Component Trees and Chaining Operators in Climb

Christopher Chedeau

Technical Report n^o1108, June 2011
revision 2295

Climb is a generic image processing library designed to be used for rapid prototyping. The implementation of two component trees algorithms impacts Climb in several ways: the definition of values is extended, new site sets are added and debugging utilities are improved.

A detour is taken to understand the Chaining design pattern popularized by the Javascript jQuery library. The pattern is adapted to both the image processing domain and Common Lisp and is extended to introduce a parallel notation as well as better control flows.

Climb est une bibliothèque de traitement d'image générique ayant pour objectif le prototypage rapide. L'implémentation de deux algorithmes d'arbre de composantes impacte Climb de plusieurs façons : la définition des valeurs est étendue, de nouveaux ensembles de sites sont ajoutés et les outils de développement sont améliorés.

Un détour est pris afin de comprendre le patron de conception de chaînage popularisé par la bibliothèque jQuery. La méthode est modifiée afin de s'adapter au traitement d'image ainsi qu'à Common Lisp. Elle est également étendue via une notation parallèle ainsi qu'avec une meilleure gestion du fil d'exécution.

Keywords
climb, image processing, dynamic language, chaining, component tree

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
`christopher.chedeau@lrde.epita.fr` – http://www.lrde.epita.fr/

Copying this document

Contents

Chapter 1

Introduction

The image processing domain requires high performance as the inputs are big: most images nowadays are composed of millions of pixels. As a consequence, most libraries are written in C or C++ with a high focus on speed. It is often done at the expense of ease of use. Climb has been designed from the ground up to be a developer friendly library. Using Common Lisp, Climb benefits from the dynamic aspect and functional abilities of the language.

Climb is in its infancy. The core aspects of the library such as sites, site sets and morphers and the implementation of most mathematical morphology operators have been established in previous work (Denuzière (2009), Chedeau (2010)). The next step is to introduce many new algorithms from various areas of image processing and see if the framework supports them.

In this article, we concentrate on two different algorithms that produce the component tree of an image. The purpose of this article is not about comparing performance nor studying use cases of the two algorithms but more about showing how their implementations impacts the design of Climb.

Eventually, we take a detour to the Javascript world and specifically the jQuery[1] library to discover a design pattern to chain operations. We adapt it to the image processing world and improve it thanks to Common Lisp expressive power in order to "Write less and do more"[2].

Acknoledgements

Thanks to Didier Verna for being both a great teacher and project manager, Laurent Senta and Simon Guillot for helping out on the Climb development, Lorene Poinsot and Alex Hamelin for the help during the preparation of this report.

[1] `http://jquery.com/`

[2] jQuery: The Write Less, Do More, JavaScript Library

Chapter 2

Component Tree

2.1 Component Tree

2.1.1 Definition

A component tree is the organization in a tree structure, thanks to the inclusion relation, of the level set connected components of an image. In other words, if we see our image in 3D where the z-coordinate is the pixel intensity. A slice of a mountain is called a connected component and two slices are connected if they are on top of each other. A visual example is available in **Figure** 2.1.

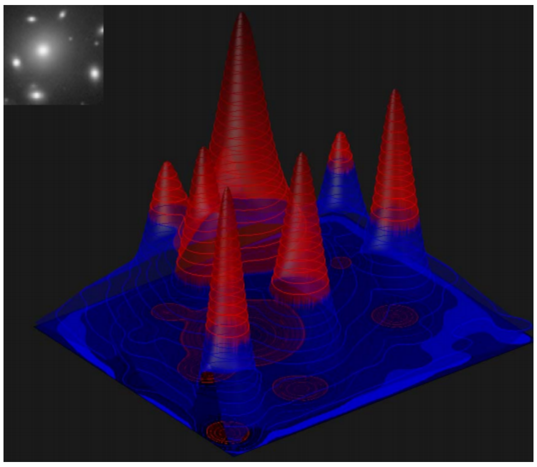

Figure 2.1: Component tree 3D representation Gonzalez et al. (2004)

We implement two algorithms to compute the component tree: Najman and Couprie (2004) and Berger et al. (2007). They are similar algorithms as they are both based on Tarjan's union-find disjoint set forest computation (Géraud (2005)). They have been chosen because the component tree they produce is stored in two different ways.

Let us take a simple example. In input, **Figure 2.2**, we have a 3x5 image in gray level. In **Figure 2.3**, we label each node (or pixel) with a capital letter.

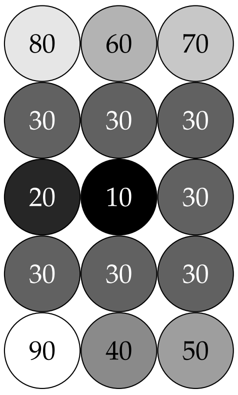

Figure 2.2: Node Values

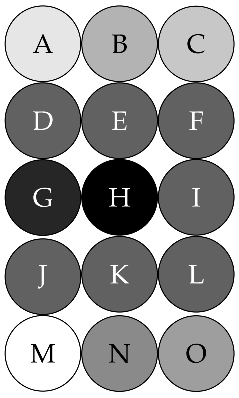

Figure 2.3: Node Names

2.1.2 Najman

The node H has the lowest value, and is therefore part of the root component of the tree. In order to climb the tree, we go to G and then there's a floor composed of D, E, F, I, L, K, J. At this step, the tree is split in two on the nodes N and B. Each of them can join two other components O, M and C, A. The first algorithm, Najman and Couprie (2004), computes the component tree in **Figure 2.4**. It also provides for each node, the canonical element of its component: **Figure 2.5**.

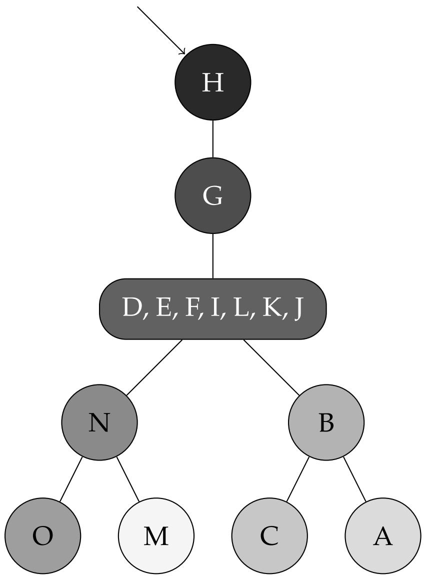

Figure 2.4: Component Tree

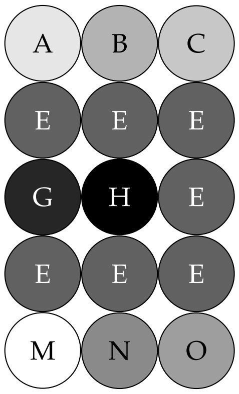

Figure 2.5: Components

2.1.3 Berger

The second algorithm, Berger et al. (2007), gives another representation of the component tree. Instead of building the tree, it provides a parent relation seen in **Figure** 2.6. If the node is canonical, the parent points to the canonical element of the parent component, else to the canonical element of its own component. In order to use the parent relation, the nodes have to be iterated in increasing order of values. Therefore, a structure containing the nodes in sorted order is also returned (**Figure** 2.7).

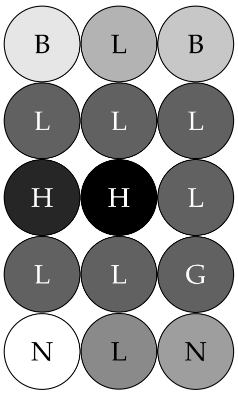

Figure 2.6: Node Parents

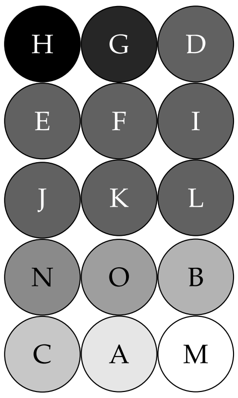

Figure 2.7: Sorted Nodes

Climb has been designed to provide an interface that matches operations used in literature. Therefore, the implementation of the algorithms is straightforward. **Figure** 2.8 shows that the lisp code is the exact translation of the pseudo-code found in Berger et al. (2007) paper.

```
CANONICALIZE-TREE($parent, f$)
for each $p \in R$ in reverse order do
    $q \leftarrow parent(p)$
    if $f(parent(q)) = f(q)$ then
        $parent(p) \leftarrow parent(q)$
return $parent$
```

```
(defun canonicalize-tree (parent sorted-sites image)
  (do-sites (p sorted-sites :backward)
    (let ((q (iref parent p)))
    (when (value-equal (iref image (iref parent q)) (iref image q))
      (setf (iref parent p) (iref parent q)))))
  parent)
```

Figure 2.8: Translation of the pseudo-code to real code

2.2 Impact on Climb

Climb has first been developed around mathematical morphology operators such as erosion, dilation or top-hat (Chedeau (2010)). Those simple operators were a good opportunity to focus on making a stable and powerful internal architecture. The implementation of those two algorithms demonstrates that overall the design was good enough to support them. The changes required are explained in this section.

2.2.1 Values

Berger algorithm requires to have a data structure that associates sites with sites while Najman algorithm requires to associate a site with a list of sites. The nearest concept available in Climb is the Image. It associates a site with a value, where the value is an array of elements that represents grayscale or RGB intensities. The natural evolution is to extend the notion of value.

An image value can be any common lisp value: numbers, lists, strings, objects and even images. Contrary to Olena[1] (in C++) and our old design, the image container is no longer required to know the type of its content. Instead, each element knows its own type. As a side-effect, it makes possible to create heterogeneous images.

The writing of morphers (Chedeau (2010)) is easier as we no longer have to manually enter the output type, which can be complicated. As we make use of classes for values, the code in **Figure 2.9** is also more meaningful for the reader.

```
; Legacy version
(image-direct-value-morpher
  image :gray
  (lambda (gray) (vector gray gray gray)) ; Reader
  (lambda (rgb) (/ (reduce #'+ rgb) 3)))  ; Writer

; Version with new Value representation
(image-direct-value-morpher
  image
  (lambda (gray) (make-rgb :r gray :g gray :b gray)) ; Reader
  (lambda (rgb) (make-gray :i (/ (+ (value-r rgb)    ; Writer
                                     (value-g rgb)
                                     (value-b rgb)) 3))))
```

Figure 2.9: Implementation of a RGB to Gray morpher

In order for algorithms to be written once and work for all types of values, we use duck typing (Martelli (2000)). All images with values that implements the generic functions `equal` and < can be used to make a component tree. It makes the addition of new value types straightforward.

The Image definition in Climb is now more in line with the image processing literature. It is nothing more than a function associates a value to a site. Each algorithm can use the best fitting implementation of the image and interpret the value as it needs to.

[1] http://olena.lrde.epita.fr/

2.2.2 Developer Utilities

make-image-[bool, gray]

Basic values such as grayscale or RGB are no longer primitive types but classes. As a consequence, we have to call a constructor function if we want to use them from primitive types. It makes the creation of small test images harder. We introduce two functions `make-image-bool` and `make-image-gray` to help the developer.

```
(make-image-bool (dimensions values))
(make-image-gray (dimensions values))
```

In order to represent boolean values, common lisp uses `nil` and `t`. As they do not have the same number of characters, it makes displaying them in a textual grid a tedious process. Looking for an alternative, we thought about `0` and `1`. However, the two symbols are hard to distinguish one from the other. We finally chose to use `.` and `#`.

Figure 2.10 and **Figure** 2.11 demonstrate how easy it is to use them.

```
(print-image-2d
(make-image-bool '(4 4)
"# . # .
 . # . #
 . . # .
 . . . #"))

; # . # .
; . # . #
; . . # .
; . . . #
```

Figure 2.10: Boolean Utilities

```
(print-image-2d
(make-image-gray '(4 4)
'(  0  10  20  30
  100 255   0   0
   10  12  18  14
    0   0   0   1)))

;   0  10  20  30
; 100 255   0   0
;  10  12  18  14
;   0   0   0   1
```

Figure 2.11: Grayscale Utilities

print-image-2d

We also add a `print-image-2d` function that prints an image to the console. The goal of the function is to provide a practical way to visualize a small image. To make it developer-friendly, the output of boolean and grayscale images uses the same format as `make-image-*`.

```
(print-image-2d (image &optional (repr 'value-repr)))
```

It is possible to provide a custom function to display the value. For example, it is useful to print the parent representation of the component tree. **Figure** 2.12 shows how to combine `make-image-gray`, `component-tree` and `print-image-2d`.

As it is hard to get a generic library right, we first start by writing non-generic functions. Only once there are multiple functions with the same goal and different types of arguments, we can merge them. In this case, we first wrote `print-image-bool-2d` and extended it. However, we did not yet have the need to print other images than 2d ones. It would be premature to write down `print-image` without real use cases.

We may also merge `make-image-bool` and `make-image-gray` with `make-image` in the future. We will consider doing this when we will discover more use cases.

```
; Transforms Site(0, 0) into 'A', Site(0, 1) into 'B' ...
(defun repr-label (site)
  (code-char
   (+ 65
      (* 3 (1- (point-coordinate site 1)))
      (1- (point-coordinate site 0)))))

(destructuring-bind
  ((tree components)
   (component-tree
     (make-image-gray
      '(3 5)
      '(80 60 70
        30 30 30
        20 10 30
        30 30 30
        90 40 50))
     (4-connectivity))
 (print-image-2d tree #'repr-label))

;   B   L   B
;   L   L   L
;   H   H   L
;   L   L   G
;   N   L   N
```

Figure 2.12: Use of `component-tree` and `print-image-2d`

2.2.3 More Site Sets

Figure 2.8 introduces the notion of iteration on sorted site sets. Climb does not have any built-in mechanism to achieve this. However, it allows to add new types of site sets. We introduce `site-set-array`. It implements generic functions required by `do-sites` to iterate over it: `reset` and `next`. And implements two additional: `push` and `sort`.

Duck typing is not only used on values but also on site sets. If an object implements both `next` and `reset`, it can be used anywhere an iterable site set is required. In order to iterate over the site-set in reverse order, we add a `:backward` optional argument to the `reset` function.

Climb features many types of site sets:

Box	: `reset` (ø, `:backward`), `next`
Array	: `reset` (ø, `:backward`), `next`, `push`, `sort`
Stack	: `reset`, `next`, `push`
Queue	: `reset`, `next`, `push`
Heap	: `reset`, `next`, `push`
Graph	: `reset` (ø, `:bfs`, `:dfs`), `next`, `push`

Figure 2.13 is an example of an image traversal as it if was a graph. We start to create a boolean image `seen` with the same dimensions as the source image using `make-image`. Then we insert the start element into our structure and set it as seen. In line 5, we iterate over each `site` of the structure. After the line 7, we add all the unseen neighbors of the current site to the structure if they haven't been seen yet.

The `struct` variable can be either a `stack`, `queue` or `heap`. The choice of this structure allows to traverse the image in various ways such as Breadth-first search, Depth-first search, "Shortest-path" or even from the lowest values to the highest.

```
 1 (defun traverse (img struct window start)
 2   (let ((seen (make-image img :initial-element nil)))
 3     (site-set-push struct start)
 4     (setf (iref seen) t)
 5     (do-sites (site struct)
 6       ; Do something with site
 7
 8       (do-sites (neighb window site)
 9         (unless (iref seen neighb)
10           (setf (iref seen neighb) t)
11           (site-set-push struct neighb))))))
```

Figure 2.13: Use of all the site-set operations

Chapter 3

Chaining Operators

jQuery is a Javascript library designed to provide unified utilities to manipulate HTML documents across all the browsers and their respective quirks. It quickly grew in popularity as it introduced a new design pattern that solves well the problems the library tries to solve. jQuery is based on the chaining ability of Javascript Harmes and Diaz (2007).

We are going to explain what this technique is, its use in jQuery in order to translate it for the image processing domain. Common Lisp expressiveness even allows to extend the pattern.

3.1 Chaining

The idea behind chaining is execute more than one action at a time on the same object. It is trivial to implement in an Object Oriented language, the functions that can be chained should return `this`. **Figure 3.1** is an example in Javascript.

```
function Object() {}

Object.prototype = {
  methodA: function () {
    // do something
    return this;
  },

  methodB: function () {
    // do something
    return this;
  }
};

var obj = new Object();
obj.methodA().methodB();
```

Figure 3.1: Javascript implementation of chaining

Method chaining removes the need to declare temporary variables, and therefore makes the code much more readable and easier to write. **Figure** 3.2 is a real world example of chaining with jQuery. We want to process all the links of a page. We give them the `external` CSS class, make them open in a new tab when clicked and plug in an analytic tracking code.

```
$('a[@href^="http://"]')
   .addClass("external")
   .attr("target", "_blank")
   .click(function () { /* analytics call */ })
```

Figure 3.2: Multiple actions on the same object

Once we understand the power of this construction, we can lift the limit of returning only `this`. **Figure** 3.3 is a jQuery example where chaining is used to traverse a HTML document: it finds the parent of the first list element that contains a sublist. **Figure** 3.4 is a sample code from the v8cgi SQL Query module[1]. The last example in **Figure** 3.5 implements the basic concept of method chaining but in an asynchronous way (Diaz (2010)).

```
$("li")
   .has("ul")
   .eq(1)
   .parent()
```

Figure 3.3: Traversing a HTML Document

```
SQL("select")
   .field("*")
   .table("users")
   .limit(10)
```

Figure 3.4: SQL Query

```
$("<div/>")
   .fetch("navigation.html")
   .addClass("column")
   .appendTo("#side")
```

Figure 3.5: Asynchronous Manipulation

3.2 Serial

In order to extract information from an image, many small algorithms are combined together into a processing chain. The chaining technique is therefore adapted for this domain. **Figure** 3.6 shows a sample processing chain we would like to write.

```
Image("lena.jpg")
   .togray()
   .otsu()
   .opening(8_connectivity)
   .save("lena_opening.png")
```

Figure 3.6: Sample processing chain

Common Lisp does not have the concept of object methods with the dot notation, instead it has generic functions. If we were to write the example it would resemble to **Figure** 3.7. The order of writing is reversed: the last actions are written in first. Also, the arguments of the functions are far away from the function call.

[1] http://code.google.com/p/v8cgi/wiki/API_Query

```
(save
  (opening
    (otsu
      (togray
        (load "lena.jpg")
      )
    )
  (8-connectivity))
"lena_opening.png")
```

Figure 3.7: Direct Common Lisp implementation

Instead, we can write it the traditional way using a temporary variable as seen in **Figure** 3.8. The code is written in the expected order but it too verbose. In this small example there are eight occurrences of `img` along with three `setf`.

```
(let ((img (load "lena.jpg")))
  (setf img (togray img))
  (setf img (otsu img))
  (setf img (opening img (8-connectivity)))
  (save img "lena_opening.png"))
```

Figure 3.8: Processing chain using temporary variable

We introduce in Climb a `$` macro (in reference to jQuery). It takes a sequence of actions and combine them such as the first argument is the result of the previous action. **Figure** 3.9 code can either be rewritten into **Figure** 3.7 and **Figure** 3.8. The `$` allows to write a code equivalent to the method chaining in Javascript: the opening parenthesis is just moved before the method name. Clojure[2], a lisp implementation on-top of the JVM[3], implements a similar operator named ->[4].

```
($ (load "lena.jpg")
   (togray)
   (otsu)
   (opening (8-connectivity))
   (save "lena_opening.png"))
```

Figure 3.9: Climb `$` macro

[2]http://clojure.org/
[3]Java Virtual Machine
[4]http://clojuredocs.org/clojure_core/clojure.core/-%3E

3.3 Paralell

Method chaining and consequently the `$` operator are limited to linear processes. jQuery has an `end` method[5] in order to simulate branching. Every time a method returns an element different than `this`, it is being pushed to a stack. The `end` method pops the latest element from the stack and returns the new top. **Figure 3.10** shows how to use it to set the color of `foo` into red and `bar` into green.

```
$("#id")

  .find(".foo")
    .css("color", "red")
  .end()

  .find(".bar")
    .css("color", "green")
  .end()
```

Figure 3.10: jQuery `end` example

The solution adopted by jQuery is the best possible option with traditional method chaining. Since we are building our operator through macros, we can bypass this limitation. This is why we introduce the `//` operator seen in **Figure 3.11**. In essence, it spawns a new `$` operation for each group of action. It allows to use one variable multiple times. The same version without `//` is available in **Figure 3.12**.

```
($ "#id"

  (//
    ((find ".foo")
     (css "color" "red"))

    ((find ".bar")
     (css "color" "green"))))
```

Figure 3.11: Climb equivalent with `//`

```
(let ((tmp ($ "#id")))

  ($ tmp
    (find ".foo")
    (css "color" "red"))

  ($ tmp
    (find ".bar")
    (css "color" "green")))
```

Figure 3.12: Climb equivalent without `//`

In the jQuery version, it is not possible to use the last value returned by the branches as they it be ignored by the `end` method. On the other hand, the `//` operator returns all the final values. They will be passed as arguments of the next action. **Figure 3.13** is an example that shows how to compare the result of two binarization algorithms: Sauvola and Otsu. The `diff` function will be called with two arguments, the first one being the image through Sauvola, while the second the image after Otsu.

[5] http://api.jquery.com/end/

```
($ (load "lena.png")
   (togray)
   (//
       ((sauvola) (save "sauvola.png"))
       ((otsu)    (save "otsu.png")))
   (diff)
   (save "difference.png"))
```

Figure 3.13: Combine the result of the `//` branches

3.4 Flow Control

It is possible with both `$` and `//` operators to achieve complex behaviors with few lines of code. However, it imposes a model where each returned value will be the first argument of the next action. It is sometimes undesirable.

For example, it is a common operation to time actions. Climb provides two functions `timer-start` and `timer-print`. They are not designed to be used in a chaining context. This is why we introduce the quote modifier: it calls the function the usual way. **Figure** 3.14 shows how to time actions and **Figure** 3.15 is a code with similar effect without the `$` operator.

```
($ (load "lena.png")
  '(timer-start)
   (otsu)
  '(timer-print "Otsu")
   (save "lena_otsu.png"))

; Otsu: 101ms
```

Figure 3.14: Quote modifier

```
(let ((tmp (load "lena.png")))
  (timer-start)
  (setf tmp (otsu tmp))
  (timer-print "Otsu")
  (save tmp "lena_otsu.png"))

; Otsu: 101ms
```

Figure 3.15: Expanded code

The quote modifier uses two distinct concepts:

- **Custom arguments**: All the previous return values are not automatically inserted as firsts arguments. They are available through the keywords `$1`, `$2` ...
- **Skip return values**: The return values of the action are not used for chaining. Instead, previous return values are used.

In order to get more flexibility, we also introduce the `#` modifier that only provides custom arguments. So far, it is possible with modifiers to express three of the four combinations: **Figure** 3.16. We may find use cases and introduce the last combination in the future.

		Custom arguments	
		Yes	No
Skip return values	Yes	'	*none yet*
	No	#	ø

Figure 3.16: Available modifiers

Figure 3.17 is a more complex example using the // operator as well as the ' and # modifiers.
The image is first being dispatched to the three blocks using the // operator. Each block applies a different algorithm, saves the processed image and prints the algorithm execution time using the ' modifier.
The three algorithms results are then dispatched into three blocks. Each one makes the difference between two images using the `diff` algorithm and the # modifier.

```
($ (load "lenagray.png")

   (//
     ('(timer-start)
        (otsu)
      '(timer-print "Otsu")
        (save "otsu.png")) ; $1

     ('(timer-start)
        (sauvola (box2d 1))
      '(timer-print "Sauvola 1")
        (save "sauvola1.png")) ; $2

     ('(timer-start)
        (sauvola (box2d 5))
      '(timer-print "Sauvola 5")
        (save "sauvola5.png"))) ; $3

   (//
     (#(diff $1 $2) (save "diff-otsu-sauvola1.png"))
     (#(diff $1 $3) (save "diff-otsu-sauvola5.png"))
     (#(diff $2 $3) (save "diff-sauvola1-sauvola5.png"))))

; Otsu: 101ms
; Sauvola 1: 263ms
; Sauvola 5: 3407ms
```

Figure 3.17: Comparison of three binarization techniques

The generated images are shown in **Figure 3.18**.

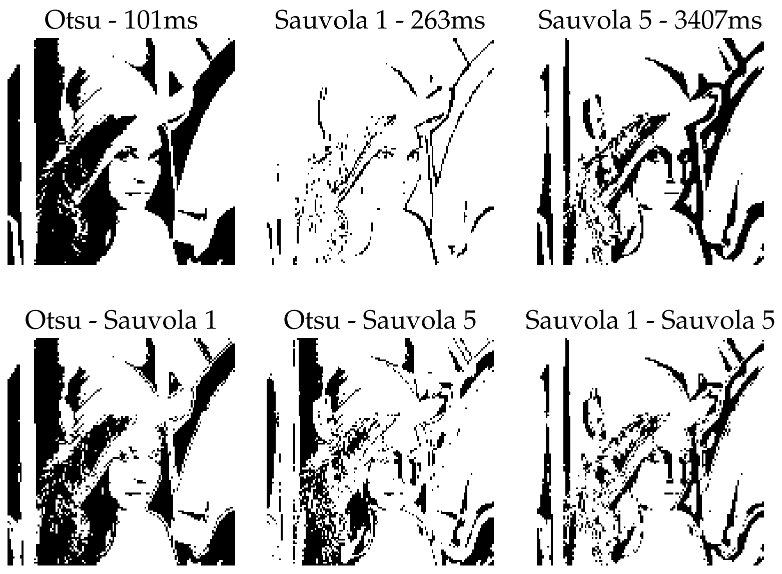

Figure 3.18: Outputs of the binarization processing chain

Chapter 4

Conclusion

We have taken two component tree algorithms from the literature and implemented them into Climb. It enlightened the parts of the library that required to be worked on. The definition of values is extended: instead of being a vector of numbers, it can now be of any type. The extensibility of site sets through a duck typing interface is used to support those algorithms. Debugging utilities are also added to improve ease of development.

In order to enhance end user ability to use our library, we introduce a design pattern called method chaining popularized by the Javascript jQuery library. As we adapted it to the Common Lisp language and to the image processing context, it was clear that the current form was too restrained. We improved it to support parallel execution as well as better flow control.

The image processing field is deeply rooted with languages such as C and C++. Being able to immerse it into a dynamic language such as Common Lisp gives us the ability to use a wide range of programming patterns that were not being considered. Climb improved with the use of lambda functions that powers morphers (Chedeau (2010)) and with the extended form of method chaining proposed in this technical report.

There are many other patterns that could potentially be taken from dynamic languages to improve the library. Future work will be focused on exploring these new possibilities. New algorithms as well as new types of images (both structures and values) will be implemented in order to strengthen our core framework and to provide solutions to real world problems.

Chapter 5

Bibliography

Berger, Ch., Géraud, Th., Levillain, R., Widynski, N., Baillard, A., and Bertin, E. (2007). Effective component tree computation with application to pattern recognition in astronomical imaging. In *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, volume 4, pages IV–41–IV–44, San Antonio, TX, USA.

Chedeau, C. (2010). Functionnal approach of image processing genericity. Technical Report 1001, EPITA Research and Development Laboratory (LRDE).

Denuzière, L. (2009). CLIMB: A dynamic approach to generic image processing. Technical Report 0906, EPITA Research and Development Laboratory (LRDE).

Diaz, D. (2010). Asynchronous method queue chaining in javascript. http://www.dustindiaz.com/async-method-queues/.

Géraud, Th. (2005). Ruminations on Tarjan's Union-Find algorithm and connected operators. volume 30 of *Computational Imaging and Vision*, pages 105–116, Paris, France. Springer.

Gonzalez, R., Woods, R., and Eddins, S. (2004). *Digital Image Processing Using MATLAB(R)*. Prentice Hall.

Harmes, R. and Diaz, D. (2007). *Pro JavaScript Design Patterns (Recipes: a Problem-Solution Ap)*. Apress, 1 edition.

Martelli, A. (2000). http://groups.google.com/group/comp.lang.python/msg/e230ca916be58835.

Najman, L. and Couprie, M. (2004). Quasi-linear algorithm for the component tree. In *IS&T/SPIE Symposium on Electronic Imaging, In Vision Geometry XII*, pages 18–22.