Practical Genericity: Writing Image Processing Algorithms Both Reusable and Efficient

Roland Levillain[1], Thierry Géraud[1], Laurent Najman[2], Edwin Carlinet[1,2]

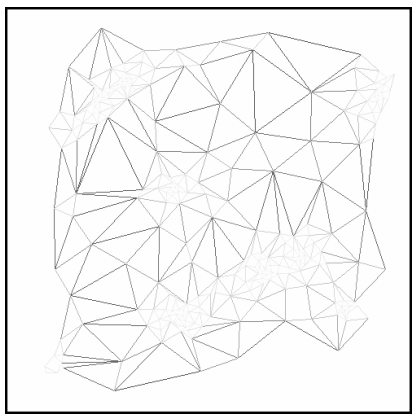
[1]EPITA Research and Development Laboratory (LRDE)
[2]Laboratoire d'Informatique Gaspard-Monge (LIGM)
first.lastname@lrde.epita.fr *first.lastname*@esiee.fr

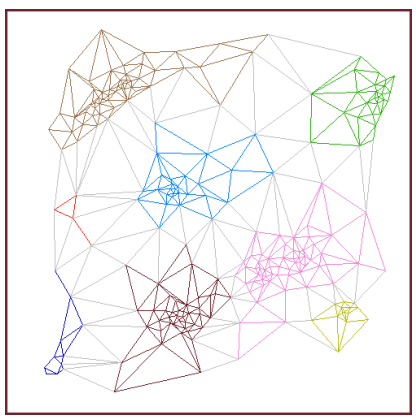
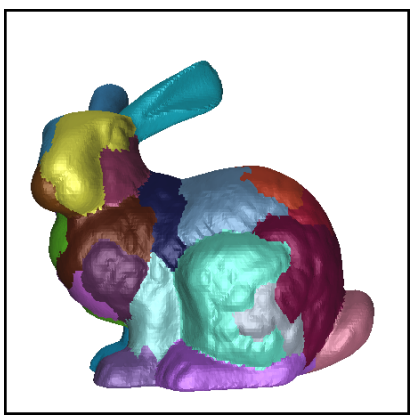
November 2014, 3rd

Objective

Be able to process *easily* and *efficiently* many kind of images.

A generic watershed transform

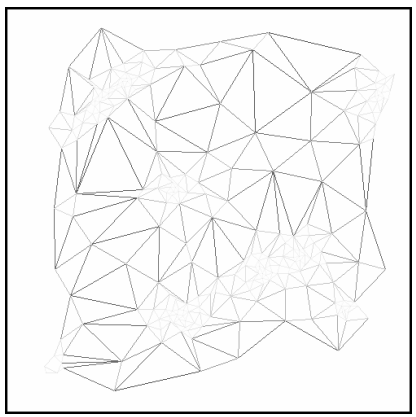




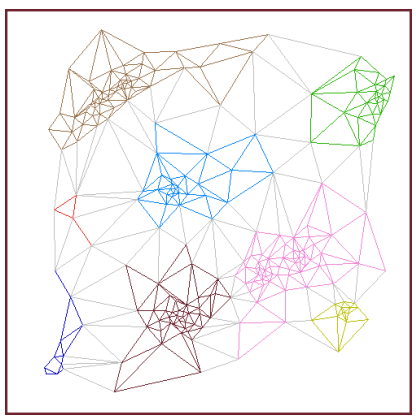
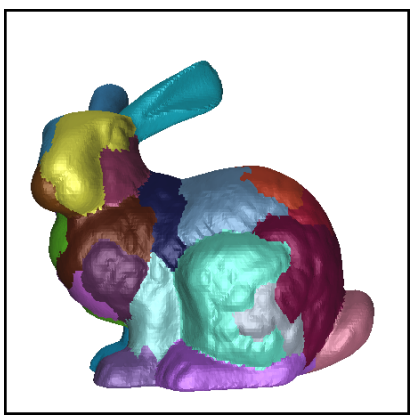
On a regular grid | On an edge-valued graph | On a 3D surface mesh

Objective

Be able to process *easily* and *efficiently* many kind of images.

A generic watershed transform

On a regular grid | On an edge-valued graph | On a 3D surface mesh

A single algorithm processes these "images" !

What about image processing algorithms?

Case study. Dilation by a structuring element (SE).

What about image processing algorithms?

Case study. Dilation by a structuring element (SE).

2D dilation of float images with a square SE

```
image
dilation(image f, int r)
  image out(f.nrows(), f.ncols());
  for i = 0 to f.nrows(); do
    for j = 0 to f.ncols(); do
      float sup = FLT_MIN;
      for k = -r to r; do
        for l = -r to r; do
          if sup < f[i+k, j+l]
            sup = f[i+k, j+l]
      out[i,j] = sup;
  return out;
```

What about image processing algorithms?

Case study. Dilation by a structuring element (SE).

2D dilation of float images with a square SE

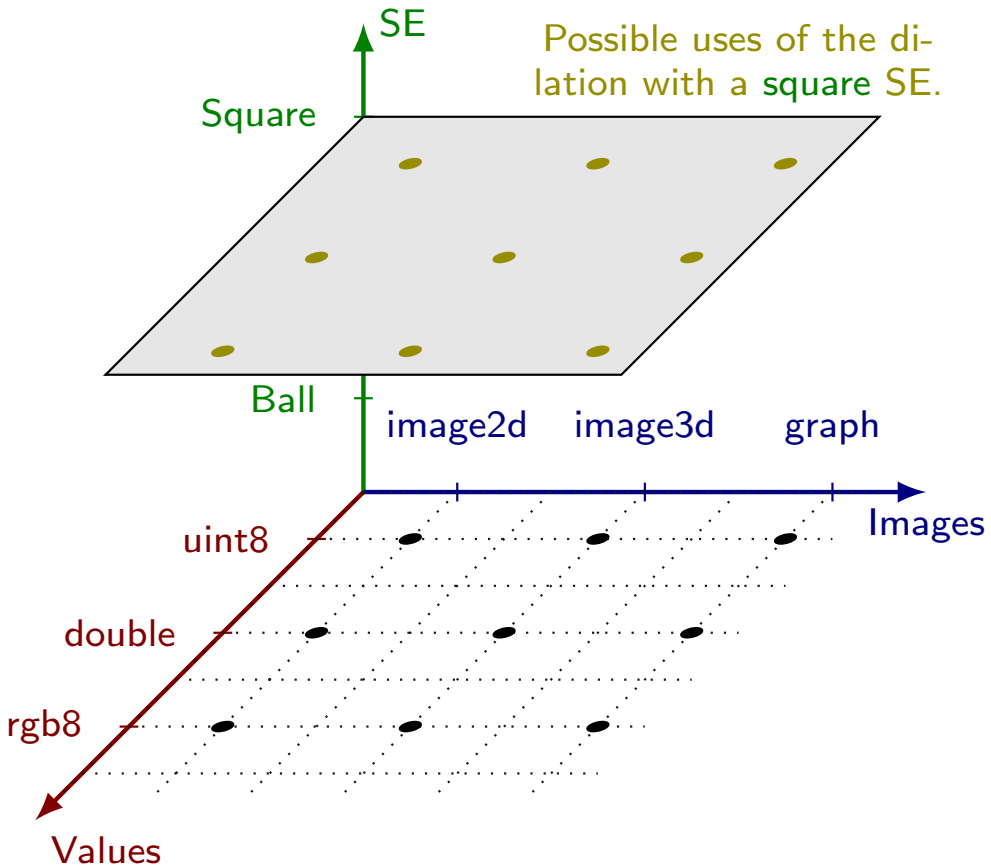
```
image
dilation(image f, int r)
  image out(f.nrows(), f.ncols());
  for i = 0 to f.nrows(); do
    for j = 0 to f.ncols(); do
      float sup = FLT_MIN;
      for k = -r to r; do
        for l = -r to r; do
          if sup < f[i+k, j+l]
            sup = f[i+k, j+l]
      out[i,j] = sup;
  return out;
```

It works but...

What about image processing algorithms?

Problem. It works but...

- what if the image is in color? (genericity in the *value space*)
- what if the image is 3D? (genericity in the *domain space*)
- what if the image is a graph? (*structural* genericity)
- what if the structuring element is a ball?



We want genericity to cover the space of possibilities!

Paradigms	Code Complexity	Efficiency	Structural Genericity	1 alg. = 1 impl.
Code duplication	✗	✓	✗	✗

Code duplication. Copy & paste and adapt the code
→ redundancy and maintainability issues...

1D dilation for 8-bits unsigned

```
image
dilation(image f, int r)
  image out(f.size());
  for i = 0 to f.size(); do
    unsigned char sup = 0;
    for k = -r to r; do
      sup = max(f[i+k], sup);
    out[i] = sup;
  return out;
```

2D dilation for float

```
image
dilation(image f, int r)
  image out(f.nrows(), f.ncols());
  for i = 0 to f.nrows(); do
    for j = 0 to f.ncols(); do
      float sup = FLT_MIN;
      for k = -r to r; do
        for l = -r to r; do
          if sup < f[i+k, j+l]
            sup = f[i+k, j+l]
      out[i,j] = sup;
  return out;
```

Paradigms	Code Complexity	Efficiency	Structural Genericity	1 alg. = 1 impl.
Code duplication	✗	✓	✗	✗

Code duplication. Copy & paste and adapt the code
→ redundancy and maintainability issues...

1D dilation for 8-bits unsigned

```
image
dilation(image f, int r)
  image out(f.size());
  for i = 0 to f.size(); do
    unsigned char sup = 0;
    for k = -r to r; do
      sup = max(f[i+k], sup);
    out[i] = sup;
  return out;
```

Bad !

2D dilation for float

```
image
dilation(image f, int r)
  image out(f.nrows(), f.ncols());
  for i = 0 to f.nrows(); do
    for j = 0 to f.ncols(); do
      float sup = FLT_MIN;
      for k = -r to r; do
        for l = -r to r; do
          if sup < f[i+k, j+l]
            sup = f[i+k, j+l]
      out[i,j] = sup;
  return out;
```

Paradigms	Code Complexity	Efficiency	Structural Genericity	1 alg. $=$ 1 impl.
Code duplication	✗	✓	✗	✗
Generalization	✓	✗	✗	✓

Generalization. e.g. consider 3D image of double for every images (the wider type).
$\rightarrow$ efficiency issues and still not *structurally* generic

Paradigms	Code Complexity	Efficiency	Structural Genericity	1 alg. = 1 impl.
Code duplication	✗	✓	✗	✗
Generalization	✓	✗	✗	✓
Object-Oriented Programming	✓	✗	✓	✓

Generalization. e.g. consider 3D image of double for every images (the wider type).
→ efficiency issues and still not *structurally* generic

Object-Oriented Programming. Generalization through type hierarchies.
→ efficiency issues (virtual methods)

Paradigms	Code Complexity	Efficiency	Structural Genericity	1 alg. = 1 impl.
Code duplication	✗	✓	✗	✗
Generalization	✓	✗	✗	✓
Object-Oriented Programming	✓	✗	✓	✓
Generic Programming	✓	✓	✓	✓

Generic programming is the way to go...

Paradigms	Code Complexity	Efficiency	Structural Genericity	1 alg. = 1 impl.
Code duplication	✗	✓	✗	✗
Generalization	✓	✗	✗	✓
Object-Oriented Programming	✓	✗	✓	✓
Generic Programming	✓	✓	✓	✓

Generic programming is the way to go...

Because the algorithm is *intrasically generic* and **so should be the code**

V is the image value type

```
dilation(Image f, SE win)
    initialize out from f
    foreach Site p in f's domain do
        out(p) ← inf(V)
        foreach Site n in win(p) do
            out(p) ← sup(out(p), f(n))
    return out
```

Paradigms	Code Complexity	Efficiency	Structural Genericity	1 alg. = 1 impl.
Code duplication	✗	✓	✗	✗
Generalization	✓	✗	✗	✓
Object-Oriented Programming	✓	✗	✓	✓
Generic Programming	✓	✓	✓	✓

Generic programming is the way to go...

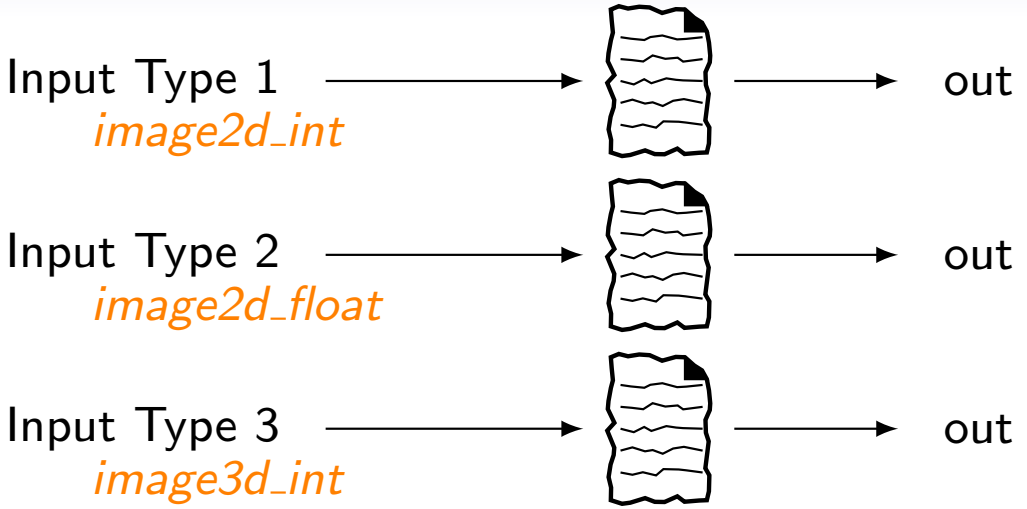
Because the algorithm is *intrasically generic* and **so should be the code**

V is the image value type

```
dilation(Image f, SE win)
    initialize out from f
    foreach Site p in f's domain do
        out(p) ← inf(V)
        foreach Site n in win(p) do
            out(p) ← sup(out(p), f(n))
    return out
```

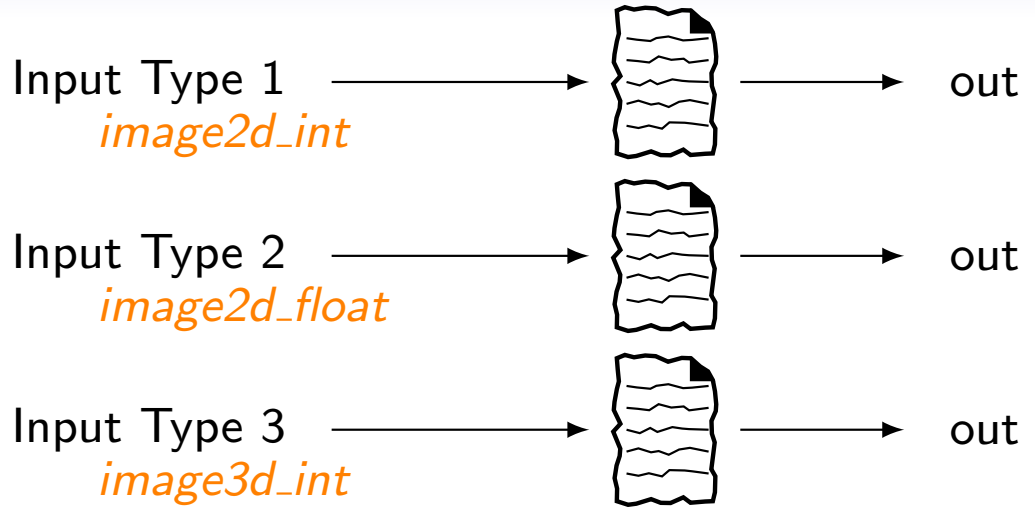
Real implementation should look like this! (see full code)

Specific algorithms.



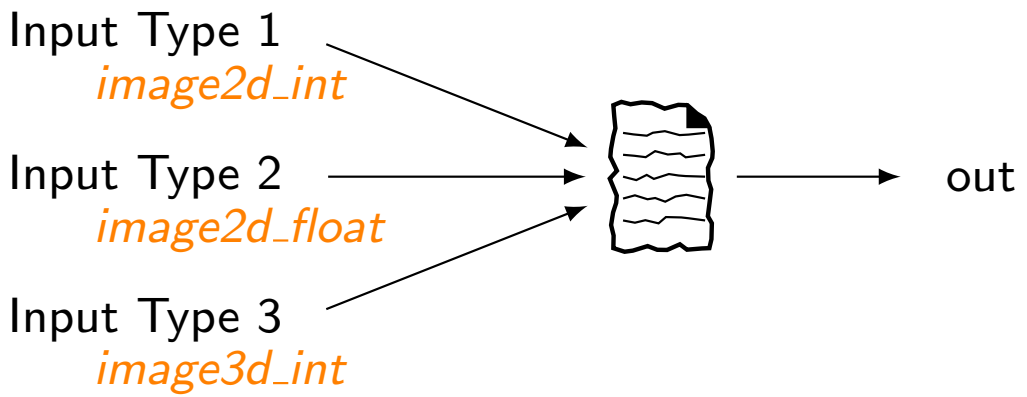
1 Input Type
= 1 Implementation

Specific algorithms.



1 Input Type
= 1 Implementation

Generic algorithm.

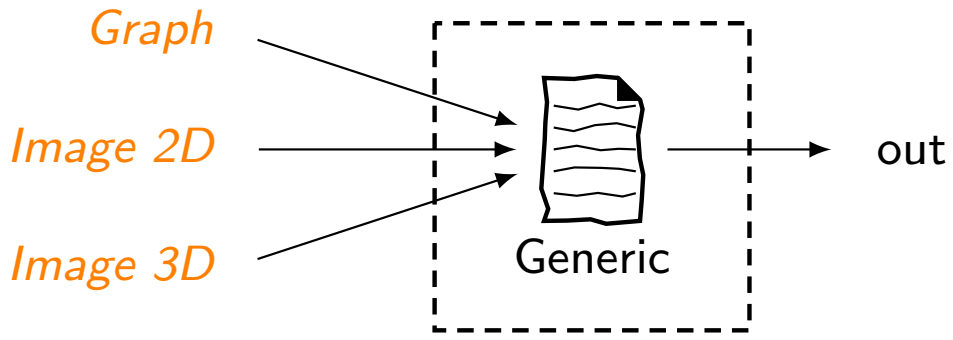


1 Algorithm
= 1 Implementation
= Many Input Types

Outline

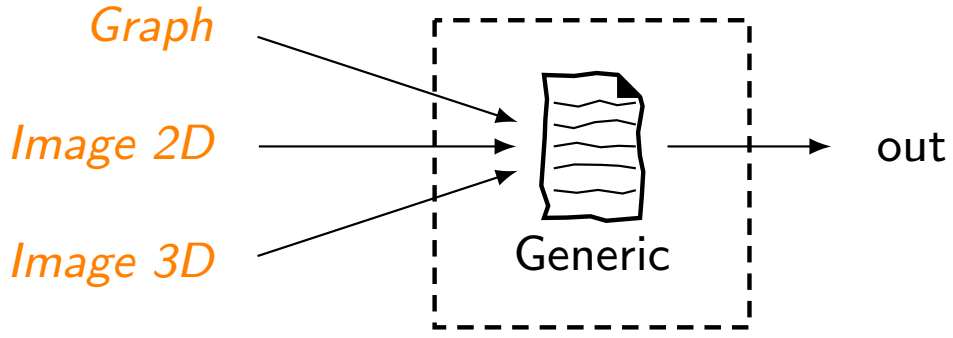
Genericity vs Efficiency Trade-Off

Before.



Genericity vs Efficiency Trade-Off

Before.

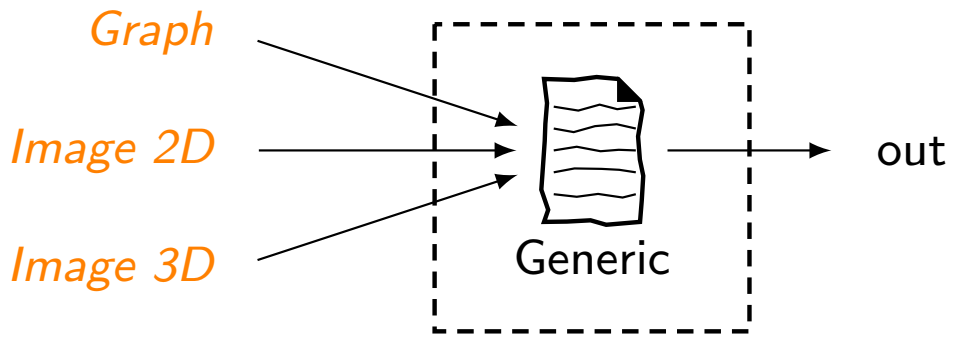


OK but
no so efficient

slower than impl. with pointers

Genericity vs Efficiency Trade-Off

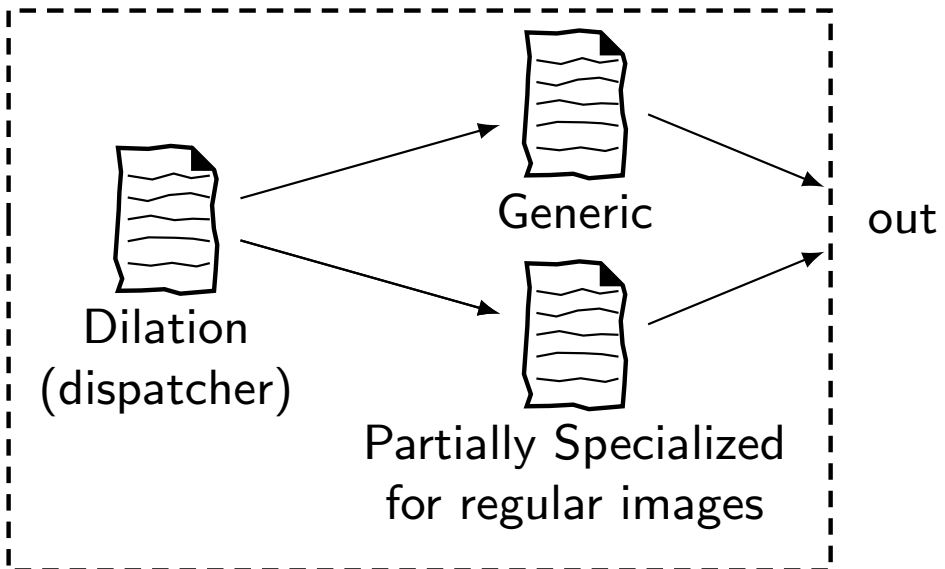
Before.



OK but
no so efficient

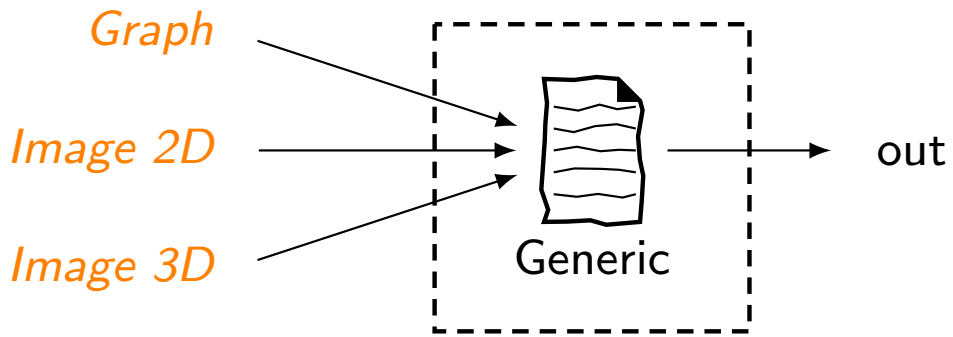
slower than impl. with pointers

After.



Genericity vs Efficiency Trade-Off

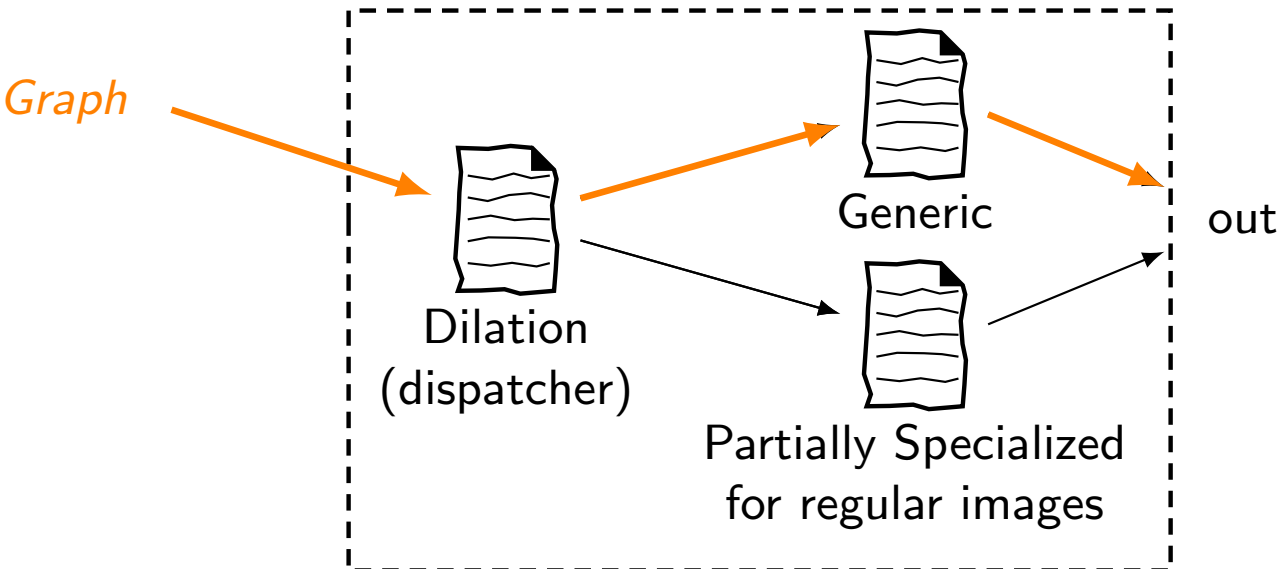
Before.



OK but
no so efficient

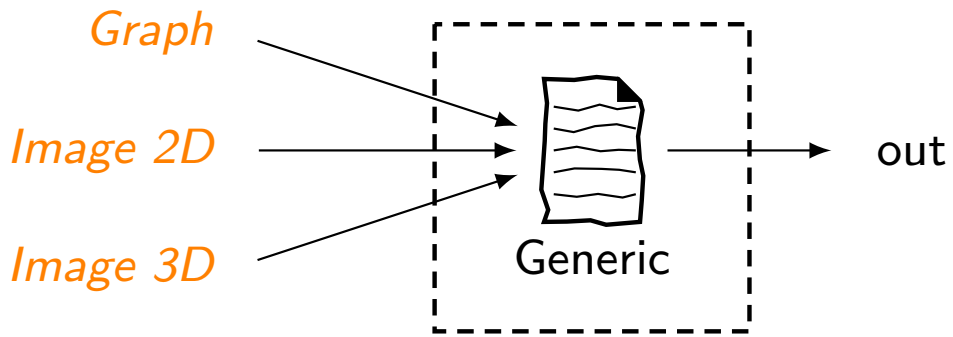
slower than impl. with pointers

After.



Genericity vs Efficiency Trade-Off

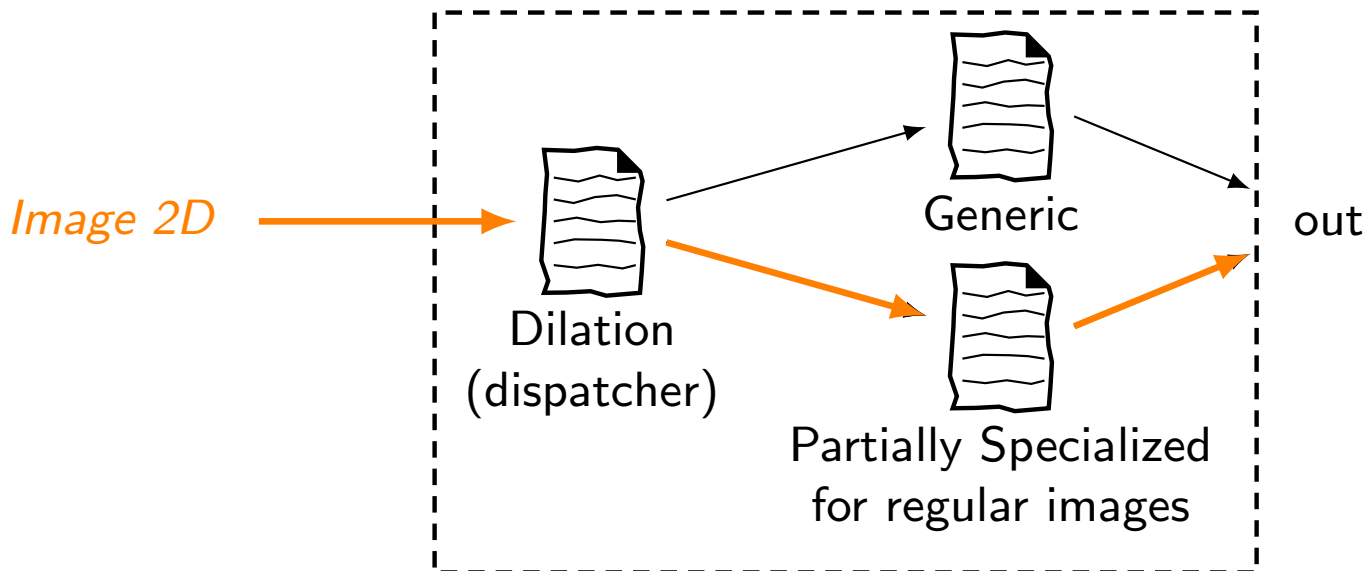
Before.



OK but
no so efficient

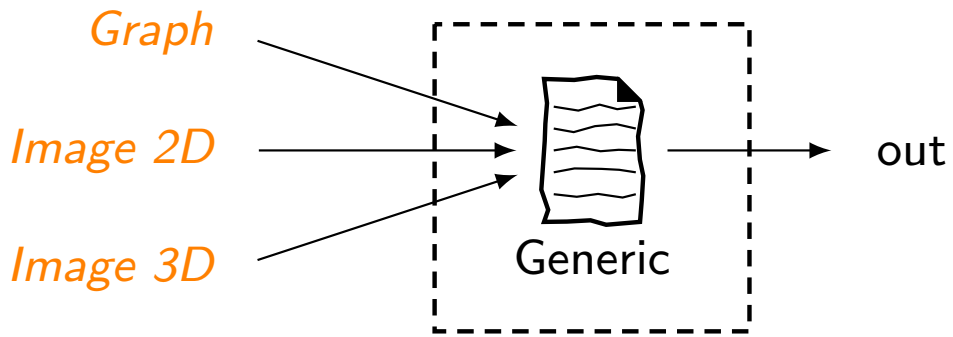
slower than impl. with pointers

After.



Genericity vs Efficiency Trade-Off

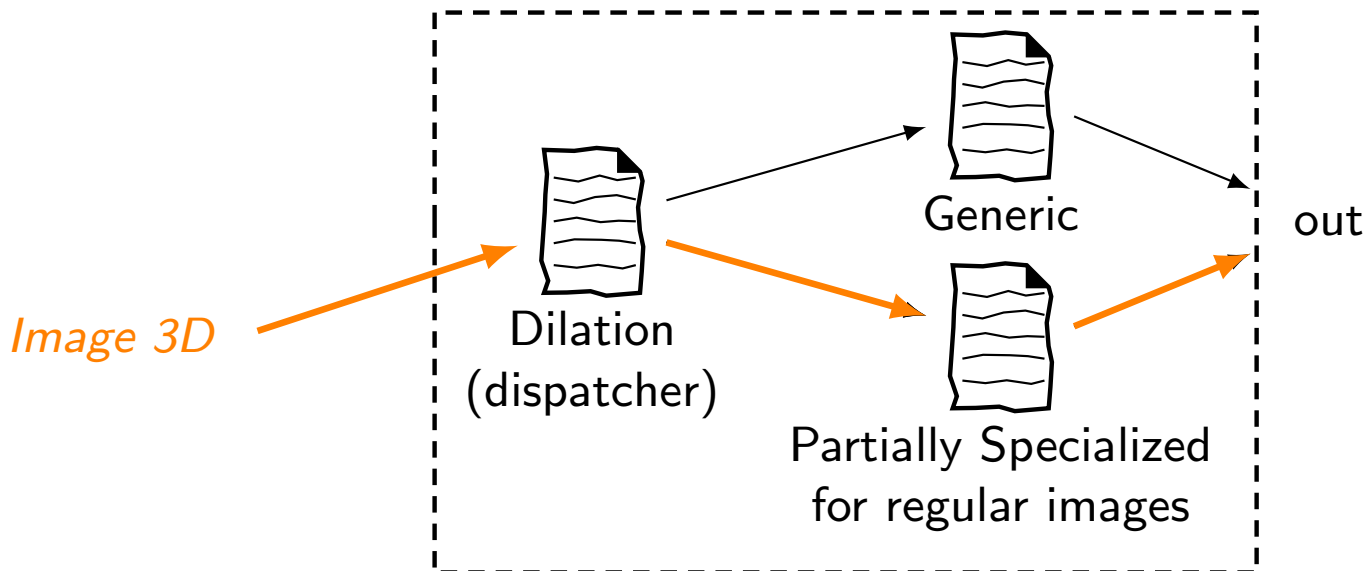
Before.



OK but
no so efficient

slower than impl. with pointers

After.



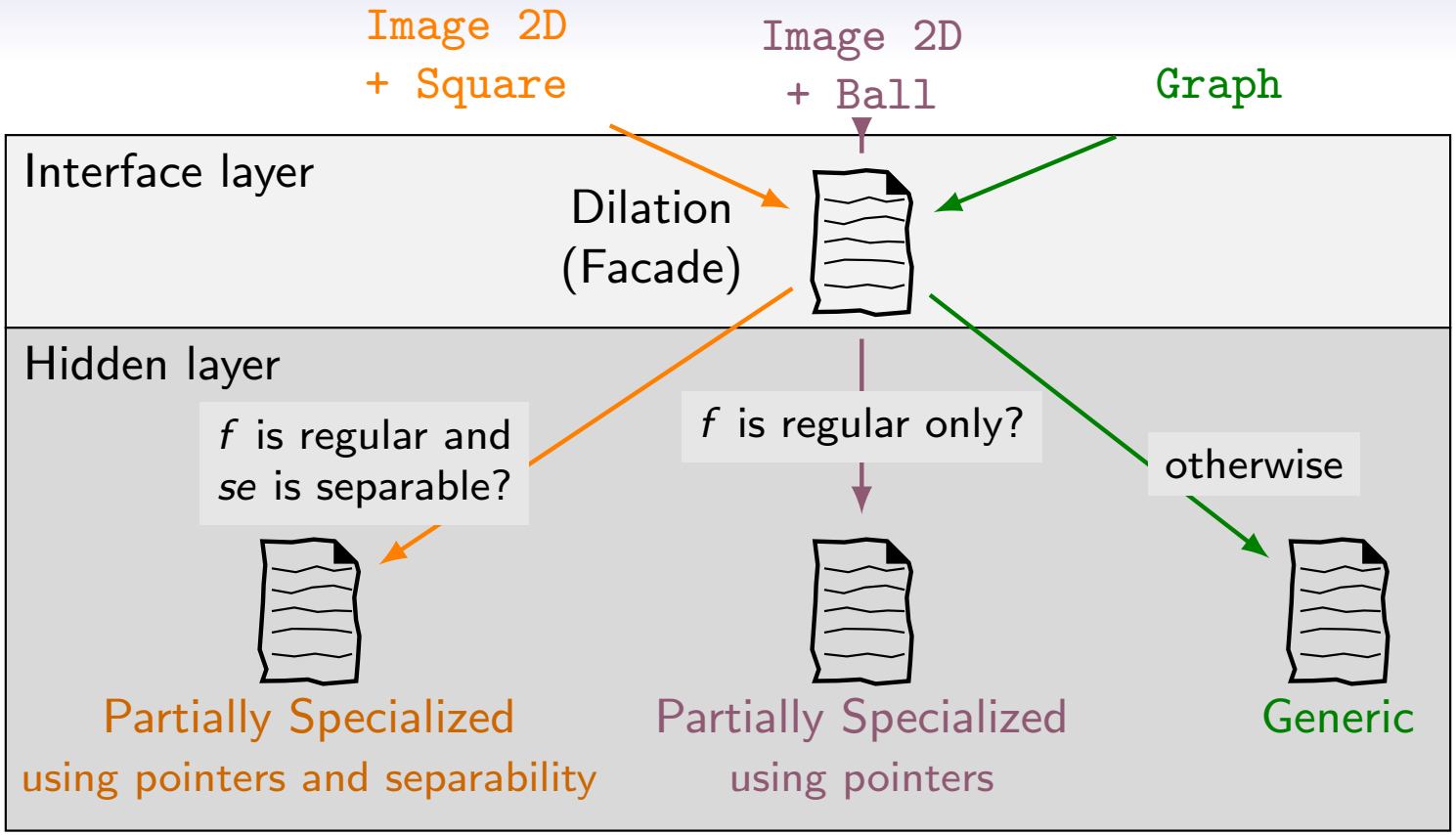
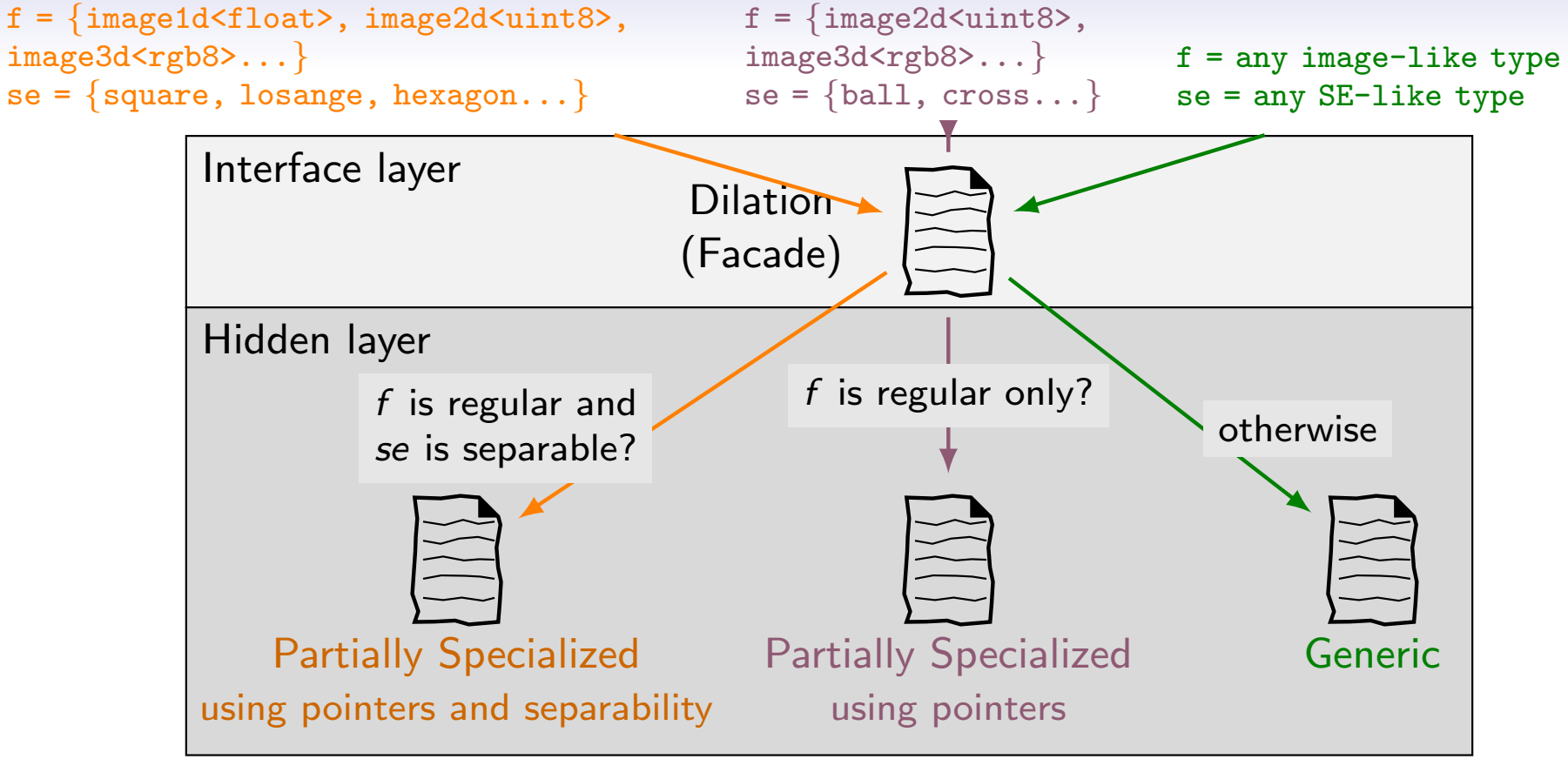


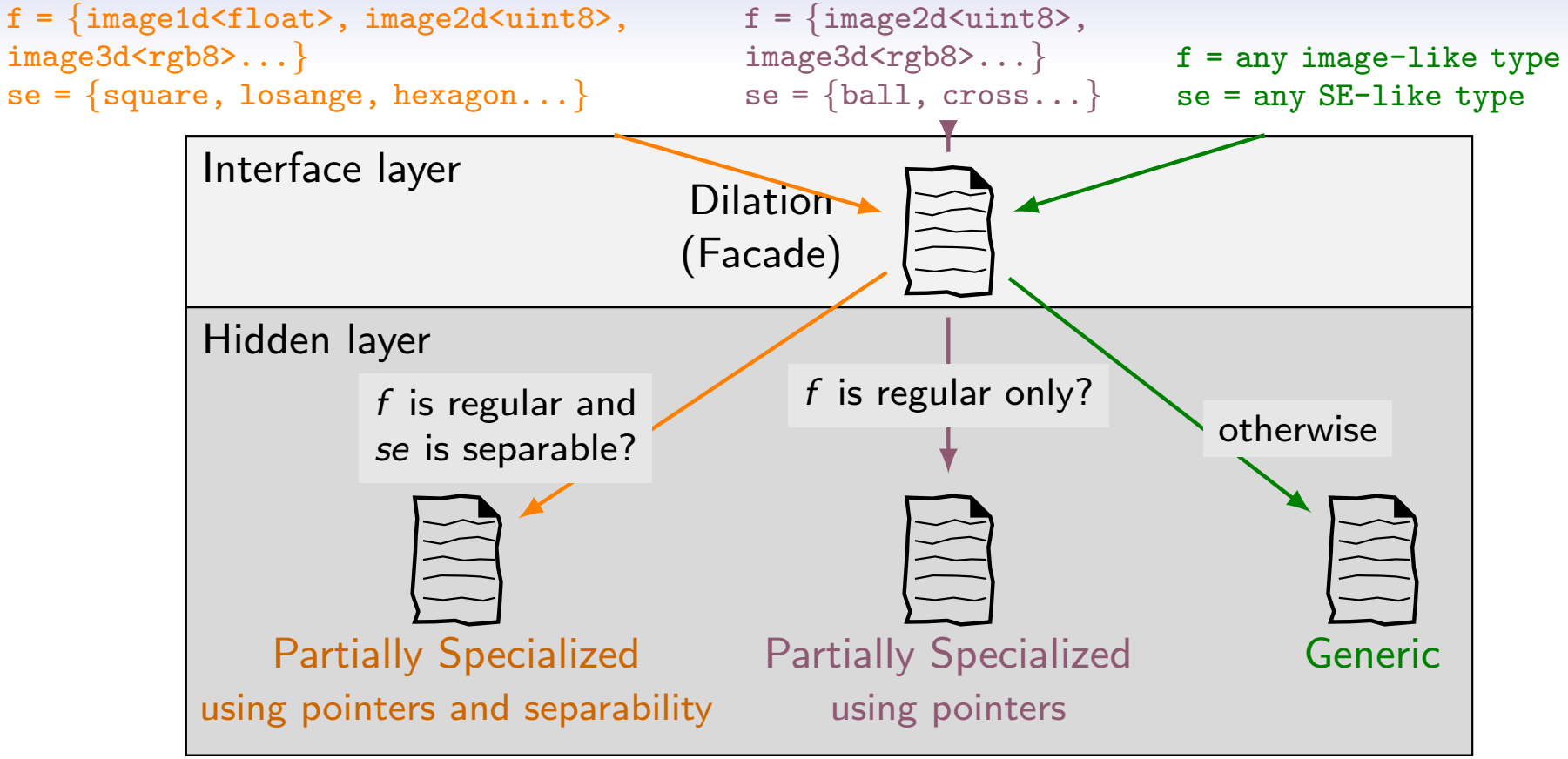
Image 2D
+ Square
Image 2D
+ Ball
Graph
Interface layer
Dilation
(Facade)
Hidden layer
f is regular and
se is separable?
f is regular only?
otherwise
Partially Specialized
using pointers and separability
Partially Specialized
using pointers
Generic



Remark 1.

Partially Specialized = Partially Generic = More efficient
≠ Specific

→ *Yet* 1 implementation = Many input types



Remark 2.

The interface *does not change*.

$\rightarrow$ 1 specialization = any dilation-based code gets optimized *for free* !

Conclusion

Why Generic Programming?

- No code duplication
- One implementation to handle any kind of images
- Somewhat efficient

Conclusion

Why Generic Programming?

- No code duplication
- One implementation to handle any kind of images
- Somewhat efficient

Reconciliation of genericity and efficiency.

- Complexity hidden and transparent from the user POV
- Partial specialization: loosing some genericity for efficiency...
- ...but we are *still* generic!

Conclusion

Why Generic Programming?

- No code duplication
- One implementation to handle any kind of images
- Somewhat efficient

Reconciliation of genericity and efficiency.

- Complexity hidden and transparent from the user POV
- Partial specialization: loosing some genericity for efficiency...
- ...but we are *still* generic!

$\rightarrow$ generic implementations can run as fast as hand-written specific implementations

Conclusion

Why Generic Programming?

- No code duplication
- One implementation to handle any kind of images
- Somewhat efficient

Reconciliation of genericity and efficiency.

- Complexity hidden and transparent from the user POV
- Partial specialization: loosing some genericity for efficiency...
- ...but we are *still* generic!

$\rightarrow$ generic implementations can run as fast as hand-written specific implementations

Implemented in the Milena IP library of the Olena project
`http://olena.lrde.epita.fr`

Thank you for your attention

Bibliography


Milena: Write generic morphological algorithms once, run on many kinds of images. Levillain, R., *Levillain, R., Géraud, T., Najman, L.* In: Proceedings of the ISMM. Lecture Notes in Computer Science, vol. 5720, pp. 295–306. Springer Berlin / Heidelberg, Groningen, The Netherlands (August 2009)


Writing reusable digital topology algorithms in a generic image processing framework. *Levillain, R., Géraud, T., Najman, L.* In: Proc. of WADGMM. Lecture Notes in Computer Science, vol. 7346, pp. 140–153. Springer-Verlag (2012)

Full C++ dilation code with Milena

```
template <class I, class W>
I dilation(I input, W win)
  I output;
  initialize(output, input);
  mln_piter(I) p(input.domain());
  mln_qiter(W) q(win, p);
  for_all(p)
    accu::supremum<mln_value(I)> sup;
    for_all(q) if (input.has(q))
      sup.take(input(q));
    output(p) = sup.to_result();
  return output;
```

Full C++ pointer-based dilation with Milena

```
template <class I, class W>
I dilation(I input, W win) {
  I output;
  initialize(output, input);
  mln_pixter(I) pi(input);
  mln_pixter(I) po(output);
  mln_qixter(I, W) q(pi, win);
  for_all_2(pi, po) {
    accu::supremum<mln_value(I)> sup;
    for_all(q)
      sup.take(q.val());
    po.val() = sup.to_result();
  }
  return output;
```