Context-Oriented Image Processing

Reconciling Genericity and Performance through Contexts

Didier Verna

didier@lrde.epita.fr
http://www.lrde.epita.fr/˜didier
http://www.facebook.com/didierverna
`@didierverna`

COP 2015 – Sunday, July 5

Table of contents

Introduction

The Common Lisp Image Manipulation Bundle

Climb

- Highly generic image processing library
- DSL / GML for complex image processing chains
- Inspired by Milena (C++ / templates)

Genericity drawbacks

- Performance degradation
- Code cluttering / OO Design breakage

Agenda

- Public: reconciling genericity and performance
- Hidden (not so) : explore the benefits of a multi-paradigm dynamic language

Generic Image Processing

Abstracting images, neighborhoods, pixels *etc*

The duality of “pixels”

- A value ? A location on a 2D grid ?

2 key concepts: sites and values

- $Image = f(site) \rightarrow value$
- Site sets: (iterators) full images, neighborhoods *etc*
- Values: (regular OO design) RGB, RGBA, bits, ints, floats, 32, 64 *etc*

Generic Image Processing

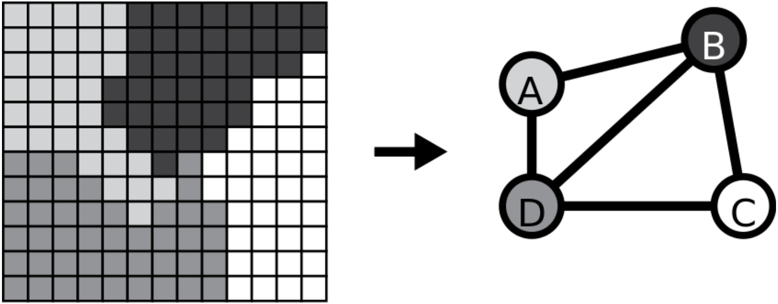
Abstracting images, neighborhoods, pixels *etc*

Generic Dilation Algorithm

```
(defun dilation
    (image &aux (result (copy image)))
  (do-sites (site (domain image))
    (let ((max no-value))
      (do-sites (neighbor (neighbors site))
        (setq max (max max
                       (iref image neighbor))))
      (setf (iref result site) max)))
  result)
```

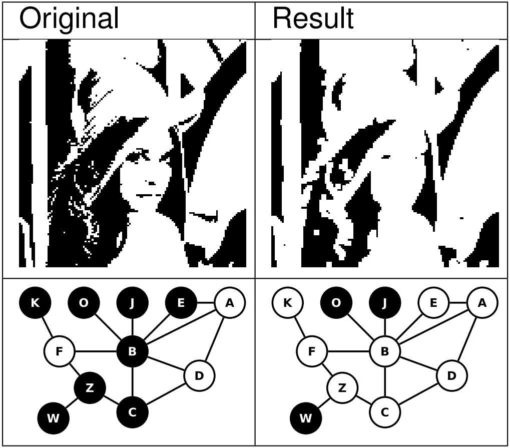
Graph-Based Image Example

Segmentation



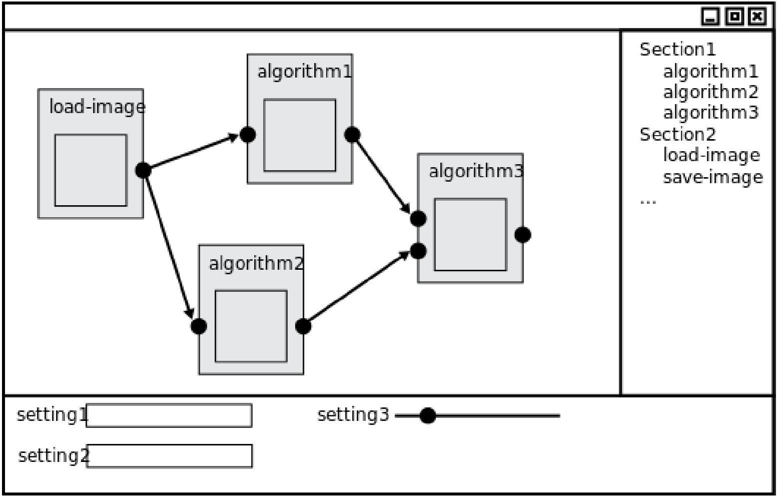
Dilation Algorithm Examples

On regular 2D and graph-based images



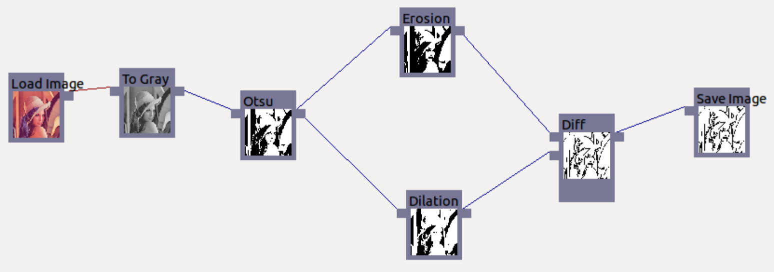
The GUI / GML

Climb also provides a textual DSL



A GML Example

Contour Detection Algorithm



Contextual Image Processing

Rationale

Generic Image Processing Drawbacks

- Image specificities not taken into account
- Runtime cost for abstraction layers (in general)
- Even worse for image processing

Image Specificities

To be taken into account

Reasonably easy

- Image formats, storage types, pixel values *etc*
- Still, code cluttering (class proliferation)

Cross-cutting

- Image *properties*
- Orthogonal to regular specificities
- Example: `speed` property for site access
  - `slow`
  - `fast` ($O(1)$)
  - `fastest` ($O(1)$ + pointer arithmetic)
  - Depends on *both* the image type and the site-set type

Introducing Contexts

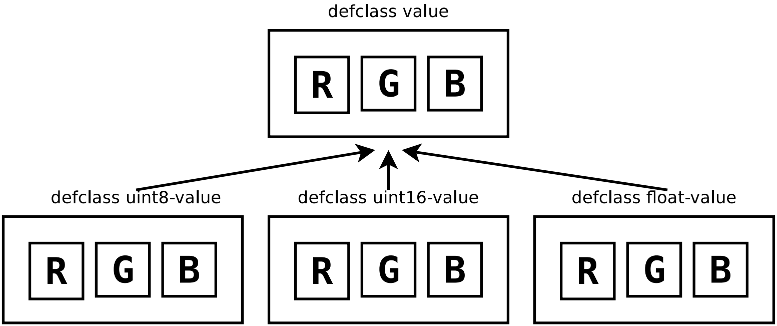
"Cross-cutting" should ring a bell!

Layering image properties/specificities

- Layered generic functions: algorithm specialization
- Layered classes: structural specialization

Behavioral Optimization Example

Static Typing



- Dynamic types ⇛ polymorphic operations (slow)
- Subclassing ⇒ class proliferation (bad)

Layering value classes

Layered static types

Layered RGB class

```
(deftype uint8-color () '(unsigned-byte 8))
(deflayer uint8-color-value)

(define-layered-class rgb
    :in-layer uint8-color-value (value)
  ((red :type uint8-color)
   (green :type uint8-color)
   (blue :type uint8-color)))
```

Layering functions

Layered static types

Optimized algorithms

```
(define-layered-method make-grayscale
  :in-layer uint8-color-value ((rgb rgb))
  (declare (optimize (speed 3) (safety 0)))
  (make-instance ’grayscale
    :intensity
    (the uint8-color
      (round (the float
               (+ (the float (* (red rgb)
                                0.299))
                  (the float (* (green rgb)
                                0.587))
                  (the float (* (blue rgb)
                                0.114))))))))
```