Programming in modern C++ for image processing

Michaël Roynard

May 3, 2021

0.1 Acknowledgement

0.2 Abstract

TODO EN
TODO FR

0.3 Long abstract

Contents

0.1 Acknowledgement 2
0.2 Abstract 2
0.3 Long abstract 2

I Context **11**

1 Introduction **13**

2 Genericity **23**
2.1 Genericity within libraries 24
2.1.1 Different approaches to get genericity 26
2.1.2 Unjustified limitations 30
2.2 Genericity in pre C++11 34
2.2.1 SFINAE: Substitution-Failure-Is-Not-An-Error 34
2.2.2 CRTP: Curiously Recurring Template Pattern 35
2.3 Genericity in post C++11 (C++20 and Concepts) 41
2.3.1 Conceptification 41
2.3.2 Simplifying code 44
2.3.3 Benchmarks 47
2.4 C++ templates in a dynamic world 51

II Contribution **55**

3 Taxonomy of Images and Algorithms **57**
3.1 Rewriting an algorithm to extract a concept 57
3.1.1 Gamma correction 57
3.1.2 Dilation algorithm 59
3.1.3 Concept definition 60
3.2 Images types viewed as Sets: version & specialization 62

3.3 Generic aspect of algorithm: canvas 65
3.3.1 Taxonomy and canvas 66
3.3.2 Heterogeneous computing: a partial solution, canvas . 68
3.4 Library concepts: listing and explanation 72
3.4.1 Index 72
3.4.2 Value 72
3.4.3 Point 73
3.4.4 Pixel 73
3.4.5 Ranges 73
3.4.6 Domain 73

4 Image views **77**
4.1 The Genesis of a new asbtraction layers: Views 77
4.2 Upgrading the way to design IP algorithms 77
4.2.1 Keeping properties 77
4.2.2 Lazy evaluation 77
4.2.3 Composability and piping 77
4.3 A practical example: border management 77
4.4 Performance discussion 77
4.5 Benchmark figures 93

5 Static dynamic bridge **95**
5.1 Coexistance between the static world and the dynamic world 95
5.2 Hybrid solution: $n * n$ dispatch thanks to variants 95
5.3 JIT-based solutions: pros. and cons. 95

III Continuation **101**

6 Conclusion **103**

IV Appendices **109**

A Bibliography **111**

B Concepts & archetypes **123**
B.0.1 Concepts 123
B.0.2 Archetypes 133

C Benchmark compilation time **145**
C.1 Common structures 145
C.2 Constraints 153
C.3 Index 160

List of Figures

1.1 Illustration of the specter of the multitude of possibilities in the image processing world. 16

2.1 Watershed algorithm applied to three different image types. . 25
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). 25
2.3 Fill algorithm skeleton with a switch/case dispatcher to ensure exhaustivity. 27
2.4 Fill algorithm for a generalized supertype. 28
2.5 Dynamic, object-oriented polymorphism (a) vs. static, parametric polymorphism (b). 29
2.6 Benchmark: dilation of a 2D image (3128x3128 ≈10Mpix) with a 2D square and a 2D disc. 33
2.7 C++0x SFINAE detection of nested sub-type. 37
2.8 C++0x cloning example with covariant return type. 38
2.9 C++0x not-working cloning example with smart pointers. . . 39
2.10 C++0x working cloning example with smart pointers. 40
2.11 Fill algorithm, generic implementation. 41
2.12 Dilation algorithm, generic implementation. 43
2.13 Benchmark GCC-10: Benchmarking compilation speed of template instanciation constrained by Concepts (vanilla) vs. Concepts (improved) vs. SFINAE. 47
2.14 Benchmark GCC-10: Benchmarking compilation speed of template instanciation constrained by Concepts vs. SFINAE. . . 48
2.15 Benchmark Clang-10: Benchmarking compilation speed of template instanciation constrained by Concepts (vanilla) vs. Concepts (improved) vs. SFINAE. 49

2.16 Benchmark Clang-10: Benchmarking compilation speed of template instanciation constrained by Concepts vs. SFINAE. 49
2.17 Benchmark Clang-11: Benchmarking compilation speed of template instanciation constrained by Concepts (vanilla) vs. Concepts (improved) vs. SFINAE. 50
2.18 Benchmark Clang-11: Benchmarking compilation speed of template instanciation constrained by Concepts vs. SFINAE. 50

3.1 Concepts in C++20 codes 62
3.2 Set of supported image type. 63
3.3 Comparison of implementation of the `fill` algorithm for two families of image type. 64
3.4 Dilate algorithm with decomposable structuring element. . . . 64
3.5 Algorithm specialization within a set. 65
3.6 Dilate vs. Erode algorithms. 65
3.7 New Dilate vs. Erode algorithms. 66
3.8 Local algorithm canvas. 67
3.9 Local algorithm canvas. 67

4.1 Range-v3's ranges (a) vs. multidimensional ranges (b). 80
4.2 Lazy-evaluation and *view* chaining. 82
4.3 Abstract Syntax Tree of the types chained by the code above 82
4.4 An image view performing a thresholding. 84
4.5 Comparison of a legacy and a modern pipeline using algorithms and views. 85
4.6 Usage of transform view: grayscale. 86
4.7 Clip and filter image adaptors that restrict the image domain by a non-regular ROI and by a predicate that selects only even pixels. 86
4.8 Example of a simple image processing pipeline. 86
4.9 Algorithm vs image view composition. 87
4.10 Using high-order primitive views to create custom view operators. 88
4.11 View composition seen as an expression tree. 88
4.12 Background substraction pipeline using algorithms and views. 89

List of Tables

2.1 Genericity approaches: pros. & cons. 30

3.1 Concepts formalization: definitions 61
3.2 Concepts formalization: expressions 61
3.3 Concepts Index: expressions 73
3.4 Concepts Value: expressions 73
3.5 Concepts Point: expressions 73
3.6 Concepts Pixel: definitions 74
3.7 Concepts Pixel: expressions 74
3.8 Concepts Ranges: definitions 74
3.9 Concepts Ranges: expressions 75
3.10 Concepts Domain: definitions 75
3.11 Concepts Domain: expressions 75

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 "rec-

ognize 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 [66] 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 improve-

ment, 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.

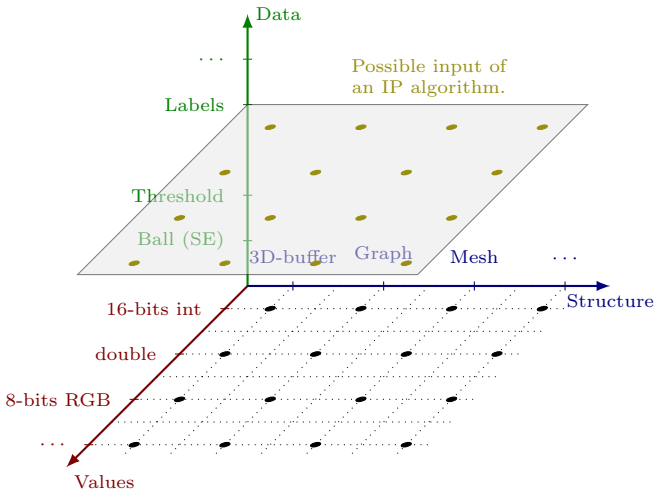


Figure 1.1: Illustration of the specter of the multitude of possibilities in the image processing world.

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 [85] or The GIMP [79]. 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 [79] or Photoshop [85]. 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 [91], GraphicsMagick [81] or MegaWave [43, 25].

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 [77] 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 [82], Scilab [83], Octave [89], Mathematica [84] and Jupyter [63] 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 [92] and Conda [80]. 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 [17], NumPy [30], scikit-image [59], Pillow [86] as well as binding for OpenCV [9].

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 [26], ITK [53], Boost.GIL [28], Vigra [14], GrAL [27], DGTal [62], OpenCV [9], CImg [49], Video++ [56], Generic Graphic Library [13] Milena [32, 35] and Olena [97, 41, 44, 58].

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 [54] and SYCL [75, 74] 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 [34], Blaze [45, 46], Blitz++ [15]

or Armadillo [65]. They leverage a possibility of the C++ programming language (*expression templates* [6]) 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 [70]. 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 [73]) 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.

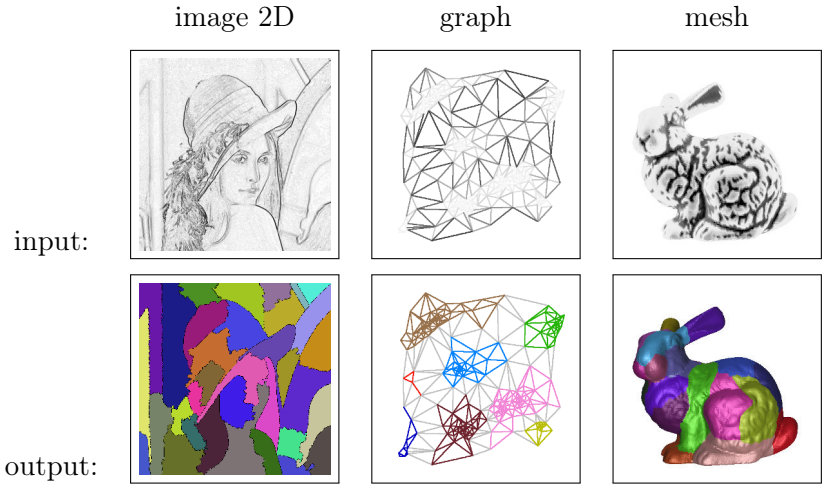
Genericity within programming language is described by the ability of the language to execute a code statement over a large amount of data structure, be they native (char, int, ...) or not (user defined). It is nowadays primordial for a programming language to be able to do so nowadays. Indeed, in a 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.

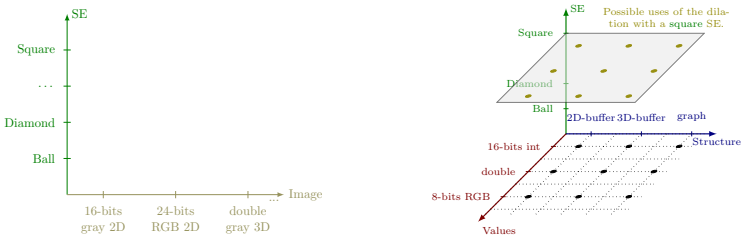
In image processing, there are 3 main axes around which genericity is working. The first axis is about the data type: grey level or RGB color (8-bits, 10-bits), decimal (double) and so on. The second axis, is about the structure of the image: a contiguous buffer (2D or 3D), a graph, a look-up table and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



(a) (b)

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, 10]. 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 [33], 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 [78] 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 [70] 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 pro-

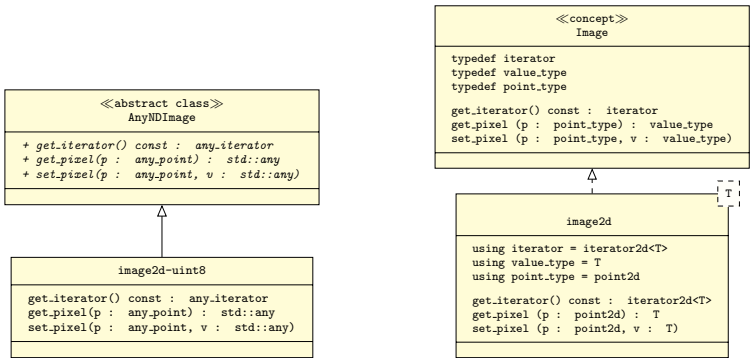
cess the supertype into the algorithm pipeline requested by the user, finally convert back the resulting image into the specific image type the user is expecting. This approach offers the advantage of being maintainable. Adding a new image type is just a matter of providing the two conversions facilities: 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 [9], scikit-image [60] use this technique to handle as many image types as possible. There are other libraries, for instance CImg [49], MegaWave [43], 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 [78], 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 [26],




```
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 [53], Boost.GIL [28], Vigra [14], GrAL [27], DGTal [62], Milena [32, 35], Olena [97, 41, 44, 58] and Pylena [70]. 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 [54] and SYCL [75, 74] both provide their own DSL to that end. Others like Eigen [34], Blaze [45, 46], Blitz++ [15] or Armadillo [65] 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* [6]. 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 [19]. 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 [48], 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.

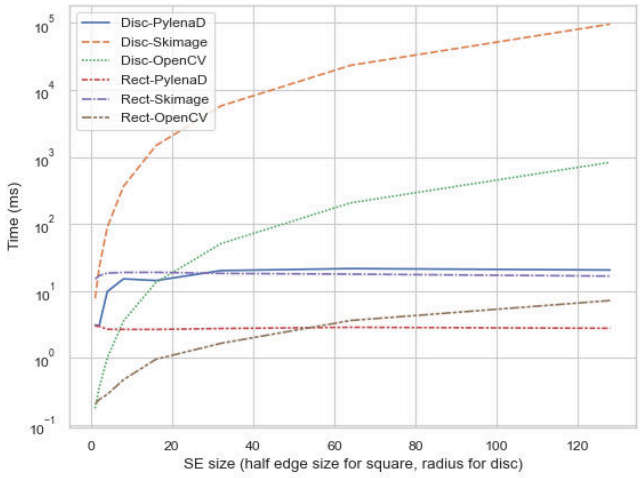


Figure 2.6: Benchmark: dilation of a 2D image (3128x3128 ≈10Mpix) with a 2D square and a 2D disc.

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 [39] came out the genericity facilities offered by the C++ programming language were already turing complete [23]. 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 [20] (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 [8] and C++03 [22] 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 [87] 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 [21, 29, 31]. 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 [97, 44] was born and carried the work

around this field of research applied to Image processing in [12, 11, 18, 24]. This design is described in details by Levillain among his work [32, 35, 36, 42, 41, 47, 58]. Finally, with the release of new C++ standards in 2011 [39], then 2014 [57], 2017 [67] 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 [70]. In the end, it was C++20 [39] that marked the shift wanted by Stepanov [1, 4, 10, 33] and Stroustrup **????????** for years with the coming of Concepts, and all the new possibilities it brings to the programmer.

```
// 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.

```cpp
#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) [10] 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 [95, 94, 96] to finally appear in the 2020 standard [98] (referred as C++20 [39]). 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 [95, 94, 96], it will finally appear in the next standard [98] (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 [33], 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, 12, 18, 35, 58]. 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 optimised 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 construct 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 over-

load 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 [88], 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.

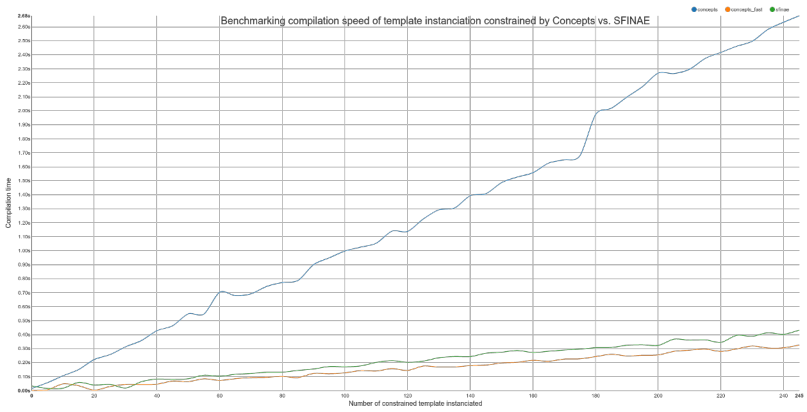


Figure 2.13: Benchmark GCC-10: Benchmarking compilation speed of template instanciation constrained by Concepts (vanilla) vs. Concepts (improved) vs. SFINAE.

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-

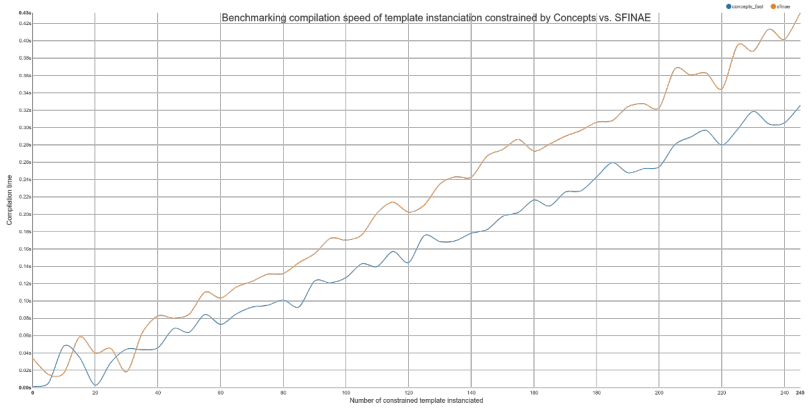


Figure 2.14: Benchmark GCC-10: Benchmarking compilation speed of template instanciation constrained by Concepts vs. SFINAE.

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.

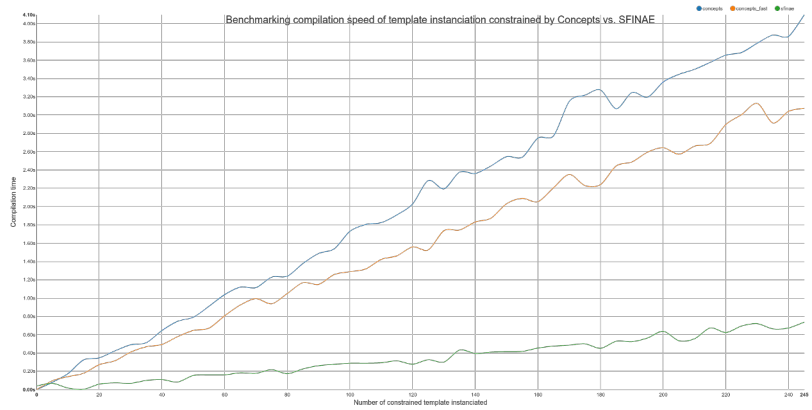


Figure 2.15: Benchmark Clang-10: Benchmarking compilation speed of template instanciation constrained by Concepts (vanilla) vs. Concepts (improved) vs. SFINAE.

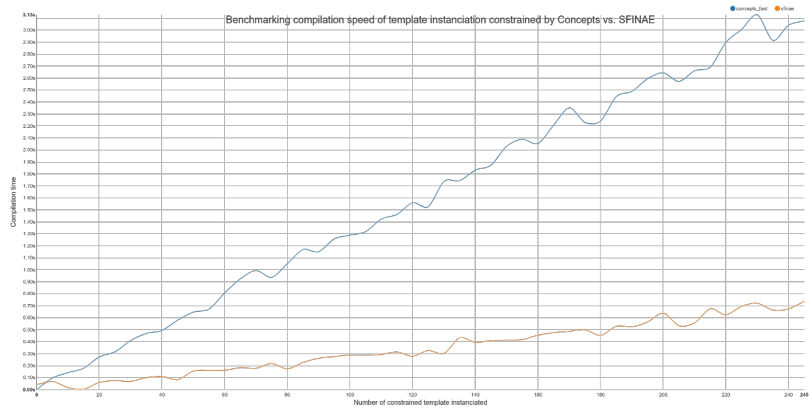


Figure 2.16: Benchmark Clang-10: Benchmarking compilation speed of template instanciation constrained by Concepts vs. SFINAE.

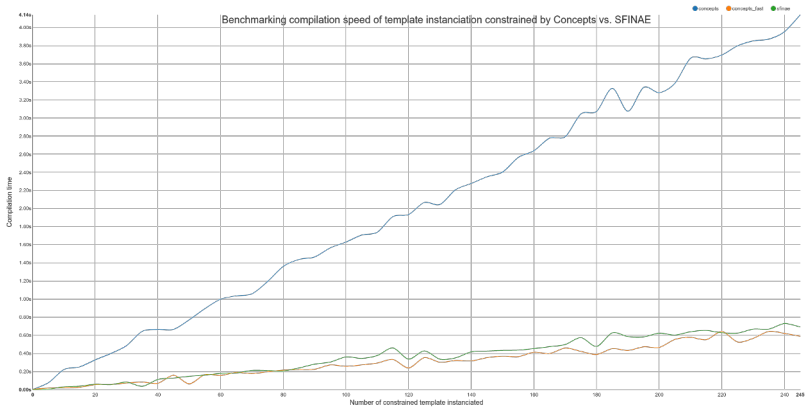


Figure 2.17: Benchmark Clang-11: Benchmarking compilation speed of template instanciation constrained by Concepts (vanilla) vs. Concepts (improved) vs. SFINAE.

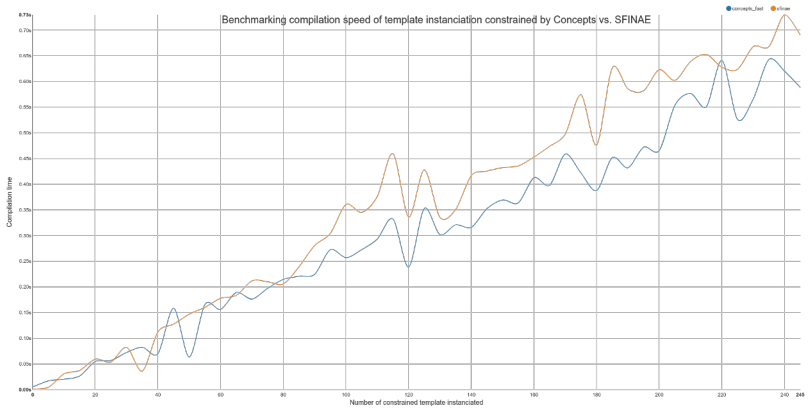


Figure 2.18: Benchmark Clang-11: Benchmarking compilation speed of template instanciation constrained by Concepts vs. SFINAE.

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 [50] 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 [7, 97] 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 [14], chose this solution.

VCSN [51] 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 [38] 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 essentialy bypass the work of a JIT compiler (that would be used by a programming using bytecode such as Java or C#) and just offload it onto well known/proven solution: the machine's C compiler.

Autowig [71], Cppyy [64] and Xeus-cling [90] are all solutions aiming to generate automaticaly 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 [63] 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

TODO : Definition of images and algorithms (introduction)

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.

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*".

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 *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 [99, 101] (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 [100]. 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 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 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 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.

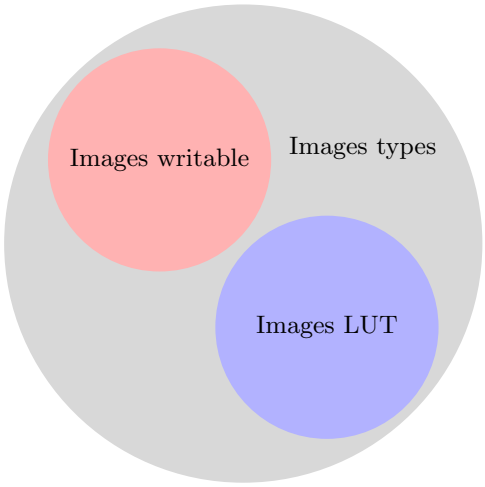


Figure 3.2: Set of supported image type.

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. *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

$fill(I, v) : \forall p \in \mathcal{D}, I(p) = v$

(a) Writable image fill algorithm

$fill(I, v) : \forall i \in I.LUT, i = v$

(b) Image LUT fill algorithm

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

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.

There are more details that go in depth when considering the distinction

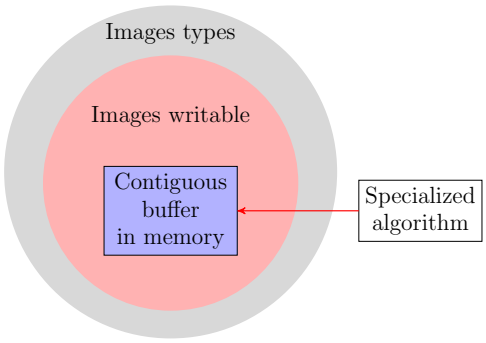


Figure 3.5: Algorithm specialization within a set.

between runtime dispatch (dynamic) and compile time dispatch (static) that the interested reader can consult in appendix (appendix.dispatch.dyn.static). Those details are not mandatory for the overall comprehension of this paper, hence why they are omitted here.

3.3 Generic aspect of algorithm: canvas

modèles de programmation pour traiter une image : * modèle type kernel/tuile (CUDA, Halide) sur GPU * modèle type pipeline pour traitement complex * modèle type deep learning (conv conv conv = filtres) + fonction de réduction dans des réseaux parrallèle entre canvas et modèles programmatic dissociation du parcours et du traitement

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. First let us study two basic algorithm: dilation and erosion. The python code of such algorithm is naively be given in figure 3.6.

```
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

```
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

```
1  def local_canvas(img, out, se):
2    # do something before outer loop
3    for pnt in img.points():
4      # do something before inner loop
5      for nx in se(pnt):
6        # do something inner loop
7      # do something after inner loop
8    # do something after outer loop
```

Figure 3.8: Local algorithm canvas.

2, 4, 6, 7 and 8. The user would then provide callbacks and the canvas would 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 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 [75, 74], Boost.SIMD [55] or even VCL [52]. 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 [54]: 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 abstract 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 [16] for similar purposes.

quote C++Now 2019: Ben Deane "Identifying Monoids: Exploiting Compositional Structure in Code"

```
## Issue: wide subject, a lot to consider

Genericity: "write once, work for every types, run everywhere"

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.

There is no simple answer to this question.

## Challenge: find the customization point

* Find what can be parallelized/distributed.
* Find an algorithm abstraction.

For us it is the local algorithms. They all have an accumulation pattern:

```cpp
  template <class In, class Out, class SE, class T, class Op>
  auto local_accumulate(In input, Out output, SE se, T init, Op op)
  {
    for(auto&& row : ranges::rows(input.pixels()))
      for(auto p : row)
      {
        auto v = init;
        for(auto nb : se(p))
          v = op(init, nb.val());
        init += v;
      }
    return init;
  }
```

Data structure requirement: monoid

\small
Monoid, definition:

1. operates over a set of values, finite or infinite
2. provides a binary operation which is closed and associative
3. provide a neutral element, the identity

Easy monoids:
```

```
* boolean, *and*, "true" -> binary erosion
* boolean, *or*, "false" -> binary dilation
* unsigned integer, *max", "0" -> dilation
* signed integer, *min", "global max" -> erosion

## Solution: algorithms canvas

\scriptsize

* Algorithm canvas: standard way to iterate over an image to perform a
  computation
* Structure similar to accumulation for local algorithms
* Friendly for factorizing code
* Decoupling image traversing from actual computation
* Change of paradigm: procedural -> event-driven
* Non intuitive way to decompose an algorithm for the user
* Allow customization points by maintainer without restraining the user.

```cpp
  Image img_in = ..., img_out = ... ;
  StructuringElement se = ... ;
  auto z = ranges::view::zip(img_in.pixels(), img_out.pixels());
  for (auto rows : ranges::rows(z)) {
    user_callbacks->ExecuteAtLineStart();
    for (auto [px_in, px_out] : rows) {
      user_callbacks->EvalBeforeLocalLoop(px_in.val(), px_out.val());
      for (auto nbhs_in : se(px_in))
        user_callbacks->EvalInLocalLoop(nbhs_in.val(), px_in.val(),
                                        px_out.val());
      user_callbacks->EvalAfterLocalLoop(px_in.val(), px_out.val());
    }
  }
```
```

3.4 Library concepts: listing and explanation

3.4.1 Index

3.4.2 Value

Let *Idx* be a type that models the concept *Index*. Let *idx* and *idy* be an instance of *Idx*. Then we have the following valid expressions:

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 3.3: Concepts Index: expressions

Let *Val*, *CmpVal* and *OrdVal* be types that models the concepts resp. *Value*, *ComparableValue* and *OrderedValue*. Let *val*, *cmp_val* and *ord_val* be instances of types resp. *Val*, *CmpVal* and *OrdVal*. Then we have the following valid expressions:

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`	`boolean`	supports inequality comparisons
	`ord_val1 <= ord_val2, ...`	`boolean`	

Table 3.4: Concepts Value: expressions

3.4.3 Point

Let *Pnt* be a type that models the concept *Point*. Let *pnt* be an instance of type *Pnt*. Then we have the following valid expressions:

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`	`boolean`	supports inequality comparisons
	`pnt1 <= pnt2, ...`	`boolean`	

Table 3.5: Concepts Point: expressions

3.4.4 Pixel

3.4.5 Ranges

3.4.6 Domain

Let *Pix* and *OPix* be types that model the concepts resp. *Pixel* and *OutputPixel*. Then *OutputPixel* inherits all types defined for *Pixel*. Then we can define:

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. Can be a proxy.	Models the concept *Pixel*
	`point_type`	Type of the pixel's point.	Models the concept *Point*
OutputPixel		Inherit *Pixel*'s definitions. No additional definition.	

Table 3.6: Concepts Pixel: definitions

Let *pix*, *opix*, *pnt*, *val* be instances of types resp. *Pix*, *OPix*, *Pix::point_type* and *Pix::value_type*. Then we have the following valid expressions:

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 3.7: Concepts Pixel: expressions

Let *MDRng*, *OMDRng* and *RMDRng* be types that model the concepts resp. *MDRange*, *OutputMDRange* and *ReversibleMDRange*. Then we can define:

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. Can be a proxy.	Models the concept `std::indirect`
OutputMDRange		Inherit *MDRange*'s definitions. No additional definition.	
ReversibleMDRange		Inherit *MDRange*'s definitions. No additional definition.	

Table 3.8: Concepts Ranges: definitions

Let *mdrng*, *omdrng*, *rmdrng* and *val* be instances of types resp. *MDRng*, *OMDRng*, *RMDRng* and *std::ranges::range_value_t<MDRng>*. Then we have the following valid expressions:

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 3.9: Concepts Ranges: expressions

Let *Dom*, *SDom*, *ShDom* be types that model the concepts resp. *Domain*, *SizedDomain* and *ShapedDomain*. Then *Domain* inherits all types defined for *MDRange*. Then *SizedDomain* inherits all types defined for *Domain* and *ShapedDomain* inherits all types defined for *SizedDomain*. Then we can define:

Concept	Definition Description Requirement
Domain	Inherit *MDRange*'s definitions. No additional definition.
SizedDomain	Inherit *Domain*'s definitions. No additional definition.
ShapedDomain	Inherit *SizedDomain*'s definitions. No additional definition.

Table 3.10: Concepts Domain: definitions

Let *dom*, *sdom*, *shdom* and *pnt* be instances of types resp. *Dom*, *SDom*, *ShDom* and *Dom::value_type*. Then we have the following valid expressions:

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 3.11: Concepts Domain: expressions

- Définitions des types/catégories de types/propriétés de types
- Conceptualisation expliquer par l'exemple (papier rrpr) la relation Image (n) <-> Implem (m) avec n » m
  - Concepts déduis des algorithmes et non des types
  - Plusieurs algos pour le même opérateur
  - Plusieurs implems pour le même algo
- Vision ensembliste des types d'images/algo
  - versions d'algorithmes
  - spécialisations d'algorithmes
- canvas d'algorithmes
  - factorisation
  - opportunités d'optimisation (utilisation de propriétés, parallélisation)
- listing et explications des concepts de la bibliothèque (images, élément structurant)

Chapter 4

Image views

4.1 The Genesis of a new asbtraction layers: Views

4.2 Upgrading the way to design IP algorithms

4.2.1 Keeping properties

4.2.2 Lazy evaluation

4.2.3 Composability and piping

4.3 A practical example: border management

4.4 Performance discussion

Material about views

Introducing range-based image traversing

Previously in Milena [32] we had a very customized way of traversing images: macro-based. It aimed at hiding the complexity of such a task while loosing as little performance as possible. For example, the dilation algorithm was written this way:

```
template<class I, class SE>
mln_concrete(I) dilate(const I& f, const SE& se)
{
  mln_concrete(I) g;
  initialize(g, f);
  mln_piter(I) p(f.domain());
  mln_qiter(SE) q(se, p);
  for_all(p) // for all p in f domain
  {
    mln_value(I) v = f(p);
    for_all(q) // for all q in se(p)
      if(f.has(q) and f(q) > v)
        v = f(q);
    g(p) = v;
  }
  return g;
}
```

In this code `mln_concrete`, `mln_piter`, `mln_qiter`, `for_all` and `mln_value` are all macros aiming at hiding the underlying complexity. Our goal is to replace those macros with existing C++ core language code to improve the user experience as well as ease the maintenance, contribution and further improvement of the library. Nowadays, a tool named range (and especially Eric Niebler's range-v3) allow seamless traversing of an image. For instance, we can rewrite the above algorithm this way:

```
template<class I, class SE>
image_concrete_t<I> dilate(const I& f, const SE& se)
{
  auto g = f.concretize();
```

```
  auto supr = accu::supremum<image_value_t<I>>();
  for(auto [f_px, g_px] : zip(f.pixels(), g.pixels()))
  {
    for(auto qx : se(f_px))
      supr.take(qx.val());
    g_px.val() = supr.result();
  }
  return g;
}
```

Here we have a much more efficient code that, in theory, enables compiler optimizations such as vectorization or inner loops unrolling. But through benchmarking, we have learned that this solution doesn't mix well with the multidimensional nature of images. The issue originates from the fact that we have no way to explicitly say in the code that the multidimensional range is made of chunk of contiguous rows of memory. Indeed, for each element we have to compute an index originating from potentially N dimensions. This disables critical optimizations such as vectorization.

We solved this problem by augmenting range-v3's ranges with our own multidimensional ranges. Indeed, we only need to have contiguity on the last dimension to provide the compiler code it can optimize. Which means that each for-loop that traverses the whole n-dimensional image can be transformed into a double for-loop whose inner loop is guaranteed to be a contiguous row. This way we have now an outer range as well as an inner range, as illustrated in figure 4.1.

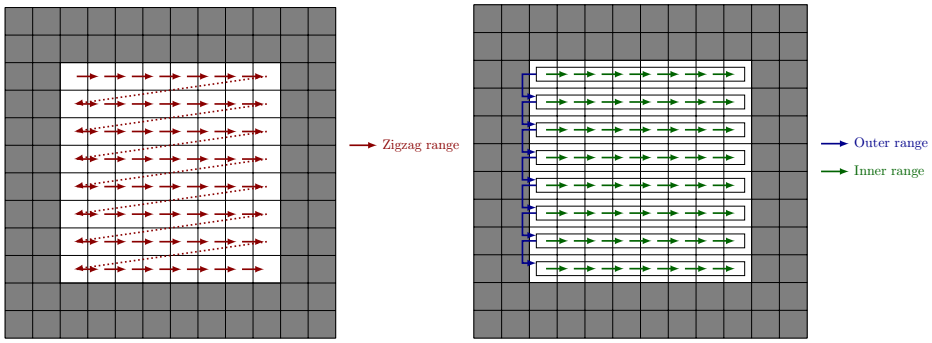


Figure 4.1: Range-v3's ranges (a) vs. multidimensional ranges (b).

Thanks to this new design we can now rewrite our algorithm with a double for-loop for the image traversing. Hopefully it stays really similar to what one would be used to when working with the classical two dimension image. As an example, we can rewrite the dilation algorithm this way:

```
template<class I, class SE>
image_concrete_t<I> dilate(const I& f, const SE& se)
{
  auto g = f.concretize();
  auto supr = accu::supremum<image_value_t<I>>();
  auto zipped_pixels = zip(f.pixels(), g.pixels());
  for(auto&& row : ranges::rows(zipped_pixels))
    for(auto [f_px, g_px] : row)
    {
      for(auto qx : se(f_px))
        supr.take(qx.val());
      g_px.val() = supr.result();
    }
  return g;
}
```

The highlight of this code is the usage a new tool: `ranges::rows` to bring out an inner range (contiguous) from the multidimensional outer range.

Introducing Views

Practical genericity for efficiency: the Views

Let us introduce another key point enabled by genericity and concepts: the *Views*. A *View* is defined by a non-owning lightweight image, inspired by the design introduced in *Ranges for the Standard Library* [93] proposal for *non-owning collections*. A similar design is also called *Morphers* in Milena [32, 44]. *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 [34]), thus it knows at compile-time the type of the composition and ensures a 0-overhead at evaluation.

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

Lazy-evaluation combined with the view *chaining* allows the user to write clear and very efficient code whose evaluation is delayed till very last moment as shown in fig. 4.2 (see [72] 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.

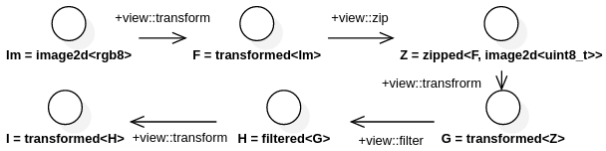


Figure 4.3: Abstract Syntax Tree of the types chained by the code above

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 [78]) that will not be presented in this paper. Additionally to concepts, C++20 introduces a new library facility called *ranges* [101] which includes a non-owning lightweight container called *views* whose design is very similar to that of *morphers*, introduced in Milena [32, 44]. 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. 4.4. 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.

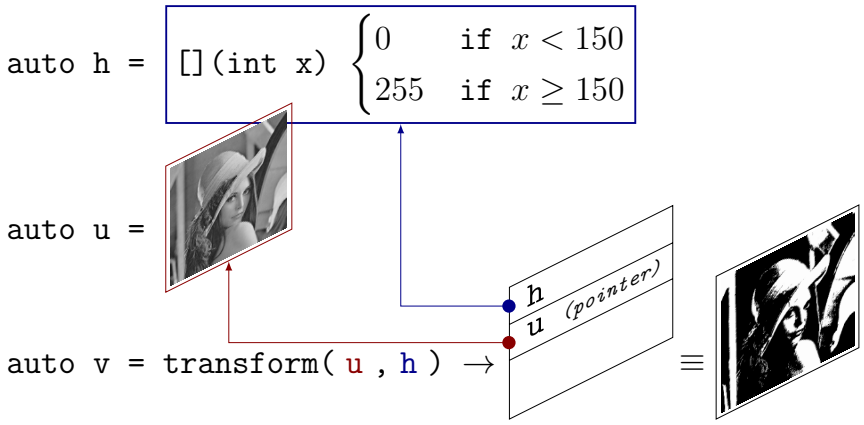
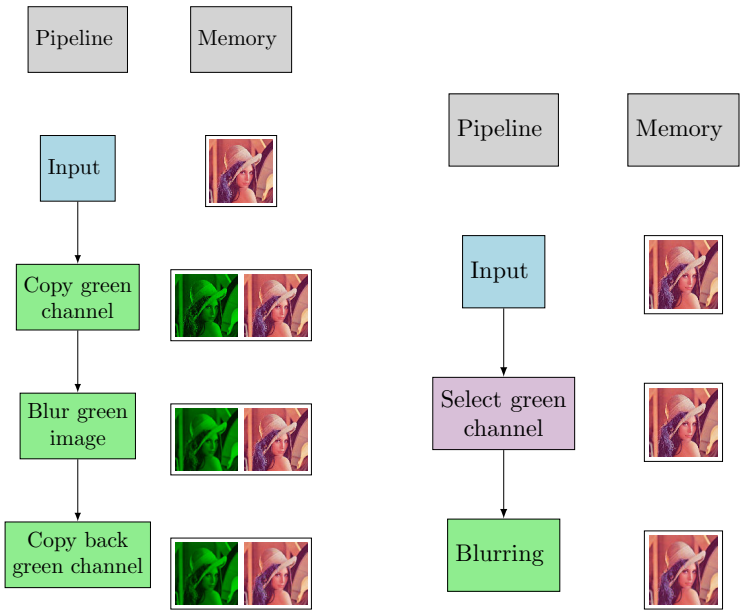


Figure 4.4: An image view performing a thresholding.

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 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.

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. 4.5.



(a) Legacy pipeline with copy

(b) Modern pipeline with view

Figure 4.5: Comparison of a legacy and a modern pipeline using algorithms and views.

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. 4.6), 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. 4.7, 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

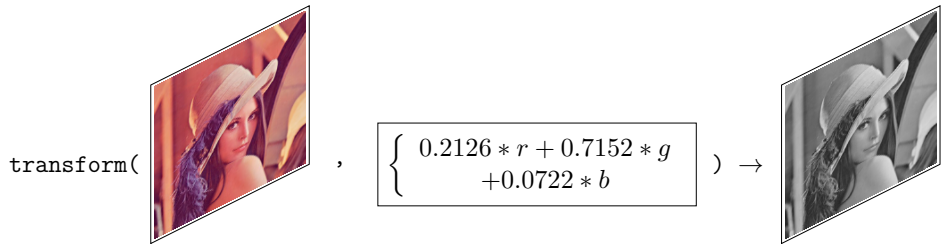


Figure 4.6: Usage of transform view: grayscale.

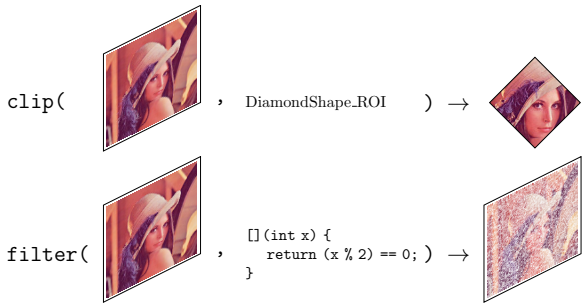


Figure 4.7: Clip and filter image adaptors that restrict the image domain by a non-regular ROI and by a predicate that selects only even pixels.

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. 4.8.

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. 4.8 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. 4.9 we show the distinction

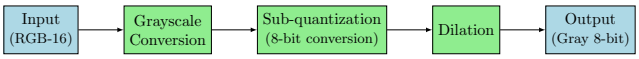


Figure 4.8: Example of a simple image processing pipeline.

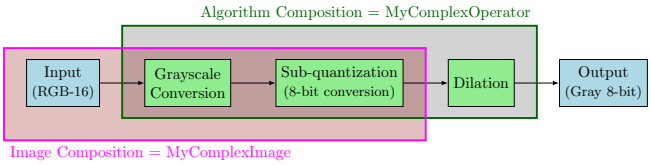


Figure 4.9: Algorithm vs image view composition.

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. 4.9 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* [5] 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. 4.10, 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. 4.10, 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

[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 4.10: Using high-order primitive views to create custom view operators.

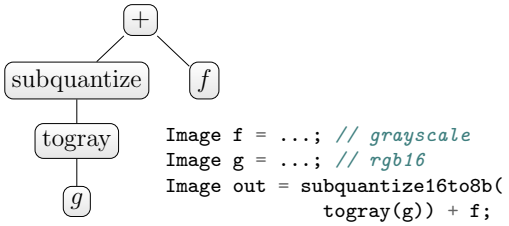


Figure 4.11: View composition seen as an expression tree.

fact, a view is an image adaptor which actually is a *template expression* [6, 15]. 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. 4.11.

Views for performance. Consider a classical design where each operation of the pipeline is implemented on "its own". Each operation requires that memory be allocated for the output image, and also, each operation requires that the image be traversed. This design is simple, flexible, composable, but is not memory efficient nor computationally efficient. With the lazy evaluation approach, the image is traversed only once (when the dilation is applied) that has two benefits. First, there are no intermediate images so this is memory effective. Second, it is faster thanks to a better memory cache usage; processing a RGB16 pixel from the dilation algorithm directly converts it in grayscale, then sub-quantized it to 8-bits, and finally make it available. It acts *as if* we were writing an optimal operator that would combine these operations.

As an experiment, we benchmarked both pipelines on a 20MPix image

RGB16 (random generated values) on a desktop computer i7-2600 CPU @ 3.40GHz, single-threaded[2]. The dilation is done with a small 3x3 square structuring element using tiling for caching input values. The pipeline using views is about 20% faster than the regular one (133 vs 106 ms). Note that views are also compatible with optimizations such as parallelization and vectorization.

Background Subtraction: The background subtraction pipeline is used to detect changes in image sequences [69]. 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 fig. 4.12.

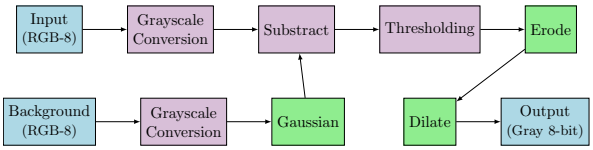


Figure 4.12: Background substraction pipeline using algorithms and views.

From local algorithm to border management

In image processing there are three main families of algorithms:

- Point-wise algorithms: they process each value of the image, individually from each others (e.g. multiplication).
- Local algorithms: they process each value of the image, helped with the knowledge of the neighboring values catched in a window: the structuring element (e.g. convolution).
- Global algorithms: they propagate the changes downstream during the algorithm unrolling. Knowledge of previously computed values is required (e.g. chamfer distance transform).

Here we are interested in particular in local algorithms. The 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

[2]Experimentation code is available at https://gitlab.lrde.epita.fr/mroynard/roynard.icip.2020.snippets

over the neighbors. 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 defacto 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 algorithm 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 couple (image and structuring element) given a set of configuration wanted by the user.

We designed the configuration to be configured 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 the values of our extension. 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: 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.
- Fill: will fill the border with a specific value.
- Mirror: will mirror the image following an axial symmetry of image's edges.
- Periodize: replicate the image around, like a mosaic.
- Clamp: expand the value at the image's edge on the border.
- Image: assume 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.
- User: Assume 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 wether 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 eased:

```
// 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.

put schemas showing different type of border management

this can be intro of chapter Our work on bringing genericity to image processing through C++ template metaprogramming and concepts is presented in [78]. Continuing this previous work, we focused on the main goals of the library. First is the zero-cost of traversing an image, overhauling the existing system which is macro-based. The new system presented in section (X) is based on range-v3 [93, 101, 99]. Following, we will show in section (X) a design around image's domain definition, extension and structuring element: how those concepts link with each other when used to write local algorithms. Another objective of the library is the ease of use and the binding with existing ecosystem, such as Python and NumPy. So we will present in section (X) our work about binding a generic C++ library to python. Finally, we will discuss in section (X) the issue of heterogeneous computing. We offer an avenue around the idea of canvas algorithm that can be exploited to bring performance without being intrusive for the user. Though it comes with its lot of disadvantage, such as a paradigm switch to event-driven programming.

4.5 Benchmark figures

- origine, parallèle avec range-v3
- Value/Ref semantics des images
- Comment préserver les propriétés
- Évaluation paresseuse
- Composabilité/chaînage
- ...
- Performances + Bench

Chapter 5

Static dynamic bridge

5.1 Coexistance between the static world and the dynamic world

5.2 Hybrid solution: $n * n$ dispatch thanks to variants

5.3 JIT-based solutions: pros. and cons.

Material about Static dynamic bridge

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 [30] 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 exists that could bring solutions to lower the number of dispatcher to write, such as multi-method [37]. 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 [76], a student co-supervised by the authors of this report, in order to type-erased the most important types (images) as well as the algorithms. We then embark runtime information about carried types in the type-erased object. When the algorithm is called on the type-erased object, we attempt to cast this object into a known set of types (dimensions: 2, 3, 4; value: int, double, rgba). Should the cast succeed then we can call the fast, optimized routine. Otherwise we fallback onto an algorithm relying on dynamic dispatch even for its most basic operation (e.g. addition).

Let us illustrate this path with an example: the stretch algorithm. First let's see the naive algorithm:

```
template <typename T>
```

```
image2d<float> stretch(const image2d<T>& src)
{
  auto res = image2d<float>(src.width(), src.height());
  auto values_span = src.values();
  std::transform(values_span.begin(), values_span.end(), res.values().begin(),
    [](T val) -> float
    {
      return static_cast<float>(val) / std::numeric_limits<T>::max();
    }
  );
  return res;
}
```

One should observe that the function parameter is templated by a type `T`. This induces that "span" further in the code is also templated. Both needs to be type erased. Furthermore, getting the max value of the type `T` is also an issue that needs to be abstracted away.

We aim at obtaining this prototype:

```
image2d<> stretch_py(const image2d<>& src);
```

Here `image2d<>` which is also `image2d<void>` is the type-erased type of `image2d<T>`. This means that we have hierarchy where `image2d<T>` inherits from `image2d<void>`. We then introduce an intermediate step:

```
1   template <typename T>
2   struct apply_stretch_t
3   {
4     auto operator()(const image2d<>& src)
5     {
6       return stretch(*src.cast_to<T>());
7     }
8   };
9
10  image2d<float> stretch(const image2d<>& src)
11  {
12    return visit<apply_stretch_t>(src.type().tid(), src);
13  };
```

This piece of code is interesting in many ways. It introduces the way we dispatch according to the known types (line 12): there is the embedded runtime type information we are going to use for the dispatch. Then there is the call to `visit` parametrized by the templated structure `apply_stretch_t`. This visitor will statically instantiate the correct `apply_stretch_t` with the correct dynamic type (src.type().tid()) and call a non type-erased version of the function stretch after having cast the values of the images. However, for this to work we still have to tweak the first implementation a little.

```
1   template <class T>
2   image2d<float> stretch(const image2d<T>& src)
3   {
4     auto        res  = image2d<float>(src.width(), src.height());
5     auto        span = src.values();
6     const auto& vs   = src.get_value_set();
7     std::transform(span.begin(), span.end(), res.values().begin(),
8                    [&vs](auto val) -> float
9                    {
10                     auto tmp = vs.max();
11                     return vs.template cast<float>(val) / vs.template cast<float>(tmp);
12                   });
13    return res;
14  }
```

We introduce a new tool: the value-set (line 6). This value-set is a type-erased way to provide basic operations on types such as casting, division, addition, getting the global max etc. This powerful tool, also embedded in the image, allow us to write the algorithm in a generic way lines 10-11.

Thanks to this design, we have been able to type-erase our image types so that our algorithms can be called through python via bindings generated by pybind [68]. As an example, we can then call our stretch algorithm this way:

```
import pylena as pln, imageio, numpy as np
img_in = imageio.imread("lena.png")
np_arr = np.array(img_in, dtype='int8')
img = pln.image2d(np_arr)
print(timeit.timeit('img_out = pln.stretch(img)', number=1000,
      globals=globals()))
>> 0.24129085899949132 # Seconds for 1000 cycles

# Using a type that isn't int8 or int16
np_arr64 = np.array(img_in, dtype='int64')
img64 = pln.image2d(np_arr64)
print(timeit.timeit('img_out64 = pln.invert(img64)', number=1000,
      globals=globals()))
>> 26.124844277999728 # Seconds for 1000 cycles
```

We can observe a hundred time factor between the fast and optimized path and the slow dynamically dispatched path.

- rappel de la problématique (backward ref depuis Généricité/4.)
- expliquer l'approche hybride (son design et les techniques de dispatch n*n avec variants)
- bench, trade-off
- continuité sur JIT avec autowig, cppyy (perspective)

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 writting 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 [61], 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 [64] 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 [64]

[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 [40] 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] 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.

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

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

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

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

[15] 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.

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

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

[18] 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.

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

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

[21] 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.

[22] 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.

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

[24] 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.

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

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

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

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

[29] 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.

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

[31] 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.

[32] 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.

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

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

[35] 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.

[36] 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/.

[37] 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.

[38] 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.

[39] 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.

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

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

[42] 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.

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

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

[45] Klaus Iglberger. *Blaze C++ Linear Algebra Library*. https://bitbucket.org/blazelib. 2012.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

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

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

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

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

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

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

[63] 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/.

[64] 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.

[65] 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.

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

[67] 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.

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

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

[70] 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.

[71] Pierre Fernique and Christophe Pradal. "AutoWIG: automatic generation of python bindings for C++ libraries". In: *PeerJ Computer Science* 4 (2018), e149.

[72] 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.

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

[74] 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.

[75] 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.

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

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

[78] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[101] 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;
  };
```

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, const Pnt pnt) {
    { cext.fit(se) }      -> bool;
    { cext.extent() }     -> int;
  };

template <typename Ext>
concept FillableExtension =
  Extension<Ext> &&
  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() }  -> bool;
  };

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

template <typename Ext>
```

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

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

template <typename Ext, typename U>
concept ExtendWithExtension =
  Extension<Ext> &&
  std::convertible_to<typename Ext::support_extend_with, std::true_type> &&
  InputImage<U> &&
  requires {
    typename Ext::value_type;
    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() }  -> bool;
  };
```

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 } -> std::same_as<image_reference_t<I>>;
    };
} // 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
{
  // 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 };
    };
} // 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

// 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 viewu
    { cima.stride(dim) } -> std::same_as<std::ptrdiff_t>;
  };


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

```
      { *(ima.data()) = v };
    };
} // namespace detail


// OutputImage
// Usage: RawImage<I> && OutputImage<I>
template <typename I>
concept OutputImage =
  (not ForwardImage<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>> &&
  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>;
```

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.inc() }  -> StructuringElement<P>;
    { se.dec() }  -> StructuringElement<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))>>;
  };
```

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!");
```

Extension

```
  // TODO
```

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
  {
    // FIXME
```

```
    };

    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>;
```

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;

    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>>;
```

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);
```

```
  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):

```
// 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;
}
```