Context-Oriented Image Processing

Reconciling genericity and performance through contexts

Didier Verna
didier@lrde.epita.fr

François Ripault
francois.ripault@lrde.epita.fr

EPITA Research and Development Laboratory
14-16, rue Voltaire
94276 Le Kremlin-Bicêtre CEDEX

ABSTRACT

Genericity aims at providing a very high level of abstraction in order, for instance, to separate the general shape of an algorithm from specific implementation details. Reaching a high level of genericity through regular object-oriented techniques has two major drawbacks, however: code cluttering (*e.g.* class / method proliferation) and performance degradation (*e.g.* dynamic dispatch). In this paper, we explore a potential use for the Context-Oriented programming paradigm in order to maintain a high level of genericity in an experimental image processing library, without sacrificing either the performance or the original object-oriented design of the application.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-Oriented Programming*; D.3.2 [**Language Classifications**]: Object-Oriented Languages; D.3.3 [**Language Constructs and Features**]: Classes and Objects; I.4.0 [**Image Processing And Computer Vision**]: General—*image processing software*

General Terms

Design, Languages

Keywords

Genericity, Performance, Context-Oriented Programming, Image Processing

1. INTRODUCTION

Climb is an experimental image processing library designed to be highly generic. The idea behind genericity is to be able to write algorithms only once, independently from the data types to which they may be applied. In the context of image processing, being fully generic means being independent from the image formats, pixel types, storage schemes *etc.* For example, Climb provides transparent support for graph-based images, that is, images where the pixel adjacency relation is represented by a graph instead of a rectangular grid. In order to ease the writing of complex processing chains, the library also provides a domain specific language and a graphical modeling language.

Climb is actually inspired by Milena[11], another library developed by a sister team in our laboratory as well. While most image processing libraries out there (Milena included) are written in C or C++ for performance, and sometimes at the expense of genericity, we take a different approach. Instead of grounding our library in a statically typed language, we arbitrarily choose to use Common Lisp[19, 18] instead, in order to explore what a multi-paradigm dynamic language has to offer in the context of genericity and performance. In particular, we are interested in studying the expressiveness of dynamic object-orientation mechanisms such as context-oriented programming.

Section 2 gives an overview of what generic image processing is all about. Sections 3 and 4 describe potential optimizations available in image processing and how it is possible to use contexts in order to implement them in a cross-cutting fashion. Section 5 provides some performance results. Finally, we give some conclusions and directions for future work.

2. GENERIC IMAGE PROCESSING

When they design image processing algorithms, researchers use abstract concepts such as “finding a maximum value” or “browsing a neighborhood”. While these concepts make perfect sense for a human being, a computer needs much more information to actually perform the tasks. For example, finding a maximum value would require knowledge about the data structure in which the set of values is stored, and the type of values the computer is looking at.

The goal of Climb is to provide a layer of abstraction allowing researchers to express abstract image processing concepts directly. By providing a small, predetermined set of high level operations, researchers are able to write only one generic implementation of an algorithm that can work on very different kinds of images, without any prior knowledge of either their structure or the manipulated values.

2.1 Image Definition

2.1.1 Sites

The most frequent kind of image uses a regular 2D grid to store its pixel values. In that case, the term “pixel” can

COP ’15 Prague, Czech Republic

be interpreted in two ways: either the position on the grid, or the actual value stored there. In order to remain generic enough, we need to clearly separate these two notions. Besides, digital images are not necessarily represented on a 2D grid. Values can be stored on hexagonal grids, 3D matrices, graphs *etc.*

We hence define an image as a function from a location, called a *site*, to a value: $image(site) \rightarrow value$. The `site` abstraction allows us to manipulate images without knowing exactly which data structure is used for storing values (2D points or nodes in a graph for example).

Based on sites, `site-set` objects are iterators representing a set of locations such as the whole image (called the image *domain*) or a neighborhood. By browsing site sets, an algorithm is able to process different kinds of areas (a set of nodes in a graph, a 3D cube *etc.*) by using the same high level iterator, and again, without any prior knowledge about the underlying data structure.

2.1.2 Values

Pixel values may come in many different types, such as floating point numbers for gray-scale images, Booleans for black and white images, RGB triplets or RGBA quadruplets for colored images.

Genericity on pixel value type is implemented in a traditional object-oriented fashion, via a `value` class hierarchy, allowing us to define generic operators such as comparison and addition. These operators can in turn be used in image processing algorithms without introducing type-dependent specialization, hence preserving complete genericity.

2.1.3 Algorithms

Listing 1 shows an implementation of the dilation[17] algorithm using the generic infrastructure provided by Climb. Note that there is no information about the underlying image data structure or pixel value type in the expression of this algorithm. It will hence work out of the box on any kind of image (including the ones for which there would be no concrete implementation yet).

```
(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))))
      (setq (iref result site) max)))
  result)
```

Listing 1: Generic dilation algorithm

2.2 Graph Images

The very high level of genericity provided by the library allows us to transparently handle very peculiar kinds of images, such as graph-based ones.

2.2.1 Description

In Image Processing, it is customary to use graphs for representing special kinds of images such as segmented ones. This is illustrated in figure 1. In the original 2D image, a *segment* is an area in which all pixels have the same value. In the graph representation, every *node* (or *vertex*) corresponds to a segment and *edges* represent the inter-segment connexity relations.

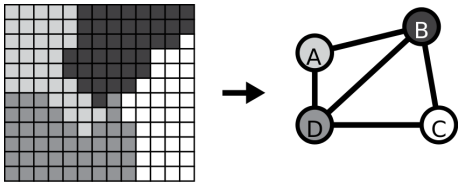


Figure 1: Segmented 2D image to graph representation

2.2.2 Usage

In Climb, graph-based images are just another kind of image and are used in the exact same fashion as any other. The `site` abstraction supports graph-based images by making graph nodes (even edges in some cases) a special kind of pixel. All other concepts in the library (neighbors, iterators *etc.*) will work equally well and transparently. As a consequence, the dilation algorithm presented in listing 1 will work on graph images without any modification to the code.

Figure 2 exhibits the result of this algorithm on two very different images. Note again that the same, unique implementation of the algorithm is used in both cases.

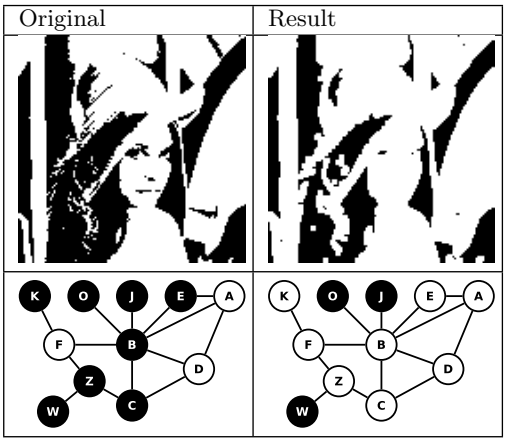


Figure 2: Dilation algorithm on different images structures

2.3 Processing Chains

Image processing applications often involve more than just one algorithm. It is in fact frequent to use a "chain" of algorithms and apply them sequentially, in parallel with merging, or even both in order to get the desired result. For example, a contour detection application may require first the conversion from color to gray-scale, and then different passes of morphological operators such as erosion and dilation.

Climb provides a domain-specific language for the building of complex image processing chains. Although this facility makes the task much easier for programmers, many "clients" of image processing software are not even programmers at all. For that reason, Climb also provides a graphical modeling language. This language allows to visually design complex chains of algorithms without any knowledge in programming. It also provides real-time visualization of the intermediate and final results of the current processing chain (figure 3). In the graphical user interface (GUI), algorithms are depicted as graphical boxes that can be moved around and interconnected with each other (figure 4).

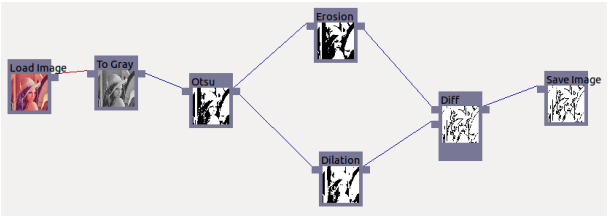


Figure 3: Visual display of the processing chains

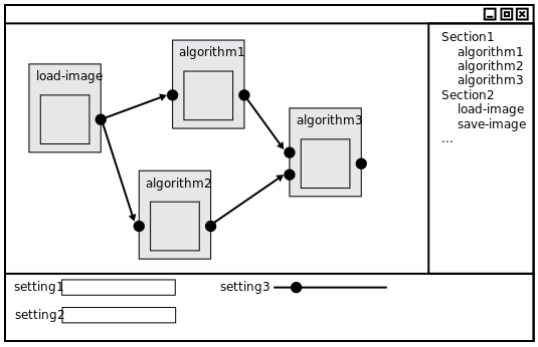


Figure 4: Interface Layout

3. CONTEXTUAL IMAGE PROCESSING

The development of generic algorithms (image processing ones in particular) by traditional object-oriented techniques is usually bound to have a negative impact on performance, for several reasons.

1. Generic algorithm implementations do not take into account image specificities that could lead to potential optimizations,
2. crossing abstraction layers at run-time is costly in general,
3. in the image processing field in particular, algorithms often involve very small/short methods called very often, hence an even greater proportional cost for mechanisms such as dynamic dispatch.

3.1 Image Specificities

Regarding the first point, one way to improve the performance of generic algorithms is to provide specialized versions that take into account the various specificities of an image.

As mentioned earlier, such specificities include image formats, pixel types, storage schemes *etc.* One problem here is that if we want to take all those specificities into account in the object-oriented design of the library, it would lead to class/method proliferation, which we want to avoid.

Images also have other interesting specificities which are orthogonal to the ones mentioned above, and which we would rather call "properties"[12]. The efficiency of site access, that we call the `speed` property, is one of them. This property may have 3 values: `slow`, meaning that the access complexity is greater than $O(1)$, `fast` for $O(1)$, and `fastest` meaning $O(1)$ plus pointer semantics (access to the pixel values of an image directly through pointers). This property is interesting for optimization purposes: because the access speed to an image can be fast, a processing chain could use a specific image traversal algorithm, faster than the generic one. However, this property is hard to express in the original oriented-object design of the library because it depends on both the image type and the site set type.

To sum up, the difficulty we face in order to preserve both genericity and performance is to avoid cluttering up the original object-oriented design of the library, and to express properties that are orthogonal to it, hence cross-cutting. That is why using context orientation seems like an interesting idea to try out.

3.2 Contextual Image Properties

Context-oriented programming[6] is a paradigm that addresses cross-cutting concerns and context-dependent behavior in a program. We argue that image properties can be seen as contextual information.

Many image processing algorithms can be specialized in different ways for specific sets of properties. By representing such properties with layers (in our case, using ContextL[2]), the applicability of such or such specialized algorithm version can be decided automatically based on the set of currently active layers.

Because ContextL allows us to layer not only methods but classes as well, we can even optimize beyond just algorithms. We can also optimize the low level implementation of our data structures (image types, values, storage *etc.*).

4. CONTEXTUAL OPTIMIZATION

This section presents two cases of contextual optimization, both at the behavioral and structural levels.

4.1 Behavioral Optimization: Static Types

The first idea is to provide low level optimizations for the representations of pixel values. In the fully generic implementation, a `value` class such as `rgb` provides 3 slots (one for each color channel). Because each component of an RGB value can in turn be represented as 8 or 12 unsigned bits, single precision floats *etc*, the slots are dynamically typed. As a consequence, all arithmetic operations in the generic versions of our algorithms involve dynamic dispatch on the actual numerical types of their arguments, which has a dramatic impact on performance.

CLOS[5, 8], the Common Lisp Object System would allow us to subclass the `rgb` class and provide (optional) static type declarations for every slot. This, however would complicate the object oriented design of the application for a concern which is admittedly cross-cutting, since it deals with optimization only. So instead, we choose to layer our pixel value classes definitions.

We first define type-specific layers: `value-simple-color` represents components defined on 8 bits unsigned integer, `value-bit` represents binary values, *etc.* Within these layers, classes such as `rgb` can be specialized with type information. Listing 2 describes one such specialization.

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

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

Listing 2: Layered definition for class `rgb`

Provided with this layered definition, functions manipulating `rgb` values can in turn be specialized with type information. Listing 3 exhibits a layered definition for the `make-grayscale` constructor, which creates a `grayscale` value from an `rgb` one.

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

Listing 3: Layered definition for method `make-grayscale`

Provided with such type information, and along with appropriate compiler optimization settings, this code will effectively be compiled as statically (and weakly) typed code with fully dedicated arithmetic operations. It has been shown that the level of performance hereby achieved can be similar to that of semantically equivalent C code[21].

Specializing the code in such a way is only a fraction of the possibilities offered by the context-oriented approach. In particular, one limitation to this approach is that the object-oriented interfaces need to remain unchanged in order to stay compatible with the rest of the library. The next section presents another approach where the interfaces themselves are modified in order to obtain further performance improvements.

4.2 Structural Optimization: Classless Representation

In "generic" Climb, a pixel value is an aggregation of several color channels. In the previous section, we demonstrated how pixel value classes can be specialized with type information while leaving the oriented-object design of the library unchanged.

Another approach towards performance, however, is to *inline* such values directly within the images themselves. By default, nD images are represented with an n-dimensional array containing indirect references to pixel values. In a fully dedicated version however, a more efficient representation would be to use a 1-dimensional array concatenating the image lines, and in which every cell represents one component of a pixel value. For instance, in the case of an RGB image, the first cell would contain the red component of the first pixel, the second cell the green component, the third cell the blue component *etc.* The advantage of this approach is a more compact memory representation and at the same time a more efficient pixel access (a slight arithmetic overhead for indexation instead of using accessor functions to intermediate `site-set`, `site` and `value` objects). Implementing this optimization in a contextual fashion is outlined below.

We define one layer for every possible number of color components in an image, and as before, one layer for every type of pixel value. For instance, there is a `channel-3` layer corresponding to RGB images. Note that as soon as we know on which particular image we're working, we have all the required information to activate the appropriate layers. Once activated, these layers induce the specialization of I/O functions such as `load-image`, which in turn use the appropriately optimized image memory representation.

With this optimized design in place, images no longer contain references to `value` objects, which is normally expected by the rest of the library. At the same time, we don't want to recreate such objects on the fly, only for the sake of protocol compatibility. So instead, we need to propagate those structural changes to the whole library.

The solution we adopt here is the following. A specialized version of the `value` *type* (not a class anymore) is defined as a pair containing a reference to the image, and the index of the first component of the concerned value (red for RGB or RGBA). Provided with this new definition, we also redefine methods that operate on values. Listing 4 details the implementation of the `vref` method which returns one component from a 3-components value. Note that values are not objects in the CLOS sense anymore. On top of this function, a small layer of macrology defines direct red, green and blue accessors by calling `vref` with the appropriate channel number.

```
(define-layered-method vref :in-layer channel-3
    (value &optional (channel 0))
  (declare (optimize (speed 3) (safety 0)))
  (let ((img (car value))
        (index (cadr value)))
    (declare (type fixnum index channel))
    (aref (image-raw-data img)
          (+ channel index))))
```

Listing 4: Implementation of `vref` in the `channel-3` layer

A similar optimization can be achieved on the `site` hierarchy. For classical nD images, sites boil down to a set of nD coordinates, and, as mentioned before, we don't want to recreate fully generic `site` objects on the fly, only for the sake of protocol compatibility with the rest of the library. So for instance, we define a `2d-coordinates` layer, in which sites are now represented by a pair of x and y coordinates. This structural change impacts the rest of the library, and functions such as `iref` (image sites accessor) are layered in order to take into account both the optimization on values and the specialization on sites.

5. PERFORMANCE RESULTS

In order to get an idea on the performance improvements entailed by the contextual optimizations described in the previous section, we analyze various low-level aspects of the library, plus one particular example of a more general image processing chain, depicted in listing 5. In this example, we load an image, convert it to a gray-scale one, apply the Sauvola binarization algorithm [15] and finally save the result back to a file.

```
(let* ((original (image-load "rgb.bmp"))
       (grayscale
         (make-image (image-domain original)
           :initfunc (lambda (site)
                       (make-grayscale
                        (iref original site)))))
       (binary (sauvola grayscale (box2d 1))))
  (image-save binary "bw.jpg"))
```

Listing 5: Example of an image processing pipeline

In this particular case, the original is a 2D RGB image, so the optimized version activates the `channel-3` and `value-simple-color` layers, as described earlier. Note that turning

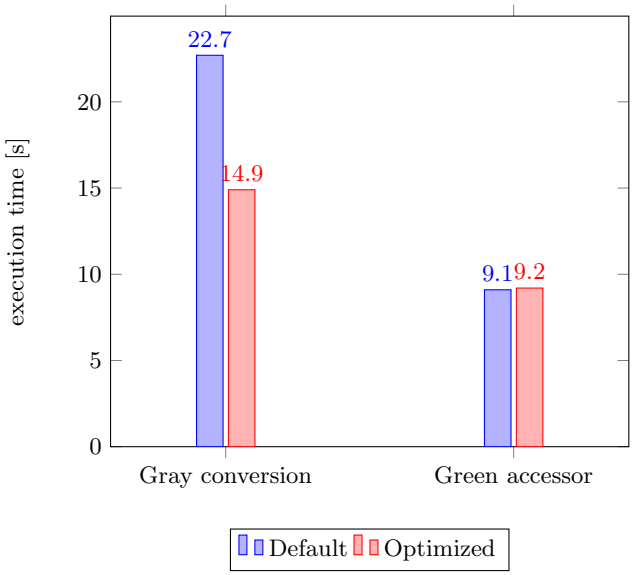


Figure 5: Performance with contextual types

optimizations on is done by activating a special layer (which, in turn, triggers the activation of the other ones), so no change to the code of listing 5 is necessary.

We first analyze the performance gain achieved with contextual static types, and then with structural changes. The benchmarks presented in this section have been conducted on an Intel Core 2Duo (3GHz) with 4GB of RAM, using Debian 3.2. The compiler used is SBCL 1.0.57.

5.1 Improvements with Static Types

Figure 5 shows the performance improvement achieved with the optimizations exposed in section 4.1. Two methods are analyzed: `make-grayscale`, the implementation of which has been detailed in listing 3 and the `green` accessor, which accesses the green component of an RGB value. These functions were called 100 million times. We notice that the specialized version of `make-grayscale` runs 1.5 times faster than the generic one. The `green` accessor, however, does not benefit from any performance improvement.

These results are to be expected. `make-grayscale` involves arithmetic operations so it is normal to witness some performance improvement when dynamic typing (hence polymorphism) is removed in favor of static (and weak) arithmetic types, as fully dedicated operations are now used.

As mentioned earlier, a specificity of image processing is the abundance of very short methods (like pixel-wise operations) called a huge number of times. This explains the relatively modest improvement observed for `make-grayscale`, as this function not only spends time on arithmetics, but on value access as well. It also explains the total lack of improvement for the `green` accessor. Even though contextual types may help the compiler unbox and inline values, the impact of this is still negligible compared to the time spent traversing generic object through their various accessors.

5.2 Improvements with Classless Representation

Figure 6 details the execution time and and memory usage improvement with the optimizations exposed in section 4.2. In listing 6b, results are presented with the unit "Mega cons", which represents a million "consing" operations, that is, the primitive allocation facility for objects in Common Lisp.

`sref` is the generic site access method for site sets. This function is called 1 million times on a 128×128 `site-set-box` and we notice a huge performance improvement both in terms of execution time and memory usage. Replacing CLOS objects with pairs in order to describe coordinates has a very beneficial impact on the performance of the library. This performance improvement also impacts image traversal facilities such as `do-sites` (called 500 times on a 128×128 image) and value access on images with `iref` (called 500 times on a 128×128 image). Finally, high level algorithms such as `sauvola`, and `pipeline`, which is the function presented in listing 5 are 3.5 times faster, with a similar gain on memory consumption (each algorithm is called once on a 128×128 image).

It is important to realize that even if this approach introduces structural changes to the library, only low level components (such as `sref` and `iref`) are affected. High level algorithms such as `sauvola` are not modified in any way and yet, benefit a great deal from these contextual optimizations.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a possibly unusual use for context-oriented programming: performance optimization in an image processing library. This study is in its very early stages, and the image processing domain is admittedly an excuse to study the more general idea of contextual optimization. Yet, even at this early stage, the results described here are very encouraging. We demonstrated that it is possible to use contexts for providing both behavioral and structural specializations to the library's building blocks, resulting in important performance improvement, all of this without sacrificing either the genericity of the user-level programming interface, or the general design of the library.

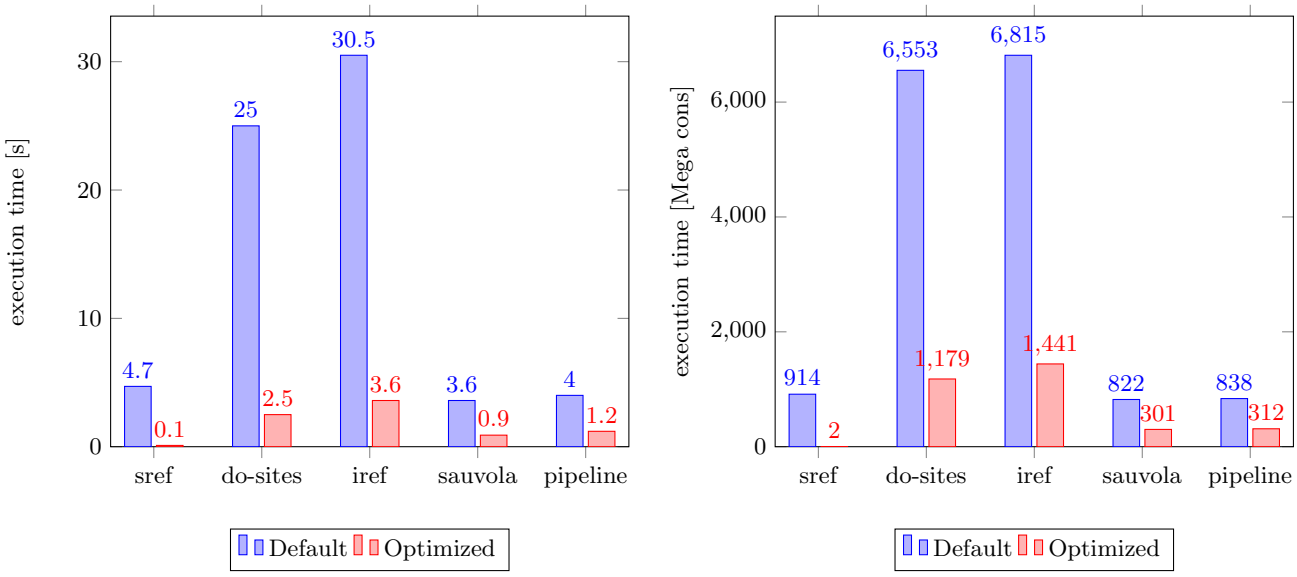
6.1 Directions with Contexts

Note that we currently don't know the potential performance impact of the ContextL infrastructure itself. ContextL goes through the CLOS meta-object protocol (MOP) [10, 14] to dynamically generate classes that represent combined layers, and dynamically dispatches on layered functions. Eventhough it is possible to efficiently implement layer activation [3], it is nevertheless possible that the cost becomes proportionally important in cases where we would end up doing a lot of context switches. We plan to investigate this in the future.

Despite the encouraging results presented here, we have already encountered several limitations that we intend to address in the future as well. Some of them are outlined below.

In ContextL, layers are only defined as symbols and do not retain state *per se*, which, in our opinion, limits their expressiveness. In Climb, we had to define layered functions to obtain information about the currently activated layers: for instance, the function `get-value-type` is necessary to introspect the currently active pixel value type (*e.g.* `bit` when the layer `value-bit` is activate). Ideally, layers should be stateful, perhaps objects or even first-class citizens.

Another related concern is the modelization of relations between layers, in particular, joint activation logic: in Climb,



(a) Execution time

(b) Memory consumption

Figure 6: Performance and memory usage with contextual structures

the layer `specialized` "contains" all contextual optimizations by encapsulating layers that describe the type and the number of components of a pixel value. This is however not explicit in the code because ContextL provides no mean to express logical relations between layers. Hence, layers supposed to be active at the same time need to be all (de)activated manually, and it is possible to put the library in an illogical state resulting in run-time errors.

Finally, we also faced the well-known "coercion problem". As mentioned earlier, some image processing chains may involve parallel branches (figures 3 and 4) in which different optimizations are active. When two or more parallel branches are joined back together, data coming from different contexts need to be "reunited", and there is currently no clean / automatic way to coerce objects from one context to another.

6.2 Directions with Other Paradigms

Context-oriented programming is one possible answer to a general question we have been investigating: expressing optimization as a cross-cutting concern, that is, without losing either genericity or the original design. On the long term, we plan to incorporate other paradigms in this study, and provide an in-depth comparison of their respective merits.

Aspect oriented programming [9] is a related paradigm which has been used for optimization already (loop fusion, memoization, pre-allocation of memory *etc.*). [13] details a case-study of aspect-oriented programming for an image processing library. We also intend to compare with a mixins [16] approach, a purely functional one. Research comparing aspects, mixins and monads already exist [7, 4].

Finally, we also intend to investigate on more recent paradigms enabled in dynamic object-oriented environments, such as predicate or filtered dispatch [20, 1].

7. REFERENCES

[1] P. Costanza, C. Herzeel, J. Vallejos, and T. D'Hondt. Filtered dispatch. In *Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008, July 8, 2008, Paphos, Cyprus*, page 4, 2008.

[2] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the 2005 Symposium on Dynamic Languages*, DLS'05, pages 1–10, New York, NY, USA, 2005. ACM.

[3] P. Costanza, R. Hirschfeld, and W. D. Meuter. Efficient layer activation for switching context-dependent behavior. In *JMLC'06: Proceedings of the Joint Modular Languages Conference*, pages 84–103. Springer, 2006.

[4] B. C. d. S. Oliveira. The different aspects of monads and mixins, 2009. Draft Paper. Last Update: 04/03/2009. Submitted to ICFP 2009.

[5] R. P. Gabriel, J. L. White, and D. G. Bobrow. Clos: integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):29–38, 1991.

[6] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[7] C. Hofer and K. Ostermann. On the relation of aspects and monads. In *Proceedings of the 6th Workshop on Foundations of Aspect-oriented Languages*, FOAL '07, pages 27–33, New York, NY, USA, 2007. ACM.

[8] S. E. Keene. *Object-Oriented Programming in* Common Lisp*: a Programmer's Guide to* Clos. Addison-Wesley, 1989.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[10] G. J. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[11] R. Levillain, Th. Géraud, and L. 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)*, pages 1941–1944, Hong Kong, Sept. 2010.

[12] R. Levillain, Th. Géraud, and L. 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)*, Bordeaux, France, Sept. 2011. In French.

[13] A. Mendhekar, A. Mendhekar, G. Kiczales, G. Kiczales, J. Lamping, and J. Lamping. RG: A case-study for aspect oriented programming. Technical report, Xerox Parc, 1997.

[14] A. Paepcke. User-level language crafting – introducing the CLOS metaobject protocol. In A. Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, chapter 3, pages 65–99. MIT Press, 1993. Downloadable version at http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps.

[15] J. Sauvola and P. M. Adaptive document image binarization. *Pattern Recognition*, 33:225–236, 2000.

[16] Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs, 2001.

[17] P. Soille. *Morphological Image Analysis: Principles and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2 edition, 2003.

[18] G. L. Steele. COMMON LISP *the Language, 2nd edition*. Digital Press, 1990. Online and downloadable version at http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html.

[19] ANSI. American National Standard: Programming Language – COMMON LISP. ANSI X3.226:1994 (R1999), 1994.

[20] A. M. Ucko. Predicate dispatching in the common lisp object system, 2001.

[21] D. Verna. Beating C in scientific computing applications. In *Third European LISP Workshop at* ECOOP, Nantes, France, July 2006.