The Cost of Dynamism in Static Languages for Image Processing

Baptiste Esteban
baptiste.esteban@lrde.epita.fr
EPITA Research Laboratory
Le Kremlin-Bicêtre, FRANCE

Edwin Carlinet
edwin.carlinet@lrde.epita.fr
EPITA Research Laboratory
Le Kremlin-Bicêtre, FRANCE

Guillaume Tochon
guillaume.tochon@lrde.epita.fr
EPITA Research Laboratory
Le Kremlin-Bicêtre, FRANCE

Didier Verna
didier.verna@lrde.epita.fr
EPITA Research Laboratory
Le Kremlin-Bicêtre, FRANCE

Abstract
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.

CCS Concepts: • Software and its engineering → Development frameworks and environments; • Computing methodologies → Image processing.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

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 operator overloading in C++ or the implementation of the trait bound Add in Rust.

// C++ version

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

// Rust version

```rust
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 $f : \Omega \rightarrow V$ be an image with $\Omega$ its domain and $V$ 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

```cpp
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));
}
```

```cpp
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

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, ac-
cessible 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 combi-
nation of parameters. Set to all kinds of possible types, it
leads to a combinatorial explosion.

3 Dynamism for static genericity

As explained above, the genericity in statically typed lan-
guages 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 explosi-
on and the code bloat resulting from the monomor-
phization 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 imple-
mentation. 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

trait Buffer2dInterface {
    type Output: ?Sized;
    fn domain(&self) -> Rect2d;
    fn at(&mut self, p: Point2d)
        -> Option<&mut Self::Output>;
}

// Generic 2D buffer
struct Buffer2d<T> { 
    domain: Rect2d, 
    data: Vec<T>
}

// Type-erased 2D buffer
struct Buffer2dAny { 
    domain: Rect2d, 
    element_size: usize, 
    data: Vec<u8>
}

Figure 5. Rust implementation of a 2D buffer

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++ stan-
dard library and used in several cases. For example, the
std::any object is a type-safe object handling any kind
of object under certain constraints. It allocates memory for
template <class T>
struct indirect2d {
    T& operator()(point2d p);
    rect2d domain() const;
    std::function<T&(point2d)> m_access;
};

Figure 6. C++ implementation of indirection

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.

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 1 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...
The Cost of Dynamism in Static Languages for Image Processing

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.

Arguments are type-erased, unknown at compile-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.

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 \( s = \{2^n \mid n \in [4 - 12]\} \). 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 flags `-O3`, `-ftree-vectorize`, `-mavx` and `-unroll-loops`. The Rust
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\textsuperscript{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 size relative 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 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.

Acknowledgments

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

\textsuperscript{1}https://github.com/google/bloaty
<table>
<thead>
<tr>
<th>Number of handled types</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size for static version (in Kb)</td>
<td>2.9</td>
<td>5.3</td>
<td>7.9</td>
<td>10.5</td>
<td>13.2</td>
<td>15.8</td>
<td>18.5</td>
<td>21.1</td>
<td>23.8</td>
<td>26.5</td>
<td>29.2</td>
</tr>
<tr>
<td>Size for dynamic version (in Kb)</td>
<td>3.7</td>
<td>3.8</td>
<td>3.9</td>
<td>4.0</td>
<td>4.1</td>
<td>4.2</td>
<td>4.4</td>
<td>4.5</td>
<td>4.6</td>
<td>4.7</td>
<td>4.9</td>
</tr>
</tbody>
</table>

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

**References**


Received 2022-08-12; accepted 2022-10-10