copies from the RAM.

Deep learning is a huge deal in image processing these last years as well. As deep neural networks are just a combinations of successive MapReduce [34], it can be offloaded onto onto executions units that can be cluster of computers through network (cloud), or local available computers, or directly localy available GPU hardware.

In every case there is a notion of pipeline where the user pipe algorithms into each others in order to achieve a result. Those algorithms can leverage all the heterogenous ressources (cloud, GPU, CPU) they can to *map* the input data. Algorithms will then finally aggregate the results (*reduce*) to output them into another algorithm, or save them, or display them. It is important to dissociate the route the data will go through and the processing pipeline logic. Both have their own specificities. In this thesis, we make a parallel, at small scale, between processing pipeline logic and image processing algorithms. First let us study two basic algorithm: dilation and erosion. The python code of such algorithm is naively be given in figure 3.6.

```python
def dilate(img, se, out):
  for pnt in zip(img.points()):
    out(pnt) = img(pnt)
    for nx in se(pnt):
      out(pnt) = \
        max(out(pnt), img(nx))
```

(a) Dilation

```python
def erode(img, se, out):
for pnt in zip(img.points()):
    out(pnt) = img(pnt)
    for nx in se(pnt):
      out(pnt) = \
        min(out(pnt), img(nx))
```

(b) Erosion

Figure 3.6: Dilate vs. Erode algorithms.

The algorithms are almost written the same way. The only change is the operation *min* and *max* when selecting the value to keep. As such, we can easily see a way to factorize code by passing the operator as an argument. The algorithms can then be rewritten as shown in figure 3.7.

```
def local_op(img, se, op, out):
  for pnt in zip(img.points()):
    out(pnt) = img(pnt)
    for nx in se(pnt):
      out(pnt) = op(out(pnt), img(nx))
```

(a) Local algorithm with custom operator

```
def dilate(img, se, out):
  local_op(img, se, max, out)
```

(b) Dilation (delegated)

```
def erode(img, se, out):
  local_op(img, se, min, out)
```

(c) Erosion (delegated)

Figure 3.7: New Dilate vs. Erode algorithms.

3.3.1 Taxonomy and canvas

This approach leads to question a way to classify algorithm in families where this factorization can be possible, more broadly. In essence there are three big families of algorithm when looking at the state of the art of image processing today. The first is the point-wise family. In essence those algorithms only need to know the current pixel to do the work. Those are the most basic algorithm. Some useful point-wise algorithms are: gamma correction, thresholding, contrast correction projection. The second family consists in all the local algorithm. To work they need to know a structuring element which is the window to consider around a pixel. Those algorithms introduce several very important notions: neighborhood (of a pixel), separability and decomposability (of a structuring element) and border management. Some useful local algorithms are: dilation, erosion, gradient, rank filter, median filter or hit or miss. Finally, the third family consists in all the algorithm that propagate their computation while traversing the image. The chamfer distance tranform is such an algorithm. Those algorithms are less friendly to factorization of code.

For the first family of algorithm, one can write them all with views so that factorizing code is hardly an issue. The second family of algorithm may be abstracted behind an algorithm canvas where the user provides the work to do at each point of the algorithm. For instance, a single pass local algorithm will always have shape given in 3.8:

This canvas can be customized to do a specific job, especially at the lines 2, 4, 6, 7 and 8. The user would then provide callbacks and the canvas would

```
def local_canvas(img, out, se):
  # do something before outer loop
  for pnt in img.points():
    # do something before inner loop
    for nx in se(pnt):
      # do something inner loop
    # do something after inner loop
  # do something after outer loop
```

Figure 3.8: Local algorithm canvas.

do the job. This is especially useful when knowing that the canvas would handle the border management (the user would provide a handling strategy like mirroring the image or filling it with a value). The canvas would also take advantage of optimization opportunities (such as the decomposability of a structuring element) that the user would probably forget, or not know, when first writing his local algorithm. Another advantage is the opportunity to do more complex optimization such as parallelizing the execution or offloading part of the calculation on a GPU. More generally, all optimization done through heterogeneous computing would be available by default even if the user is not an area expert.

Despite all these advantages, one big disadvantage is the readability of the algorithm user-side. For instance, the dilation algorithm is rewritten in figure 3.9.

```
def dilate(img, out, se):
  do_nothing = lambda *args, **kwargs: None

  def before_inner_loop(img, out, pnt):
    out(pnt) = img(pnt)

  def inner_loop(ipix, opix, nx):
    out(pnt) = max(out(pnt), img(nx))

  local_canvas(img, out, se,
    before_outer_loop = do_nothing,
    before_inner_loop = before_inner_loop,
    inner_loop        = inner_loop,
    after_inner_loop  = do_nothing,
    after_outer_loop  = do_nothing
  )
```

Figure 3.9: Local algorithm canvas.

This way of thinking algorithms is far less readable than the classic way. The user does not see the loops happening and it can becomes very messy when several passes are happening (closing, opening, hit or miss, etc.)

3.3.2 Heterogeneous computing: a partial solution, canvas

One of the key aspect driving genericity is performance. We still have the following mantra: "write once, work for every types, run everywhere". However when considering the *run* aspect, one has a lot to do. Indeed, nowadays, exploiting the available resources to their maximum is long standing issue. There are many ongoing work on the subject, such as SyCL [79, 78], Boost.SIMD [59] or even VCL [56]. After taking some distance to study the subject, we can infer that there are three main aspects to consider when optimizing performance.

The first one, the most important one is the algorithm to use in function of certain set of data. This aspect is covered by the C++ language and its builtin genericity tool: template metaprogramming. Indeed, we select the most optimized algorithm for a particular set of data.

The second one is the ability for the code to be understood by the compiler so that is is further optimized during the generation of the binary. Indeed, when compiling for the native architecture of a recent processor, one can use the most recent assembly instructions to use wide vectorized registries (AVX512). The use of a recent compiler also brings the help much needed.

Finally, the third aspect is not as trivial as the first two ones. It consists in studying the structure of an algorithm to allow distributed computation. Sometimes algorithm are friendly to be distributed on several processing unit that compute a part of the result concurrently. This is what we call parallelism. There exists several way to take advantage of parallelism. First there is the use of several CPU unit on the host computer. Then there is the use of GPU units working in combination with the CPU units to take advantage of the massive amount of core a graphic card can provide. Finally there is the use of cloud computing which consists in using several "virtual" computers, each of them offering of CPU and GPU units in order to compute a result. One should be aware that each time we introduce a new layer of abstraction, there is a cost to orchestrate the computation, send the input data and retrieve the results. It is thus very important to study case by case what is needed. Some solutions exists that abstract away completely the hardware through a DSL [1] such as Halide [58]: the DSL compiler's job will be to try very hard to make the most out of both the available (or targeted) hardware and the code. Those solutions are not satisfactory for us as we want to avoid DSL and remain at code level. We are not developing a compiler: we are working with it.

There is one true issue when studying parallel algorithm: it is wether they can be parallelized or not. Not all algorithms can be parallelized. Some just intrinsically cannot, typically, algorithms that immediately need the result at the previous iteration to compute the next iteration. There are still way to parallelize those one but it is not trivial and will not be treated in this paper. What interest us are the algorithms whose structure is an accumulation over a data type that can be defined as a monoid. We assert that every algorithm that can be rewritten as an accumulation over a monoid can be parallelized and/or distributed. This model that consists in distributing computation like an accumulation over a monoid data structure is also call the map-reduce. This model has two steps: the distribution (map) and then the accumulation (reduce).

The map step will dispatch computation on sub-units with small set of data. The reduce step will retrieve and accumulate all those resulting data, as soon as they are ready.

The accumulation algorithm has this form:

```
1 template <class In, class T, class Op>
2 auto accumulate(In input, T init, Op op)
3 {
4   for(auto e : input)
5   {
6     init = op(init, e);
7   }
8   return init;
9 }
```

The loop line 4 can be split into several calculation units which are going to be distributed, and then be accumulated later once the units have finished their computation.

The issue left here is the monoid. What exactly is a monoid here? A monoid is a data structure which operates over a set of values, finite or infinite. This data structure must provide a binary operation which is closed and associative. Finally, this data structure must also provide a neutral element (aka the identity). Some trivial monoids comes to mind:

- boolean. For binary operation "and", identity is "true" whereas for binary operation "or", identity is "false".

[1] Domain Specific Language

- integer. For binary operation "-" and "+", identity is "0" whereas for binary operation "*" and "/", identity is "1".
- string. For concatenation, identity is empty string.
- optional value (also known as monadic structure in haskell programming language).

There are many more monoids, less trivial but very handy, such as the unsigned integer/max/0 set and the signed integer/min/global max set.

This theory is extremely benefic to image processing as the most commonly used algorithms, the local algorithms, can all be written in the form of an accumulation over the pixels of an image. The fact that finding an identity for the operation processed by the algorithm is often quite trivial led us to the idea of canvas. A canvas is a standard way to write an iteration over an image which abstracts the underlying data structure. A canvas is a tool for the user to provide its computation model based on events such as: "entering inner loop" or "exiting inner loop". The user can then provide its operations as if he was writing his algorithm himself (restricted to the accumulation model). As the maintainer of the library provides the canvas of execution, he can know also make change to take advantage of it. For instance, computing a CUDA kernel at one point and dispatching it on GPU units is totally within scope and transparent for the user of the library. Although there is a caveat: rewriting our algorithm in an accumulate form and chunking it in fragments to feed to the canvas is definitely not intuitive. Indeed, we require our user to change his way of thinking from the procedural paradigm to the event-driven paradigm. This approach is not new and is used in other libraries such as Boost.Graph [18] for similar purposes. Dean talks about this recurring monoid pattern more in-depth in his talk [80].

In image processing, we quickly come to identify *local* algorithms, that reason about a group of pixel around a given of coordinate. All those algorithms can be abstracted behind an accumulation of some sort and they all have the same morphology. Thus leading to the following abstraction:

```
template <class In, class Out, class SE, class T, class Op>
auto local_accumulate(In input, Out output, SE se, T init, Op op)
{
    auto zipped_imgs = ranges::view::zip(input.pixels(), output.pixels())
                                                // (1)
    for(auto&& rows : ranges::rows(zipped_imgs))
    for(auto [px_in, px_out] : rows)
    {
        auto v = op(init, px_in.val());         // (2)
        for(auto nb : se(px_in))
```

```
      v = op(v, nb.val());                        // (3)
      px_out.val() = v;                           // (4)
    }
                                                  // (5)
}
```

From this code we can deduce some very useful and easy monoids by the following triplets:

- $(type = boolean, operator = and, neutral = true)$ is a binary erosion
- $(type = boolean, operator = or, neutral = false)$ is a binary dilation
- $(type = unsignedinteger, operator = max, neutral = 0)$ is a dilation
- $(type = signedinteger, operator = min, neutral = globalmax)$ is an erosion

Now if we want to rewrite the `local_accumulate` in an event driven paradigme, we need to identify the different callbacks to expose our user on the call site. Especially, what will be of the callback parameters. There are five callback event we have identified:

1. before entering outer loop (no work is done)
2. before entering inner loop (iteration over the pixel's neighbor)
3. inner loop (actual operation to perform, result is accumulated)
4. after exiting inner loop (iteration over the neighbor is over, what to do with the accumulated result?)
5. after exiting outer loop (iteration over the image is over)

```
template <class In, class Out, class SE, class T, class BeforeOuterLoopCB,
          class BeforeInnerLoopCB, class InnerLoopCB class AfterInnerLoopCB,
          class AfterOuterLoopCB>
auto local_accumulate(In input, Out output, SE se, T init,
                      BeforeOuterLoopCB bolCB, BeforeInnerLoopCB bilCB,
                      InnerLoopCB ilCB, AfterInnerLoopCB ailCB,
                      AfterOuterLoopCB aolCB)
{
  auto zipped_imgs = ranges::view::zip(input.pixels(), output.pixels()
  bolCB(input, output);                           // (1)
  for(auto&& row : ranges::rows(zipped_imgs))
    for(auto [px_in, px_out] : rows)
    {
      bilCB(px_out.val(), init, px_in.val())      // (2)
      for(auto nb : se(px_in))
        ilCB(px_out.val(), px_out.val(), nb.val())  // (3)
```

```
    ailCB(px_out.val(), init, px_in.val())       // (4)
  }
  aolCB(input, output)                           // (5)
}
```

In this code, we can see that all the callbacks do not take the same type and/or number of parameters. Here is what the call site would like if the user wants to perform a dilation:

```
local_accumulation(
  input,                                        // input image
  output,                                       // output image
  se,                                           // structuring element
  0,                                            // monoid's neutral element
  [](auto I, auto& O) { /* do nothing */ }, // (1) entering outer loop callback
  [](auto& o, auto init, auto in){          // (2) entering inner loop callback:
    o = std::max(init, in);                 // initialize with neutral element
  },
  [](auto& o, auto cur, auto nbh) {         // (3) inner loop callback:
    o = std::max(cur, nbh);                 // keep the local maximum
  },
  [](auto& o, auto init, auto in) {         // (4) exiting inner loop callback
    /* do nothing */
  },
  [](auto I, auto& O) { /* do nothing */ }  // (5) exiting outer loop callback
);
```

It is very verbose and non-intuitive but hopefully, once the compiler optimize out the empty callbacks, the generated code is as fast as a non-generic handwritten dilation.

3.4 Library concepts: listing and explanation

Let us now delve into concepts related to the Image Processing area. Indeed, this domain has his specificities and we want to improve generic image processing library design by learning from our past experiments and working with new techniques. The most basic usage of an image is the famous algebraic formula $y = f(x)$ where y is a *value* generated by the *image* f for the *emph* x. Aside from generating a value, an image can also *store* a value, as in $f(x) = y$ where the value is *assigned* to the image for a given point. Those notions are the basis of our work and will drive the entire design.

3.4.1 The fundamentals

First, let us introduce the fundamental concepts deriving from the basis notion. The *Value* concept is refined into three distincts one in section 3.4.1.

Concept	Modeling type	Inherit behavior from	Instance of type
Value	`Val`	∅	`val`
ComparableValue	`CmpVal`	Value	`cmp_val1, cmp_val2`
OrderedValue	`OrdVal`	ComparableValue	`ord_val1, ord_val2`

Concept	Expression	Return Type	Description
Value	`std::semiregular<Val>`	`std::true_type`	`Val` is a semiregular type. It can be: copied, moved, swapped, and default constructed.
ComparableValue	`std::regular<CmpVal>`	`std::true_type`	`CmpVal` is a regular type. It is a semiregular type that is equality comparable.
	`cmp_val1 == cmp_val2`	`boolean`	Supports equality comparison
OrderedValue	`std::totally_ordered<OrdVal>`	`std::true_type`	`CmpVal` is a totally ordered as well as a regular type. Additionally the expressions must be equality preserving.
	`ord_val1 < ord_val2` `ord_val1 <= ord_val2, ...`	`boolean` `boolean`	Supports inequality comparisons

Table 3.3: Concepts Value: expressions

There are the basic *Value* but also the *ComparableValue* and the *OrderedValue* which are useful when it comes to comparison or ordering algorithms.

Then we have the concept of *Point*, detailed in section 3.4.1 which is a bit more constrained as it must be totally ordered. Indeed, when accessing a value stored in an image, wether it be reading of mutating, it is important that there is only one accessed value.

Concept	Modeling type	Inherit behavior from	Instance of type
Point	`Pnt`	∅	`pnt1, pnt2`

Concept	Expression	Return Type	Description
Point	`std::regular<Pnt>`	`std::true_type`	`Pnt` is a regular type. It can be: copied, moved, swapped, and default constructed. It also is equality comparable.
	`std::totally_ordered<OrdVal>`	`std::true_type`	`Pnt` is a totally ordered as well as a regular type. Additionally the expressions must be equality preserving.
	`pnt1 < pnt2` `pnt1 <= pnt2, ...`	`boolean` `boolean`	supports inequality comparisons

Table 3.4: Concepts Point: expressions

Now we introduce an abstract way to represent this relation $Value *$ *Point*: the Pixel. This is a well known notion in image processing and it represent a couple $(point, value)$. This facility is easy to move around and contains facilities to read and mutate the pixel's value if possible. Indeed, not all pixel are able to mutate their value. If the pixel is yielded by an image that only generate values on the fly then it cannot be mutated. Henceforth, we introduce two new concepts: *Pixel* and *OutputPixel* in section 3.4.1.

Concept	Modeling type	Inherit behavior from	Instance of type
Pixel	`Pix`	∅	`pix`
OutputPixel	`OPix`	Pixel	`opix`

Concept	Definition	Description	Requirement
Pixel	`value_type`	Type of the value contained in the pixel. Cannot be constant or reference.	Models the concept `Value`.
	`reference_type`	Type used to mutate the pixel's value if non-const. Can be a proxy.	Models the concept `std::indirectly_writable` if non-const.
	`point_type`	Type of the pixel's point.	Models the concept `Point`

Table 3.5: Concepts Pixel: definitions

Those two concepts have a very similar interface described in section 3.4.1. They can both access the stored informations: the point and the value. On top of that, the *OutputPixel* can mutate the value.

Type	Instance of type
`Pix::value_type`	`val`
`Pix::point_type`	`pnt`

Concept	Expression	Return Type	Description
Pixel	`pix.val()`	`Pix::reference_type`	Access the pixel's value for read and/or write purpose.
	`pix.point()`	`Pix::point_type`	Read the pixel's point.
	`pix.shift(pnt)`	`void`	Shift pixel's point coordinate base on `pnt`'s coordinates.
OutputPixel	`opix.val() = val`	`void`	Mutate pixel's value.

Table 3.6: Concepts Pixel: expressions

Now we need a helper concept: the ranges. Ranges [99, 77, 107, 105] are a set of concepts defined in the C++ standard library shipped with the ISO C++20 norm in 2020 [87]. They allow the user to abstract away iterators to only iterate over one object: the range. This allow the user to migrate his source code from:

```
template <class IteratorBegin, class IteratorEnd>
void my_algorithm(IteratorBegin beg, IteratorEnd end)
{
  for(; beg != end; ++beg)
    // ...
}
```

To:

```
template <class Range>
void my_algorithm(Range rng)
```

```
{
  for(auto e : rng)
    // ...
}
```

In image processing, we refine further this concept by introducing multi-dimensional ranges (*MDRange*). Indeed, in image processing the user is used to write double loop to iterate over a bi-dimensional image. And abstracting away this aspect under standard ranges induce performance loss. That is why we needed this concept to exist. A multi-dimentional range can be split with a library function, `mln::ranges::rows(mdrng)` to fit the double-loop pattern and keep its performance. This topic is tackle in-depth later in **??**. For now, let us consider multi-dimensional ranges as an image processing extension for performance. They are defined in section 3.4.1 and their interface is the same as standard ranges, as seen in section 3.4.1.

Concept	Modeling type	Inherit behavior from	Instance of type
MDRange	`MDRng`	∅	`mdrng`
OutputMDRange	`OMDRng`	MDRange	`omdrng`
ReversibleMDRange	`RMDRng`	MDRange	`rmdrng`

Concept	Definition	Description	Requirement
MDRange	`value_type`	Type of the value contained in the range. Cannot be constant or reference.	Models the concept `Value`.
	`reference_type`	Type used to mutate the pixel's value if non-const. Can be a proxy.	Models the concept `std::indirectly_writable` if non-const.

Table 3.7: Concepts Ranges: definitions

Type	Instance of type
`std::ranges::range_value_t<MDRng>`	`val`

Concept	Expression	Return Type	Description
MDRange	`mdrng.begin()`	`unspecified`	Return a forward iterator allowing a traversing of the range.
	`mdrng.end()`	`unspecified`	Return a sentinel allowing to know when the end is reached.
OutputMDRange	`auto it = omdrng.begin()` `*it++ = val`	`void`	Mutate a value inside the range then increment the iterator's position
ReversibleMDRange	`rmdrng.rbegin()`	`unspecified`	Return a forward iterator allowing a traversing of the range starting from the end.
	`rmdrng.rend()`	`unspecified`	Return a sentinel allowing to know when the end is reached.

Table 3.8: Concepts Ranges: expressions

From an algebraic point of view, the definition of an image is not complete without considering a definition domain on which it is defined. In image processing, the same rule applies. We cannot consider an image without considering the set of points that are valid for this image. Henceforth we must define the concept of *Domain* in *table:concept.domain.definitions*.

Concept	Modeling type	Inherit behavior from	Instance of type
Domain	`MDRng`	MDRange	`dom`
SizedDomain	`OMDRng`	Domain	`sdom`
ShapedDomain	`RMDRng`	SizedDomain	`shdom`

Table 3.9: Concepts Domain: definitions

The *Domain* concept is refined into two sub-concepts which are *SizedDomain* and *ShapedDomain*. This emphasis the possibility of existence of possible infinite domain and domains that may be defined over non-continuous intervals in space. This allows algorithms to requires the domain to have certain shape if needed. The domain behavior is described in section 3.4.1

Type	Instance of type
`Dom::value_type`	`pnt`

Concept	Expression	Return Type	Description
Domain	`Point<Dom::value_type>`	`std::true_type`	Domain's value models the `Point` concept
	`dom.has(pnt)`	`bool`	Check if a points is included in the domain.
	`dom.empty()`	`void`	Read the pixel's point.
	`dom.dim()`	`void`	Returns the domain's dimension.
SizedDomain	`sdom.size()`	`unsigned int`	Returns the number of points inside the domain.
ShapedDomain	`shdom.extends()`	`std::forward_range`	Return a range that yields the number of elements for each dimension.

Table 3.10: Concepts Domain: expressions

Now we have all the tools to introduce our main concept: *Image*. As for *Pixel*, we have the distinction over image whose value can be mutated in a sub-concept named *WritableImage*. These concepts are defined in table 3.11.

In addition to this definition we can infer the behavior described in section 3.4.1. There are complicated requirements written in template metaprogramming code. But in the end it is just to requires the ranges returned by the member functions `pixels()` and `values()` to iterate over element whose type are the same as those declared in the parent image.

In addition, we introduce two facilities which are the member function `concretize()` and `ch_value<V>()`. The first is a way to turn a view into a concrete type. This will be seen more in-depth in chapter 4. The last is

Concept	Modeling type	Inherit behavior from	Instance of type
Image (InputImage, ForwardImage)	`Img`	∅	`img`
WritableImage	`WImg`	Image	`wimg`

Concept	Definition	Description	Requirement
	`pixel_type`	Type of the image's pixel.	Models the concept `Pixel`.
	`point_type`	Type of the image's point.	Models the concept `Point`.
	`value_type`	Type of the image's value. Cannot be constant or reference	Models the concept `Value`.
	`domain_type`	Type of the image's domain.	Models the concept `Domain`.
	`reference`	Type used to mutate an image pixel's value if non-const	Models the concept `std::indirectly_writable` if non-const.
	`concrete_type`	Image concrete type (that holds data).	Models the concept `Image`.
	`ch_value_type<V>`	Facility to return a new image type. that casts the underlying `value_type` into `V`	Models the concept `Image`.

Table 3.11: Concepts Image: definitions (1)

Concept	Expression	Return Type	Description
Image	`img.concretize()`	`std::convertible_to< concrete_type>`	Return a concrete image that holds data.
	`img.ch_value<V>()`	`std::convertible_to< ch_value_type<V> >`	Return an image whose values are casted to `V`.
	`img.domain()`	`std::convertible_to< domain_type>`	Return the image's domain.
	`img.pixels()`	`MDRange`	Return a range that yields all the image pixels.
	`img.values()`		Return a range that yields all the image values.
	`std::convertible_to< std::ranges::ranges_value_t< decltype(img.pixels())>, pixel_type>` `std::convertible_to< std::ranges::ranges_value_t< decltype(img.values())>, value_type>`	`std::true_type`	Ranges converts to compatible element types.
WritableImage	`wimg.values()`	`OutputMDRange`	Return a range that yields all the image values (mutable).
	`OutputPixel< std::ranges::ranges_value_t< decltype(wimg.pixels())> >`	`std::true_type`	Ranges whose elements are mutable.

Table 3.12: Concepts Image: expressions (1)

Image

a way to cast values from one type to another. It forms a new image type whose underlying values will be returned after being casted to a new value type.

3.4.2 Advanced way to access image data

Being able to iterate over ranges of pixels or values is good and all but we are still lacking fundamental facilities to access an element directly from the image. First we need to define the concept of *Index* in section 3.4.2 which we will use afterwards.

Concept	Modeling type	Inherit behavior from	Instance of type
Index	`Idx`	∅	`idx, idy`

Concept	Expression	Return Type	Description
Index	`std::signed_integral<Idx>`	`std::true_type`	Idx is a signed integral arithmetic type
	`idx + idy, idx - idy, ...`	`Idx`	Supports all trivial arithmetical operations

Table 3.13: Concepts Index: expressions

The first fundamental tool is represented as an *IndexableImage*. An element can be accessed simply by providing its index number. This concept is defined in table 3.14.

Concept	Modeling type	Inherit behavior from	Instance of type
IndexableImage	`IdxImg`	Image	`idximg`
WritableIndexableImage	`WIdxImg`	IndexableImage, WritableImage	`widximg`

Concept	Definition	Description	Requirement
IndexableImage	`index_type`	Type of the image's buffer index.	Models the concept `Index`.

Table 3.14: Concepts Image: definitions (2)

This introduces a simple behavioral pattern described in section 3.4.2.

Additionally we want to be able to access a value by providing a point, the same way as in the algebraic definition $val = image(point)$. To do so, we introduce the concept of accessibility through *AccessibleImage*. This concept is defined in table 3.16.

This introduces new behavior that is described in section 3.4.2. We can notice facilities specifically including bound checking. Indeed, we suppose, for fast access, that the user is always picking element from the image's domain but it is possible to bound check elements if needed on access for specific usages.

Type	Instance of type
Img::value_type	val
IdxImg::index_type	k

Concept	Expression	Return Type	Description
IndexableImage	idximg[k]	std::same_as<reference>	Access a value at a given index.
Writable IndexableImage	widximg[k] = val	void	Mutate a value at a given index.

Table 3.15: Concepts Image: expressions (2)

Concept	Modeling type	Inherit behavior from	Instance of type
AccessibleImage	AccImg	Image	accimg
WritableAccessibleImage	WAccImg	AccessibleImage, WritableImage	waccimg

Table 3.16: Concepts Image: definitions (3)

Type	Instance of type
Img::point_type	pnt

Concept	Expression	Return Type	Description
AccessibleImage	accimg(pnt)	std::same_as<reference>	Access a value for a given point.
	accimg.at(pnt)	std::same_as<reference>	Access a value for a given point. No bound checking.
	accimg.pixel(pnt)	std::same_as<pixel_type>	Access a pixel for a given point.
	accimg.pixel_at(pnt)	std::same_as<pixel_type>	Access a pixel for a given point. No bound checking.
Writable AccessibleImage	img(pnt) = val	void	Mutate a value at a given point.
	waccimg.at(pnt) = val	void	Mutate a value at a given point. No bound checking.
	OutputPixel<decltype(waccimg.pixel(pnt))>	std::true_type	The returned pixel models OutputPixel.
	OutputPixel<decltype(waccimg.pixel_at(pnt))>	std::true_type	The returned pixel models OutputPixel. No bound checking.

Table 3.17: Concepts Image: expressions (3)

Once we know that an image is both *indexable* and *accessible* we can deduce new behaviors (described in section 3.4.2) that we put behind the concept of *IndexableAndAccessibleImage* defined in *table:concept.image.definitions.4*. This behavior is related to accessing index from points and vise versa.

Concept	Modeling type	Inherit behavior from	Instance of type
IndexableAnd AccessibleImage	`IdxAccImg`	IndexableImage, AccessibleImage	`idxaccimg`
WritableIndexable AndAccessibleImage	`WIdxAccImg`	IndexableAndAccessibleImage, WritableIndexableImage, WritableAccessibleImage	`widxaccimg`

Table 3.18: Concepts Image: definitions (4)

Concept	Expression	Return Type	Description
IndexableAnd AccessibleImage	`img.point_at_index(k)`	`point_type`	Get the point corresponding to the given index.
	`idxaccimg.index_of_point(pnt)`	`index_type`	Get the linear index (offset in the buffer) of multi-dimensional point.
	`idxaccimg.delta_index(pnt)`	`index_type`	Get the linear index offset for the given point.

Table 3.19: Concepts Image: expressions (4)

Additionally we assume that an image can be traversed in a forward way as well as in a backward way. This may not be true for all images so this notion needs to be refined into a new concept *BidirectionalImage*. This concept is defined in table 3.20 and its behavior is described in section 3.4.2.

Concept	Modeling type	Inherit behavior from	Instance of type
BidirectionalImage	`BidirImg`	Image	`bidirimg`
WritableBidirectionalImage	`WBidirImg`	BidirectionalImage, WritableImage	`wbidirimg`

Table 3.20: Concepts Image: definitions (5)

Now we need a way, when possible, to iterate over a continuous data buffer for very fast and optimized calculation. That is what the concept of *RawImage* is for: an image whose data buffer can be accessed, as well as its mutable counterpart. It is defined defined in table 3.22.

Having a raw image whose data buffer can be accessed allow ust to expose two more member function to access the data buffer and its strides for correct pointer arithmetic. They behave as described in section 3.4.2.

Type	Instance of type
`Img::point_type`	`pnt`
`Img::value_type`	`val`
`IdxImg::index_type`	`k`
`int`	`dim`

Concept	Expression	Return Type	Description
BidirectionalImage	`bidirimg.pixels()`	`ReversibleMDRange`	Return a reversible range that yields all the image pixels.
	`bidirimg.values()`		Return a reversible range that yields all the image values.

Table 3.21: Concepts Image: expressions (5)

Concept	Modeling type	Inherit behavior from	Instance of type
RawImage	`RawImg`	IndexableAndAccessibleImage, BidirectionalImage	`rawimg`
WritableRawImage	`WRawImg`	RawImage, WritableIndexableImage, WritableBidirectionalImage	`wrawimg`

Table 3.22: Concepts Image: definitions (6)

Concept	Expression	Return Type	Description
RawImage	`rawimg.data()`	`std::convertible_to< const value_type*>`	Get a constant pointer to the first element of the domain.
	`rawimg.stride(dim)`	`std::ptrdiff_t`	Get the stride (in number of elements) between two consecutive elements in the given `dim`.
Writable RawImage	`img.data()`	`std::convertible_to< value_type*>`	Get a pointer to the first element of the domain.
	`*(wrawimg.data() + k) = val`	`void`	Access an element from the data buffer at index `k` and mutate it to `val`.
WithExtension Image	`wextimg.extension()`	`std::convertible_to< extension_type>`	Get the extension of the image.

Table 3.23: Concepts Image: expressions (6)

3.4.3 Local algorithm concepts: structuring elements and extensions

From the beginning concepts are emerging from behavioral patterns extracted from algorithms. In image processing, there is a family of algorithms called the *local algorithms*. They work by considering a specific pixel as well as all the pixels among a window having a specific shape centered in this first pixel. The window is called the *structuring element* and the pixels considered by this window are called the *neighborhood*. This leads us to introduce the concept of *StructuringElement* which is defined in table 3.24.

Concept	Modeling type	Inherit behavior from	Instance of type
StructuringElement	`SE, Pnt`	$\varnothing$, Point	`se, pnt`
DecomposableStructuringElement	`DSE, Pnt`	StructuringElement, Point	`dse`
SeparableStructuringElement	`SSE, Pnt`	StructuringElement, Point	`sse`
IncrementalStructuringElement	`ISE, Pnt`	StructuringElement, Point	`ise`

Concept	Definition	Description	Requirement

StructuringElement	`incremental`
	`decomposable`
	`separable`
		`std::bool_constant`	`std::true_type` if supported
			`std::false_false` otherwise.

Table 3.24: Concepts Structuring Elements: definitions

This concept is refined into three subconcepts that are related to properties the structuring element can offer. Those properties are:

- decomposability: ability to split a complex structuring element into several smaller and simpler structuring element. There is an equivalence in behavior when the algorithm is recursively run for each smaller structuring element one after another, in a multipass way.

- separability: ability to split a complex structuring element into several smaller and simpler structuring element. There is an equivalence in behavior when the convolution is recursively run for each smaller structuring element one after another, in a precise order, in a multipass way.

- incremental: ability to tell the points that are added to or removed from the range when the structuring element is shifted by a basic displacement (e.g. for a *2D point*, the basic deplacement is $(0, 1)$). Usually used to compute attributes over a sliding structuring element in linear time.

Type	Instance of type	Requirement
Pix	pix	std::same_as<Pix::point_type, Pnt>
Pnt	pnt	

Concept	Expression	Return Type	Description
StructuringElement	std::regular_invocable<SE, Pnt> std::regular_invocable<SE, Pix>	std::true_type	se can called using std::invoke, is equality preserving and does not modify function object nor arguments.
	se(pnt)	std::forward_range	Return a range that yeilds the neighboring points of pnt.
	se.offsets()	std::forward_range	Return a range that yeilds the neighboring points, in relative coordinates.
	se(pix)	std::forward_range	Return a range that yeilds the neighboring pixels of pix.
	se.radial_extent()	int	Returns the radial extent of the SE, the radius of the disc (square).
	std::convertible_to< std::ranges::range_value_t< decltype(se(pnt))>, Pnt> std::convertible_to< std::ranges::range_value_t< decltype(se.offsets())>, Pnt>	std::true_type	Converts to a compatible point type
	Pixel<std::ranges::range_value_t< decltype(se(pix))> >		Is a range of compatible pixel type
Decomposable StructuringElement	dse.is_decomposable()	std::bool_constant	std::true_type if supported std::false_false otherwise.
	dse.decompose()	std::forward_range	Return a range that yeilds simpler structuring elements.
	StructuringElement< std::ranges::range_value_t< decltype(dse.decompose())> >	std::true_type	Is a range of compatible structuring elements types.
	dse.is_decomposable()	bool	Wether the decompose facility is supported
Separable StructuringElement	dse.is_decomposable()	std::bool_constant	std::true_type if supported std::false_false otherwise.
	sse.separate()	std::forward_range	Return a range that yeilds simpler structuring elements.
	StructuringElement< std::ranges::range_value_t< decltype(sse.separate())> >	std::true_type	Is a range of compatible structuring elements types.
	sse.is_separable()	bool	Wether the separate facility is supported
Incremental StructuringElement	dse.is_decomposable()	std::bool_constant	std::true_type if supported std::false_false otherwise.
	ise.inc()	StructuringElement< SE, Pnt>	Return the next simpler structuring elements.
	ise.dec()	StructuringElement< SE, Pnt>	Return the previous simpler structuring elements.
	ise.is_incremental()	bool	Wether the incremental facility is supported

Table 3.25: Concepts Structuring Elements: expressions

The behavioral for those concepts is discribed in section 3.4.3.

Additionally, we introduce the concept of *Neighborhood* in table 3.26. This concept has facilities to know what points/pixels are placed before or after another point/pixel inside the window of a specific structuring element. It behaves as described in section 3.4.3.

Concept	Modeling type	Inherit behavior from	Instance of type
Neighborhood	SE, Pnt	StructuringElement, Point	se, pnt

Table 3.26: Concepts Neighborhood: definitions

Type	Instance of type	Requirement
Pix	pix	std::same_as<Pix::point_type, Pnt>
Pnt	pnt	

Concept	Expression	Return Type	Description
Neighborhood	se.before(pnt)	std::forward_range	Return a range that yields the points before pnt.
	se.after(pnt)		Return a range that yields the points after pnt.
	se.before(pix)		Return a range that yields the pixels before pix.
	se.after(pix)		Return a range that yields the pixels after pnt.
	std::convertible_to< std::ranges::ranges_value_t< decltype(se.before(pnt))>, Pnt> std::convertible_to< std::ranges::ranges_value_t< decltype(se.after(pnt))>, Pnt>	std::true_type	Ranges converts to compatible element types.
	Pixel< std::ranges::ranges_value_t< decltype(se.before(pix))>, Pix> Pixel< std::ranges::ranges_value_t< decltype(se.after(pix))>, Pix>		Ranges that are of compatible element types.

Table 3.27: Concepts Neighborhood: expressions

And the last side concept to introduce to our current galaxy is the extension. Indeed, extension management is very important when dealing with local algorithm as pixels on the border need to be processed and the behavior near the border of the image must be defined and well-specified. There are several strategies when it comes to borders and extension. We refine concept for each strategy we identified:

- fillable: fill the border with a specific value.
- mirrorable: mirror the image as if there was an axial symmetry, with the border being the axis.

- periodizeable: repeat the image, as if a modulo size wasa applied to the coordinates.
- clampable: extend the value at the image's border into the extension.
- extent with: used when tiling. it consider the current image as a subimage of another bigger image and pick the extension values there.

Those concepts are defined in table 3.28 and their behavior is described in section 3.4.3.

Concept	Modeling type	Inherit behavior from	Instance of type
Extension	`Ext, Pnt`	∅, Point	`ext, pnt`
FillableExtension	`FExt, Pnt`	Extension, Point	`fext`
MirrorableExtension	`MExt, Pnt`	Extension, Point	`mext`
PeriodizableExtension	`PExt, Pnt`	Extension, Point	`pext`
ClampableExtension	`CExt, Pnt`	Extension, Point	`cext`
ExtendWithExtension	`EwExt, Pnt, U`	Extension, Point, Image	`ewext, u`

Concept	Definition	Description	Requirement
Extension	`value_type`	Type of value contained in the range. Cannot be constant or reference.	Models the concept `Value`.
	`support_fill` `support_mirror` `support_periodize` `support_clamp` `support_extend_with`	`std::bool_constant`	`std::true_type` if supported `std::false_false` otherwise.
ExtendWithExtension	`point_type`	Type of point in the extended image.	Converts to the sub-image points type.

Table 3.28: Concepts Extensions: definitions

This allows us to introduce the final refined image concepts: *WithExtensionImage* as well as *ConcreteImage* and *ViewImage*. Those two last will be seen in detail in the next chapter 4. Those concepts are defined in table 3.30. Their behavior is described in section 3.4.3.

Finally, we introduce a helper concept to centralize the detection of the "writability" of an image. Indeed, we do not want the user to have to use the writable counterpart of each concept for each and every case. That is why we introduce this final concept, *OutputImage* in section 3.4.3, that will tell the user if an image is well-specified.

The correct way to use it is:

```
template <class Img>
requires RawImage<Img> && OutputImage<Img>
void my_algorithm(Img img) {
    // ...
  }
```

Type	Instance of type
`SE<Pnt, Pix>`	`se`
`Ext::value_type`	`val`
`Ext::point_type`	`offset`

Concept	Expression	Return Type	Description
Extension	`ext.fit(se)`	`bool`	Wether the extension fit the structuring element.
	`ext.extent()`	`int`	Extension's border width. std::numeric_limits<int>::max() for infinite size.
FillableExtension	`fext.fill(val)`	`void`	Fill the extension with the value `val`.
	`fext.is_fill_supported`	`bool`	Wether the fill facility is supported.
MirrorableExtension	`mext.mirror()`	`void`	Fill the extension with mirrored image's values.
	`mext.is_mirror_supported()`	`bool`	Wether the mirror facility is supported.
PeriodizeableExtension	`mext.periodize()`	`void`	Fill the extension with periodic copies of image's values.
	`mext.is_periodize_supported()`	`bool`	Wether the mirror facility is supported.
ClampableExtension	`mext.clamp()`	`void`	Fill the extension by extending image's values at extremities.
	`mext.is_clamp_supported()`	`bool`	Wether the clamp facility is supported.
ExtendWithExtension	`std::convertible_to<point_type, U::point_type>`	`std::true_type`	Converts to the sub-image points type.
	`mext.extend_with(u, offset)`	`void`	Fill the extension with sub-image `u`'s values starting from `offset`.
	`mext.is_extent_with_supported()`	`bool`	Wether the extend-with-sub-image facility is supported.

Table 3.29: Concepts Extensions: expressions

Concept	Modeling type	Inherit behavior from	Instance of type
WithExtensionImage	`WExtImg`	Image	`wextimg`
ConcreteImage	`CImg`	Image	`cimg`
ViewImage	`VImg`	Image	`vimg`

Concept	Definition	Description	Requirement
WithExtensionImage	`extension_type`	Type of the image's extension.	Models the concept `Extension`.

Table 3.30: Concepts Image: definitions (7)

Concept	Expression	Return Type	Description
WithExtension Image	`wextimg.extension()`	`std::convertible_to< extension_type>`	Get the extension of the image.

Table 3.31: Concepts Image: expressions (7)

$$
\begin{aligned}
OutputImage = {} & (Image \implies WritableImage) \land \\
& (IndexableImage \implies WritableIndexableImage) \land \\
& (AccessibleImage \implies WritableAccessibleImage) \land \\
& (IndexableAndAccessibleImage \implies WritableIndexableAndAccessibleImage) \land \\
& (BidirectionalImage \implies WritableBidirectionalImage) \land \\
& (RawImage \implies WritableRawImage)
\end{aligned}
\tag{3.1}
$$

Figure 3.10: Concepts OutputImage: definition

Chapter 4

Image views

4.1 The Genesis of a new abstraction layers: Views

This concept of views is not new and naturally appeared in Image processing with Milena under the name of morphers [36, 48]. It was always useful to be able to project an image through a prism that could extract specific information about it without the need to copy the underlying data buffer. In modern days, the language C++ (20) also introduce this mechanism with the ranges [99] facility for *non-owning collections*. It is named *views* and allows the user to access a container's content (vector, map) through a prism. In Pylene, we decided to align the naming system after what was decided in C++20 in order not to confuse the user. This way, a `transform` view in image processing will do the same thing on an image that the transform view in the standard range library does on a container. *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 [38]), 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*, `clip(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. From those four fundamentals come out more very useful views such as `cast<T>(input)` or `mask(input, msk)` that are more specific to the image processing area.

In Pylena, the fractionner can use a large array of views. Those views comes into different form and allow the practitioner to seamlessly use arithmetic or logic operators on images like he would using expression template.

The transform view is the most important view of all. It consists in applying a function to each image's pixel. For instance, writing the grayscale algorithm with a transform view is as simple as the following code:

```
auto grayscale_transform = [](mln::rgb8 val) -> uint8_t {
  return 0.2126 * v[0]    // red
         + 0.7152 * v[0]  // green
         + 0.0722 * v[0]; // blue
};
mln::image2d<mln::rgb8> ima_rgb = /* ... */;
mln::image2d<uint8_t>   ima_grayscale = mln::view::transform(ima_rgb, grayscale_transform);
```

There is no loop in this code, just the pixel-wise transformation function. Furthermore, the code will not compute the resulting image. The computation will happen on-the-fly each time a value from `ima_grayscale` is yielded. This view allows the practitioner to quickly write and adapt any pixel-wise algorithm he needs for his more complex calculation in an efficient way.

The filter view is also a fundamental view which consists in keeping only the values that satisfy a predicate. This is very useful when working with thresholds as shown in the following code:

```
auto my_threshold = 145;
auto inferior_to [my_threshold](uint8_t val) { return val <= my_threshold; };
auto superiorstrict_to = [](uint8_t val) { return not inferior_to(val); };
mln::image2d<uint8_t> ima_grayscale = /* ... */;
auto ima_inferior = mln::view::filter(ima_grayscale, inferior_to);
auto ima_superiorstrict = mln::view::filter(ima_grayscale, superiorstrict_to);
mln::fill(ima_inferior, 0u8);
mln::fill(ima_superiorstrict, 255u8);
```

This code shows a way to binarise `ima_grayscale` with a custom threshold using the filter view. It is important to note that the resulting filtered image has its domain of definition changed. And the new domain of definition will most likely not be in a regular usual shape (such as a 2D rectangle). This implies that the usage of this view inside certain algorithms may be limited.

The clip view is a convenient way to extract a sub-image from a base image. This view essentially redefine the domain of definition to restrict it into a smaller one. It does not change anything else which means it proxies every access to the image. For instance, we make use of this view to easily process tiles in an efficient way in the following code:

```
mln::image2d<mln::rgb8> large_image = /* ... */;
point2d shape = large_image.domain().shape();
auto chunk_size = 1024;
auto nb_tile_x = shape.x() / chunks_size + 1;
auto nb_tile_y = shape.y() / chunks_size + 1;
for (int x = 0; x < nb_tile_x; ++x)
  for(int y = 0; y < nb_tile_y; ++y) {
    point2D tile_tl { large_image.domain().tl().x() + x*chunk_size,
                      large_image.domain().tl().y() + y*chunk_size };
    point2D tile_bl { large_image.domain().tl().x() + (x+1)*chunk_size,
                      large_image.domain().tl().y() + (y+1)*chunk_size };
    auto tile_domain = mln::box2d{tile_tl,
          tile_bl <= large_image.domain().bl() ? tile_bl : large_image.domain().bl()};
    auto tile = mln::view:clip(ima, tile_domain);
    process_tile(tile);
  }
```

The zip view is one of the most useful view and allow the practitioner to iterate over a set of image at the same time. The basic use-case consists in iterating over a set of input image and the output image to be able to consistently assign output values to a resulting computation from input values. Its usage is shown in the following code:

```
mln::image2d<uint8_t> input = /* ... */;
mln::image2d<uint8_t> output{input.domain()};
auto zipped_ima = mln::view::zip(input, output);
for (auto&& [v_in, v_out] : zipped_ima.values())
  v_out = v_in < 145 ? 0 : 255; // binarisation
```

This code is another example of how to compute a binary threshold image.

The mask view is very image-processing oriented as it allows the practitioner to provide a boolean image the same size as the original image to select only the pixels whose corresponding value in the mask is true. Its usage is shown in the following code:

```
mln::image2d<mln::rgb8> ima = /* ... */;
auto mask = ima > 127;
mln::fill(mln::view::mask(ima, mask), 255);
```

This code set all the values that are superior to 127 to the max value 255. It shows that it can both be used with read and write access.

The channel/RGB views is a projector to access a specific color channel of an image. There exists image with many more channels than just the standard red/green/blue ones, from the astrophysics or medical area for instances. This view is a tool to restrict an image and only access a specific channel. Its usage is shown in the following code:

```
mln::image2d<mln::rgb8> ima = /* ... */;
mln::copy(mln::view::red(ima), mln::view::green(ima));
```

This code copies the red component into the green component. It shows that the view can be used in both read and write access. Another more generic view exists; `mln::view::channel(ima, k)`, that access the `k`-th channel in `ima`.

The cast views is a way to convert an image's underlying type to another type, by performing a cast. As this does not modify the underlying value in itself, the access cannot be a write access. This view can be used as shown in the following code:

```
mln::image2d<double> ima = /* ... */;
mln::image2d<uint8_t> ima_8bits = mln::view::cast<uint8_t>(ima);
```

The arithmetical operators $+, -, *, /, \%$ are implemented in the form of transformation views that operate point-wise between two images whose size is identical. For instance, writing the following code:

```
mln::image2d<uint8_t> ima1 = /* ... */;
mln::image2d<uint8_t> ima2 = /* ... */;
auto ret = ima1 + ima2;
```

Is equivalent to writing the following code:

```
auto ret = mln::view:transform(ima1, ima2, [](auto v1, auto v2){ return v1 + v2; });
```

It is important to note that the `-` unary operator is also supported: `-ima1`.

The logical operators $<, <=, ==, !=, >, >=$ are implemented in the same way that arithmetical operators are. Both unary and binary operators are expressed as transform views such as writing the following code:

```
auto ret = !ima1 && ima2;
```

is equivalent to writing the following code:

```
auto tmp = mln::view::transform(ima1, [](auto v){ return !v; });
auto ret = mln::view::transform(tmp, ima2, [](auto v1, auto v2){ return v1 && v2; });
```

It is far more expressive and more comprehensible by the practitioner. Also, a new facility is introduced to express the logic behind a ternary expression (if C then A else B): the operator $ifelse(C, A, B)$. The rationale is to be able to swap between values depending on a boolean mask. This way, a mathematical morphology algorithm such as *hit or miss* can be implemented in the following simple manner:

```
mln::image2d<uint8_t> ima = /* ... */;
auto ero = erode(ima);
auto dil = dilate(ima);
uint8_t zero = 0;
auto ret = mln::view::ifelse(dil < ero, ero - dil, zero);
```

Everything is taken care of and the practitioner just has to write down his algorithm to get it done.

The mathematical operators are implemented in the form of views that operates point-wise. The supported mathematical operators are the following: abs, pow, sqr, cbrt, sqrt, sum, prod, min, max, dot, cross, l0norm, l1norm, l2norm, l2norm_sqr, linfnorm, lpnorm, l0dist, l1dist, l2dist, l2dist_sqr, linfdist, lpdist. Calling an operator onto an image is equivalent to calling a transform view on each value of this image:

```
auto ima = /* ... */;
auto ret = view::maths::abs(ima);
```

Is equivalent to calling:

```
auto ima = /* ... */;
auto ret = view::transform(ima, [](auto v){ return std::abs(v); });
```

4.2 Upgrading the way to design IP algorithms

As seen in chapter 3, there are three main categories of algorithms. First there are the *pixel-wise* algorithms which perform operations for each pixel without knowing about the other pixels. Second, there are the *local* algorithms which perform an operation on a pixel with the knowledge of a window of pixels around this pixel. Finally, there are *global* algorithms which perform an operation on a pixel with the knowledge of the whole image. *Views* introduce an abstraction layer that is able to completely handle the first case: pixel-wise algorithm. Indeed, every pixel-wise loop can be rewritten using the four fundamental views. This is not the case for local and global algorithms though.

Another key point to note is the lazy-evaluation. Indeed, often an IP practitioner will transform a whole image and only make use of a small subpart of it. Thanks to the lazy-evaluation property, only the exploited subimage will be transformed on-the-fly at the call site. However, if there are several call site browsing the view the same way, it can incur a performance loss as the same calculation is performed several time, each time the image is browsed. In this case, it is better to create a temporary concrete subimage (with a data buffer) that contains the values that will be asked several times.

The other key part about *views* is the composability. Image processing is an area in which a practitioner design pipelines which, by nature compose several operations on top of each other. Views have the ability to be composable with each others to get the wanted result.

Finally, a view is a *non-owning* and *cheap-to-copy* image, and that design has consequences on its usage.

4.2.1 Data ownership

The concept of *View* brought to us a fundamental issue when dealing with images: "*What is an image?*". More precisely: should an image always be the owner of its data buffer? Should we have a shared ownership of the data buffer between all the images using it? Then what happens when the data changes? The issue about the semantic of an image is crucial but also very similar to the issue there is to differentiate a *container* (such as `std::vector`, that is to say the data buffer) and a *view* on this container in the *Ranges TS*.

From here we have considered two approaches. The first one is to have *shared ownership* of the data buffer for the image and its derived views. However this does not allow the differentiation between an already computed image and a lazy image. To be able to make this differentiation is crucial in an *Image Processing library* as we want to make the most out of the data we already have and we do not want to compute data we do not need. Also, we cannot distinguish when the *copyability* property is required. This is the main reason why we did not adopt this approach.

The second one is to make the differentiation between a *concrete image* which owns the data (like the standard containers) and the *views* that are lightweight cheap-to-copy objects. Not all *concrete image* may be *copyable*, but all *views* are. This is a very important property as it simplify greatly the reasoning when performance is needed. It also enables us to have a library design similar to the standard library which the user is familiar with and, why not, have standard algorithm and standard view work on our images types. All of these are the main reason why we decided to adopt this design. Henceforth from now on the *Image* concept is similar to the *View* concept from which we refines a *ConcreteImage* concept that requires a specific behavior as it owns data.

In (subsec.gen.concept), we saw that what is truly important is the behavior: an algorithm will require its input to be able to behave a certain way, and if those requirements are fulfilled, then the algorithm can be used with this set of inputs. This enables non-standard type of image to be input in algorithms, providing they still behave correctly. The way how we can check if the required behavior is satisfied is a new C++20 feature called *concept* (the authors show how to leverage them in [83]) that will not be presented in this paper. Additionally to concepts, C++20 introduces a new library facility called *ranges* [107, 105] which includes a non-owning lightweight container called *views* whose design is very similar to that of *morphers*, introduced in Milena [36, 48]. Views are completely transferable to the image processing world. Also, views feature interesting properties that an image processing practitioner will find to his taste.

A view is a *lightweight object* that behaves exactly the same as an image: let V be a view of an image defined on $\mathcal{D}$ then we have $\forall p \in \mathcal{D}, v = V(p)$. It can be a random generator that yields a random number each time $V(p)$ is called; a proxy to the underlying image that records the number of times each pixel is accessed in order, for instance, to compare algorithm performance; a projection to a specific color channel; applying an automatic gamma correction; restricting the definition domain $\mathcal{D}$; and so on.

A view is a *non-owning*, *cheap-to-copy* lightweight object that basically only *records an operation* and stores a pointer to an image. For instance, let us consider the view transform defined as follow $v = transform(u_1, u_2, \cdots, u_n, f)$ where u_i are input images and h a n-ary function. $transform$ returns an image generated from the other image(s) as show in fig. 4.1. Also, we can see that the view itself does not own any image but just stores pointers as well as the operation (h). This means that, for instance, modifying the original values of the image(s) will impact the values yield by the view. Finally, as the view is cheap-to-copy, it features a pointer semantic that help the practitioner passing around his images by copy to his algorithms without worrying about heavy buffer copies in the background.

4.2.2 Lazy evaluation, composability and piping

Another key point of views is the lazy evaluation. When an image is piped through a view, no computation is done. The computation happens when

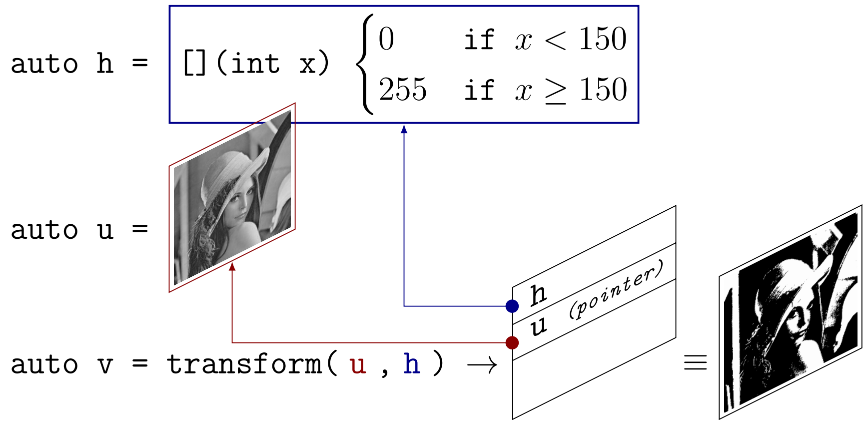


Figure 4.1: An image view performing a thresholding.

the practitioner requests a value by doing $val = V(p)$. The implications are multiples: an image can be piped into several computation-heavy views, some of which can be discarded later on, it won't impact the performance. Also, when processing large images, applying a transformation on a part of the image is as simple as restricting the domain with a view and applying the transformation to this resulting sub-image.

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 fig. 4.2 (see [76] 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;
});
```

Figure 4.2: Lazy-evaluation and *view* chaining.

The tree of type resulting from this view chaining is illustrated by fig. 4.3.

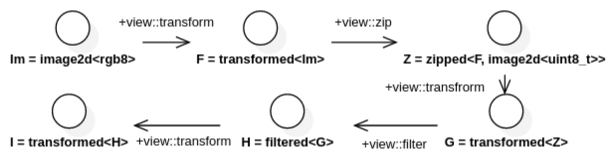


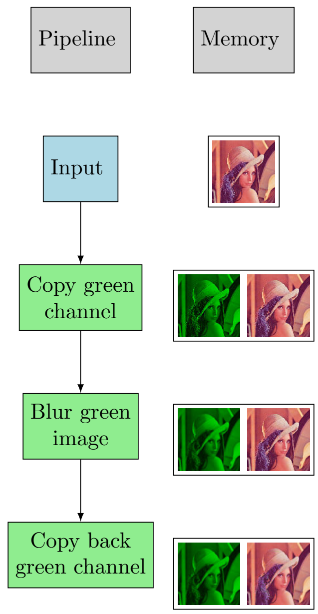
Figure 4.3: Abstract Syntax Tree of the types chained by the code above

Views will also try to preserve properties of the original image when they can. That means that views can preserve the ability of the practitioner to, for instance, write into this image. This may be a trivial property to preserve when considering a view that restrict a domain, but when considering a view that transforms the resulting values, it is not. Let us consider the projection $h : (r, g, b) \mapsto g$ that selects the green component of an RGB triplet. When piping the resulting view into, for instance, a blurring algorithm, the computation will take place in place thanks to still having the ability to write into the image. A legacy way of obtaining the same result would have been to create a temporary single-channel image consisting of the green channel of the original RGB image so that the temporary image could then be blurred. Then one would have needed to copy the values of the temporary image back into the green channel of the original image. The comparison between the legacy way and the in-place way of doing this computation is shown in fig. 4.4.

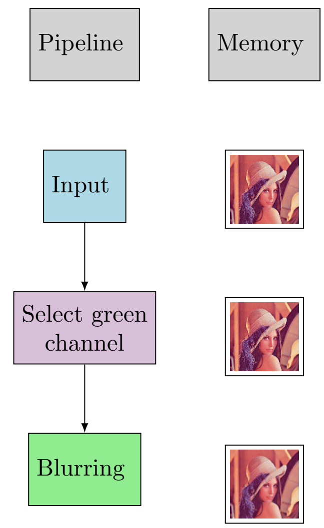
On the other hand, when considering the view $g : (r, g, b) \mapsto 0.2126 * r + 0.7152 * g + 0.0722 * b$ that compute the gray level of a color triplet (as shown in fig. 4.5), the ability to write a value into the image is not preserved. One would need an inverse function that is able to deduce the original color triplet from the gray level to be able to write back into the original image.

Following the same principle, a view can apply a restriction on an image domain. In fig. 4.6, we show the adaptor `clip(input, roi)` that restricts the image to a non-regular `roi` and `filter(input, predicate)` that restricts the domain based on a predicate. All subsequent operations on those images will only affect the selected pixels.

Views feature many interesting properties that change the way we program an image processing application. To illustrate those features, let us consider the following image processing pipeline: (Start) Load an input RGB-8 2D image (a classical HDR photography) (A) Convert it in grayscale (B) Sub-quantize to 8-bits (C) Perform the grayscale dilation of the image (End) Save the resulting 2D 8-bit grayscale image; as described in fig. 4.7.



(a) Legacy pipeline with copy



(b) Modern pipeline with view

Figure 4.4: Comparison of a legacy and a modern pipeline using algorithms and views.

4.2.3 Keeping properties

While is it tempting to use views everywhere, it is also important to note that you may converse or loose some properties from the base concrete image (the one holding the data buffer). To give a quick example, let us consider a classic RGB image and a view that calculate the grayscale on-the-fly. When only seeing the view, it is not possible to modify the base values which are RGB colors. Indeed, we cannot infer how a modification on the grayscale value will impact the base color of the image. Passing from a grayscale image to a colored image is a research area on its own and cannot be done deterministically on-the-fly in a fixed time. This induces that piping an image into this grayscale views turns the image *read-only*. in this case, The image loose the ability to be mutable. Another case would be a view that performs a projections of one of the color channel. This way the resulting viewed image is still mutable and modifying its values would only modify the color channel projected. Keeping properties is not an easy subject and requires the attention of the practitioner to pipe the correct operations to

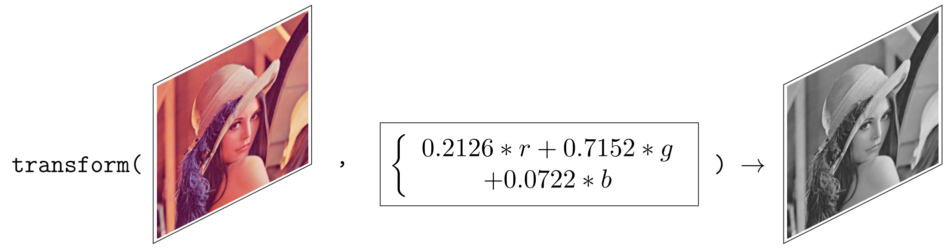


Figure 4.5: Usage of transform view: grayscale.

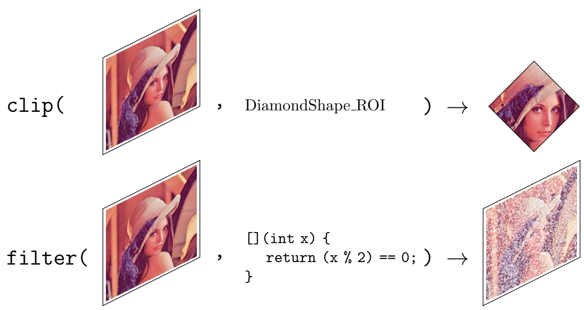


Figure 4.6: Clip and filter image adaptors that restrict the image domain by a non-regular ROI and by a predicate that selects only even pixels.

keep the properties he wants to keep.

There are six properties one want to keep track when working with views: forward, bidirectional, raw, writable, accessible and indexable. Those properties echo to the concepts seen in section 3.4. And image is *forward* when it can be traversed in a forward way. It is *bidirectional* when it can be traversed in both a forward and a backward way. It is *raw* when its data buffer can is contiguous and can directly be accessed with information about strides. It is *writable* when the values are mutable. It is accessible whenever it allows to access the value associated to a point (i.e. it allows to write the expression $v = ima(p)$). Finally, an image is *indexable* whenever its values can be accessed through an index localisator (i.e. it allows to write the expression $v = ima[idx]$). Usually, accessing through an index is faster than accessing by a point. The table table 4.1 present all the views and how they conserve

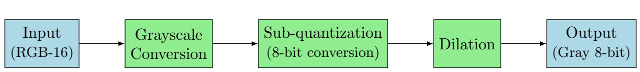


Figure 4.7: Example of a simple image processing pipeline.

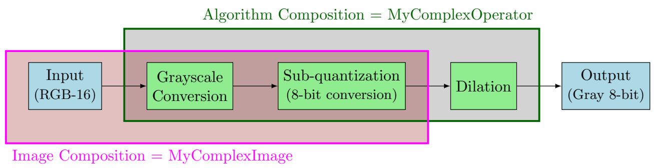


Figure 4.8: Algorithm vs image view composition.

the base properties of a concrete image.

Table 4.1: Views: property conservation

View type	Property / Expression	Forward	Bidirectional	Raw	Writable	Accessible	Indexable
Image	`ima1, ima2`	✓	✓	✓	✓	✓	✓
Cast	`cast<T>(ima)`	✓	✓	✗	✗	✓	✓
Transform	`transform(ima, func)`	✓	✓	✗	✓[1]	✓	✓
Filter	`filter(ima, pred)`	✓	✓	✗	✗	✓	✓
Clip	`clip(ima, dom)`	✓	✓	✗	✓	✓	✓
mask	`mask(ima, mask)`	✓	✓	✗	✓	✓	✓
Zip	`zip(ima1, ima2)`	✓	✓	✗	✓	✓	✓
Channel	`red(ima)`	✓	✓	✗	✓	✓	✓
Arithmetic	`ima1 + ima2`	✓	✓	✗	✗	✓	✓
Logical	`ima > 125`	✓	✓	✗	✗[2]	✓	✓
Mathematical	`abs(ima)`	✓	✓	✗	✗	✓	✓

[1]: writability is preserved only if `func` is a projection.
[2]: writability not preserved except for the expression `ifelse(ima, ima1, ima2)`.

4.2.4 Summary

Views are composable. Chaining operations has always been a very important feature in image processing as well as in software engineering in general (known object composition). Being able to weave simple blocks together into more complex blocks in a way that the resulting block can still be treated as a simple block is a most wanted feature. The fig. 4.7 features an example of a pipeline using 3 basic operations *Image* → *Image*: a grayscale conversion, a sub-quantization and a dilation. It is important to note that we can consider there is only one complex operation composed of 3 basic algorithms in which an image is piped. A view thus carries both information about the image and the transformations. In fig. 4.8 we show the distinction between the composition of algorithms and the compositions of views which carry both the image and the transformations.

Views improve usability. The code featuring the pipeline in fig. 4.8 can almost be implemented the following way:

```
auto input = imread(...);
auto A = transform(input, [](rgb16 x) -> float {
    return (x.r + x.g + x.b) / 3.f; };
auto MyComplexImage = transform(A, [](float x)
    -> uint8_t { return (x / 256 + .5f); };
```

When one is familiar to functional programming, it is quite easy to draw the parallel between *transform*, *map*, *filter* and the sequence operators. Views are, in reality, higher-order functions built from an image as well as the function(s) (operator or predicate) to apply for each pixel. It is not required to make the iteration over each pixel of the image oneself, we just provide the function to morph the image into another one. The technique used when composition several sequence operators is called *currying* [6] in the functional programming world.

Views improve re-usability. When looking at the code snippets above, one could see that they are simple though not very re-usable. However, keeping the functional programming paradigm in mind, one can easily define new views just by considering that a view is a *higher-order function*. Then, as shown in fig. 4.9, the primitive *transform* serves as the basis to build three new views: one that performs the summation of two images, one that performs the grayscale conversion and one that performs the sub-quantization. All those three views can then be reused afterwards[1].

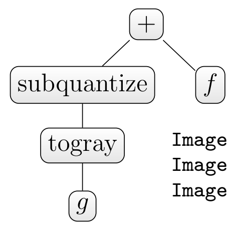
Views for lazy computing. One fundamental point of views is that they embed the operation within themselves, meaning that in fig. 4.9, the creation of the views does not incur any computation. The computation is delayed until the invocation of the `v(p)` expression. Also, the computation can be delayed quite far thanks to the composition capability of views. In fact, a view is an image adaptor which actually is a *template expression* [7, 17]. Indeed, the *expression* used to generate the image is recorded as a template parameter. A view is represented by an *expression tree*, as shown in fig. 4.10.

[1]A more generic implementation could have been provided for these views for even more re-usability, but this is not the purpose here.

```
auto operator+(Image A, Image B) {
  return transform(A, B, std::plus<>());
}
auto togray = [](Image A) { return transform(A, [](auto x)
  { return (x.r + x.g + x.b) / 3.f; };)
};
auto subquantize16to8b = [](Image A) { return transform(A,
  [](float x) { return uint8_t(x / 256 +.5f); });
};

auto input = imread(...);
auto MyComplexImage = subquantize16to8b(togray(A));
```

Figure 4.9: Using high-order primitive views to create custom view operators.




```
Image f = ...; // grayscale
Image g = ...; // rgb16
Image out = subquantize16to8b(
              togray(g)) + f;
```

Figure 4.10: View composition seen as an expression tree.

4.3 A practical example: border management

When looking at local algorithms, we notice that a long recurring issue is about the behavior on the border of the image. There are many ways of dealing with this problem. One is to allocate additional memory for the border and paste values in it. Another is to check the bounds when looping over the neighbors inside the computational window. We can also decorate the image to return a correct lazily computed value when accessing out-of-image's-bound value still inside the extension. The point is: all these methods have advantages as well as disadvantages.

Memory allocated border The border width is fixed at the image's creation and cannot be augmented without doing a reallocation. There is also a cost when computing border's values (to fill it) which is proportional to the border's width and to the image's size. On the other hand, the access time of a border value during the algorithm unrolling is as fast as a native access time within the image itself. The last issue remaining would be that the border is not infinite. We cannot process a local algorithm with a structuring element that does not fit in the extension. This method is especially adapted when there is medium structuring elements with a known size which will yield a lot of out-of-image's bound accesses. When speed is required, this method is a de facto standard.

Bound checking Assuming there is no border and we are not allowed to access out-of-image's-bound values, a bound check is required when accessing each values. Another way to do would be to decorate the facility that yields the neighbors of a pixel: do not yield out-of-image's-bound pixels. This removes the need to bound check for each pixel's value which is relatively faster. The caveats of this method are that it induces a slight slow down when yielding the pixel's neighbors from the structuring element, and that it is not always viable: some algorithms do need to access values in an extension to produce proper results.

Image decoration The border is infinite and we make a view of our image to decorate it with the required extension. This method has the advantage to *always work*. Given any structuring element of any size, any algorithm will work. The disadvantage is that we need to check for out-of-bound access at the image level, and lazily compute the value in case of out-of-image's-bound access. The slowness induced is not negligible and should be weighted carefully.

It is important to note the very close relation between an image's domain (to perform out-of-bound checks), the structuring element (notably its size) and the extension (its width). A user may require, for a specific set of those three elements, to decorate the image, and/or the structuring element and/or to perform computation and/or reallocation. To resolve this issue, we decided to provide the user with a new facility: the *border manager* whose job is to prepare a suitable pair (image and structuring element) given a set of configuration wanted by the user.

We designed the configuration to be constructed from a given set of a policy and a method. We currently offer two policies: native and auto.

- Native: if the border is large enough: forward the image as-is to the algorithm to allow the fastest access possible. Otherwise, the border manager fails and halt the program.
- Auto: if the border is large enough: forward the image as-is to the algorithm to allow the fastest access possible. Otherwise, decorate the

image with a view whose extension will emulate what is required by the algorithm with the given structuring element.

We also provide seven different methods to fill up our extension with the wanted values. It is important to note that not all the methods are available for both policies. The policies are: none, fill, mirror, periodize, clamp, image and user.

None Enforces enforce a policy where there is no border to use. The method cannot fail as it enforces the border to vanish. To enforce this method, the border manager decorate the structuring element in a view that checks the domain inclusion of each neighboring point. And example is given in figure 4.11.

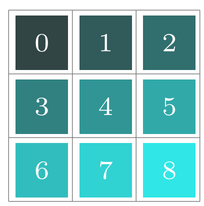


Figure 4.11: Border method: none.

Fill Enforces that the border is filled with a specific value. The figure 4.12 shows an image whose border is filled with the value 0.

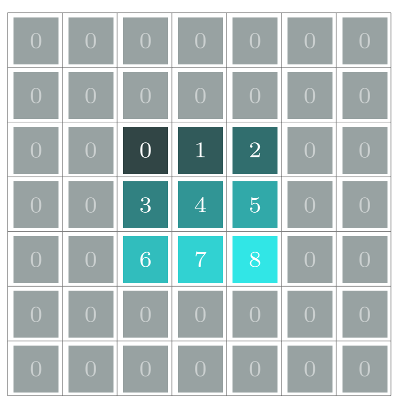


Figure 4.12: Border method: fill.

Fill Enforces that the border is filled with a mirrored value from an axial symmetry relative to the image's edges. The figure 4.13 shows an image whose border is filled with mirrored values.

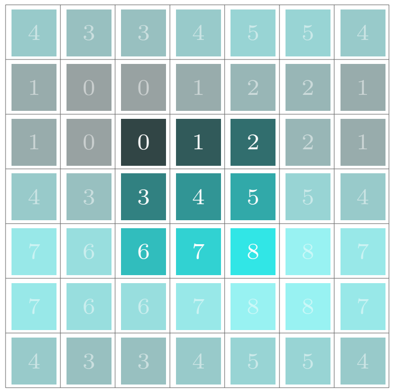


Figure 4.13: Border method: mirror.

Periodize Enforces that the border replicate the image, like a mosaic. The figure 4.14 shows an image whose border is filled with periodized values.

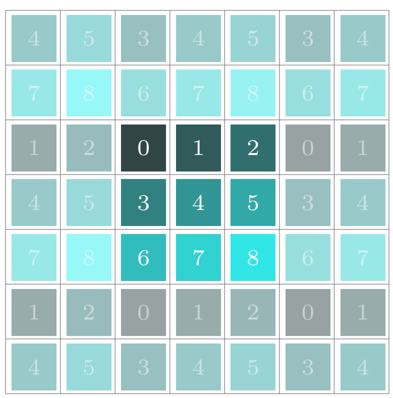


Figure 4.14: Border method: periodize.

Clamp Enforces that the border is filled with values expanded from the values at the image's edge. The figure 4.15 shows an image whose border is filled with clamped values.

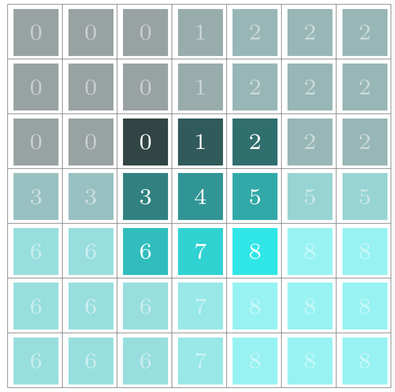


Figure 4.15: Border method: clamp.

Image Enforces all points out of the current image's domain are to be picked inside another image. A basic use-case is preparing tiles from a large image. The position of our image can be offset in the image acting as an extension which ease the ease of usage. The figure 4.16 shows how a subimage (tile) can consider the base image as its border.

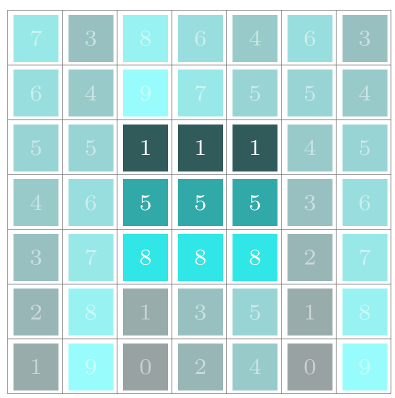


Figure 4.16: Border method: image.

User Assumes the user knows what he is doing and do not touch nor decorate the given image in any way. Both policies lead to the same behavior: check whether the structuring element fit and then forward the image as-is if it fits. An exception is raised if it does not.

As a consequence the usage of a local algorithm becomes very simple:

```
// default border width is 3
image2d<int> ima = {{0, 1, 0}, {0, 1, 1}, {0, 1, 0}};
auto disc_se = se::disc{1}; // radius is 1
auto bm = extension::bm::fill(0); // fill border with 0 with policy auto

local_algorithm(ima, disc_se, bm); // will handle the border for you
```

The border manager `bm` is set with the method fill (with value 0) and policy `auto` (wich is the default policy). To use the policy `native`, one would write `extension::bm::native::fill(0)` instead.

In the implementation of the local algorithm, a dispatch is made with the pattern *visitor*, relying on the standard facilities `std::visit` and `std::variant` so that the performance overhead as well as the complexity of use remain minimal. Let us assume we have a local algorithm implemented this way:

```
template <class Ima, class SE>
local_algorithm(Ima ima, SE se)
{
  // assume ima has a large enough border for the given se
  // use ima & se in loop
}
```

We can rewrite it leveraging the border manager facility this way :

```
template <class Ima, class SE, class BM>
local_algorithm(Ima ima, SE se, BM bm)
{
  auto [managed_ima, managed_se] = bm.manage(ima, se);
  std::visit([&](auto&& ima_, auto&& se_) {
    // use ima_ & se_ in loop
   }, managed_ima, managed_se);
}
```

The overhead is kept minimal thanks to using `std::variant` and `std::visit` and the algorithm implementer delegate the border management to the border manager. This is made possible thanks to the views. Indeed, under the hood the border manager may pipe the orginal image into a view that will behave accordingly to the policy chosen by the user. This will be transparent from both practitioner and maintener points of views.

4.4 Performance discussion

In order to have a relevant discussion on performances, we decided to implement a real world image processing pipeline: the background subtraction. It is used to detect changes in image sequences [73]. It is mainly used when regions of interest are foreground objects. The pipeline components include: subtraction, Gaussian filtering, threshold, erode and dilate, as shown in 4.17.

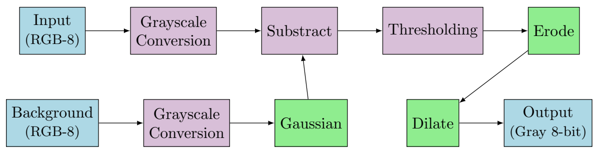


Figure 4.17: Background substraction pipeline using algorithms and views.

We have decided to run the algorithm on an original set of image to detect a changing foreground. Here are the 10 sets we have considered.

Set #1: castle The figure 4.18 presents the first candidate set we have run the algorithm with.

Figure 4.18: Background detection: castle results.

We have run benchmarks on this set comparing multiple ways of achieving this result, both using Pylene and OpenCV as well as varying the size and the shape of the structuring element window.The breakdown of these benchmarks are presented in the figures 4.19 and 4.20.

Set #2: garden We have also run the algorithm on another another original set of images presented in figure 4.21.

The breakdown of the benchmarks corresponding to this original set is presented in the figures 4.22.

Set #3: pathway Finally, we have run our algorithm on a last original set of image presented in figure 4.23.

The breakdown of the benchmarks corresponding to this original set is presented in the figures 4.24.

For small structuring elements ($radius < 15$), OpenCV is faster thanks to its handwritten code optimization. However, when the size of the structuring elements increase, so does the execution time, especially for structuring

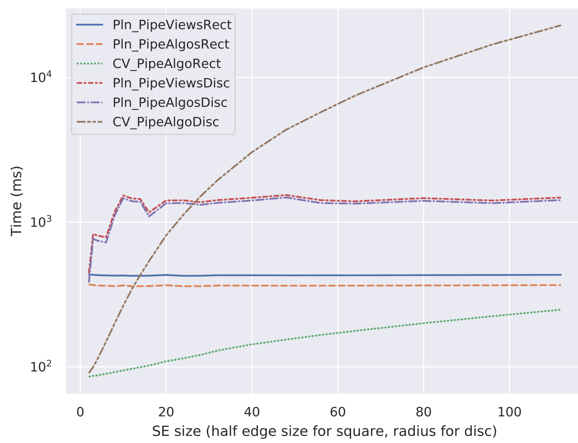


Figure 4.19: Background detection: castle benchmark.

elements shaped as discs. We can see that the way OpenCV decomposes its structuring elements is not efficient for a disc whereas it is very efficient for a rectangle. Pylene, on the other hand has a starting cost higher even for small structuring elements but is extremely stable over the increase of structuring elements' size. Indeed, the only variation is seen when we are working with the disc shape and this variation stabilise for larger size. For a disc whose $radius > 25$, Pyelene will be faster than OpenCV.

When looking at the performance of algorithms expressed in term of views, we can see that expressing an algorithm in term of view incurs a very small overhead that stays constant and stable over the increase of structuring elements' radius. Furthermore, it does not show on the graphics but the memory consumption is reduced when using views. Indeed, more computation is done on the fly and the image is copied less during the algorithm. This implies that caching values is harder when using views which may explain the overhead.

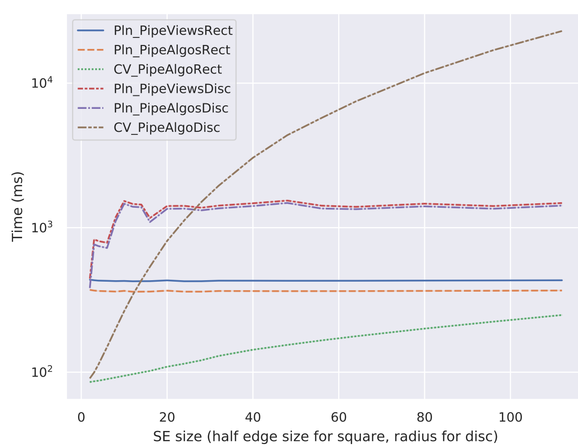


Figure 4.20: Background detection: castle benchmark, Pylene only.

Figure 4.21: Background detection: garden results.

Result | Benchmark | Benchmark Pln only

Figure 4.22: Background detection: garden results.

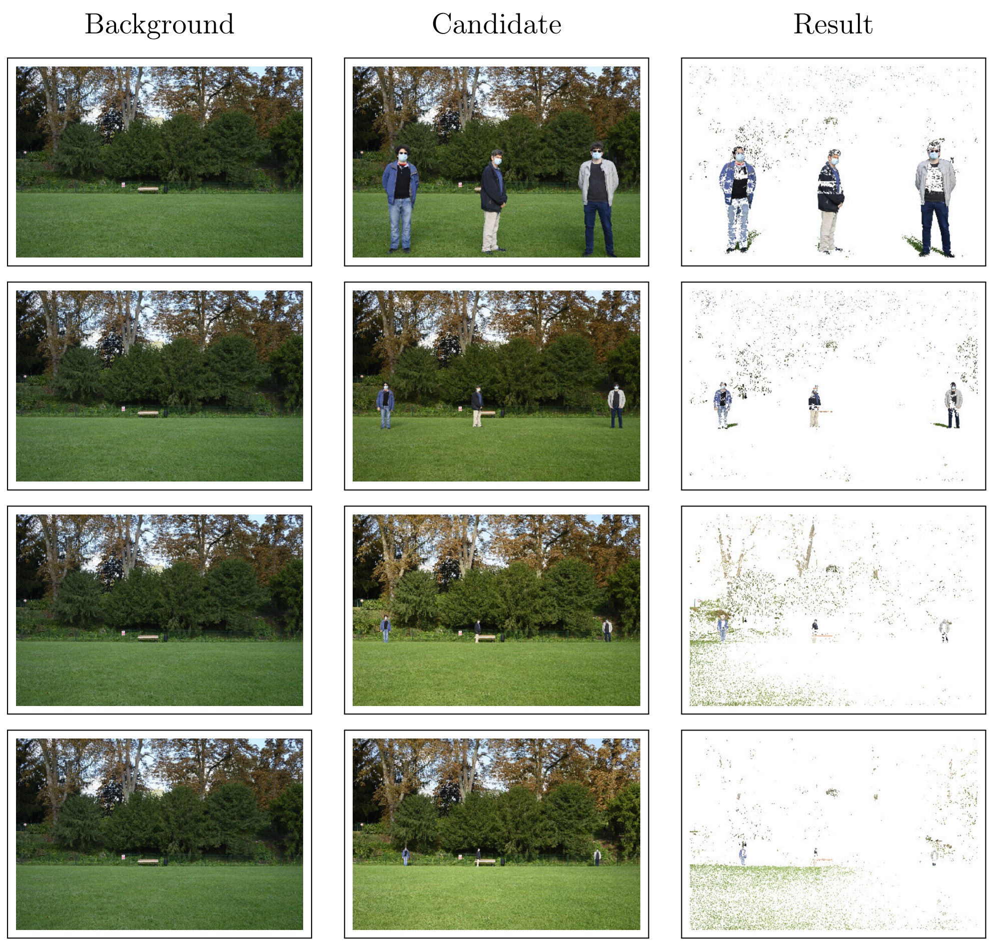


Figure 4.23: Background detection: pathway results.

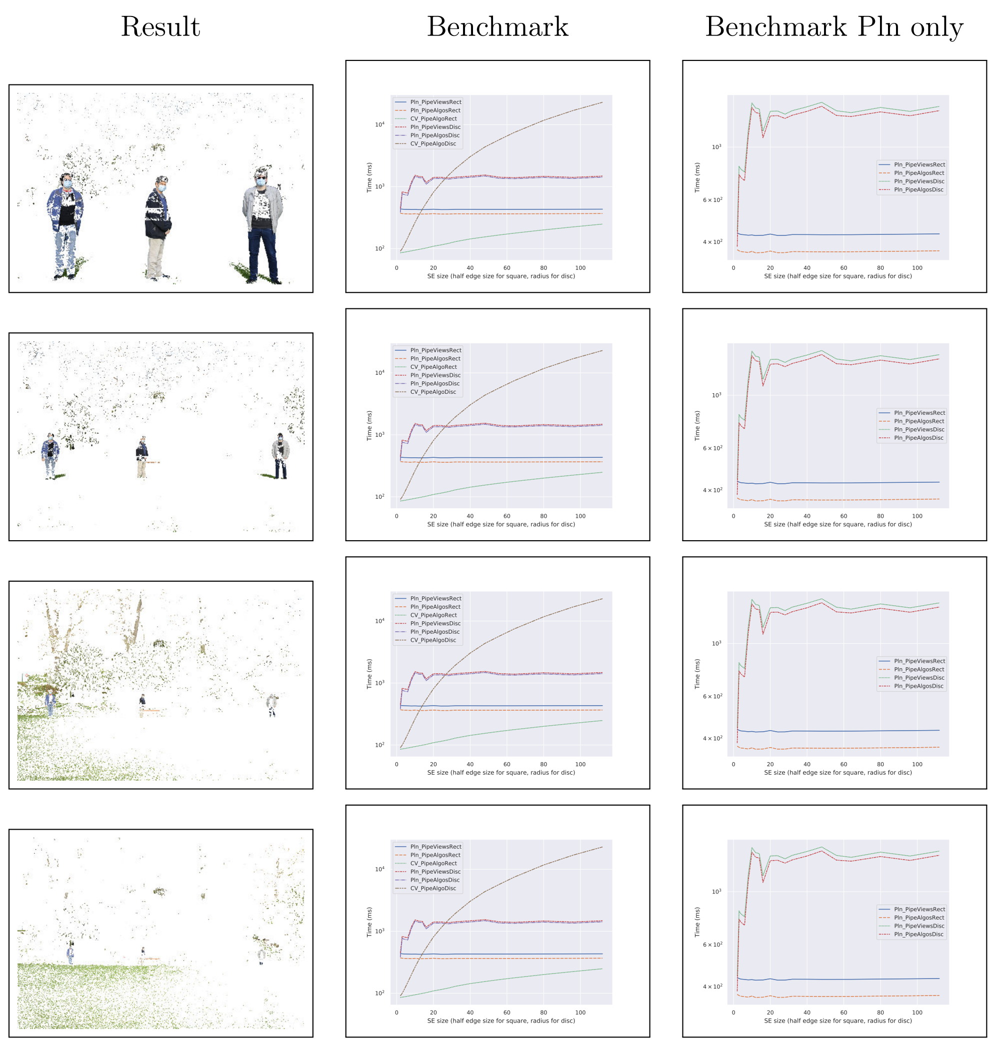


Figure 4.24: Background detection: pathway results.

Chapter 5

Static dynamic bridge

In the programming world, there are two main families of programming language. There are the *compiled* programming, such as C, C++, Rust or Go. There are also the *interpreted* programming languages, such as Python, PHP, Lisp or Javascript. Finally, there are languages such as Java that are both at the same time.

The *compiled* programming languages have the advantage of being very end-user friendly. Indeed, the implementer distribute compiled self-sufficient binaries and the user select the binary that is compatible with his operating system. Then, the program is supposed to work out of the box without more work than that, as illustrated in fig. 5.1.

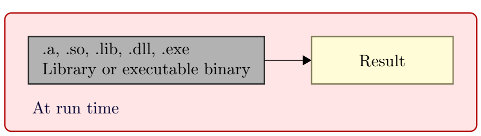


Figure 5.1: Compiled languages: run-time

Opposite to this apparent simplicity for the end-user, all the burden is shouldered by the programmer. Indeed, to generate a binary, there are many steps, as illustrated in fig. 5.2. There is a first pass with the compiler to generate intermediate machine code. Then there is a linker pass to resolve any dependency between machine code and system code into one or multiple final distributable binaries. However, this last pass tie the binary with a distribution. Indeed, the location of system libraries may vary between operating system, between version of the same operating system, between compiler variants etc. Also, as the developer wants to distribute efficient programs, he will use last optimized vectorized instructions if possible, which can tie further the binary to a certain set of hardware supporting some assembly instructions (SSE4, XOP, FMA4, AVX-512, etc.). Upstream from those issues, there are also issues with code. Indeed, many libraries are not cross-platform and leveraging all the equivalences from one OS to another incurs an increase in code quantity, tests and maintenance cost to support many platforms. For instance, the native GUI Windows libraries does not exist in Linux and must be rewritten with another framework, such as GTK or Qt. Or else, the developer can choose to use a cross-platform GUI library from start however this decision may not have been viable if the software was first windows-only and the cross-platform support was added at a later date. Another caveat is that using code introspection is often very difficult at compile-time because only few information are available (only static information). Dynamic reflection at runtime is impossible.

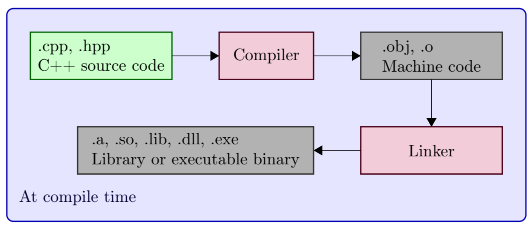


Figure 5.2: Compiled languages: compile-time

The *interpreted* programming language are a little less end-user friendly but are much more comfortable for developer to distribute their software. Indeed, as shown in fig. 5.3, everything happens at runtime. The maintainer only distribute the source code, the dependency list and the assets necessaries for his program. The burden is mostly shouldered by the end-user this time. He must download and install all the language interpreter and environment in order to execute the program from the source code. He must resolve the dependencies and be able to execute the source code on his computer. This has the advantage of having a very rich ecosystem as distributing, maintaining and using programs is very easy once integrated in a package manager (often delivered with the language SDK natively). However, the main disadvantage is the performance. As the source code is not compiled into optimized assembly code ready to be executed, the interpreter must do all the work in one go and very often this is slow. Nowadays, all the interpreter embarks a Just-In-Time (JIT) compiler that detected the portions of a program that are used heavily (a.k.a. hot code) and will compile them into native machine code to increase the performance drastically without having the user pay for a long compilation time. Also, those languages usually have very developed introspection facilities. Dynamic reflection at runtime is possible and some language, such as Common Lisp, even go further by allowing the developer to mutate the program Abstract Syntax Tree (AST) at runtime (macro). This allows very powerful integrations such as defining one's own DSL as if it was part of the core language itself.

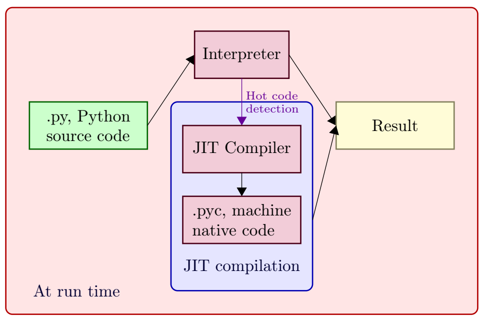


Figure 5.3: Interpreted languages: run-time

Image processing communities like to have bridges with interpretable language such as Python or Matlab, to interface with their favorite tools, algorithms and/or facilities. As an example, with Python, the module NumPy [32] is community standard which is heavily used. Henceforth, to broaden the usage of our library, we should be able to provide a way to communicate between our library and NumPy. However, here is a showstopper: we only distribute source code, we don't hand over binaries. Indeed, genericity in C++ is achieved via usage of template metaprogramming. One caveat of it is that the C++ compiler cannot generate a binary until it knows which type (of image, of value) will be used. But we don't know this information: the user (on Python's end) is not going to recompile our library each time he has another set of types to exercise. From here, there are still multiple ways to achieve our goal.

First option is to embark a JIT (Just-in-time) compiler whose job would be to generate the binaries and bindings just as they are used. This solution brings speed (excluding the first run that includes the compilation time) and unrestrained genericity. However we are now bound to specificities of a compiler vendor and loose platform portability.

Another option is to type-erase our types to enables the use of various concrete types through a single generic interface. This would translate into a class hierarchy whose concrete classes are on the leaves (thus, whose value type and dimension are known). This induces a non-negligible slow down but allow us to keep the genericity and portability at the cost of maintaining the class hierarchy.

Type generalization can also be considered: cast everything into a super-type that is suitable for the vast majority of cases. For instance, we could say that we have a super-type `image4D<double>` into which we can easily cast sub-types such as `image2D<int>` or `image3D<float>`. Of course we would loose the generic aspect and induce non negligible speed cost. Although portability is kept.

And finally there is the dynamic dispatch. It consists in embarking dynamic information at runtime about types, and dispatch (think of switch/case) to the correct facility which can handle those types. The obvious caveat is the cost of maintenance induced by the genericity as we would have a number of possible dispatches that grow in a multiplicative way with the number of handled types. Which is not very generic. On the other hand there is almost no speed loss and the portability is guaranteed. Theoretical models exist that could bring solutions to lower the number of dispatcher to write, such as multi-method [41]. Unfortunately they are currently not part of C++.

In Pylene we have chosen an hybrid solution between type-erasure and dynamic dispatch. The aim is to have a set of known types for which we have no speed cost as well as continuing to handle other types to remain generic. To achieve this goal, we have worked together with Célian Gossec [81], a student co-supervised by the authors of this thesis, in order to provide a facility to expose our generic code to Python. As seen in the previous chapter, it is not possible to bind C++ source code to Python. We need to have a compiled binary implementing Python binding (we chose Pybind11 [72]) in order to be able to call C++ code from Python. In order to achieve the binding without sacrificing the genericity and the performances, we have designed a solution in two steps. We do not want to provide an abstract interface that will resolve the calls to access data on the call-site via virtual call because it would be very slow when the C++ code is executed. This would defeat the purpose of having to rely on C++ in a first place. However, it is possible to convert an abstract class into an instantiated concrete generic class whose template parameter are known. This requires, however, to enumerate all the possible cases. With modern C++, it has become possible to design $n * n$ dispatch without gigantic switch-case clauses.

5.1 Hybrid solution's first step: type-erasure

The first step of our solution consists in designing a buffer class that holds all the informations about an image: dimension, underlying type, strides and pointer to data buffer. This class is named `ndimage_buffer`. When interfacing with Python, it is necessary to convert the Python image which is a `NumpPy.array` into our image type. The purpose of this buffer image is to holds all the information from the `NumpPy.array` to then instantiate a concrete C++ type. The first pitfall here is due to a limitation from the abstraction interface used in Python. Indeed, when using for instance *Scikit-Image*, it is not possible to differentiate a 2D multichannel image from a 3D grayscale image. Indeed, the image is always broken down to its most simple value and a 3D multichannel image is turned into a 3-dimensional `NumpPy.array` containing 8-bits values, the last dimension contains only 3 elements at max but can theoretically contain more as there is no limitation from the used abstraction to prevent that. A 3D grayscale image will be broken down into a 3-dimensional `NumpPy.array` containing 8-bits values, the last dimension will contain many values as it is expected of a 3D-image. To prevent this confusion, there is a need to explicitly say to the Python/C++ wrapper wether the image is multichannel or not. This information must be carried through the `ndimage_buffer` into C++ for a correct instanciation. This process is illustrated in fig. 5.4.

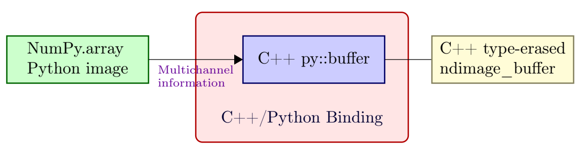


Figure 5.4: Bridge from Python to C++ via Pybind11 and a type-erased C++ class.

From the point of view of a practitioner, the code on the call-site (python side) should be as follow:

```python
from skimage import data
import numpy as np
import Pylena as pln # our python binding
img = data.astronaut() # 2D-rgb8 image -> Numpy.array(ndim=3, dtype='uint8')
pln_img = pln.ndimage(img, multichannel=True) # manualy point out multichannel information
# use(pln_img) for any pln.<algorithm> exposed to Python
```

From the point of view of the library implementer, the code to expose the binding looks like this one:

```
#include "ndimage.hpp"
#include <pybind11/pybind11.h>

// Expose pybind module
void init_class_ndimage(pybind11::module& m);
PYBIND11_MODULE(Pylena, m) { init_class_ndimage(m); }

// declare the conversion function
namespace mln::py {
  mln::ndbuffer_image   ndimage_from_buffer(pybind11::buffer b, bool is_multichannel = false);
  pybind11::buffer_info ndimage_to_buffer(const mln::ndbuffer_image& img);
}

// expose the python ndimage class with the conversion from/to py::buffer (numpy.array buffer)
void init_class_ndimage(py::module& m) {
  using namespace pybind11::literals;

  py::class_<mln::ndbuffer_image>(m, "ndimage", py::buffer_protocol(), "is_multichannel"_a = false)
    .def(py::init(
        [](py::buffer b, bool is_multichannel = false) {
            return ndimage_from_buffer(b, is_multichannel); }))
    .def_buffer(ndimage_to_buffer);
}
```

This code declares a new module named `Pylena`. It then declares a class named `ndimage` which is a bridge to Python's `buffer_protocol`. This `buffer_protocol` is an abstraction to allow the usage of NumPy's array. Finally, the code declares that the class is convertible to and from the `buffer_protocol` thanks to provided callbacks. The code of those callbacks is as follow:

```
// implement the conversion from Python to C++
mln::ndbuffer_image ndimage_from_buffer(::py::buffer b, bool is_multichannel) {
  /* Request a buffer descriptor from Python */
  ::py::buffer_info info = b.request();

  std::ptrdiff_t      strides[16];
  int                 dims[16];
  int                 ndim = info.ndim - is_multichannel ? 1 : 0;
  int                 bpp  = info.itemsize;

  // convert the type information from Python to C++ (string to enum) for faster dispatch
  mln::sample_type_id st   = get_sample_type(info.format, is_multichannel);

  for (int i = 0; i < ndim; ++i) {
    dims[i]    = info.shape[ndim - 1 - i];
    strides[i] = info.strides[ndim - 1 - i];
  }

  if (strides[0] != bpp)
    throw std::runtime_error("Unsupported image stride along the last dimension.");

  // construct the type-erased image with all informations
  return mln::ndbuffer_image::from_buffer(
```

```
    reinterpret_cast<std::byte*>(info.ptr), st, ndim, dims, strides);
}

// implement the conversion from C++ to Python
::py::buffer_info ndimage_to_buffer(const mln::ndbuffer_image& img)
{
  // return the python format of the underlying type, as well as information about multichanneling
  auto [ti, is_multichannel, channel_count, channel_stride] =
          get_sample_type_id_traits(img.sample_type());

  int ndim = img.pdim() + is_multichannel ? 1 : 0;

  std::vector<ssize_t> dims(ndim);
  std::vector<ssize_t> strides(ndim);

  for (int i = 0; i < ndim; ++i) {
    dims[i]    = img.size(ndim - 1 - i);
    strides[i] = img.byte_stride(ndim - 1 - i);
  }
  dims[ndim - 1]    = channel_count;
  strides[ndim - 1] = channel_stride;
  // construct the python buffer with all the information
  return ::py::buffer_info(img.buffer(), ti.size(),
          get_sample_type(img.sample_type(), is_multichannel), ndim, dims, strides);
}
```

This code forward informations about the buffer and handle the special case of multichannel images which Python treat as 3D images.

5.2 Hybrid solution's second step: multi-dispatcher (a.k.a. $n * n$ dispatch)

The second step of our hybrid solution is to dispatch the type-erased code to efficient generic code. The naive way of doing so would be to include a gigantic switch-case clause in each algorithm implementation and dispatch to the correct instantiated generic algorithm from there. Aside from being a nightmare to maintain, the size of those clause can grow several fold depending on the cardinality of the generic implementation. For instance, for a generic dilation, there are 3 axis of cardinality: the underlying type, the dimension, the structuring element shape. In the case where the library support 5 different structuring element shape, 10 underlying types and 6 dimension for the image, the switch-case statement will have 300 clauses to dispatch. And each algorithm will have to dispatch. This solution is not viable, defeat the purpose of genericity which is to write less code in the first place. We had to find a solution to have those dispatch while keeping our code short and efficient. The idea we took to solve this problem comes from the design of a C++ feature, the variant, and especially the visitor. We need to have a way to write the implementation of the algorithm once while enumerating all the possible cases. Also, if possible, the list of supported types should be written once at one place for maintenance purpose.

We then had the idea of writing a dispatcher. This dispatcher list all the supported types and call the given callbacks forwarding the given arguments by instantiating a specific type. For instance, let us first expose the binary threshold operator to Python. The Python call-site code will look like this:

```
img_grayscale = skimage.data.grass()
pln.operators.binary_threshold(img_grayscale)
```

On C++ side, we need to expose the function with this code:

```
void init_module_operators(pybind11::module& m);

mln::ndbuffer_image binary_threshold(::py::buffer buffer)

void init_module_operators(::py::module& m) {
  using namespace pybind11::literals;

  m.def("binary_threshold", binary_threshold,
        "Perform a binary threshold.\n",
        "Input"_a);
}

PYBIND11_MODULE(Pylena, m) {
  /* ... */

  auto operators = m.def_submodule("operators", "Image processing operators.");
  init_module_operators(operators);
}
```

Now that our python submodule and that our `binary_threshold` operator are declared, let us have a look to the operator's implementation:

```
mln::ndbuffer_image dilate(::py::buffer buffer)
{
  // grayscale image mandatory
  auto input = mln::py::ndimage_from_buffer(buffer);

  // dispatch along dimension (dimension is a valued template parameter, hence _v)
  return dispatch_v<binary_threshold_operator_t>(input.pdim(), input);
}
```

We have replaced the gigantic switch-case clause by a dispatcher templated by an operator. This operator will cast the input image into the concrete generic type and call the fast generic algorithm on it. Let us have a look to what this operator look like:

```
// Operator templated by the dimension
template <auto Dim>
struct binary_threshold_operator_t {
  // Function templated by the image type
  template <typename Img>
  mln::ndbuffer_image operator()(Img&& img) const {
    // Cast to a grayscale (information known) of the correct dimension
    if (auto* image_ptr = std::forward<Img>(img).template cast_to<std::uint8_t, Dim>(); image_ptr)
      // call generic algorithm
      return mln::operators::binary_threshold(*image_ptr);
    else {
      std::runtime_error("Unable to convert the image to the required type.");
      return {};
    }
  }
};
```

This operator will attempt to cast the given image into a grayscale image of the correct dimension and then use the resulting concrete type to pass it the fast generic `binary_threshold` operator. Now let us have a look at where the magic happen, at the dispatcher which list all the supported type.

```
template <template <auto> class V, typename... Args>
auto dispatch_v(std::size_t dim, Args&&... args) {
  switch (pdim) {
  case (1):
    return F<1>{}(std::forward<Args>(args)...);
  case (2):
    return F<2>{}(std::forward<Args>(args)...);
  case (3):
    return F<3>{}(std::forward<Args>(args)...);
  case (4):
    return F<4>{}(std::forward<Args>(args)...);
  case (5):
    return F<5>{}(std::forward<Args>(args)...);

  /* ... */

  case (0):
    [[fallthrough]];
  default:
    throw std::runtime_error("Unsupported dimension.");
}
```

The dispatcher is instantiate the given type by the correct dimension number and then call the operator parenthesis (function call) forwarding all the given parameters. In our case, it will instantiate the type `binary_threshold_operator_t<2>` and then call the function `binary_threshold_operator_t<2>.operator()(input)`, forwarding the input image to the underlying algorithm.

The main advantage of this approach is that all the supported features are to be listed only in one place, the dispatcher, while any number of dispatcher can be piped to achieve the cardinality wanted. Let us push our example to implement the mathematical morphology operator dilation. We now have two more generic axis to cover: the structuring element shape and the underlying datatype. First, let us expose the operator to the Python code. Here is what the Python call-site look like:

```
img_grayscale = skimage.data.grass()
rect = pln.se.rect2d(width=3, height=3)
pln.operators.dilate(img_grayscale, se)
```

We need to expose the new structuring element's sub-module for usage in the dilation operator:

```
void init_module_se(pybind11::module& m);

void init_module_se(::py::module& m) {
  ::py::class_<mln::se::disc>(m, "disc").def(
      ::py::init([](float radius) { return mln::se::disc{radius}; }));

  ::py::class_<mln::se::sphere>(m, "sphere").def(::py::init([](float radius) {
    return mln::se::sphere{radius};
  }));

  ::py::class_<mln::se::rect2d>(m, "rect2d").def(::py::init([](int width, int height) {
    return mln::se::rect2d{width, height};
  }));

  ::py::class_<mln::se::rect3d>(m, "rect3d").def(::py::init([](int width, int height, int depth) {
    return mln::se::rect3d{width, height, depth};
  }));
}

PYBIND11_MODULE(Pylena, m) {
  /* ... */

  auto mse = m.def_submodule("se", "Structuring elements module.");
  init_module_se(mse);
}
```

Now we need to expose the dilate function into the operator submodule:

```
// using std::variant
using se_t = std::variant<mln::se::disc, mln::se::sphere,
                          mln::se::rect2d, mln::se::cube>;

mln::ndbuffer_image dilate(::py::buffer buffer, const se_t& se)

void init_module_operators(::py::module& m) {
  /* ... */

  m.def("dilate", dilate,
    "Perform a morphological dilation.\n"
    "\n"
    "structuring element must be valid.",
    "Input"_a, "se"_a);
}
```

We are all set to now implement the dilation operator. First, let us have a look at the underlying operator that will be dispatched:

```
template <auto Dim, typename T>
struct dilate_operator_t {
  template <typename Img, typename SE>
  mln::ndbuffer_image operator()(Img&& img, SE se) const {
    if (auto* image_ptr = std::forward<Img>(img).template cast_to<T, Dim>(); image_ptr)
      return mln::dilation(*image_ptr, se);
    else {
      std::runtime_error("Unable to convert the image to the required type.");
      return {};
    }
  }
};
```

Here we can see that we need a double dispatch. Also, the structuring element is no longue a variant and needs to be dispatched before instantiating this operator. Finally, there is an issue here because there are two template parameter and our dispatcher `dispatch_v` does only handle one. We workaround this issue by writing another intermediate operator dispatcher `dilate_operator_intermediate_t` serving as trampoline that will partially instantiate the final operator `dilate_operator_t` along the dimension template parameter to feed it to the last dispatcher, `dispatch_t`:

```
template <auto Dim>
struct dilate_operator_intermediate_t {
  template <typename Img, typename SE>
  mln::ndbuffer_image operator()(Img&& img, SE&& se) const {
    // Partial instantiation
    return double_dispatch_t<dilate_operator_t, Dim>(
            input.sample_type(), std::forward<Img>(input), std::forward<SE>(se));
  }
};
```

The final function implementation will look like this:

```
mln::ndbuffer_image dilate(::py::buffer buffer, const se_t& se) {
  auto input = mln::py::ndimage_from_buffer(buffer);
  // dispatch the structuring elements through using std::visit for std::variant
  return std::visit(
      [&input](const auto& se_) {
        return dispatch_v<dilate_operator_intermediate_t>(input.pdim(), input, se_);
      }, se);
}
```

The final piece of our puzzle would be the double dispatch function that will handle the last dispatch along the underlying data while forwarding the first dispatch along the dimension. Here is how we implemented our double dispatch:

```
template <template <auto, typename> class F, auto Dim, typename... Args>
auto double_dispatch_t(mln::sample_type_id tid, Args&&... args) {
  switch (tid) {
    case (mln::sample_type_id::INT8):
      return F<Dim, std::int8_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::INT16):
      return F<Dim, std::int16_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::INT32):
      return F<Dim, std::int32_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::UINT8):
      return F<Dim, std::uint8_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::UINT16):
      return F<Dim, std::uint16_t>{}(std::forward<Args>(args)...);
    case (sample_type_id::UINT32):
      return F<Dim, std::uint32_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::DOUBLE):
      return F<Dim, double>{}(std::forward<Args>(args)...);

      /* ... */

    case (mln::sample_type_id::OTHER):
      [[fallthrough]];
    default:
      throw std::runtime_error("Unhandled data type");
  }
}
```

Now we have all the pieces to build operators that are agnostic from the supported data-types. Indeed, the maintainer has gathered all the logic about listing supported data types and dimension into variant or custom dispatcher. He just need to maintain those to enable, by default, all exposed algorithm to support them. This hybrid solution mixes type-erasure and modern C++ facilities to allow maximum performance. Indeed, the dispatch is done before entering algorithms and the buffer protocol facility allows us to plug directly into the Python image without having any unnecessary copies. The only caveat would be the code bloat incurred by all the explicit instanciation leading to compiling a large binary. Another point not covered right now would be a way to inject Python types into C++. Indeed, our hybrid solution only support the types provided by the library. It will instantiate all the code relative to them and support all of the combinations. But the user may be tempted to plug a user-defined type from Python as an underlying data-type. To allow this use-case, we introduce a new concept: the *value-set*. The value-set is a standard way manipulate the underlying values. Through type-erasure, we can either manipulate a known underlying value with native facilities (near-zero overhead) or fallback on a virtual call that may report an error or callback user-provided Python routine to manipulate an unknown user value.

5.3 Hybrid solution's third and final step: the value-set

The *value-set* is an abstraction layer around common operations needed when implementing an image processing algorithm such as an addition, a multiplication, a type conversion, getting the maximum etc. It can be defined in C++ as a class template whose parameter is the manipulated type. The following code shows how to define a value-set:

```
template <class T = void>
struct value_set {
  template <class U>
  U cast(T&& v) const { return static_cast<U>(std::forward<T>(v)); }

  T max() const noexcept { return std::numeric_limits<T>::max(); }
  T min() const noexcept { return std::numeric_limits<T>::min(); }
  /* inf, sup, ... */

  T plus(T&& v) const noexcept { return +std::forward<T>(v); }
  T minus(T&& v) const noexcept { return -std::forward<T>(v); }

  T add(T&& l, T&& r) const noexcept { return std::forward<T>(l) + std::forward<T>(r); }
  T sub(T&& l, T&& r) const noexcept { return std::forward<T>(l) - std::forward<T>(r); }
  /* mod, pow, min, max, ... */
};
```

We can see that the default parameter of the class template is `void`. Indeed, we are inspired by what was implemented in the standard library for `std:less` and providing a default (void) specialization in order to improve the usability. The following code shows how to implement this specialization:

```
template <>
struct value_set<void> {
  template <class U, class T>
  U cast(T&& t) const { return static_cast<U>(std::forward<T>); }

  template <class T>
  T max() const noexcept { return std::numeric_limits<T>::max();}
  template <class T>
  T min() const noexcept { return std::numeric_limits<T>::min(); }
  /* ... */

  template <class T, class U>
  auto add(T&& l, U&& r) const noexcept { return std::forward<T>(l) + std::forward<U>(r); }
  template <class T, class U>
  auto sub(T&& l, U&& r) const noexcept { return std::forward<T>(l) - std::forward<U>(r); }
  /* ... */
};
```

The template parameter is shifted from the class to the member functions. It is also important to note that the member function are not static, which requires to instantiate the `value-set` before using it. It may sound like a disadvantage at first glance but it can be turned into an advantage later on. Indeed, this design allows a subclass to hold member variables which will be crucial.

Now that we have designed how our value-set is intended to work, we can deduce that an image is able to provide its own value-set. Indeed, an image knows what values it holds and thus is able to instantiate the proper value-set corresponding to this type. The member function returning the value-set in the class template ndimage<T, D> is then implemented as follow:

```
template <class T, std::size_t D>
class ndimage {
  /* ... */
  auto get_value_set() const noexcept {
    return value_set<T>{};
  }
};
```

For the sake of example, we are going to implement the linear stretch algorithm in order to augment the contrast. First this algorithm construct an histogram to get both actual minimum and maximum value in the image. Then the algorithm get the maximum and minimum values possible in the space, construct a ratio from these informations and then apply this ratio to the image. For the sake of simplicity, we restrict our example to mono channel image. Here is how it can be naively implemented:

```
1   template <class T = float, class V, std::size_t D>
2   mln::ndimage<T, D> stretch(mln::ndimage<V, D> img) {
3     // histogram
4     auto hist = std::vector<std::size_t>{std::numeric_limits<V>::max()+1, 0};
5     mln::for_each(img, [&hist](V v){ ++hist[v]; });
6
7     // construct ratio
8     auto [m, M] = std::minmax_element(begin(hist), end(hist));
9     double min = (not std::is_floating_point_v<V>) ? static_cast<double>(std::numeric_limits<V>::min()) : 0.0
10    double max = (not std::is_floating_point_v<V>) ? static_cast<double>(std::numeric_limits<V>::max()) : 1.0
11    double ratio = (max - min) / (M - m);
12
13    // construct and apply scaling functor
14    auto scale_fn = [m, x, r](double v) -> V { return static_cast<V>(x + (v - m) * r); };
15    return mln::transform(img, scale_fn);
16  }
```

Now, we can see line 4 that we are gathering the maximum value of the input image's space. Then, line 9 and 10, we are gathering the maximum and minimum value of the input image's space. Finally, at lines 11 and 14, we are doing computation mixing input image's type and resulting image's type. In order to be agnostic from the way those computations are done, we can rewrite our algorithm using value-sets as in the following code:

```
template <class T = float, class V, std::size_t D>
mln::ndimage<T, D> stretch(mln::ndimage<V, D> img) {
  auto img_vs = img.get_value_set();
  aut default_vs = value_set<>{}; // void specialization

  // histogram
  auto hist = std::vector<std::size_t>{img_vs.max()+1, 0};
  mln::for_each(img, [&hist](V v){ ++hist[v]; });

  // construct ratio
  auto [m, M] = std::minmax_element(begin(hist), end(hist));
  double min = (not std::is_floating_point_v<V>) ?
                 img_vs.template cast<double>(img_vs.min()) : 0.0;
  double max = (not std::is_floating_point_v<V>) ?
                 img_vs.template cast<double>(img_vs.max()) : 1.0;
  // equiv to (max - min) / (M - m);
  double ratio = default_vs.div(default_vs.sub(max, min), default_vs.sub(M, m));

  // construct and apply scaling functor
  auto scale_fn = [m, x, r, default_vs](double v) -> V {
    // equiv to static_cast<V>(x + (v - m) * r)
    return default_vs.template cast<V>(default_vs.add(x, default_vs.mult(default_vs.sub(v, m), r)));
  };
  return mln::transform(img, scale_fn);
}
```

Despite loosing a little bit of expressivity (calling explicit function such as `vs.mult(..., ...)`) we are now completely agnostic from the underlying value-type when doing any computation. In this example we are using both the input image's value-set to gather informations about the value space limits as well as the default value-set in order to get the computations right. Now we are able to write an algorithm independently from its underlying type on the C++ side. This feat enables one fundamental feature: type-injection from Python. Indeed, it is now possible to provide a value-set from Python. This feat is realized simply by specializing the base value-set class over the type `pybind::object` which is the generic way to refer to a non-trivially-convertible Python type. This specialization is able to call the operators on any input `pybind11::object` by using a value-set coming from Python at the construction of the image. Here is how this value-set specialization looks like on C++ side:

```
template <>
struct value_set<pybind11::object> {
value_set(pybind11::object python_vs_instance)
  : vs_instance_(python_vs_instance)
{}

```

```
7  template <typename U>
8  pybind11::object cast(pybind11::object v) const {
9    return static_cast<U>(vs_instance_.attr("cast")(v, get_python_type<U>()));
10 }
11
12 pybind11::object max() const {
13   return vs_instance_.attr("max")();
14 }
15 pybind11::object min() const {
16   return vs_instance_.attr("min")();
17 }
18 /* ... */
19
20 pybind11::object add(pybind11::object l, pybind11::object r) const {
21   return vs_instance_.attr("add")(l, r);
22 }
23 pybind11::object sub(pybind11::object l, pybind11::object r) const {
24   return vs_instance_.attr("sub")(l, r);
25 }
26 /* ... */
27
28 private:
29   pybind11::object vs_instance_;
30 };
```

In this code we can clearly see that line 27 we are storing our Python's value-set instance into our class. This is possible due to the fact that our value-set abstraction is not providing static class function but member function. Hence, it is possible to offload the work of the value-set to a member variable at lines 11, 14, 19 and 22 that will call the Python's value-set and get the wanted result. Also, at line 7 we use multiples techniques at once to get the correct resulting cast from a Python type. First we call a function `get_python_type` that will return a string containing the python-compatible representation of the resulting type we want to cast the variable into. This function can be implemented with the C++ facilites contained in the `typeinfo` header such as the `typeid` operator and the `std::type_index` helper class as in the following code:

```
#include <cinttypes>
#include <string>
#include <typeinfo>

template <class U>
std::string get_python_type() {
  // C++ type -> Python type
  static std::unordered_map<std::type_index, std::string> type_names {
    { std::type_index(typeid(bool{})),          "bool"  },
    { std::type_index(typeid(int8_t{})),        "int"   },
    { std::type_index(typeid(int16_t{})),       "int"   },
    { std::type_index(typeid(int32_t{})),       "int"   },
    { std::type_index(typeid(int64_t{})),       "int"   },
```

```
    { std::type_index(typeid(uint8_t{})),       "int"   },
    { std::type_index(typeid(uint16_t{})),      "int"   },
    { std::type_index(typeid(uint32_t{})),      "int"   },
    { std::type_index(typeid(uint64_t{})),      "int"   },
    { std::type_index(typeid(float{})),         "float" },
    { std::type_index(typeid(double{})),        "float" },
    { std::type_index(typeid((char*){})),       "str"   },
    { std::type_index(typeid((const char*){})), "str"   },
    { std::type_index(typeid(std::string{})),   "str"   }
  };
  return type_names[std::type_index(typeid(U{}))];
}
```

This code perform the conversion between the type information extracted from `U` with `typeid` which is compiler specific and the corresponding Python type in order to very easily perform the type-cast on the Python side.

On this particular matter, the user will find a Python abstract class to implement in order for his value-set to be usable by the library. This abstract class is defined by the following Python code:

```python
from abc import ABC, abstractmethod
from typing import Any
import math, importlib

class AbstractValueSet(ABC):

  @abstractmethod
  def cast(self, value: Any, type_): pass
    if type_ in ["int", "float", "bool", "str"]:
      module = importlib.import_module('builtins')
      cls = getattr(module, type_)
      return cls(value)
    else:
      raise ValueError()

  @abstractmethod
  def max(self): return math.inf

  @abstractmethod
  def min(self): return -math.inf

  # ...

  @abstractmethod
  def add(self, lhs: Any, rhs: Any) -> Any: return lhs + rhs

  @abstractmethod
  def sub(self, lhs: Any, rhs: Any) -> Any: return lhs - rhs

  # ...
```

This abstract class provide a facility to cast a value into a given type from its representation as a string. It also provides default / standard way of computing values. Those methods needs to be overridden by a child class as they are all tagged with the `@abstractmethod` attribute.

Now, let us make our own custom Python data structure containing a value. Let us name our class `MyStruct` as in the following code:

```
from typing import Any
class MyStruct:
  v_: Any
  def __init__(self, v: Any): self.v_ = v
  def getV(self) -> Any:      return self.v_
  def setV(self, v: Any):     self.v_ = v
```

Now we want to use this custom structure in an image we pass to the C++ library. The following Python code will not work:

```
img = np.array(
  [MyStruct(1), MyStruct(2), MyStruct(6.5), MyStruct(3.14)],
  ndmin=1)
pln_img = pln.ndimage(img, is_multichannel=false)
```

Indeed, the image's value-type is a `pybind11::object` which requires the C++ code to fallback on the corresponding value-set specialization. However, in order to construct a value-set of that specialization, we are missing a parameter: the `pybind11::object` value-set offloading the work to Python. The next step is then to declare our custom value-set on Python side shown on the following code:

```
from typing import Any

class MyValueSet(AbstractValueSet):
  def get_MyStruct_val__(self, v: Any):
    return v.getV() if isinstance(v, MyStruct) else v

  def cast(self, value: Any, type_):
    return super().cast(self.get_MyStruct__(value), type_)

  def max(self): return super().max()

  def min(self): return super().min()

  def add(self, lhs: Any, rhs: Any) -> Any:
    return MyStruct(super().add(self.get_MyStruct__(lhs), self.get_MyStruct__(rhs)))

  def sub(self, lhs: Any, rhs: Any) -> Any:
    return MyStruct(super().sub(self.get_MyStruct__(lhs), self.get_MyStruct__(rhs)))
```

Now it is possible to write the following code:

```
  img = np.array(
    [MyStruct(1), MyStruct(2), MyStruct(6.5), MyStruct(3.14)],
    ndmin=1)
  pln_img = pln.ndimage(img, is_multichannel=false, value_set=MyValueSet())
```

And on the C++ side there are just small trivial adaptations to do to forward the `pybind11::object` to the `ndimage_from_buffer` function so that it is then correctly forwarded into the resulting `mln::ndbuffer_image`, thus accessible from any algorithms. There is another way of achieving the exact same result which consist of having a concrete value-set Python class inheriting an abstract value-set C++ class. This is rendered possible by using a trampoline on the C++ side to define a special C++ class (with macros provided by pybind). Afterwards, it is possible to define a Python class in Python code inheriting from the trampoline intermediate C++ class. Then the user implement the pure virtual member function with Python code. Thanks to polymorphism, it is then possible to pass this child class back to a C++ function as if it was the C++ parent class. Whichever solution is selected, the performances remain equally bad as Python code do the work in both case.

Indeed, While it works and enables the user to construct `NumPy.array` of custom Python type and pass them to the library with the corresponding value-set for it to "just work", the performance is greatly impacted. As a matter of fact, the computation is no longer done on the C++ side with optimized, vectorized instructions. Instead, a callback to Python is done in order to get the result. It is important that the user keep in mind that custom python types are be supported by the library by providing a value-set at the cost that the resulting performances will literally be blown away. This may be sufficient for prototyping and tinkering however the user must consider implementing his own type on the C++ side when time comes to write production code.

5.4 Performances & overhead

Performance-wise, the hybrid solution aims at being very competitive when comparing to the other "standard" libraries. We want to compare to *OpenCV* but also *scikit-image* which are both widely use in image processing. For our benchmark, we are simply calling a dilation on a sample image whose data are randomized but kept identical for all the libraries. In order to conduct this benchmark, we use the Python *timeit* module to evaluate the calls. Here is the code which generate the randomized image for the benchmarks:

```python
import numpy as np
import random as rnd

def setup_test_img():
  sizes = {"width": 3138, "height": 3138}  # 10Mo
  number = 100
```

```
  percent = 20
  ref = np.zeros((sizes["width"], sizes["height"]), dtype="uint8")
  rnd.seed(42)
  for x in range(0, sizes["width"]):
    for y in range(0, sizes["height"]):
      ref[x, y] = rnd.randint(0, 255)
  return ref
```

Now that our base image is setup, let us what we are going to mesure. We want to first compare the dilation algorithm relative to the radius of the structuring element (disc or rectangle), as well as distinguish the shapes.

For OpenCV, the benchmarked functions will be as follow:

```
import cv2

def bench_cv2_disc(ref, radius):
  disc = cv2.getStructuringElement(
    cv2.MORPH_ELLIPSE, (radius*2+1, radius*2+1))
  cv2.dilate(ref, disc, iterations=1)

def bench_cv2_rect(ref, width, height):
  disc = cv2.getStructuringElement(
    cv2.MORPH_RECT, (rect_width, rect_height))
  cv2.dilate(ref, disc, iterations=1)
```

Now, for scikit-image, the benchmarked functions will be as follow:

```
import skimage.morphology as skimorph

def bench_sckimage_disc(ref, radius):
  disc = skimorph.disk(radius, dtype="uint8")
  skimorph.dilation(ref, disc)

def bench_sckimage_rect(ref, width, height):
  rect = skimorph.rectangle(width, height, dtype="uint8")
  skimorph.dilation(ref, rect)
```

And finally, for Pylena, we will use benchmark the following code:

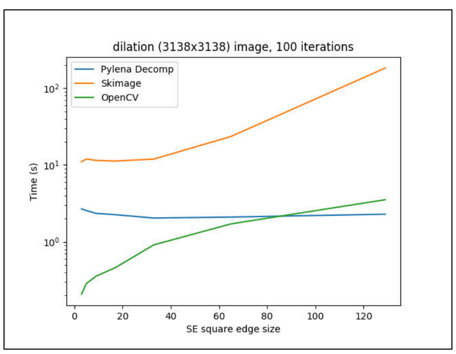
```
import Pylena as pln

def bench_pylena_rect(ref, width, height):
  rect = pln.se.rect2d(width, height)
  pln.morpho.dilate(ref, rect)

def bench_pylena_disc(ref, radius):
  disc = pln.se.disc(float(radius))
  pln.morpho.dilate(ref, disc)
```

The resulting performance are shown in fig. 5.5. We can see notably that for tiny structuring elements (small radius), OpenCV is very fast thanks to hand-made optimization in the core code. Scikit-image which rely on NumPy and then SciPy is consistently slower than both Pylena and OpenCV. It also remains stable only for a rectangle structuring element. For a disc, the execution time grows alongside the radius of the disc. OpenCV does not remain stable the larger the structuring element is, both for a rectangle and a disc. This may be due to the incremental nature of the decomposition of the structuring element algorithm which decompose it in smaller 3x3 structuring elements. However, OpenCV remains faster than Scikit-image. This algorithm remain slow for large structuring elements. Finally, Pylena is very stable both for a rectangle and a disc. It also is consistently faster than Scikit-image image in both cases. Compared to OpenCV, it may be slower for very small structuring elements in the case of a disc but only gets faster than OpenCV for medium sized rectangle shaped structuring elements. We conclude that the strict decomposition algorithm performed by Pylena allows the user to have very stable performances accross the mathematical morphology algorithms.

Rectangle



Disc

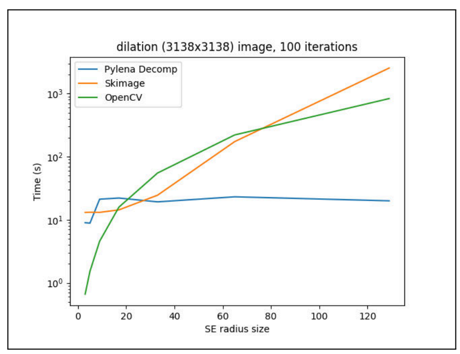


Figure 5.5: Benchmark results: OpenCV vs. Scikit-image vs. Pylena (in a dilation).

Indeed, the fig. 5.6 shows the performance difference with the Pylena library when this one does not the decomposability of its structuring elements. The algorithm using a non-decomposable structuring element has its performances heavily impacted. Also, the algorithm is no longer stable and grows slower and slower with the radius of the structuring elements.

Rectangle Disc

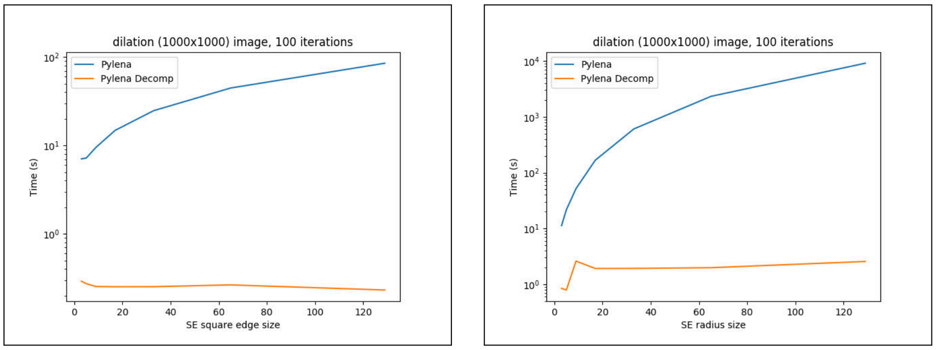


Figure 5.6: Benchmark results: Pylena structuring elements decomposable vs. non-decomposable (in a dilation).

5.5 JIT-based solutions: pros. and cons.

Our hybrid solution certainly has advantages but the huge disadvantage is the slowness of injecting our own types from the Python side. There exists another solution that this thesis did not have the opportunity to study in-depth. This solution is based on a known technology: the Just-In-Time (JIT) compilation which has been previously illustrated in fig. 5.3. Indeed, it is a technology already used by interpreted languages such as Java or PHP to generate on-the-fly native and optimized machine code for the section of the source code that is considered "hot" by the interpreter. A source code is "hot" when it is executed a lot: the end-user would gain paying the compilation time once to have this code executed faster several times later on. When applying this strategy to our problematic, it would mean that the user must be able to compile native machine code from the templated generic C++ code by injected the requested type when it is used. Such an operation shift heavily the burden on the user and it is well-known that compiling C++ code is notably *complicated* and *slow*. In addition, the library needs to be able to auto-generate python-binding once the code is compiled. There are several solutions to achieve this process.

The first solution is to basically use system call to the compilers to actually *compile* C++ code once the templated types are known and explicitly instanciated in the source code. This solution requires careful code-generation design and that the user actually possess a compiler on his computer. Furthermore, the user must resolve all the library dependencies, such as *freeim-*

age for IO etc. This solution was engineered in the library [55]. Indeed, each time the user declared a new automata in his jupyter notebook, corresponding source code is compiled in the foreground and then cached. It is a very perilous solution to implement when the final execution environment (OS, installed software) is not well-known in advance. Nowadays, the issue may be lesser, however, it still requires to maintain both the library and the container solution to use it.

The second solution is to use Cython [42]. It is a transpiling infrastructure which transform a Python source code directly into C-language source code so that it can be compiled by a standard C compiler just by linking against the Python/C API. This remove the burden of writting the careful code-generation routine, system-calls to the C++ compiler and removes the need to resolve all the dependencies. This infrastructure takes care of everything for the user. Also, by transpiling it into C code, it is faster because a C compiler is faster than a C++ compiler. The big issue here is that C code has no support for templates. This means that instead of having to compile instanciated C++ templated code directly, we need to write some glue code to call templated C++ code from the generated C code from Python. Sadly the main issue here is not solved, just shifted.

The third solution consists in relying on recent projects that are all relying on the LLVM infrastructure. We can notably note Autowig [75], Cppyy [68] and Xeus-cling [96]. Autowig has in-house code based on LLVM/Clang to parse C++ code in order to generate and compile a Swig Python binding using the Mako templating engine. Autowig, coupled with Cython would permit the user to, for instance, generate C code related to a custom Python structure. Then a simple call to Autowig will parse the C code and inject it into the C++ library to generate the appropriate bindings for the user. As for Cppyy, it is based on LLVM/Cling, a C++ interpreter, and can directly interpret C++ code from a python string. This allow for easy injection of custom types, be they in Python code (transpiled with Cython) or C++ code (directly interpreted by Cling). Afterwards, the infrastructure generates the appropriate binding from the templated C++ library for the injected type. Finally, Xeus-cling is a ready-to-use jupyter kernel and allow the usage of C++ code directly from within a notebook. This completely bypass the need of a Python binding in the first place and allow the user to use the library from within the notebook as if he was using a Python library. However all those infrastructure come with a hefty cost in term of binary size. Indeed, a C++ compiler is not small and embarking it alongside the image processing library can easily impact greatly the final binary. Without the LLVM infrastructure the binary may weight around 3MB. With the LLVM infrastructure, the binary weight at the bare minimum 50MB. Also, these solutions may not be immediately faster. Indeed, when prototyping back and forth with a variety of types, the user may not be eager to wait for long compilations times each time he is testing with a an iteration of his work. Despite those facts, those solutions offers great avenue of research for the future and the author is eager to thread those paths.

Part III

Continuation

Chapter 6

Conclusion

Through concrete benchmarks and examples, we have shown how to leverage genericity nowadays without slowing down the performances. There are several types of genericity which have been presented, as well as several widely used implementation of them in the industry. Our take offer a new approach to reach the goal of having one code for several algorithm, one algorithm for several image types. Furthermore, we introduce meta-algorithms (canvas) that are based on behavior patterns known once the image type is known. We also show how C++ template metaprogramming techniques allow not to impact the performances despites the indirections induced. Finally, we show how an approach based on properties (on image type as well as on external data such as structuring elements) can be beneficial to introduce customization points to take advantage of opportunities to increase performances. When coupling both properties and algorithm canvas, it becomes standard for a user to write efficient and generic algorithm by default.

It is also shown how we were able to abstract two of the three main families of algorithms. First are point-wise algorithms that can all be expressed through the *views.Views* enables streamlining the writing process of image processing pipelines so that it is shorter, efficient and expressive. Second are local algorithm whose problematics (border management, structuring elements, pass number) can all be abstracted away behind a canvas hiding the complexity and taking advantage of opportunities to increase performance for you.

The solutions presented in this paper do have some disadvantages, such as the readability when using local algorithm canvas, code-bloat due to heavy instantiation of C++ templates especially in the views and finally the lack of availability to the dynamic (prototyping) world by default. Indeed, C++

metaprogramming needs to be compiled at the time of its usage preventing direct link with dynamic languages such as python. There is on-going work to introduce dynamic dispatch at some key points to reduce code size without impairing the performance. For instance, a dynamic dispatch to select the correct version of an algorithm has much less impact than a dynamic dispatch when accessing the value of a pixel. Further work is required to improve the static-dynamic bridge and bring the capabilities of the techniques presented in this paper to the dynamic world, and in particular, python which is vastly used in the image processing world.

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 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.

The main issue, if we consider having an effective Python binding as our future main objective, is intrinsic to the way generic C++ code works. To be able to bind Python and C++, we need a compiled binary that will be called from Python. However, generic code is, by essence, *header-only*. That means an abstraction layer that instantiate our generic code with predefined types is needed to generate the binary code required by Python. However this induces two disadvantages. First, there is no more *zero-overhead* as

[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

the abstraction layer based on virtual dispatch will impact performances. Second, it will be very hard to consider injecting new types from the Python side into the library.

A solution to the last disadvantage would be to consider solutions that compiles our C++ code on the fly when we know the types injected from the Python side. There already exists a tool based on Clang/LLVM that makes this integration relatively feasible. Namely there is Pythran [65], an ahead of time compiler that compiles annotated Python code into optimized native code (with SIMD instructions). This would allow the injection of Python code into our C++ at runtime. Henceforth this would provide a solution to inject new types defined on python side into the already compiled C++ binary code and have the virtual dispatch somehow use it.

Another approach would be to consider a solution using another very promising tool: cppyy [68] which is a an automatic Python/C++ binding generator designed for large scale programs. The C++ interpreter is based on Cling [2] (an interpreter based on Clang/LLVM and maintained by CERN). This would allow us to compile our generic C++ code directly from our Python code and have our bindings automatically generated. As a future work, we will implement both approaches to benchmark them in order to measure which one is the most effective for the many use-cases we have.

During our 2^{nd} year of work on our Ph.D. we have broaden our horizon and seen many issues as well as studied potential leads of solutions.

For traversing of an image, a satisfactory solution has been found that does not hinder performances, though some small issues remain to be studied later.

Concerning the way to handle an image's borders when processing a local algorithm, we have studied the question in depth and exercised the presented design against different kind of problem such as structuring elements which are static, dynamic or adaptive. All of these are within scope and handled by the solution presented in this paper.

The static dynamic bridge proved to be a big challenge as there was very little experience feedback from existing solution experimented by other libraries. It took a lot of incremental trial and error to design the hybrid solution that would perform with satisfactory performances for common use cases without duplicating code. This question still remains open as we would like to study the usability of bringing in a C++ interpreter such as Cppyy [68]

[2] https://root.cern.ch/cling

(based on clang) to dynamically generate python bindings on the fly when those are missing for specific types. This would allows us to inject user type from python into the C++ core library code. The solution of code generation through tools like AsmJIT [44] also remains open for future work.

Finally the reflection about how we can bring parallelism and more generally heterogeneous computing into the library brought us to study different approaches, different paradigms and think about how our data structure operate with the on-going computation. We attempted to bring a solution with the algorithm canvas though we did not took advantage of it for the moment. For instance, we do not dispatch CUDA kernel yet. This question remains open and can still be studied further.

To conclude, the priority in the future will be put into having a first stable release of the library. Indeed, we want to provide the library for the students so that they can give feedbacks on its usability. We will study both the usability of the C++ aspect and the python aspect.

- Synthèse générale
- Réponse à la problématique d'Introduction
- confrontation à d'autres travaux de recherche ayant donné naissance à une bibliothèque de TI
- Ouvertures, perspectives, limites
  - continuité JIT
  - ce qu'il reste à faire

Part IV

Appendices

Appendix A

Bibliography

Bibliography

[1] David R. Musser and Alexander A. Stepanov. "Generic programming". In: *Intl. Symp. on Symbolic and Algebraic Computation*. Springer. 1988, pp. 13–25.

[2] Gerhard X. Ritter, Joseph N. Wilson, and Jennifer L. Davidson. "Image algebra: An overview". In: *Computer Vision, Graphics, and Image Processing* 49.3 (1990), pp. 297–331.

[3] Marcel van Herk. "A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels". In: *Pattern Recognition Letters* 13.7 (1992), pp. 517–521. ISSN: 0167-8655. DOI: https://doi.org/10.1016/0167-8655(92)90069-C. URL: http://www.sciencedirect.com/science/article/pii/016786559290069C.

[4] David R. Musser and Alexander A. Stepanov. "Algorithm-oriented generic libraries". In: *Software: Practice and Experience* 24.7 (1994), pp. 623–642. DOI: 10.1002/spe.4380240703. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380240703. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380240703.

[5] Bjarne Stroustrup. *The Design and Evolution of C++*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1994. ISBN: 0-201-54330-3.

[6] Michael Hanus, Herbert Kuchen, and Juan Jose Moreno-Navarro. "Curry: A Truly Functional Logic Language". In: *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*. Vol. 95. 1995, pp. 95–107.

[7] Todd Veldhuizen. "Expression templates". In: *C++ Report* 7.5 (1995), pp. 26–31.

[8] David M Beazley et al. "SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++." In: *Tcl/Tk Workshop*. Vol. 43. 1996, p. 74.

[9] ISO. *ISO/IEC 14882:1998: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Sept. 1998, p. 732. URL: https://www.iso.org/standard/25845.html.

[10] Bjarne Stroustrup. "An overview of the C++ programming language". In: *Handbook of Object Technology* (1999).

[11] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[12] James C. Dehnert and Alexander Stepanov. "Fundamentals of Generic Programming". In: *Generic Programming*. Vol. 1766. LNCS. Springer. 2000, pp. 1–11.

[13] Alexandre Duret-Lutz. "Olena: a component-based platform for image processing, mixing generic, generative and OO programming". In: *symposium on Generative and Component-Based Software Engineering, Young Researchers Workshop*. Vol. 10. Citeseer. 2000.

[14] Thierry Géraud et al. "Obtaining genericity for image processing and pattern recognition algorithms". In: *Proceedings of the 15th International Conference on Pattern Recognition (ICPR)*. Vol. 4. Barcelona, Spain: IEEE Computer Society, Sept. 2000, pp. 816–819.

[15] Øyvind Kolås and et al. *Generic Graphic Library*. Available at http://www.gegl.org. 2000.

[16] Ullrich Köthe. "STL-Style Generic Programming with Images". In: *C++ Report Magazine* 12.1 (2000). https://ukoethe.github.io/vigra, pp. 24–30.

[17] Todd L. Veldhuizen. "Blitz++: The Library that Thinks it is a Compiler". In: *Advances in Software Tools for Scientific Computing*. Ed. by Hans Petter Langtangen, Are Magnus Bruaset, and Ewald Quak. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 57–87. ISBN: 978-3-642-57172-5.

[18] Andrew Lumsdaine Jeremy Siek Lie-Quan Lee. *The Boost Graph library*. 2001. URL: http://cds.cern.ch/record/1518180/files/0201729148_TOC.pdf.

[19] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. 2001. URL: http://www.scipy.org/.

[20] Jérôme Darbon, Thierry Géraud, and Alexandre Duret-Lutz. "Generic implementation of morphological image operators". In: *Mathematical Morphology, Proceedings of the 6th International Symposium (ISMM)*. Sydney, Australia: CSIRO Publishing, Apr. 2002, pp. 175–184.

[21] Delores M Etter, David C Kuncicky, and Douglas W Hull. *Introduction to MATLAB*. Prentice Hall, 2002.

[22] David Vandevoorde and Nicolai M Josuttis. *C++ Templates: The Complete Guide, Portable Documents*. Addison-Wesley Professional, 2002.

[23] Nicolas Burrus et al. "A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming". In: *Proceedings of the Workshop on Multiple Paradigm with Object-Oriented Languages (MPOOL)*. Anaheim, CA, USA, Oct. 2003.

[24] ISO. *ISO/IEC 14882:2003: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Oct. 2003, p. 757. URL: https://www.iso.org/standard/38110.html.

[25] Todd L. Veldhuizen. *C++ Templates are Turing Complete*. Tech. rep. 2003.

[26] Jérôme Darbon, Thierry Géraud, and Patrick Bellot. "Generic algorithmic blocks dedicated to image processing". In: *Proceedings of the ECOOP Workshop for PhD Students*. Oslo, Norway, June 2004.

[27] Jacques Froment. *MegaWave2 user's guide*. 2004.

[28] Stewart Taylor. *Intel integrated performance primitives*. Intel Press, 2004.

[29] Guntram Berti. "GrAL–the Grid Algorithms Library". In: *Future Generation Computer Systems* 22.1-2 (2006), pp. 110–122.

[30] Lubomir Bourdev and Hailin Jin. *Boost Generic Image Library*. Adobe stlab. Available at https://stlab.adobe.com/gil/index.html. 2006.

[31] Thierry Géraud. *Advanced Static Object-Oriented Programming Features: A Sequel to SCOOP*. http://www.lrde.epita.fr/people/theo/pub/olena/olena-06-jan.pdf. Jan. 2006.

[32] Travis Oliphant. *NumPy: A guide to NumPy*. USA: Trelgol Publishing. 2006. URL: http://www.numpy.org/.

[33] Bjarne Stroustrup. "Evolving a Language in and for the Real World: C++ 1991-2006". In: *Proc. of the 3rd ACM SIGPLAN Conf. on History of Programming Languages*. Vol. 4. San Diego, California, 2007, pp. 1–59. ISBN: 9781595937667. DOI: 10.1145/1238844.1238848. URL: https://doi.org/10.1145/1238844.1238848.

[34] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[35] Thierry Géraud and Roland Levillain. “Semantics-Driven Genericity: A Sequel to the Static C++ Object-Oriented Programming Paradigm (SCOOP 2)”. In: *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL)*. Paphos, Cyprus, July 2008.

[36] Roland Levillain, Thierry Géraud, and Laurent Najman. “Milena: Write Generic Morphological Algorithms Once, Run on Many Kinds of Images”. In: *Mathematical Morphology and Its Application to Signal and Image Processing – Proceedings of the Ninth International Symposium on Mathematical Morphology (ISMM)*. Ed. by Michael H. F. Wilkinson and Jos B. T. M. Roerdink. Vol. 5720. Lecture Notes in Computer Science. Groningen, The Netherlands: Springer Berlin / Heidelberg, Aug. 2009, pp. 295–306.

[37] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, June 2009.

[38] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. Available at http://eigen.tuxfamily.org. 2010.

[39] Roland Levillain, Thierry Géraud, and Laurent Najman. “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)*. Hong Kong, Sept. 2010, pp. 1941–1944.

[40] Roland Levillain, Thierry Géraud, and Laurent Najman. “Writing Reusable Digital Geometry Algorithms in a Generic Image Processing Framework”. In: *Proceedings of the Workshop on Applications of Digital Geometry and Mathematical Morphology (WADGMM)*. Istanbul, Turkey, Aug. 2010, pp. 96–100. URL: http://mdigest.jrc.ec.europa.eu/wadgmm2010/.

[41] Peter Pirkelbauer, Yuriy Solodkyy, and Bjarne Stroustrup. “Design and evaluation of C++ open multi-methods”. In: *Science of Computer Programming* 75.7 (2010). Generative Programming and Component Engineering (GPCE 2007), pp. 638–667. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2009.06.002. URL: http://www.sciencedirect.com/science/article/pii/S016764230900094X.

[42] S. Behnel et al. “Cython: The Best of Both Worlds”. In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 31–39. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.118.

[43] ISO. *ISO/IEC 14882:2011: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Sept. 2011, p. 1338. URL: https://www.iso.org/standard/50372.html.

[44] P Kobalicek. *asmjit-complete x86/x64 JIT assembler for C++ language*. 2011.

[45] Roland Levillain. “Towards a Software Architecture for Generic Image Processing”. PhD thesis. Marne-la-Vallée, France: Université Paris-Est, Nov. 2011.

[46] Roland Levillain, Thierry Géraud, and Laurent Najman. “Une approche générique du logiciel pour le traitement d’images préservant les performances”. In: *Proceedings of the 23rd Symposium on Signal and Image Processing (GRETSI)*. In French. Bordeaux, France, Sept. 2011.

[47] Jacques Froment. “MegaWave”. In: *IPOL 2012 Meeting on Image Processing Libraries*. France, June 2012. URL: https://hal.archives-ouvertes.fr/hal-00907378.

[48] Thierry Géraud. “Outil logiciel pour le traitement d’images: Bibliothèque, paradigmes, types et algorithmes”. In French. Habilitation Thesis. Université Paris-Est, June 2012.

[49] Klaus Iglberger. *Blaze C++ Linear Algebra Library*. https://bitbucket.org/blaze-lib. 2012.

[50] Klaus Iglberger et al. “Expression Templates Revisited: A Performance Analysis of Current Methodologies”. In: *SIAM Journal on Scientific Computing* 34(2) (2012), pp. C42–C69.

[51] Roland Levillain, Thierry Géraud, and Laurent Najman. “Writing Reusable Digital Topology Algorithms in a Generic Image Processing Framework”. In: *WADGMM 2010*. Ed. by Ullrich Köthe, Annick Montanvert, and Pierre Soille. Vol. 7346. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2012, pp. 140–153.

[52] S. Matuska, R. Hudec, and M. Benco. “The comparison of CPU time consumption for image processing algorithm in Matlab and OpenCV”. In: *2012 ELEKTRO*. May 2012, pp. 75–78. DOI: 10.1109/ELEKTRO.2012.6225575.

[53] David Tschumperlé. “The CImg Library”. In: *IPOL 2012 Meeting on Image Processing Libraries*. Cachan, France, June 2012, 4 pp. URL: https://hal.archives-ouvertes.fr/hal-00927458.

[54] V. Vassilev et al. “Cling – The New Interactive Interpreter for ROOT 6”. In: vol. 396. 5. IOP Publishing, Dec. 2012, p. 052071. DOI: 10.1088/1742-6596/396/5/052071. URL: https://iopscience.iop.org/article/10.1088/1742-6596/396/5/052071/pdf.

[55] Akim Demaille et al. “Implementation Concepts in Vaucanson 2”. In: *Implementation and Application of Automata*. Ed. by Stavros Konstantinidis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 122–133. ISBN: 978-3-642-39274-0.

[56] Agner Fog. *C++ vector class library*. http://www.agner.org/optimize/vectorclass.pdf. 2013.

[57] Hans J. Johnson et al. *The ITK Software Guide*. Third. *In press*. Kitware, Inc. 2013. URL: http://www.itk.org/ItkSoftwareGuide.pdf.

[58] Jonathan Ragan-kelley et al. “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines”. In: *PLDI 2013* (2013).

[59] Pierre Estérie et al. “Boost.SIMD: Generic Programming for Portable SIMDization”. In: *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. WPMVP ’14. Orlando, Florida, USA: ACM, 2014, pp. 1–8. ISBN: 978-1-4503-2653-7. DOI: 10.1145/2568058.2568063. URL: http://doi.acm.org/10.1145/2568058.2568063.

[60] Matthieu Garrigues and Antoine Manzanera. “Video++, a modern image and video processing C++ framework”. In: *Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE. 2014, pp. 1–6.

[61] ISO. *ISO/IEC 14882:2014: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Dec. 2014, p. 1358. URL: https://www.iso.org/standard/64029.html.

[62] Roland Levillain et al. “Practical Genericity: Writing Image Processing Algorithms Both Reusable and Efficient”. In: *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications – Proceedings of the 19th Iberoamerican Congress on Pattern Recognition (CIARP)*. Ed. by Eduardo Bayro and Edwin Hancock.

Vol. 8827. Lecture Notes in Computer Science. Puerto Vallarta, Mexico: Springer-Verlag, Nov. 2014, pp. 70–79.

[63] Stéfan van der Walt et al. "scikit-image: image processing in Python". In: *PeerJ* 2 (June 2014), e453. ISSN: 2167-8359. DOI: 10.7717/peerj.453. URL: https://doi.org/10.7717/peerj.453.

[64] Stéfan van der Walt et al. "scikit-image: image processing in Python". In: *PeerJ* 2 (June 2014), e453. ISSN: 2167-8359. DOI: 10.7717/peerj.453. URL: https://doi.org/10.7717/peerj.453.

[65] Serge Guelton et al. "Pythran: Enabling static optimization of scientific python programs". In: *Computational Science & Discovery* 8.1 (2015), p. 014001.

[66] D Coeurjolly, JO Lachaud, and B Kerautret. *DGtal: Digital geometry tools and algorithms library*. 2016.

[67] Thomas Kluyver et al. "Jupyter Notebooks - a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by Fernando Loizides and Birgit Scmidt. Netherlands: IOS Press, 2016, pp. 87–90. URL: https://eprints.soton.ac.uk/403913/.

[68] W. T. L. P. Lavrijsen and A. Dutta. "High-Performance Python-C++ Bindings with PyPy and Cling". In: *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. Nov. 2016, pp. 27–35. DOI: 10.1109/PyHPC.2016.008.

[69] Conrad Sanderson and Ryan Curtin. "Armadillo: a template-based C++ library for linear algebra". In: *Journal of Open Source Software* 1.2 (2016), p. 26.

[70] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV].

[71] ISO. *ISO/IEC 14882:2017: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Dec. 2017, p. 1605. URL: https://www.iso.org/standard/68564.html.

[72] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11—Seamless operability between C++ 11 and Python*. https://github.com/pybind/pybind11. 2017.

[73] *Background Subtraction*. https://docs.opencv.org/3.4/d1/dc5/tutorial_background_subtraction.html. Dec. 2018.

[74] Edwin Carlinet et al. *Pylena: a Modern C++ Image Processing Generic Library*. EPITA Research and Developement Laboratory. Available at https://gitlab.lrde.epita.fr/olena/pylene. 2018.

[75] Pierre Fernique and Christophe Pradal. "AutoWIG: automatic generation of python bindings for C++ libraries". In: *PeerJ Computer Science* 4 (2018), e149. URL: https://peerj.com/articles/cs-149.pdf.

[76] Thierry Géraud and Edwin Carlinet. *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. URL: https://www.lrde.epita.fr/~theo/talks/geraud.2018.gtgdmm_talk.pdf.

[77] Eric Niebler and Casey Carter. *P1037R0: Deep Integration of the Ranges TS*. https://wg21.link/p1037r0. May 2018.

[78] Michael Wong and Hal Finkel. "Distributed & Heterogeneous Programming in C++ for HPC at SC17". In: *Proceedings of the International Workshop on OpenCL*. IWOCL '18. Oxford, United Kingdom: ACM, 2018, 20:1–20:7. ISBN: 978-1-4503-6439-3. DOI: 10.1145/3204919.3204939. URL: http://doi.acm.org/10.1145/3204919.3204939.

[79] Gordon Brown, Ruyman Reyes, and Michael Wong. "Towards Heterogeneous and Distributed Computing in C++". In: *Proceedings of the International Workshop on OpenCL*. IWOCL'19. Boston, MA, USA: ACM, 2019, 18:1–18:5. ISBN: 978-1-4503-6230-6. DOI: 10.1145/3318170.3318196. URL: http://doi.acm.org/10.1145/3318170.3318196.

[80] Ben Dean. *Identifying Monoids: Exploiting Compositional Structure in Code*. Slides available at https://github.com/boostcon/cppnow_presentations_2019/blob/master/05-09-2019_thursday/Identifying_Monoids_Exploiting_Compositional_Structure_in_Code__Ben_Deane_cppnow_05092019.pdf, Talk available on youtube at https://youtu.be/INnattuluiM. June 2019. URL: https://cppnow2019.sched.com/event/b60160aa659270370279c5acb6196fb6.

[81] Celian Gossec. "Binding a high-performance C++ image processing library to Python". In: *Student LRDE tech reports*. 2019.

[82] PTC. *GraphicsMagick*. Version 6.0. Oct. 1, 2019. URL: https://www.mathcad.com.

[83] Michaël Roynard, Edwin Carlinet, and Thierry Géraud. "An Image Processing Library in Modern C++: Getting Simplicity and Efficiency with Generic Programming". In: *Reproducible Research in Pattern Recognition*. Ed. by Bertrand Kerautret et al. Cham: Springer International Publishing, 2019, pp. 121–137. ISBN: 978-3-030-23987-9.

[84] The GIMP Development Team. *GIMP*. Version 2.10.12. June 12, 2019. URL: https://www.gimp.org.

[85] Anaconda, Inc. *Anaconda*. Version 2020.11. Nov. 19, 2020. URL: https://anaconda.com.

[86] GraphicsMagick Group. *GraphicsMagick*. Version 1.3.36. Dec. 26, 2020. URL: https://http://www.graphicsmagick.org.

[87] ISO. *ISO/IEC 14882:2020: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Dec. 2020, p. 1853. URL: https://www.iso.org/standard/79358.html.

[88] MathWorks. *MATLAB*. Version R2020b. Sept. 22, 2020. URL: https://fr.mathworks.com/products/matlab.html.

[89] Scilab Enterprises. *Scilab*. Version 6.1.0. Feb. 25, 2020. URL: https://www.scilab.org.

[90] wolfram Research. *Mathematica*. Version 12.2. Dec. 16, 2020. URL: https://www.wolfram.com/mathematica/.

[91] Adobe. *Adobe Photoshop*. Version 22.2. Feb. 9, 2021. URL: https://photoshop.com/fr.

[92] Alex Clark and al. *Pillow*. Version 8.1.2. Mar. 6, 2021. URL: https://python-pillow.org.

[93] Boost. *Boost C++ Libraries*. Version 1.76.0. Apr. 16, 2021. URL: https://www.boost.org/.

[94] Louis Dionne and Bruno Dutra. *Metabench*. Version head. Apr. 1, 2021. URL: https://github.com/ldionne/metabench.

[95] GNU Project. *Octave*. Version 6.2.0. Feb. 20, 2021. URL: https://www.gnu.org/software/octave/index.

[96] QuantStack. *xeus-cling*. Version 0.12.1. Mar. 16, 2021. URL: https://github.com/QuantStack/xeus-cling.

[97] The ImageMagick Development Team. *ImageMagick*. Version 7.0.10. Jan. 4, 2021. URL: https://imagemagick.org.

[98] The PyPi Developement Team. *PyPi*. Version 21.0.1. Jan. 31, 2021. URL: https://pypi.org.

[99] Andrew Sutton Eric Niebler Sean Parent. *Ranges for the Standard Library: Revision 1*. Oct. 2014. URL: https://ericniebler.github.io/std/wg21/D4128.html.

[100] Bjarne Stroustrup and Gabriel Dos Reis. *Concepts - Design choices for template argument checking*. WG21, Oct. 2003. URL: https://wg21.link/n1522.

[101] Bill Seymour. *LWG Papers to Re-Merge into C++0x After Removing Concepts*. WG21, July 2009. URL: https://wg21.link/n2929.

[102] Andrew Sutton. *Working Draft, C++ extensions for Concepts*. WG21, June 2017. URL: https://wg21.link/n4674.

[103] EPITA Research and Developpement Laboratory (LRDE). *The Olena image processing platform*. http://olena.lrde.epita.fr. 2000.

[104] Ville Voutilainen. *Merge the Concepts TS Working Draft into the C++20 working draft*. WG21, June 2017. URL: https://wg21.link/p0724r0.

[105] Eric Niebler and Casey Carter. *Merging the Ranges TS*. WG21, May 2018. URL: https://wg21.link/p0896r1.

[106] Casey Carter and Eric Niebler. *Standard Library Concepts*. WG21, June 2018. URL: https://wg21.link/p0898r3.

[107] Eric Niebler and Casey Carter. *Deep Integration of the Ranges TS*. WG21, May 2018. URL: https://wg21.link/p1037r0.

Appendix B

Concepts & archetypes

B.0.1 Concepts

Index

```
// Index
template <typename Idx>
concept Index = std::signed_integral<Idx>;
```

Value

```
// Value
template <typename Val>
concept Value = std::semiregular<Val>;

// ComparableValue
template <typename RegVal>
concept ComparableValue =
  std::regular<RegVal>;

// OrderedValue
template <typename STORegVal>
concept OrderedValue =
  std::regular<STORegVal> &&
  std::totally_ordered<STORegVal>;
```

Point

```
// Point
template <typename P>
concept Point =
  std::regular<P> &&
  std::totally_ordered<P>;
```

Pixel

```
// Pixel
template <class Pix>
concept Pixel =
  std::is_base_of_v<mln::details::Pixel<Pix>, Pix> &&
  std::copy_constructible<Pix> &&
  std::move_constructible<Pix> &&
  requires {
    typename pixel_value_t<Pix>;
    typename pixel_reference_t<Pix>;
    typename pixel_point_t<Pix>;
  } &&
  std::semiregular<pixel_value_t<Pix>> &&
  Point<pixel_point_t<Pix>> &&
  !std::is_const_v<pixel_value_t<Pix>> &&
  !std::is_reference_v<pixel_value_t<Pix>> &&
  requires(const Pix cpix, Pix pix, pixel_point_t<Pix> p) {
    { cpix.point() } -> std::convertible_to<pixel_point_t<Pix>>;
    { cpix.val() }   -> std::convertible_to<pixel_reference_t<Pix>>;
    { pix.shift(p) };
  };

// WritablePixel
template <typename WPix>
concept WritablePixel =
  Pixel<WPix> &&
  requires(const WPix cpix, pixel_value_t<WPix> v) {
    // Not deep-const, view-semantic.
    { cpix.val() = v };
    // Proxy rvalues must not be deep-const on their assignement semantic (unlike tuple...)
    { const_cast<typename WPix::reference const &&>(cpix.val()) = v };
  };

// OutputPixel
template <typename Pix>
concept OutputPixel = detail::WritablePixel<Pix>;
```

Ranges

```
template <class C>
concept MDCursor =
  std::ranges::detail::forward_cursor<C> &&
  std::ranges::detail::forward_cursor<std::ranges::detail::begin_cursor_t<C>> &&
  requires (C c)
  {
    { c.read() } -> std::ranges::forward_range;
    c.end_cursor();
  };

template <class C>
concept NDCursor = std::semiregular<C> &&
  requires (C c)
  {
    { C::rank } -> std::same_as<int>;
    c.read();
```

```
    c.move_to_next(0);
    c.move_to_end(0);
  };

template <class C>
concept MDBidirectionalCursor = MDCursor<C> &&
  requires (C c)
  {
    c.move_to_prev();
    c.move_to_prev_line();
  };

template <class R>
concept MDRange =
  requires(R r)
  {
    { r.rows() } -> std::ranges::forward_range;
    { r.begin_cursor() } -> MDCursor;
    { r.end_cursor() } -> std::same_as<std::ranges::default_sentinel_t>;
  };

template <class R>
concept MDBidirectionalRange = MDRange<R> &&
  requires (R r)
  {
    { r.rrows() } -> std::ranges::forward_range;
    { r.rbegin_cursor() } -> MDCursor;
    { r.rend_cursor() } -> std::same_as<std::ranges::default_sentinel_t>;
  };

template <class R>
concept mdrange = MDRange<R> || std::ranges::range<R>;

template <class R, class V>
concept output_mdrange = mdrange<R> && std::ranges::output_range<mdrange_row_t<R>, V>;

template <class R>
concept reversible_mdrange = MDBidirectionalRange<R> || std::ranges::bidirectional_range<R>;
```

Domain

```
// Domain
template <typename Dom>
concept Domain =
  mln::ranges::mdrange<Dom> &&
  Point<mln::ranges::mdrange_value_t<Dom>> &&
  requires(const Dom cdom, mln::ranges::mdrange_value_t<Dom> p) {
    { cdom.has(p) }   -> std::same_as<bool>;
    { cdom.empty() }  -> std::same_as<bool>;
    { cdom.dim() }    -> std::same_as<int>;
  };


// SizedDomain
template <typename Dom>
concept SizedDomain =
```

```
  Domain<Dom> &&
  requires(const Dom cdom) {
    { cdom.size() } -> std::unsigned_integral;
  };

// ShapedDomain
template <typename Dom>
concept ShapedDomain =
  SizedDomain<Dom> &&
  requires(const Dom cdom) {
    { cdom.extents() }  -> std::ranges::forward_range;
  };
```

Structuring Element

```
namespace details
{
  template <typename SE>
  concept DynamicStructuringElement =
    requires (SE se) {
    { se.radial_extent() }  -> std::same_as<int>;
    };


  constexpr bool implies(bool a, bool b) { return !a || b; }
}


template <typename SE, typename P>
concept StructuringElement =
  std::convertible_to<SE, mln::details::StructuringElement<SE>> &&
  std::ranges::regular_invocable<SE, P> &&
  std::ranges::regular_invocable<SE, mln::archetypes::PixelT<P>> &&
  requires {
    typename SE::category;
    typename SE::incremental;
    typename SE::decomposable;
    typename SE::separable;
  } &&
  std::convertible_to<typename SE::category, mln::adaptative_neighborhood_tag> &&
  details::implies(std::convertible_to<typename SE::category, mln::dynamic_neighborhood_tag>,
                   details::DynamicStructuringElement<SE>) &&
  requires (SE se, const SE cse, P p, mln::archetypes::PixelT<P> px) {
    { se(p) }         -> std::ranges::forward_range;
    { se(px) }        -> std::ranges::forward_range;
    { cse.offsets() } -> std::ranges::forward_range;

    requires std::convertible_to<std::ranges::range_value_t<decltype(se(p))>, P>;
    requires std::Pixel<std::ranges::range_value_t<decltype(se(px))>>;
    requires std::convertible_to<std::ranges::range_value_t<decltype(cse.offsets())>, P>;
  };

namespace details
{
  template <typename R, typename P>
  concept RangeOfStructuringElement =
```

```
    StructuringElement<std::ranges::range_value_t<R>, P>;
}
```

```
template <typename SE, typename P>
concept DecomposableStructuringElement =
  StructuringElement<SE, P> &&
  std::convertible_to<typename SE::decomposable, std::true_type> &&
  requires(const SE se) {
    { se.is_decomposable() }  -> std::same_as<bool>;
    { se.decompose() }        -> std::ranges::forward_range;
    requires details::RangeOfStructuringElement<decltype(se.decompose()), P>;
  };
```

```
template <typename SE, typename P>
concept SeparableStructuringElement =
  StructuringElement<SE, P> &&
  std::convertible_to<typename SE::separable, std::true_type> &&
  requires(const SE se) {
    { se.is_separable() } -> std::same_as<bool>;
    { se.separate() }     -> std::ranges::forward_range;
    requires details::RangeOfStructuringElement<decltype(se.separate()), P>;
  };
```

```
template <typename SE, typename P>
concept IncrementalStructuringElement =
  StructuringElement<SE, P> &&
  std::convertible_to<typename SE::incremental, std::true_type> &&
  requires(const SE se) {
    { se.is_incremental() } -> std::same_as<bool>;
    { se.inc() }  -> StructuringElement<SE, P>;
    { se.dec() }  -> StructuringElement<SE, P>;
  };
```

Neighborhood

```
template <typename SE, typename P>
concept Neighborhood =
  StructuringElement<SE, P> &&
  requires (SE se, P p, mln::archetypes::PixelT<P> px) {
    { se.before(p) }  -> std::ranges::forward_range;
    { se.after(p) }   -> std::ranges::forward_range;
    { se.before(px) } -> std::ranges::forward_range;
    { se.after(px) }  -> std::ranges::forward_range;

    requires std::convertible_to<std::ranges::range_value_t<decltype(se.before(p))>, P>;
    requires std::convertible_to<std::ranges::range_value_t<decltype(se.after(p))>, P>;
    requires std::Pixel<std::ranges::range_value_t<decltype(se.before(px))>>;
    requires std::Pixel<std::ranges::range_value_t<decltype(se.after(px))>>;
  };
```

Image

```
template <typename I>
concept Image =
  // Minimum constraint on image object
  // Do not requires DefaultConstructible
  std::is_base_of_v<mln::details::Image<I>, I> &&
  std::copy_constructible<I> &&
  std::move_constructible<I> &&
  std::derived_from<image_category_t<I>, forward_image_tag> &&
  requires {
    typename image_pixel_t<I>;
    typename image_point_t<I>;
    typename image_value_t<I>;
    typename image_domain_t<I>;
    typename image_reference_t<I>;
    typename image_concrete_t<I>;
    typename image_ch_value_t<I, mln::archetypes::Value>;
    // traits
    typename image_indexable_t<I>;
    typename image_accessible_t<I>;
    typename image_extension_category_t<I>;
    typename image_category_t<I>;
    typename image_view_t<I>;
  } &&
  Pixel<image_pixel_t<I>> &&
  Point<image_point_t<I>> &&
  Value<image_value_t<I>> &&
  Domain<image_domain_t<I>> &&
  std::convertible_to<pixel_point_t<image_pixel_t<I>>, image_point_t<I>> &&
  std::convertible_to<pixel_reference_t<image_pixel_t<I>>, image_reference_t<I>> &&
  // Here we don't want a convertible constraint as value_type is the decayed type and should really be the s
  std::same_as<pixel_value_t<image_pixel_t<I>>, image_value_t<I>> &&
  std::common_reference_with<image_reference_t<I>&&, image_value_t<I>&> &&
  std::common_reference_with<image_reference_t<I>&&, image_value_t<I>&&> &&
  std::common_reference_with<image_value_t<I>&&, const image_value_t<I>&> &&
  requires(I ima, const I cima, image_domain_t<I> d, image_point_t<I> p) {
    { cima.template ch_value<mln::archetypes::Value>() }
        -> std::convertible_to<image_ch_value_t<I, mln::archetypes::Value>>;
    { cima.concretize() } -> std::convertible_to<image_concrete_t<I>>;
    { cima.domain() }     -> std::convertible_to<image_domain_t<I>>;
    { ima.pixels() }  -> mln::ranges::mdrange;
    { ima.values() }  -> mln::ranges::mdrange;
    requires std::convertible_to<mln::ranges::mdrange_value_t<decltype(ima.pixels())>, image_pixel_t<I>>;
    requires std::convertible_to<mln::ranges::mdrange_value_t<decltype(ima.values())>, image_value_t<I>>;
  };

namespace detail
{
  // WritableImage
  template <typename I>
  concept WritableImage =
    Image<I> &&
    OutputPixel<image_pixel_t<I>> &&
    requires(I ima) {
    { ima.values() }  -> mln::ranges::output_mdrange<image_value_t<I>>;
      // Check Writability of each pixel of the range
      requires OutputPixel<
```

```
                std::common_type_t<
                  mln::ranges::mdrange_value_t<decltype(ima.pixels())>,
                  image_pixel_t<I>>>;
    };
} // namespace detail


// InputImage
template <typename I>
concept InputImage = Image<I>;


// ForwardImage
template <typename I>
concept ForwardImage = InputImage<I>;


// IndexableImage
template <typename I>
concept IndexableImage =
  Image<I> &&
  requires {
    typename image_index_t<I>;
  } &&
  image_indexable_v<I> &&
  requires (I ima, image_index_t<I> k) {
    { ima[k] }  -> std::same_as<image_reference_t<I>>; // For concrete image it returns a const_reference
  };


namespace detail
{
  // WritableIndexableImage
  template <typename I>
  concept WritableIndexableImage =
    WritableImage<I> &&
    IndexableImage<I> &&
    requires(I ima, image_index_t<I> k, image_value_t<I> v) {
      { ima[k] = v }
    };
} // namespace detail

// AccessibleImage
template <typename I>
concept AccessibleImage =
  Image<I> &&
  image_accessible_v<I> &&
  requires (I ima, image_point_t<I> p) {
    { ima(p) }              -> std::same_as<image_reference_t<I>>; // For concrete image it returns a const_reference
    { ima.at(p) }           -> std::same_as<image_reference_t<I>>; // idem
    { ima.pixel(p) }    -> std::same_as<image_pixel_t<I>>; // For concrete image pixel may propagate constness
    { ima.pixel_at(p) } -> std::same_as<image_pixel_t<I>>; // idem
  };


namespace detail
```

```cpp
{
  // WritableAccessibleImage
  template <typename I>
  concept WritableAccessibleImage =
    detail::WritableImage<I> &&
    AccessibleImage<I> &&
    requires(I ima, image_point_t<I> p, image_value_t<I> v) {
      { ima(p) = v };
      { ima.at(p) = v };

      requires OutputPixel<decltype(ima.pixel(p))>;
      requires OutputPixel<decltype(ima.pixel_at(p))>;
    };
} // namespace detail

// IndexableAndAccessibleImage
template <typename I>
concept IndexableAndAccessibleImage =
  IndexableImage<I> &&
  AccessibleImage<I> &&
  requires (const I cima, image_index_t<I> k, image_point_t<I> p) {
    { cima.point_at_index(k) }  -> std::same_as<image_point_t<I>>;
    { cima.index_of_point(p) }  -> std::same_as<image_index_t<I>>;
    { cima.delta_index(p) }     -> std::same_as<image_index_t<I>>;
  };


namespace detail
{
  // WritableIndexableAndAccessibleImage
  template <typename I>
  concept WritableIndexableAndAccessibleImage =
    IndexableAndAccessibleImage<I> &&
    detail::WritableImage<I> &&
    detail::WritableIndexableImage<I>;
} // namespace detail

// BidirectionalImage (not in STL term)
template <typename I>
concept BidirectionalImage =
  Image<I> &&
  std::derived_from<image_category_t<I>, bidirectional_image_tag> &&
  requires (I ima) {
  { ima.pixels() }  -> mln::ranges::reversible_mdrange;
  { ima.values() }  -> mln::ranges::reversible_mdrange;
};


namespace detail
{
  // WritableBidirectionalImage
  template <typename I>
  concept WritableBidirectionalImage =
    WritableImage<I> &&
    BidirectionalImage<I>;
} // namespace detail
```

```cpp
// RawImage (not contiguous, stride = padding)
template <typename I>
concept RawImage =
  IndexableAndAccessibleImage<I> &&
  BidirectionalImage<I> &&
  std::derived_from<image_category_t<I>, raw_image_tag> &&
  requires (I ima, const I cima, int dim) {
    { ima.data() }        -> std::convertible_to<const image_value_t<I>*>; // data() may be proxied by a view
    { cima.stride(dim) } -> std::same_as<std::ptrdiff_t>;
  };


namespace detail
{
  // WritableRawImage
  template <typename I>
  concept WritableRawImage =
    WritableIndexableAndAccessibleImage<I> &&
    WritableBidirectionalImage<I> &&
    RawImage<I> &&
    requires(I ima, image_value_t<I> v, image_index_t<I> k) {
      { ima.data() }        -> ::concepts::convertible_to<image_value_t<I>*>;
      { *(ima.data() + k) = v };
    };
} // namespace detail


// OutputImage
// Usage: RawImage<I> && OutputImage<I>
template <typename I>
concept OutputImage =
  (not Image<I> || (detail::WritableImage<I>)) &&
  (not IndexableImage<I> || (detail::WritableIndexableImage<I>)) &&
  (not AccessibleImage<I> || (detail::WritableAccessibleImage<I>)) &&
  (not IndexableAndAccessibleImage<I> ||
    (detail::WritableIndexableAndAccessibleImage<I>)) &&
  (not BidirectionalImage<I> || (detail::WritableBidirectionalImage<I>)) &&
  (not RawImage<I> || (detail::WritableRawImage<I>));


template <typename I>
concept WithExtensionImage =
  Image<I> &&
  requires {
    typename image_extension_t<I>;
  } &&
  Extension<image_extension_t<I>, image_point_t<I>> &&
  not std::same_as<mln::extension::none_extension_tag, image_extension_category_t<I>> &&
  requires (I ima, image_point_t<I> p) {
    { ima.extension() } -> std::convertible_to<image_extension_t<I>>;
  };


// ConcreteImage
template <typename I>
```

```
concept ConcreteImage =
  Image<I> &&
  std::semiregular<I> &&  // A concrete image is default constructible
  not image_view_v<I>;


// ViewImage
template <typename I>
concept ViewImage =
  Image<I> &&
  image_view_v<I>;
```

Extension

```
  template <typename Ext, typename Pnt>
  concept Extension =
    std::is_base_of_v<mln::Extension<Ext>, Ext> &&
    requires {
      typename Ext::support_fill;
      typename Ext::support_mirror;
      typename Ext::support_periodize;
      typename Ext::support_clamp;
      typename Ext::support_extend_with;
    } &&
    Value<typename Ext::value_type> &&
    requires (const Ext cext,
        mln::archetypes::StructuringElement<
          Pnt,
          mln::archetypes::Pixel> se) {
      { cext.fit(se) }      -> std::same_as<bool>;
      { cext.extent() }     -> std::same_as<int>;
    };

  template <typename Ext, typename Pnt>
  concept FillableExtension =
    Extension<Ext, Pnt> &&
    std::convertible_to<typename Ext::support_fill, std::true_type> &&
    requires {
      typename Ext::value_type;
    } &&
    requires (Ext ext, const Ext cext, const typename Ext::value_type& v) {
      { ext.fill(v) };
      { cext.is_fill_supported() }  -> std::same_as<bool>;
    };

  template <typename Ext, typename Pnt>
  concept MirrorableExtension =
    Extension<Ext, Pnt> &&
    std::convertible_to<typename Ext::support_mirror, std::true_type> &&
    requires (Ext ext, const Ext cext) {
      { ext.mirror() };
      { cext.is_mirror_supported() }  -> std::same_as<bool>;
    };

  template <typename Ext, typename Pnt>
  concept PeriodizableExtension =
```

```
    Extension<Ext, Pnt> &&
    std::convertible_to<typename Ext::support_periodize, std::true_type> &&
    requires (Ext ext, const Ext cext) {
      { ext.periodize() };
      { cext.is_periodize_supported() }  -> std::same_as<bool>;
    };

  template <typename Ext, typename Pnt>
  concept ClampableExtension =
    Extension<Ext, Pnt> &&
    std::convertible_to<typename Ext::support_clamp, std::true_type> &&
    requires (Ext ext, const Ext cext) {
      { ext.clamp() };
      { cext.is_clamp_supported() }  -> std::same_as<bool>;
    };

  template <typename Ext, typename Pnt, typename U>
  concept ExtendWithExtension =
    Extension<Ext, Pnt> &&
    std::convertible_to<typename Ext::support_extend_with, std::true_type> &&
    InputImage<U> &&
    requires {
      typename Ext::point_type;
    } &&
    std::convertible_to<typename U::value_type, typename Ext::value_type> &&
    std::convertible_to<typename Ext::point_type, typename U::point_type> &&
    requires (Ext ext, const Ext cext, U u, typename Ext::point_type offset) {
      { ext.extend_with(u, offset) };
      { cext.is_extend_with_supported() }  -> std::same_as<bool>;
    };
```

B.0.2 Archetypes

Index

```
using Index = int;

static_assert(mln::concepts::Index<Index>, "Index archetype does not model the Index concept!");
```

Value

```
struct Value
{
};

struct ComparableValue
{
};
bool operator==(const ComparableValue&, const ComparableValue&);
bool operator!=(const ComparableValue&, const ComparableValue&);


struct OrderedValue
{
```

```
};
bool operator==(const OrderedValue&, const OrderedValue&);
bool operator!=(const OrderedValue&, const OrderedValue&);
bool operator<(const OrderedValue&, const OrderedValue&);
bool operator>(const OrderedValue&, const OrderedValue&);
bool operator<=(const OrderedValue&, const OrderedValue&);
bool operator>=(const OrderedValue&, const OrderedValue&);

static_assert(mln::concepts::Value<Value>, "Value archetype does not model the Value concept!");
static_assert(mln::concepts::ComparableValue<ComparableValue>, "ComparableValue archetype does not model the
static_assert(mln::concepts::OrderedValue<OrderedValue>, "OrderedValue archetype does not model the OrderedVa
```

Point

```
struct Point final
{
};

bool operator==(const Point&, const Point&);
bool operator!=(const Point&, const Point&);
bool operator<(const Point&, const Point&);
bool operator>(const Point&, const Point&);
bool operator<=(const Point&, const Point&);
bool operator>=(const Point&, const Point&);

static_assert(mln::concepts::Point<Point>, "Point archetype does not model the Point concept!");
```

Pixel

```
namespace details
{
  template <class P, class V>
  struct PixelT
  {
    using value_type = V;
    using point_type = P;
    using reference  = const value_type&;

    PixelT()               = delete;
    PixelT(const PixelT&) = default;
    PixelT(PixelT&&)      = default;
    PixelT& operator=(const PixelT&) = delete;
    PixelT& operator=(PixelT&&) = delete;

    point_type point() const;
    reference  val() const;
    void       shift(const P& dp);
  };

  struct OutputPixel : PixelT<Point, Value>
  {
    using reference = Value&;
    reference val() const;
  };
```

```
  template <class Pix>
  struct AsPixel : Pix, mln::details::Pixel<AsPixel<Pix>>
  {
  };
} // namespace details

template <class P, class V = Value>
using PixelT      = details::AsPixel<details::PixelT<P, V>>;
using Pixel       = PixelT<Point, Value>;
using OutputPixel = details::AsPixel<details::OutputPixel>;

static_assert(mln::concepts::Pixel<Pixel>, "Pixel archetype does not model the Pixel concept!");
static_assert(mln::concepts::OutputPixel<OutputPixel>, "OutputPixel archetype does not model the OutputPixel concept!");
```

Ranges

```
  // TODO
```

Domain

```
struct Domain
{
  using value_type = Point;
  using reference  = Point&;

  value_type* begin();
  value_type* end();

  bool has(value_type) const;
  bool empty() const;
  int  dim() const;
};

static_assert(mln::concepts::Domain<Domain>, "Domain archetype does not model the Domain concept!");

struct SizedDomain : Domain
{
  unsigned size() const;
};

static_assert(mln::concepts::SizedDomain<SizedDomain>,
                          "SizedDomain archetype does not model the SizedDomain concept!");

struct ShapedDomain final : SizedDomain
{
  static constexpr std::size_t  ndim = 1;
  value_type                    shape() const;
  std::array<std::size_t, ndim> extents() const;
};

static_assert(mln::concepts::ShapedDomain<ShapedDomain>,
                          "ShapedDomain archetype does not model the ShapedDomain concept!");
```

Structuring Element

```
namespace details
{
  template <class P, class Pix>
  requires mln::concepts::Point<P>&& mln::concepts::Pixel<Pix>
  struct StructuringElement
  {
    using category     = adaptative_neighborhood_tag;
    using incremental  = std::false_type;
    using decomposable = std::false_type;
    using separable    = std::false_type;

     std::ranges::subrange<P*> operator()(P p);

     std::ranges::subrange<Pix*> operator()(Pix px);
     std::ranges::subrange<P*>   offsets() const;
  };


  template <class SE>
  struct AsSE : SE, mln::details::Neighborhood
  helper<AsSE<SE>>
  {
  };
} // namespace details

template <class P = Point, class Pix = PixelT<P>>
using StructuringElement = details::AsSE<details::StructuringElement<P, Pix>>;


namespace details
{
  template <class P, class Pix>
  struct DecomposableStructuringElement : StructuringElement<P, Pix>
  {
    using decomposable = std::true_type;

    bool                                                              is_decomposable() const;
    std::ranges::subrange<mln::archetypes::StructuringElement<P, Pix>*> decompose() const;
  };

  template <class P, class Pix>
  struct SeparableStructuringElement : StructuringElement<P, Pix>
  {
    using separable = std::true_type;

    bool                                                              is_separable() const;
    std::ranges::subrange<mln::archetypes::StructuringElement<P, Pix>*> separate() const;
  };

  template <class P, class Pix>
  struct IncrementalStructuringElement : StructuringElement<P, Pix>
  {
    using incremental = std::true_type;

    bool                                   is_incremental() const;
```

```
    archetypes::StructuringElement<P, Pix> inc()          const;
    archetypes::StructuringElement<P, Pix> dec()          const;
  };
} // namespace details


template <class P = Point, class Pix = PixelT<P>>
using DecomposableStructuringElement = details::AsSE<details::DecomposableStructuringElement<P, Pix>>;

template <class P = Point, class Pix = PixelT<P>>
using SeparableStructuringElement = details::AsSE<details::SeparableStructuringElement<P, Pix>>;

template <class P = Point, class Pix = PixelT<P>>
using IncrementalStructuringElement = details::AsSE<details::IncrementalStructuringElement<P, Pix>>;
```

Neighborhood

```
namespace details
{
  template <class P, class Pix>
  requires mln::concepts::Point<P>&& mln::concepts::Pixel<Pix>
  struct Neighborhood : StructuringElement<P, Pix>
  {
     std::ranges::iterator_range<P*>   before(P p);
     std::ranges::iterator_range<P*>   after(P p);
     std::ranges::iterator_range<Pix*> before(Pix px);
     std::ranges::iterator_range<Pix*> after(Pix px);
  };

  template <class N>
  struct AsNeighborhood : N, mln::details::Neighborhood<AsNeighborhood<N>>
  {
  };

} // namespace details

template <class P = Point, class Pix = PixelT<P>>
using Neighborhood = details::AsSE<details::Neighborhood<P, Pix>>;
```

Image

```
namespace details
{
  template <class I>
  struct AsImage : I, mln::details::Image<AsImage<I>>
  {
    using I::I;

    using concrete_type = AsImage<typename I::concrete_type>;
    concrete_type concretize() const;


    template <typename V>
    using ch_value_type = AsImage<typename I::template ch_value_type<V>>;
```

```
  template <typename V>
  ch_value_type<V> ch_value() const;
};


struct ConcreteImage
{
  using pixel_type = archetypes::Pixel;
  using value_type     = pixel_value_t<mln::archetypes::Pixel>;
  using reference      = pixel_reference_t<mln::archetypes::Pixel>;
  using point_type     =  std::ranges::range_value_t<Domain>;
  using domain_type    = Domain;
  using category_type  = forward_image_tag;
  using concrete_type  = ConcreteImage;

  template <class V>
  using ch_value_type = ConcreteImage;

  // additional traits
  using extension_category = mln::extension::none_extension_tag;
  using indexable          = std::false_type;
  using accessible         = std::false_type;
  using view               = std::false_type;

  ConcreteImage()                     = default;
  ConcreteImage(const ConcreteImage&) = default;
  ConcreteImage(ConcreteImage&&)      = default;
  ConcreteImage& operator=(const ConcreteImage&) = default;
  ConcreteImage& operator=(ConcreteImage&&) = default;

  domain_type domain() const;


  struct pixel_range
  {
    const pixel_type* begin();
    const pixel_type* end();
  };
  pixel_range pixels();


  struct value_range
  {
    const value_type* begin();
    const value_type* end();
  };

  value_range values();
};


struct ViewImage : ConcreteImage
{
  using view = std::true_type;

  ViewImage()                 = delete;
```

```
  ViewImage(const ViewImage&) = default;
  ViewImage(ViewImage&&)      = default;
  ViewImage& operator=(const ViewImage&) = delete;
  ViewImage& operator=(ViewImage&&) = delete;
};

using Image = ViewImage;


struct OutputImage : Image
{
  using pixel_type = archetypes::OutputPixel;
  using reference      = pixel_reference_t<mln::archetypes::OutputPixel>;

  struct pixel_range
  {
    const pixel_type* begin();
    const pixel_type* end();
  };

  pixel_range pixels();

  struct value_range
  {
    value_type* begin();
    value_type* end();
  };

  value_range values();
};


struct OutputIndexableImage : OutputImage
{
  using index_type = int;
  using indexable  = std::true_type;

  using concrete_type = OutputIndexableImage;

  template <class V>
  using ch_value_type = OutputIndexableImage;

  reference operator[](index_type);
};


struct IndexableImage : Image
{
  using index_type = int;
  using indexable  = std::true_type;

  using concrete_type = OutputIndexableImage;

  template <class V>
  using ch_value_type = OutputIndexableImage;
```

```
  reference operator[](index_type);
};

struct OutputAccessibleImage : OutputImage
{
  using accessible    = std::true_type;
  using concrete_type = OutputAccessibleImage;

  template <class V>
  using ch_value_type = OutputAccessibleImage;


  reference      operator()(point_type);
  reference      at(point_type);
  pixel_type pixel(point_type);
  pixel_type pixel_at(point_type);
};


struct AccessibleImage : Image
{
  using accessible    = std::true_type;
  using concrete_type = OutputAccessibleImage;

  template <class V>
  using ch_value_type = OutputAccessibleImage;

  reference      operator()(point_type);
  reference      at(point_type);
  pixel_type pixel(point_type);
  pixel_type pixel_at(point_type);
};


struct OutputIndexableAndAccessibleImage : OutputAccessibleImage
{
  using index_type = int;
  using indexable  = std::true_type;

  using concrete_type = OutputIndexableAndAccessibleImage;

  template <class V>
  using ch_value_type = OutputIndexableAndAccessibleImage;


  reference  operator[](index_type);
  point_type point_at_index(index_type) const;
  index_type index_of_point(point_type) const;
  index_type delta_index(point_type) const;
};


struct IndexableAndAccessibleImage : AccessibleImage
{
  using index_type = int;
```

```
  using indexable  = std::true_type;

  using concrete_type = OutputIndexableAndAccessibleImage;

  template <class V>
  using ch_value_type = OutputIndexableAndAccessibleImage;

  reference  operator[](index_type);
  point_type point_at_index(index_type) const;
  index_type index_of_point(point_type) const;
  index_type delta_index(point_type) const;
};


struct BidirectionalImage : Image
{
  using category_type = bidirectional_image_tag;

  struct pixel_range
  {
    const pixel_type* begin();
    const pixel_type* end();
    pixel_range           reversed();
  };

  pixel_range pixels();

  struct value_range
  {
    const value_type* begin();
    const value_type* end();
    value_range       reversed();
  };

  value_range values();
};


struct OutputBidirectionalImage : BidirectionalImage
{
  using pixel_type = archetypes::OutputPixel;
  using reference      = pixel_reference_t<mln::archetypes::OutputPixel>;

  struct value_range
  {
    value_type* begin();
    value_type* end();
    value_range reversed();
  };
  value_range values();


  struct pixel_range
  {
    const pixel_type* begin();
    const pixel_type* end();
```

```
    pixel_range           reversed();
  };

  pixel_range pixels();
};


struct RawImage : IndexableAndAccessibleImage
{
  using category_type = raw_image_tag;
  using pixel_range   = BidirectionalImage::pixel_range;
  using value_range   = BidirectionalImage::value_range;

  pixel_range pixels();
  value_range values();


  const value_type* data() const;
  std::ptrdiff_t    strides(int) const;
};


struct OutputRawImage : OutputIndexableAndAccessibleImage
{
  using category_type = raw_image_tag;
  using pixel_range   = OutputBidirectionalImage::pixel_range;
  using value_range   = OutputBidirectionalImage::value_range;

  pixel_range pixels();
  value_range values();

  value_type*    data() const;
  std::ptrdiff_t strides(int) const;
};


struct WithExtensionImage : Image
{
  struct Extension : ::mln::Extension<Extension>
  {
    using support_fill = std::false_type;
    using support_mirror = std::false_type;
    using support_periodize = std::false_type;
    using support_clamp = std::false_type;
    using support_extend_with = std::false_type;
    using value_type = image_value_t<Image>;
    bool fit(mln::archetypes::StructuringElement<image_point_t<Image>, mln::archetypes::Pixel> se) const;
    int extent() const;
  };

  using extension_type = Extension;

  using extension_category = mln::extension::custom_extension_tag;

  extension_type extension() const;
};
```

```
} // namespace details


using Image         = details::AsImage<details::Image>;
using ConcreteImage = details::AsImage<details::ConcreteImage>;
using ViewImage     = details::AsImage<details::ViewImage>;

using ForwardImage       = Image;
using BidirectionalImage = details::AsImage<details::BidirectionalImage>;
using RawImage           = details::AsImage<details::RawImage>;

using InputImage                  = Image;
using IndexableImage              = details::AsImage<details::IndexableImage>;
using AccessibleImage             = details::AsImage<details::AccessibleImage>;
using IndexableAndAccessibleImage = details::AsImage<details::IndexableAndAccessibleImage>;


using OutputImage              = details::AsImage<details::OutputImage>;
using OutputForwardImage       = OutputImage;
using OutputBidirectionalImage = details::AsImage<details::OutputBidirectionalImage>;
using OutputRawImage           = details::AsImage<details::OutputRawImage>;

using OutputIndexableImage              = details::AsImage<details::OutputIndexableImage>;
using OutputAccessibleImage             = details::AsImage<details::OutputAccessibleImage>;
using OutputIndexableAndAccessibleImage = details::AsImage<details::OutputIndexableAndAccessibleImage>;

using WithExtensionImage = details::AsImage<details::WithExtensionImage>;
```

Extension

```
  // TODO
```

Appendix C

Benchmark compilation time

First is the code containing containing the structures that will be used in the program against both SFINAE and concept contraints.

C.1 Common structures

```
// dummy_structs.hpp

template <int N>
struct boolean_struct
{
  boolean_struct()                      = default;
  boolean_struct(const boolean_struct&) = default;
  boolean_struct(boolean_struct&&)      = default;
  boolean_struct& operator=(const boolean_struct&) = default;
  boolean_struct& operator=(boolean_struct&&) = default;

  explicit boolean_struct(bool b)
    : b_(b)
  {
  }

  operator bool() const { return b_; }

private:
  bool b_;
};

template <int N>
inline bool operator!(const boolean_struct<N>& b)
{
  return not static_cast<bool>(b);
}

template <int N, int M>
inline bool operator&&(const boolean_struct<N>& lhs, const boolean_struct<M>& rhs)
```

```
{
  return static_cast<bool>(lhs) && static_cast<bool>(rhs);
}

template <int N>
inline bool operator&&(const boolean_struct<N>& lhs, bool rhs)
{
  return static_cast<bool>(lhs) && rhs;
}

template <int M>
inline bool operator&&(bool lhs, const boolean_struct<M>& rhs)
{
  return rhs && lhs;
}

template <int N, int M>
inline bool operator||(const boolean_struct<N>& lhs, const boolean_struct<M>& rhs)
{
  return static_cast<bool>(lhs) || static_cast<bool>(rhs);
}

template <int N>
inline bool operator||(const boolean_struct<N>& lhs, bool rhs)
{
  return static_cast<bool>(lhs) || rhs;
}

template <int M>
inline bool operator||(bool lhs, const boolean_struct<M>& rhs)
{
  return lhs || static_cast<bool>(rhs);
}

template <int N, int M>
inline bool operator==(const boolean_struct<N>& lhs, const boolean_struct<M>& rhs)
{
  return static_cast<bool>(lhs) == static_cast<bool>(rhs);
}

template <int N>
inline bool operator==(const boolean_struct<N>& lhs, bool rhs)
{
  return static_cast<bool>(lhs) == rhs;
}

template <int M>
inline bool operator==(bool lhs, const boolean_struct<M>& rhs)
{
  return rhs == lhs;
}

template <int N, int M>
inline bool operator!=(const boolean_struct<N>& lhs, const boolean_struct<M>& rhs)
{
  return not(static_cast<bool>(lhs) == static_cast<bool>(rhs));
```

```
}

template <int N>
inline bool operator!=(const boolean_struct<N>& lhs, bool rhs)
{
  return not(static_cast<bool>(lhs) == rhs);
}

template <int M>
inline bool operator!=(bool lhs, const boolean_struct<M>& rhs)
{
  return not(rhs == lhs);
}


template <int N>
struct non_boolean_struct
{
  non_boolean_struct()                          = default;
  non_boolean_struct(const non_boolean_struct&) = default;
  non_boolean_struct(non_boolean_struct&&)      = default;
  non_boolean_struct& operator=(const non_boolean_struct&) = default;
  non_boolean_struct& operator=(non_boolean_struct&&) = default;

  non_boolean_struct(bool b)
    : b_(b)
  {
  }

  operator bool() const { return b_; }

private:
  bool b_;
};

template <int N>
inline bool operator!(const non_boolean_struct<N>& b)
{
  return not static_cast<bool>(b);
}

template <int N, int M>
inline bool operator&&(const non_boolean_struct<N>& lhs, const non_boolean_struct<M>& rhs)
{
  return static_cast<bool>(lhs) && static_cast<bool>(rhs);
}

template <int N>
inline bool operator&&(const non_boolean_struct<N>& lhs, bool rhs)
{
  return static_cast<bool>(lhs) && rhs;
}

template <int M>
inline bool operator&&(bool lhs, const non_boolean_struct<M>& rhs)
{
```

```
  return rhs && lhs;
}

template <int N, int M>
inline bool operator||(const non_boolean_struct<N>& lhs, const non_boolean_struct<M>& rhs)
{
  return static_cast<bool>(lhs) || static_cast<bool>(rhs);
}

template <int N>
inline bool operator||(const non_boolean_struct<N>& lhs, bool rhs)
{
  return static_cast<bool>(lhs) || rhs;
}

template <int M>
inline bool operator||(bool lhs, const non_boolean_struct<M>& rhs)
{
  return lhs || static_cast<bool>(rhs);
}

template <int N, int M>
inline bool operator==(const non_boolean_struct<N>& lhs, const non_boolean_struct<M>& rhs)
{
  return static_cast<bool>(lhs) == static_cast<bool>(rhs);
}

template <int N>
inline bool operator==(const non_boolean_struct<N>& lhs, bool rhs)
{
  return static_cast<bool>(lhs) == rhs;
}

template <int M>
inline bool operator==(bool lhs, const non_boolean_struct<M>& rhs)
{
  return rhs == lhs;
}

template <int N, int M>
inline bool operator!=(const non_boolean_struct<N>& lhs, const non_boolean_struct<M>& rhs)
{
  return not(static_cast<bool>(lhs) == static_cast<bool>(rhs));
}

template <int N>
inline bool operator!=(const non_boolean_struct<N>& lhs, bool rhs)
{
  return not(static_cast<bool>(lhs) == rhs);
}

// The non-existance of this overload will render the struct non-boolean
template <int M>
inline bool operator!=(bool lhs, const non_boolean_struct<M>& rhs) = delete;
/*
{
```

```
  return not (rhs == lhs);
}
*/
```

We then have the canonical implementation of the standard library concepts in case the compilers does not provide library support for it yet.

```
// std_concepts.hpp

#ifdef _MSVC
#pragma message ( "Your compiler does not provide <concepts> header yet. Using in-house concepts implementation!" )
#else
#warning "Your compiler does not provide <concepts> header yet. Using in-house concepts implementation!"
#endif

#include <type_traits>
#include <utility>

namespace std_
{
  using namespace std;

  // clang-format off

  template<class T, class U>
  concept __SameImpl = is_same_v<T, U>;  // exposition only

  template<class T, class U>
  concept same_as = __SameImpl<T, U> && __SameImpl<U, T>;

  template<class Derived, class Base>
  concept derived_from =
    is_base_of_v<Base, Derived> &&
    is_convertible_v<const volatile Derived*, const volatile Base*>;

  template<class From, class To>
  concept convertible_to =
    is_convertible_v<From, To> &&
    requires(add_rvalue_reference_t<From> (&f)()) {
      static_cast<To>(f());
    };

  template<class B>
  concept __boolean_testable_impl =                // exposition only
    convertible_to<B, bool>;

  template<class B>
  concept boolean_testable =                       // exposition only
    __boolean_testable_impl<B> &&
    requires (B&& b) {
        { !forward<B>(b) } -> __boolean_testable_impl;
    };

  template<class T, class U>
  concept common_reference_with =
```

```
  same_as<common_reference_t<T, U>, common_reference_t<U, T>> &&
  convertible_to<T, common_reference_t<T, U>> &&
  convertible_to<U, common_reference_t<T, U>>;


template<class T, class U>
concept common_with =
  same_as<common_type_t<T, U>, common_type_t<U, T>> &&
  requires {
    static_cast<common_type_t<T, U>>(declval<T>());
    static_cast<common_type_t<T, U>>(declval<U>());
  } &&
  common_reference_with<
    add_lvalue_reference_t<const T>,
    add_lvalue_reference_t<const U>> &&
  common_reference_with<
    add_lvalue_reference_t<common_type_t<T, U>>,
    common_reference_t<
      add_lvalue_reference_t<const T>,
      add_lvalue_reference_t<const U>>>;

template<class T>
concept integral = is_integral_v<T>;

template<class T>
concept signed_integral = integral<T> && is_signed_v<T>;

template<class T>
concept unsigned_integral = integral<T> && !signed_integral<T>;

template<class T>
concept floating_point = is_floating_point_v<T>;

template<class LHS, class RHS>
concept assignable_from =
  is_lvalue_reference_v<LHS> &&
  common_reference_with<
    const remove_reference_t<LHS>&,
    const remove_reference_t<RHS>&> &&
  requires(LHS lhs, RHS&& rhs) {
    { lhs = forward<RHS>(rhs) } -> same_as<LHS>;
  };

template<class T>
concept swappable = requires(T& a, T& b) { ranges::swap(a, b); };

template<class T, class U>
concept swappable_with =
  common_reference_with<const remove_reference_t<T>&, const remove_reference_t<U>&> &&
  requires(T&& t, U&& u) {
    ranges::swap(forward<T>(t), forward<T>(t));
    ranges::swap(forward<U>(u), forward<U>(u));
    ranges::swap(forward<T>(t), forward<U>(u));
    ranges::swap(forward<U>(u), forward<T>(t));
  };
```

```
template<class T>
concept destructible = is_nothrow_destructible_v<T>;

template<class T, class... Args>
concept constructible_from = destructible<T> && is_constructible_v<T, Args...>;

template<class T>
concept default_initializable =
  constructible_from<T> &&
  requires { T{}; } &&
  requires { ::new (static_cast<void*>(nullptr)) T; };

template<class T>
concept move_constructible = constructible_from<T, T> && convertible_to<T, T>;

template<class T>
concept copy_constructible =
  move_constructible<T> &&
  constructible_from<T, T&> && convertible_to<T&, T> &&
  constructible_from<T, const T&> && convertible_to<const T&, T> &&
  constructible_from<T, const T> && convertible_to<const T, T>;

template<class T, class U>
concept __WeaklyEqualityComparableWith = // exposition only
  requires(const remove_reference_t<T>& t,
           const remove_reference_t<U>& u) {
    { t == u } -> boolean_testable;
    { t != u } -> boolean_testable;
    { u == t } -> boolean_testable;
    { u != t } -> boolean_testable;
  };

template<class T>
concept equality_comparable = __WeaklyEqualityComparableWith<T, T>;

template<class T, class U>
concept equality_comparable_with =
  equality_comparable<T> && equality_comparable<U> &&
  common_reference_with<const remove_reference_t<T>&, const remove_reference_t<U>&> &&
  equality_comparable<
    common_reference_t<
      const remove_reference_t<T>&,
      const remove_reference_t<U>&>> &&
  __WeaklyEqualityComparableWith<T, U>;

template<class T>
concept totally_ordered =
  equality_comparable<T> &&
  requires(const remove_reference_t<T>& a,
           const remove_reference_t<T>& b) {
    { a <  b } -> boolean_testable;
    { a >  b } -> boolean_testable;
    { a <= b } -> boolean_testable;
    { a >= b } -> boolean_testable;
  };
```

```
template<class T, class U>
concept totally_ordered_with =
  totally_ordered<T> && totally_ordered<U> &&
  common_reference_with<const remove_reference_t<T>&, const remove_reference_t<U>&> &&
  totally_ordered<
    common_reference_t<
      const remove_reference_t<T>&,
      const remove_reference_t<U>&>> &&
  equality_comparable_with<T, U> &&
  requires(const remove_reference_t<T>& t,
           const remove_reference_t<U>& u) {
    { t <  u } -> boolean_testable;
    { t >  u } -> boolean_testable;
    { t <= u } -> boolean_testable;
    { t >= u } -> boolean_testable;
    { u <  t } -> boolean_testable;
    { u >  t } -> boolean_testable;
    { u <= t } -> boolean_testable;
    { u >= t } -> boolean_testable;
  };

template<class T>
concept movable = is_object_v<T> && move_constructible<T> &&
                  assignable_from<T&, T> && swappable<T>;
template<class T>
concept copyable = copy_constructible<T> && movable<T> && assignable_from<T&, T&> &&
                   assignable_from<T&, const T&> && assignable_from<T&, const T>;

template<class T>
concept semiregular = copyable<T> && default_initializable<T>;

template<class T>
concept regular = semiregular<T> && equality_comparable<T>;

template<class F, class... Args>
concept invocable = requires(F&& f, Args&&... args) {
  invoke(forward<F>(f), forward<Args>(args)...);
    // not required to be equality-preserving
};

template<class F, class... Args>
concept regular_invocable = invocable<F, Args...>;

template<class F, class... Args>
concept predicate =
  regular_invocable<F, Args...> && boolean_testable<invoke_result_t<F, Args...>>;

template<class R, class T, class U>
concept relation =
  predicate<R, T, T> && predicate<R, U, U> &&
  predicate<R, T, U> && predicate<R, U, T>;

template<class R, class T, class U>
concept equivalence_relation = relation<R, T, U>;

template<class R, class T, class U>
```

```
  concept strict_weak_order = relation<R, T, U>;

  // clang-format on

} // namespace std_
```

C.2 Constraints

First are the concepts we are going to benchmark the common structure against:

```
// concepts.hpp

#include "std_concepts.hpp"

#if __has_include(<concepts>)
#include <concepts>
#else
#include "stl_concepts.hpp"
namespace std { using namespace std_; }
#endif

#include <type_traits>

template<class B>
  concept boolean_c =
  std::movable<std::remove_cvref_t<B>> &&
  requires(const std::remove_reference_t<B>& b1,
           const std::remove_reference_t<B>& b2, const bool a) {
    { b1 } -> std::convertible_to<bool>;
    { !b1 } -> std::convertible_to<bool>;
    { b1 && b2 } -> std::same_as<bool>;
    { b1 &&  a } -> std::same_as<bool>;
    {  a && b2 } -> std::same_as<bool>;
    { b1 || b2 } -> std::same_as<bool>;
    { b1 ||  a } -> std::same_as<bool>;
    {  a || b2 } -> std::same_as<bool>;
    { b1 == b2 } -> std::convertible_to<bool>;
    { b1 ==  a } -> std::convertible_to<bool>;
    {  a == b2 } -> std::convertible_to<bool>;
    { b1 != b2 } -> std::convertible_to<bool>;
    { b1 !=  a } -> std::convertible_to<bool>;
    {  a != b2 } -> std::convertible_to<bool>;
  };

template<class B>
  concept boolean_fast_c =
  // Slow code commented and replaced by builtin compiler traits
  // std_::movable<std::remove_cvref_t<B>> &&
  std::is_object_v<B> && std::is_move_assignable_v<B> &&
  std::is_move_constructible_v<B> && std::is_swappable_v<B> &&
  requires(const std::remove_reference_t<B>& b1,
           const std::remove_reference_t<B>& b2, const bool a) {
    { b1 } -> std::convertible_to<bool>;
```

```
    { !b1 } -> std::convertible_to<bool>;
    { b1 && b2 } -> std::same_as<bool>;
    { b1 &&  a } -> std::same_as<bool>;
    {  a && b2 } -> std::same_as<bool>;
    { b1 || b2 } -> std::same_as<bool>;
    { b1 ||  a } -> std::same_as<bool>;
    {  a || b2 } -> std::same_as<bool>;
    { b1 == b2 } -> std::convertible_to<bool>;
    { b1 ==  a } -> std::convertible_to<bool>;
    {  a == b2 } -> std::convertible_to<bool>;
    { b1 != b2 } -> std::convertible_to<bool>;
    { b1 !=  a } -> std::convertible_to<bool>;
    {  a != b2 } -> std::convertible_to<bool>;
  };
```

We then provide an equivalent in SFINAE of the above concepts, written with detectors in order to be used with classic `std::void_t` and/or `std::enable_if` facilities.

```
// sfinae.hpp

#include <type_traits>
#include <utility>


// b -> bool
template <typename B, typename = void>
struct boolean_sfinae_impl_conv : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_conv<B, std::enable_if_t<std::is_convertible_v<B, bool>>> : std::true_type
{
};


// !b
template <typename B, typename = void>
struct boolean_sfinae_impl_not : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_not<B, std::void_t<decltype(!std::declval<B>())>> : std::true_type
{
};

// !b -> bool
template <typename B, typename = void>
struct boolean_sfinae_impl_not_conv : std::false_type
{
};

template <typename B>
```

```
struct boolean_sfinae_impl_not_conv<B, std::enable_if_t<std::is_convertible_v<decltype(!std::declval<B>()), bool>>>
  : std::true_type
{
};


// b1 && b2
template <typename B1, typename B2, typename = void>
struct boolean_sfinae_impl_and : std::false_type
{
};

template <typename B1, typename B2>
struct boolean_sfinae_impl_and<B1, B2, std::void_t<decltype(std::declval<B1>() && std::declval<B2>())>> : std::true_type
{
};

// b1 && b2 -> bool
template <typename B1, typename B2, typename = void>
struct boolean_sfinae_impl_and_conv : std::false_type
{
};

template <typename B1, typename B2>
struct boolean_sfinae_impl_and_conv<
    B1, B2, std::enable_if_t<std::is_same_v<decltype(std::declval<B1>() && std::declval<B2>()), bool>>> : std::true_type
{
};

// b1 && bool
template <typename B, typename = void>
struct boolean_sfinae_impl_and_rb : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_and_rb<B, std::void_t<decltype(std::declval<B>() && std::declval<bool>())>> : std::true_type
{
};

// b1 && bool -> bool
template <typename B, typename = void>
struct boolean_sfinae_impl_and_rb_conv : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_and_rb_conv<
    B, std::enable_if_t<std::is_same_v<decltype(std::declval<B>() && std::declval<bool>()), bool>>> : std::true_type
{
};

// bool && b2
template <typename B, typename = void>
struct boolean_sfinae_impl_and_lb : std::false_type
{
```

```
};

template <typename B>
struct boolean_sfinae_impl_and_lb<B, std::void_t<decltype(std::declval<bool>() && std::declval<B>())>> : std:
{
};

// bool && b2 -> bool
template <typename B, typename = void>
struct boolean_sfinae_impl_and_lb_conv : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_and_lb_conv<
    B, std::enable_if_t<std::is_same_v<decltype(std::declval<bool>() && std::declval<B>()), bool>>> : std::tr
{
};


// b1 || b2
template <typename B1, typename B2, typename = void>
struct boolean_sfinae_impl_or : std::false_type
{
};

template <typename B1, typename B2>
struct boolean_sfinae_impl_or<B1, B2, std::void_t<decltype(std::declval<B1>() || std::declval<B2>())>> : std:
{
};

// b1 || b2 -> bool
template <typename B1, typename B2, typename = void>
struct boolean_sfinae_impl_or_conv : std::false_type
{
};

template <typename B1, typename B2>
struct boolean_sfinae_impl_or_conv<
    B1, B2, std::enable_if_t<std::is_same_v<decltype(std::declval<B1>() || std::declval<B2>()), bool>>> : std
{
};

// b1 || bool
template <typename B, typename = void>
struct boolean_sfinae_impl_or_rb : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_or_rb<B, std::void_t<decltype(std::declval<B>() || std::declval<bool>())>> : std::
{
};

// b1 || bool -> bool
template <typename B, typename = void>
```

```
struct boolean_sfinae_impl_or_rb_conv : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_or_rb_conv<
    B, std::enable_if_t<std::is_same_v<decltype(std::declval<B>() || std::declval<bool>()), bool>>> : std::true_type
{
};

// bool || b2
template <typename B, typename = void>
struct boolean_sfinae_impl_or_lb : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_or_lb<B, std::void_t<decltype(std::declval<bool>() || std::declval<B>())>> : std::true_type
{
};

// bool || b2 -> bool
template <typename B, typename = void>
struct boolean_sfinae_impl_or_lb_conv : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_or_lb_conv<
    B, std::enable_if_t<std::is_same_v<decltype(std::declval<bool>() || std::declval<B>()), bool>>> : std::true_type
{
};


// b1 == b2
template <typename B1, typename B2, typename = void>
struct boolean_sfinae_impl_eq : std::false_type
{
};

template <typename B1, typename B2>
struct boolean_sfinae_impl_eq<B1, B2, std::void_t<decltype(std::declval<B1>() == std::declval<B2>())>> : std::true_type
{
};

// b1 == b2 -> bool
template <typename B1, typename B2, typename = void>
struct boolean_sfinae_impl_eq_conv : std::false_type
{
};

template <typename B1, typename B2>
struct boolean_sfinae_impl_eq_conv<
    B1, B2, std::enable_if_t<std::is_convertible_v<decltype(std::declval<B1>() == std::declval<B2>()), bool>>>
  : std::true_type
{
```

```
};

// b1 == bool
template <typename B, typename = void>
struct boolean_sfinae_impl_eq_rb : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_eq_rb<B, std::void_t<decltype(std::declval<B>() == std::declval<bool>())>> : std::
{
};

// b1 == bool -> bool
template <typename B, typename = void>
struct boolean_sfinae_impl_eq_rb_conv : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_eq_rb_conv<
    B, std::enable_if_t<std::is_convertible_v<decltype(std::declval<B>() == std::declval<bool>()), bool>>>
  : std::true_type
{
};

// bool == b2
template <typename B, typename = void>
struct boolean_sfinae_impl_eq_lb : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_eq_lb<B, std::void_t<decltype(std::declval<bool>() == std::declval<B>())>> : std::
{
};

// bool == b2 -> bool
template <typename B, typename = void>
struct boolean_sfinae_impl_eq_lb_conv : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_eq_lb_conv<
    B, std::enable_if_t<std::is_convertible_v<decltype(std::declval<bool>() == std::declval<B>()), bool>>>
  : std::true_type
{
};


// b1 != b2
template <typename B1, typename B2, typename = void>
struct boolean_sfinae_impl_neq : std::false_type
{
};
```

```
template <typename B1, typename B2>
struct boolean_sfinae_impl_neq<B1, B2, std::void_t<decltype(std::declval<B1>() != std::declval<B2>())>> : std::true_type
{
};

// b1 != b2 -> bool
template <typename B1, typename B2, typename = void>
struct boolean_sfinae_impl_neq_conv : std::false_type
{
};

template <typename B1, typename B2>
struct boolean_sfinae_impl_neq_conv<
    B1, B2, std::enable_if_t<std::is_convertible_v<decltype(std::declval<B1>() != std::declval<B2>()), bool>>>
  : std::true_type
{
};

// b1 != bool
template <typename B, typename = void>
struct boolean_sfinae_impl_neq_rb : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_neq_rb<B, std::void_t<decltype(std::declval<B>() != std::declval<bool>())>> : std::true_type
{
};

// b1 != bool -> bool
template <typename B, typename = void>
struct boolean_sfinae_impl_neq_rb_conv : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_neq_rb_conv<
    B, std::enable_if_t<std::is_convertible_v<decltype(std::declval<B>() != std::declval<bool>()), bool>>>
  : std::true_type
{
};

// bool != b2
template <typename B, typename = void>
struct boolean_sfinae_impl_neq_lb : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_neq_lb<B, std::void_t<decltype(std::declval<bool>() != std::declval<B>())>> : std::true_type
{
};

// bool != b2 -> bool
template <typename B, typename = void>
```

```
struct boolean_sfinae_impl_neq_lb_conv : std::false_type
{
};

template <typename B>
struct boolean_sfinae_impl_neq_lb_conv<
    B, std::enable_if_t<std::is_convertible_v<decltype(std::declval<bool>() != std::declval<B>()), bool>>>
  : std::true_type
{
};


template <typename B>
using boolean_sfinae = std::conjunction<
    std::is_object<B>, std::is_move_assignable<B>, std::is_move_constructible<B>, std::is_swappable<B>,
    boolean_sfinae_impl_conv<B>, boolean_sfinae_impl_not<B>, boolean_sfinae_impl_not_conv<B>,
    boolean_sfinae_impl_and<B, B>, boolean_sfinae_impl_and_conv<B, B>, boolean_sfinae_impl_and_rb<B>,
    boolean_sfinae_impl_and_rb_conv<B>, boolean_sfinae_impl_and_lb<B>, boolean_sfinae_impl_and_lb_conv<B>,
    boolean_sfinae_impl_or<B, B>, boolean_sfinae_impl_or_conv<B, B>, boolean_sfinae_impl_or_rb<B>,
    boolean_sfinae_impl_or_rb_conv<B>, boolean_sfinae_impl_or_lb<B>, boolean_sfinae_impl_or_lb_conv<B>,
    boolean_sfinae_impl_eq<B, B>, boolean_sfinae_impl_eq_conv<B, B>, boolean_sfinae_impl_eq_rb<B>,
    boolean_sfinae_impl_eq_rb_conv<B>, boolean_sfinae_impl_eq_lb<B>, boolean_sfinae_impl_eq_lb_conv<B>,
    boolean_sfinae_impl_neq<B, B>, boolean_sfinae_impl_neq_conv<B, B>, boolean_sfinae_impl_neq_rb<B>,
    boolean_sfinae_impl_neq_rb_conv<B>, boolean_sfinae_impl_neq_lb<B>, boolean_sfinae_impl_neq_lb_conv<B>>;

template <typename B>
using boolean_sfinae_t = typename boolean_sfinae<B>::type;

template <typename B>
inline constexpr auto boolean_sfinae_v = boolean_sfinae<B>::value;
```

C.3 Index

Finally we have the three following benchmark programs using the above code.

Benchmark SFINAE:

```
// bench_sfinae.cpp.erb

#include "dummy_structs.hpp"
#include "sfinae.hpp"

#include <type_traits>
#include <utility>


template <typename T, std::enable_if_t<boolean_sfinae_v<T>, void*> = nullptr>
bool foo_sfinae(T&& t) {
  return static_cast<bool>(std::forward<T>(t));
}

template <typename T, std::enable_if_t<not boolean_sfinae_v<T>, void*> = nullptr>
```

```
bool foo_sfinae(T&& t) {
  return false;
}

template <int N>
constexpr auto instantiate_both(){
  boolean_struct<N> b(true);
  non_boolean_struct<N> nb(true);

  auto a = foo_sfinae(b);
  auto c = foo_sfinae(nb);

  return a && c;
}

int main() {

#if defined(METABENCH)

  // This is ruby template syntax loop to unroll it from 0 to n=250 with a step of 5
  <% (0..n).each do |i| %>
    [[maybe_unused]] auto ret<%= i %> = instantiate_both<<%= i %>>();
  <% end %>

#endif

  return 0;
}
```

Benchmark Concept:

```
// bench_concept.cpp.erb

#include "dummy_structs.hpp"
#include "concepts.hpp"

#include <type_traits>
#include <utility>


template <typename T>
  requires boolean_c<T>
bool foo_c(T&& t) {
  return static_cast<bool>(std::forward<T>(t));
}

template <typename T>
bool foo_c(T&&) {
  return false;
}

template <int N>
constexpr auto instantiate_both(){
  boolean_struct<N> b(true);
  non_boolean_struct<N> nb(true);
```

```cpp
  auto a = foo_c(b);
  auto c = foo_c(nb);

  return a && c;
}


int main() {

#if defined(METABENCH)

  // This is ruby template syntax loop to unroll it from 0 to n=250 with a step of 5
  <% (0..n).each do |i| %>
    [[maybe_unused]] auto ret<%= i %> = instantiate_both<<%= i %>>();
  <% end %>

#endif

  return 0;
}
```

Benchmark Concept (fast):

```cpp
// bench_concept_fast.cpp.erb

#include "dummy_structs.hpp"
#include "concepts.hpp"

#include <type_traits>
#include <utility>


template <typename T>
  requires boolean_c<T>
bool foo_c(T&& t) {
  return static_cast<bool>(std::forward<T>(t));
}

template <typename T>
bool foo_c(T&&) {
  return false;
}

template <int N>
constexpr auto instantiate_both(){
  boolean_struct<N> b(true);
  non_boolean_struct<N> nb(true);

  auto a = foo_c(b);
  auto c = foo_c(nb);

  return a && c;
}


int main() {
```

```
#if defined(METABENCH)

  // This is ruby template syntax loop to unroll it from 0 to n=250 with a step of 5
  <% (0..n).each do |i| %>
    [[maybe_unused]] auto ret<%= i %> = instantiate_both<<%= i %>>();
  <% end %>

#endif

  return 0;
}
```