The Cost of Dynamism in Static Languages for Image Processing

Baptiste Esteban, Edwin Carlinet, Guillaume Tochon, Didier Verna

This article is reformated in order to not be an editor version and uploaded on HAL. Its original version is available with this DOI: 10.1145/3564719.3568693

The Cost of Dynamism in Static Languages for Image Processing

Baptiste Esteban, Edwin Carlinet, Guillaume Tochon and Didier Verna
EPITA Research Laboratory
14-16 Rue Voltaire
94270 Le Kremlin-Bicêtre
FRANCE

Generic programming is a powerful paradigm abstracting data structures and algorithms to improve their reusability, as long as they respect a given interface. Coupled with a performance-driven language, it is a paradigm of choice for scientific libraries where the implementation of manipulated objects may change depending on their use case, or for performance purposes. In those performance-driven languages, genericity is often implemented statically to perform some optimization. This does not fit well with the dynamism needed to handle objects which may only be known at runtime. Thus, in this article, we evaluate a model that couples static genericity with a dynamic model based on type erasure in the context of image processing. Its cost is assessed by comparing the performance of the implementation of some common image processing algorithms in C++ and Rust, two performance-driven languages supporting some form of genericity. Finally, we demonstrate that compile-time knowledge of some specific information is critical for performance, and also that the runtime overhead depends on the algorithmic scheme in use.

Keywords: Genericity, static languages, type erasure, image processing

1 Introduction

Scientific tooling is an important part of the research process. In image processing, three criteria are of prime importance: genericity, performance, and interactivity. Genericity [11] allows using a single algorithm on several kinds of objects while they respect a predefined interface. The genericity may be on the type of the values of an image [8], but also its spatial definition [9] and its implementation (sparse-matrix, constant image). Performance is an important criterion to be able to handle large images in an image processing pipeline or for real-time applications. Finally, interactivity allows the researcher to change the pipeline at runtime without having to perform a new compilation at every change. This kind of interactivity is usually reached by bridging the functionalities written in a compiled language such as C++ into a dynamic language, more popular for algorithm prototyping, such as Python, having a large range of scientific utilities such as NumPy [5] for multi-dimensional arrays manipulation, Matplotlib [6] for visualization or IPython/Jupyter for interactive programming.

A lot of image processing libraries meet these criteria. They are implemented in C++, a performance-driven language, well-suited for efficient scientific applications through optimizations at compile time such as partial specialization [4], and endowed with genericity capabilities, through the use of templates. Those libraries handle a large range of image processing objects such as multi-dimensional arrays with generic values for Vigra [8], different kinds of graphs for Higra [12], or any kind of image for Olena [9]. Furthermore, using some libraries wrapping the CPython API into a more interactive API such as Pybind11 [7] or Boost.Python [1], their functionalities are easily usable in Python. However, Python extensions being compiled modules, C++ templated algorithms must be instantiated to allow their bridging. While Higra instantiates its functionalities for a large range of types, generating a large amount of code due to the monomorphization process of the C++ genericity mechanism, Vigra limits the choice of input arguments type, reducing the genericity of the exposed functionalities.

Thus, our main objective is to reduce the amount of generated machine code while keeping the generic abilities of the implemented algorithm. To this aim, we present in this article a model for generic image processing algorithms allowing both static and dynamic genericity based on operation functions and we evaluate its cost. We focus in this paper on two programming languages with a genericity mechanism relying on monomorphization, the C++ language, widely used for image processing, and the Rust language, which is increasingly becoming popular for its safety in terms of memory usage, multi-processing, and performance.

This article is structured as follows: we first recall the basis of genericity and we explain its application in image processing in section 2. Then, we explain our proposed model to handle both static and dynamic genericity in section 3. We compare the performance of the static and dynamic implementation in section 4. We finally conclude in section 5.

2 Generic Programming

2.1 Preliminaries

Reusability is important in the development process. It reduces the amount of code to be written and maintained. Generic programming [11] improves the ability of programming language to reuse existing components of a library. It abstracts an algorithm through a predefined interface for the objects used by it. As long as an object respects this interface, it can be used by this algorithm.

Genericity in the C++ and Rust languages is particularly interesting in terms of performance. Their mechanism for genericity is based on monomorphization, a process producing machine code for each concrete algorithm. This machine code is optimized by the compiler for each combination of type parameters, resulting in performant libraries and applications. Genericity in C++ is based on *templates*, a list of parameters containing either types or constant values which are filled at the instantiation of a concrete data structure or algorithm. Thus, each combination of parameters results in a new concrete type or algorithm. The Rust language genericity is based on a similar parameter list named *generics*. The illustration in Figure 1 shows a generic implementation of the `sum` function in C++ and Rust traversing an array, summing all its elements, and returning the results. In these two cases, the type parameter must respect the interface ensuring the usage of the operator `+`, either by an operator overloading in C++ or the implementation of the *trait bound* `Add` in Rust.

```cpp
// C++ version
template <class T>
T sum(std::vector<T> lst) {
  T res = T();
  for (const auto e : lst)
    res += e;
  return res
}
```

```rust
// Rust version
fn sum<T: Add<Output = T> + Default>(
  lst: Vec<T>,
) -> T {
  let mut res : T = Default::default();
  for e in lst {
    res = e + res;
  }
  res
}
```

Figure 1: C++ and Rust implementation of `sum`

2.2 Application to Image Processing

Let $f : \Omega \to \mathcal{V}$ be an image with Ω its domain and $\mathcal{V}$ its value space. The domain of definition of the image differs depending on which kind of image is used: it may be a multi-dimensional grid, the index of a node in a graph, etc... In the same way, the values of an image may be univariate (boolean for binary image, numerical for grayscale image) or multivariate (for color image). Algorithms may be used on all these kinds of images. An example of algorithms on specific images is shown in Figure 2. The two functions compute the elementwise maximum between two objects. The first function takes in argument a 2D image with the value encoded as an unsigned integer on 8 bits and the second function takes a node-valued graph. The pattern of the algorithm is the same for the two functions: the object is traversed, computing the maximum between two elements and storing it in a third object.

These functions are limited: if a new type is introduced, a new function has to be implemented. However, due to the common pattern of the algorithm, they can be implemented as a single generic function by imposing a common interface for all the objects being accepted by the function. This is illustrated in Figure 3. From the definition of an image given above, the interface of an image is composed of a domain, accessible by the `domain` function, and the values are accessed using the callable operator, taking into argument an element of the domain. This element is an n-D point for an n-D image, the index of a node for a node valued graph, etc... Thus, this generic function has only two constraints: the domain of the object should be the same and the type of the values of the images should be compatible for the maximum operation. That means the value type of the images may be different.

As explained in section 2.1, C++ and Rust generate code for each instantiated function. The function `generic_max` in Figure 3 takes three template parameters; each combination of types generates machine code specialized for the combination of parameters. Set to all kinds of possible types, it leads to a combinatorial explosion.

```
void image_max(image2d a, image2d b,
               image2d& out) {
  for (int y = 0; y < a.height(); y++)
    for (int x = 0; x < a.width(); x++)
      out(x, y) = max(a(x, y), b(x, y));
}

graph graph_max(graph a, graph b,
                graph& out) {
  for (int i = 0; i < a.num_nodes(); i++)
    out(i) = max(a(i), b(i));
}
```

Figure 2: Non-generic maximum functions

```
template <class I, class J, class O>
void generic_max(I a, J b, O& out) {
  for (auto p : a.domain())
    out(p) = max(a(p), b(p));
}
```

Figure 3: C++ implementation of generic maximum function

3 Dynamism for Static Genericity

As explained above, the genericity in statically typed languages has two major drawbacks:

- The generic parameters cannot be set at runtime.
- When the generic parameters are not known at compile time, several generic objects and algorithms have to be instantiated, resulting in a combinatorial explosion and the code bloat resulting from the monomorphization process.

To solve these issues in the image processing context, we propose to observe four models of image and apply them to generic algorithms with few modifications in their implementation. In this section and the following ones, we only focus on 2D images, but this method is extendible to any kind of image, as described in Section 2.2.

3.1 Image Models

In static languages such as C++ or Rust, a practical and basic way to implement a 2D image encoded as a buffer is to store the values in an array contained inside an object and to have access to the size of the image. Furthermore, to make the image reusable, a generic parameter handles the type of values in the buffer. Thus, the user has access to two elements: the domain of the image (width and height for a 2D image) and the value encoded at a given position. This is illustrated in C++ in Figure 4 and Rust in Figure 5 as the templated and generic version of a 2D buffer. However, as specified above, this kind of image is unusable in a dynamic context.

One way to remove this generic parameter relies on *type erasure*. This mechanism is widely used in the C++ standard library and used in several cases. For example, the `std::any` [3] object is a

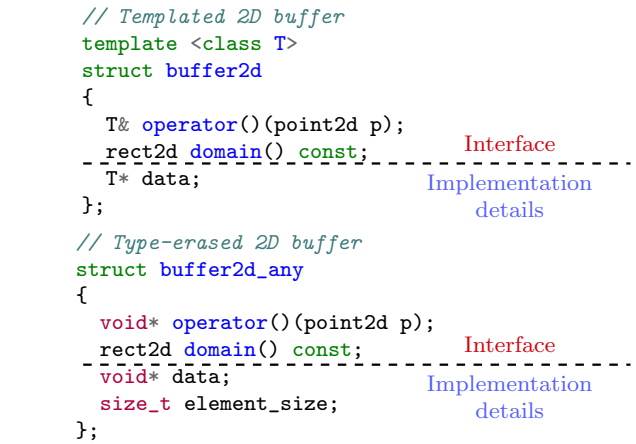


Figure 4: C++ implementation of a 2D buffer

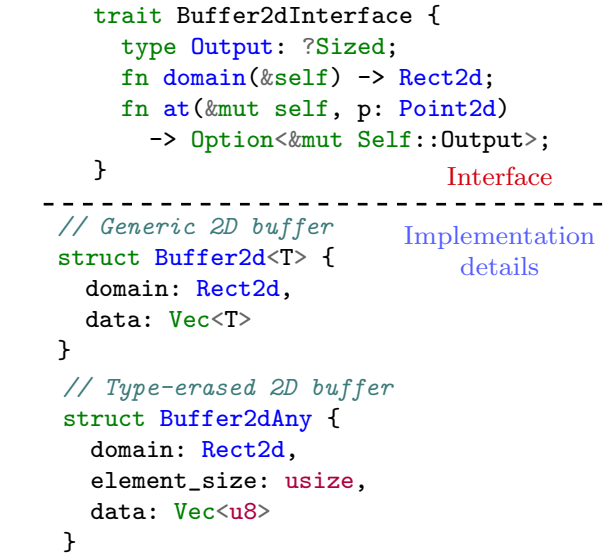


Figure 5: Rust implementation of a 2D buffer

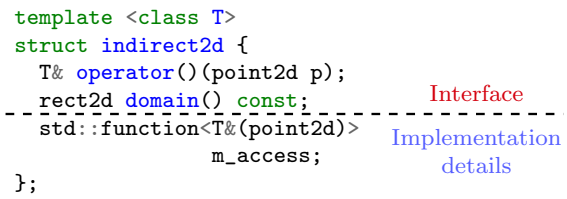


Figure 6: C++ implementation of indirection

type-safe object handling any kind of object under certain constraints. It allocates memory for this object and stores it without any static type information, but keeps a dynamic identifier of the type internally so that the conversion from the typed-erased object to the statically typed one is a type-safe operation by computing some type checking. Another use case of type erasure is the `std::function` object. It stores any kind of callable inside the object such as a function pointer or a functor storing some data since their call operation respects a given list of argument types and the return type.

Value type information This second model type-erases the values of the image. Some information about the type of values such as the size in bytes of one value must be stored in the implementation details in order to traverse the image value by value. However, the interface does not change: the type erasure does not change the nature of the definition domain and the access operator still returns information related to the value. In C++, this information is a pointer to the first byte of the value and in Rust, it is an unsized slice containing the bytes of the value. This slight difference between the two implementations is due to the fact that Rust syntax for pointers is different than its syntax for concrete values. These type-erased versions are illustrated in Figures 4 and 5 as `buffer2d_any` and `Buffer2dAny`.

Data encoding information The two previous models are based on the type of values, but not on their implementation. However, an image may have a different implementation according to the context, such as one value for a constant image, a sparse matrix for an image storing a few values different from 0, or a C++ view returning a modified image value at access. To handle all of these implementations, a model inspired by the C++ `std::function` functionality is used. Instead of encoding the implementation of the image directly in the structure handling the interface, it is handled by an external object, encoding the access of the value to the image, which is itself stored in the interface. This method makes an *indirection* to access the image values. This can be done by using a `std::function` in C++ or a dynamic trait object in Rust. These structures are denoted by `indirect2d<T>` for the statically typed values version and `indirect2d_any` for the dynamic one. The C++ implementation of `indirect2d<T>` is illustrated in Figure 6.

3.2 Application to Generic Algorithm

The image models described above have a common interface, which is a criterion for generic programming. However, the untyped models return values that cannot be directly manipulated by the algorithm, such as the addresses returned by the access operation in the C++ implementation of a 2D untyped buffer. Thus, these values should be converted into a statically typed one to be manipulated by the algorithm.

```
void qsort(void *tab, size_t nmemb, size_t size, int (*compare)(const void *, const void *))
```

Figure 7: `qsort` function prototype

```
// Generic operation
template <class I, class Op>
void elemwise_op(I a, I b, I& out, Op& op) {
  for (auto p : a.domain())
    op(a(p), b(p), out(p));
}
```

Figure 8: Elementwise operations function

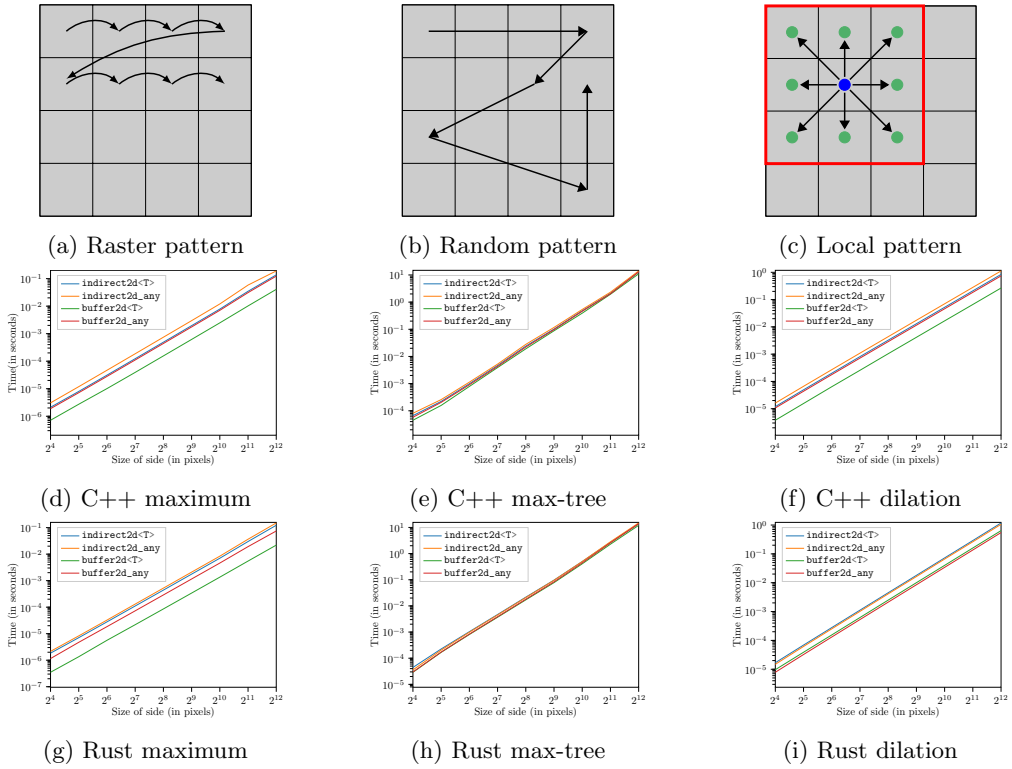
The model used to manipulate these algorithms is based on the `qsort` function from the C standard library whose prototype is recalled in Figure 7. This function sorts the elements of the table pointed by the address `tab`, composed of `nmemb` elements of `size` bytes. It sorts the elements according to the result returned by the function pointed by the `compare` pointer. This approach has several advantages: first, the `qsort` function does not require the knowledge of the static type of the object being sorted at compile time. Then, the sorting criterion can be changed at runtime without having to generate too much code. However, as pointed out by Meyers in [10], the `std::sort` function from the C++ Standard Template Library (STL) is 670% faster than the `qsort` function on a table containing one million of `double` typed elements. This is due to the fact that statically typed code is optimized at compile time and the cost of a function call is reduced by function inlining for the `std::sort` function.

This model is adapted to generic programming by adding operations to the algorithm manipulating the values of an object in the same way as the `qsort` function. The operator interface used in Figure 3 is changed to support type erasure. It changes from `(T, T) -> T` to `(T, T, T&) -> void` to store the result of the computation in the last function argument. It yields the generic `elementwise_op` depicted in Figure 8, that traverses `a` and `b` and stores the result of the operation `op` in the `out` image.

Table 2 summarizes the different image structures presented and their properties. The **Operation type** column shows two kinds of `max` operation, one where the operand types are known at compile-time, and a "dynamic" one whose arguments are type-erased, unknown at compite-time. In the context of the `elementwise_op` function, these two operations are valid as they respect the required interface. The statically-typed operation can be inlined because both the operator and the operand type are known at compile-time. However, the second one requires an indirection because the operand type is not known.

4 Results

In order to measure the cost of our models, we implemented in C++ and Rust three image processing algorithms following different algorithmic schemes, illustrated in Figures 9a to 9c. The *raster* pattern is implemented using an elementwise maximum operation between two images. The *random* pattern is used in the construction of a max-tree [13] using Berger's algorithm [2]. Finally, the *local* pattern is implemented by a dilation [14] using a square of size 1 as a structuring element. Each operation has been run on square images of side $s = \{2^n | \ n \in [\![4-12]\!]\}$. The experiments have been performed on a Linux Debian 11 machine equipped with a processor Intel i7-3770, 3.40GHz. The C++ benchmarks have been run using the Google Benchmark library from binary compiled with GCC 10.2.1 using the optimization



(a) Raster pattern (b) Random pattern (c) Local pattern

(d) C++ maximum (e) C++ max-tree (f) C++ dilation

(g) Rust maximum (h) Rust max-tree (i) Rust dilation

Figure 9: Benchmarks results of the implementation of the three algorithmic pattern in C++ and Rust

flags *-03, -ftree-vectorize, -mavx* and *-unroll-loops*. The Rust benchmarks have been compiled with the Rustc compiler using the third optimization level (*-C opt-level=3*) and the measurements have been performed using the Criterion.rs library.

The results of the benchmark are displayed in Figure 9. Each plot displays the performance of an algorithm in seconds related to the size of the side of an image. The second row is the result of the algorithms implemented in C++ and the third one of the algorithms implemented in Rust. Except for the Rust implementation of the dilation (Figure 9i), the statically typed buffer is the fastest implementation of the algorithm, which is particularly true when traversing the image in raster order as for the elementwise operation (Figures 9d and 9g). This is due to the number of cache misses, low for the elementwise operation, due to the fact the images are traversed in the same order they are stored in memory. Furthermore, the knowledge of the type at compile time enables optimizations by the compiler such as automatic vectorization of the instruction in the produced binary. Finally, the implementations of the algorithms with the `buffer2d<T>` knowing the nature of the input object values type and the input object implementation details, the operations given to the algorithm are processed by the compiler, enabling its inlining and avoiding the indirection induced by a function call.

Furthermore, we observe in Figures 9e and 9h that the algorithmic scheme is an important criterion to choose the generic model to use. For the max-tree algorithm, whatever the model used, the performance of its computation is similar for each one. Indeed, the *random* algorithmic scheme does not access the memory in the same order as the memory is used. Thus, it results in several cache

Table 1: Execution time overhead (in percentage) of the algorithmic schemes compared to the statically-typed with direct access image of side size of 4096

	Statically Typed	Yes		No	
	Direct Access	Yes	No	Yes	No
C++	Raster	+0%	+176%	+183%	+367%
	Random	+0%	+20%	+18%	+29%
	Local	+0%	+208%	+174%	+283%
Rust	Raster	+0%	+388%	+251%	+574%
	Random	+0%	+9%	+6%	+15%
	Local	+0%	+61%	-14%	+66%

misses, but also the compiler is unable to optimize the generated machine code.

We can conclude from these benchmarks that the static information of the image values type is important, but also the algorithmic scheme. This is even more obvious in Table 1, where the dynamism overhead is shown. In the context of a bridge between the C++ or Rust language and Python, specializing generic algorithms to a wide variety of types in the case of a pattern such as the random pattern is not necessary in term of performance.

The second experiment performed in this paper is the measurement of the size evolution of the generated machine code from the C++ implementation of the max-tree algorithm related to the number of handled image value types. To make it, we used the *Bloaty*[1] profiler which measures the size of different elements in binaries. The max-tree algorithm has been chosen because its compilation generates the largest amount of machine code from the three previous algorithms, but also because the cost of dynamism of its algorithmic scheme is negligible and permits the usage of its dynamic version. The result of this measurement is shown Table 3. Both version exhibit a linear increase with the addition of new image types. However, the quantity of new code generated in the dynamic version (100b/type) is 26 times lower than in the static version (2.6Kb/type) where a new full algorithm is instantiated. Therefore, the dynamic version prevents code bloat.

5 Conclusion

In this paper, we presented different models of generic programming and we compared them applied to image processing by implementing different algorithmic schemes in C++ and Rust. We showed that the performance of each model was dependent on the algorithmic scheme used, and we highlight the fact that some information such as the type of the values of an image, was more important to be known statically by the compiler in some cases. To reduce the loss of performance induced by the lack of static information knowledge, some leads may be explored: first, the usage of external modules downloaded at runtime and linked to the application, with precompiled algorithms, optimized for this particular use case. The second lead may be the usage of Just-In-Time compilation to generate optimized assembly code such as SIMD instruction at runtime for critical operations. Then, as observed in the benchmark's result Figure 9, for a small square image, the difference in performance is negligible. Looking for the best side size related to the performance of an algorithm for distributed tile-based image processing algorithm would be a means to reduce this gap in performance. Finally, this work is intended to be

[1] `https://github.com/google/bloaty`

Table 2: Summary of the different image structures and their properties

Image type	**Access policy**	**Static value type**	$f(p)$ **return type**	**Operation type**
`buffer2d<T>`	Direct to the buffer	✓	`T&`	`void max<T>(T a, T b, T& out)`
`indirect2d<T>`	Indirect	✓	`T&`	`void max<T>(T a, T b, T& out)`
`buffer2d_any`	Direct to the buffer	✗	`void*`	`void (*max)(const void* a, const void* b, void* out)`
`indirect2d_any`	Indirect	✗	`void*`	`void (*max)(const void* a, const void* b, void* out)`

Table 3: Max-tree generated machine code related to the number of handled types

Number of handled types	1	2	3	4	5	6	7	8	9	10	11
Size for static version (in Kb)	2.9	5.3	7.9	10.5	13.2	15.8	18.5	21.1	23.8	26.5	29.2
Size for dynamic version (in Kb)	3.7	3.8	3.9	4.0	4.1	4.2	4.4	4.5	4.6	4.7	4.9

used in the context of a bridge from a static language to a dynamic one to provide an interface for dynamic environments without a loss of performance for a C++ image processing library. Thus, we will use it as a basis for efficient bindings of our algorithms from C++ to Python.

Acknoledgements

The authors would like to acknowledge Antoine Martin for his useful advice about the Rust programming language.

References

[1] Abrahams, D., and Grosse-Kunstleve, R. W. Building hybrid systems with boost.python. *C/C++ Users Journal 21*, LBNL-53142 (2003).

[2] Berger, C., Géraud, T., Levillain, R., Widynski, N., Baillard, A., and Bertin, E. Effective component tree computation with application to pattern recognition in astronomical imaging. In *2007 IEEE International Conference on Image Processing* (2007), vol. 4, IEEE, pp. IV–41.

[3] Dawes, B., and Meredith, A. P0220r1: Adopt library fundamentals v1 ts components for c++17 (r1), 2016.

[4] Glück, R., Nakashige, R., and Zöchling, R. Binding-time analysis applied to mathematical algorithms. In *System Modelling and Optimization*. Springer, 1996, pp. 137–146.

[5] Harris, C. R., Millman, K. J., Van Der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., et al. Array programming with numpy. *Nature 585*, 7825 (2020), 357–362.

[6] Hunter, J. D. Matplotlib: A 2d graphics environment. *Computing in science & engineering 9*, 03 (2007), 90–95.

[7] Jakob, W., Rhinelander, J., and Moldovan, D. pybind11 – seamless operability between c++11 and python, 2017. https://github.com/pybind/pybind11.

[8] Köthe, U. Reusable software in computer vision. *Handbook of computer vision and applications 3* (1999), 103–132.

[9] Levillain, R., Géraud, T., and Najman, L. Milena: Write generic morphological algorithms once, run on many kinds of images. In *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing* (2009), Springer, pp. 295–306.

[10] Meyers, S. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison Wesley, 2001.

[11] Musser, D. R., and Stepanov, A. A. Generic programming. In *International Symposium on Symbolic and Algebraic Computation* (1988), Springer, pp. 13–25.

[12] Perret, B., Chierchia, G., Cousty, J., Guimaraes, S. J. F., Kenmochi, Y., and Najman, L. Higra: Hierarchical graph analysis. *SoftwareX 10* (2019), 100335.

[13] Salembier, P., Oliveras, A., and Garrido, L. Antiextensive connected operators for image and sequence processing. *IEEE Transactions on Image Processing 7*, 4 (1998), 555–570.

[14] Serra, J. *Image Analysis and Mathematical Morphology.* Academic press, 1982.