*Proceedings of ELS 2012*

# 5th European Lisp Symposium

## Interoperability: Systems, Libraries, Workflows

**April 30 – May1 2012**
**University of Zadar, Zadar, Croatia**




# Contents

# Organization

## Programme Chair

Marco Antoniotti, Universita Milano Bicocca, ITALY

## Local Chair

Franjo Pehar, University of Zadar, CROATIA
Damir Kero, University of Zadar, CROATIA

## Programme Committee

Giuseppe Attardi, Università degli Studi di Pisa, Pisa, ITALY
Pascal Costanza, Intel, Bruxelles, BELGIUM
Damir Ćavar, Eastern Michigan University, USA
Marc Feeley, Université de Montreal, Montreal, CANADA
Scott McKay, Google, U.S.A.
Kent Pitman, Hypermeta Inc., U.S.A.
Christophe Rhodes, Department of Computing, Goldsmiths, University of London, London, UNITED KINGDOM
Robert Strandh, LABRI, Université de Bordeaux, Bordaux, FRANCE
Didier Verna, EPITA / LRDE, FRANCE
Taiichi Yuasa, Kyoto University, JAPAN

# About ELS2012

The purpose of the European Lisp Symposium is to provide a forum for the discussion and dissemination of all aspects of design, implementation and application of any of the Lisp and Lisp-inspired dialects, including Common Lisp, Scheme, Emacs Lisp, AutoLisp, ISLISP, Dylan, Clojure, ACL2, ECMAScript, Racket, SKILL, and so on. We encourage everyone interested in Lisp to participate.

The main theme of the 2012 European Lisp Conference is “Interoperability: Systems, Libraries, Workflows”. Lisp based and functional-languages based systems have grown a variety of solutions to become more and more integrated with the wider world of Information and Communication Technologies in current use. There are several dimensions to the scope of the solutions proposed, ranging from “embedding” of interpreters in C-based systems, to the development of abstractions levels that facilitate the expression of complex context dependent tasks, to the construction of exchange formats handling libraries, to the construction of theorem-provers for the “Semantic Web”. The European Lisp Symposium 2012 solicits the submission of papers with this specific theme in mind, alongside the more traditional tracks which have appeared in the past editions.

# Sponsors

We would like to thank our sponsors for making the event possible.




# Conference schedule

| | |
|---|---|
| **Sunday April 29.** | |
| 11:00 | Sea & Sun & Coffee (meeting place: The Sea organ & Greeting to the Sun) |
| 19:30 – 21:30 | 19:30 – 21:30 Welcome Cocktail at the Maraschino Bar |
| **Monday April 30.** | |
| 8:30 – 09:00 | Registration |
| 09:30 – 10:00 | Welcome |
| 10:00 – 11:00 | Juan Jose Garcia-Ripoll: *Embeddable Common Lisp* |
| 11:00 – 11:30 | Coffee Break |
| 11:30 – 13:00 | Session 1 |
| 11:30 – 12:15 | Laurent Senta, Christopher Chedeau and Didier Verna: *Generic Image Processing with Climb* |
| 12:15 – 13:00 | Giovanni Anzani: *An iterative method to solve overdetermined systems of nonlinear equations applied to the restitution of planimetric measurements* |
| 13:00 – 15:00 | Lunch Break |
| 15:00 – 17:30 | Session 2 |
| 15:00 – 15:45 | Alessio Stalla: *Java interop with ABCL, a practical example* |
| 15:45 – 16:30 | Nils Bertschinger: *Embedded probabilistic programming in Clojure* |
| 16:30 – 17:30 | Pascal Costanza, *A little history of metaprogramming and reflection* |
| 17:30 – 18:00 | Lightening Talk (1)<br>Zoltan Varju, Richard Littauer and Peteris Ernis: *Using Clojure in Linguistic Computing* |
| 18:00 – 20:00 | Buffet at the University of Zadar |
| **Tuesday May 1.** | |
| 10:00 – 11:00 | Ernst van Waning. *Aura (Automated User-centered Reasoning and Acquisition System)*. |
| 11:00 – 11:30 | Coffee Break |
| 11:30 – 13:00 | Session 3 |
| 11:30 – 12:15 | Marco Benelli: *Scheme in Industrial Automation* |
| 12:15 – 13:00 | Gunnar Völkel, Johann M. Kraus and Hans A. Kestler: *Algorithm Engineering with Clojure* |
| 13:00 – 15:00 | Lunch Break |
| 15:00 – 16:30 | Session 4 |
| 15:00 – 15:45 | Irène Anne Durand: *Object enumeration* |
| 15:45 – 16:30 | Alessio Stalla: *Doplus, the high-level, Lispy, extensible iteration construct* |
| 16:30 – 17:00 | Lightening Talk (2)<br>Mikhail Raskin: *QueryFS, a virtual filesystem based on queries, and related tools* |
| 17:00 – 18:00 | Announcements & Wrap-up |
| 20:00 – 22:00 | Dinner |

# Invited Talks

## Embeddable Common Lisp

*Juan Jose Garcia-Ripoll*

This talk presents an implementation of the Common Lisp language that is built as a statically or dynamically loadable C library, ready to be used in third party applications as an utility language (for instance, to run Maxima in the Sage environment), but also fully functional as the core of arbitrarily complex standalone applications. This paper discusses the history of this implementation, starting with its roots in the ECoLisp, KCL and AKCL programmes, and the difficulties faced in upgrading this software to make it fully standalone, self-bootstrapable, thread- and async-interrupt safe and capable of interoperating with modern operating systems (Windows, Unix, iOS, etc).

## Aura (Automated User-centered Reasoning and Acquisition System)

*Ernst van Waning*

This talk will be about Aura (Automated User-centered Reasoning and Acquisition System), a step towards an application containing large volumes of scientific knowledge and capable of applying sophisticated problem-solving methods to answer novel questions: the Digital Aristotle. The current focus of AURA is to capture a significant fraction of a Biology textbook and embed the resulting knowledge base into an Intelligent Electronic Textbook reader that could be used by teachers and students.
The talk will further focus on work done to improve Aura's performance, its stability and its documentation.

# Session I

# Generic Image Processing With Climb

Laurent Senta
senta@lrde.epita.fr

Christopher Chedeau
christopher.chedeau@lrde.epita.fr

Didier Verna
didier.verna@lrde.epita.fr

Epita Research and Development Laboratory
14-16 rue Voltaire, 94276 Le Kremlin-Bicêtre
Paris, France

## ABSTRACT

We present Climb, an experimental generic image processing library written in Common Lisp. Most image processing libraries are developed in static languages such as C or C++ (often for performance reasons). The motivation behind Climb is to provide an alternative view of the same domain, from the perspective of dynamic languages. More precisely, the main goal of Climb is to explore the dynamic way(s) of addressing the question of genericity, while applying the research to a concrete domain. Although still a prototype, Climb already features several levels of genericity and ships with a set of built-in algorithms as well as means to combine them.

## Categories and Subject Descriptors

I.4.0 [**Image Processing And Computer Vision**]: General—*image processing software*; D.2.11 [**Software Engineering**]: Software Architectures—*Data abstraction, Domain-specific architectures*

## General Terms

Design, Languages

## Keywords

Generic Image Processing, Common Lisp

## 1. INTRODUCTION

Climb is a generic image processing library written in Common Lisp. It comes with a set of built-in algorithms such as erosion, dilation and other mathematical morphology operators, thresholding and image segmentation.

The idea behind genericity is to be able to write algorithms only once, independently from the data types to which they will be applied. In the Image Processing domain, being fully generic means being independent from the image formats, pixels types, storage schemes (arrays, matrices, graphs) *etc.* Such level of genericity, however, may induce an important performance cost.

In order to reconcile genericity and performance, the LRDE[1] has developed an Image Processing platform called Olena[2]. Olena is written in C++ and uses its templating system abundantly. Olena is a ten years old project, well-proven both in terms of usability, performance and genericity [2].

In this context, the goal of Climb is to use this legacy for proposing another vision of the same domain, only based on a dynamic language. Common Lisp provides the necessary flexibility and extensibility to let us consider several alternate implementations of the model provided by Olena from the dynamic perspective.

First, we provide a survey of the algorithms readily available in Climb. Next, we present the tools available for composing basic algorithms into more complex Image Processing chains. Finally we describe the generic building blocks used to create new algorithms, hereby extending the library.

## 2. FEATURES

Climb provides a basic image type with the necessary loading and saving operations, based on the lisp-magick library. This allows for direct manipulation of many images formats, such as PNG and JPEG. The basic operations provided by Climb are loading an image, applying algorithms to it and saving it back. In this section, we provide a survey of the algorithms already built in the library.

### 2.1 Histograms

A histogram is a 2D representation of image information. The horizontal axis represents tonal values, from 0 to 255 in common grayscale images. For every tonal value, the vertical axis counts the number of such pixels in the image. Some Image Processing algorithms may be improved when the information provided by a histogram is known.

*Histogram Equalization.* An image (*e.g.* with a low contrast) does not necessarily contain the full spectrum of intensity. In that case, its histogram will only occupy a narrow band on the horizontal axis. Concretely, this means that some level of detail may be hidden to the human eye.

[1]EPITA Research and development laboratory
[2]`http://olena.lrde.epita.fr`

Histogram equalization identifies how the pixel values are spread and transforms the image in order to use the full range of gray, making details more apparent.

*Plateau Equalization.* Plateau equalization is used to reveal the darkest details of an image, that would not otherwise be visible. It is a histogram equalization algorithm applied on a distribution of a clipped histogram. The pixel values greater than a given threshold are set to this threshold while the other ones are left untouched.

## 2.2 Threshold

Thresholding is a basic binarization algorithm that converts a grayscale image to a binary one. This class of algorithms works by comparing each value from the original image to a threshold. Values beneath the threshold are set to `False` in the resulting image, while the others are set to `True`.

Climb provides 3 threshold algorithms:

*Basic.* Apply the algorithm using a user-defined threshold on the whole image.

*Otsu.* Automatically find the best threshold value using the image histogram. The algorithm picks the value that minimizes the spreads between the `True` and `False` domains as described by Nobuyuki Otsu [4].

*Sauvola.* In images with a lot of variation, global thresholding is not accurate anymore, so adaptive thresholding must be used. The Sauvola thresholding algorithm [5] picks the best threshold value for each pixel from the given image using its neighbors.

## 2.3 Watershed

Watershed is another class of image segmentation algorithms. Used on a grayscale image seen as a topographic relief, a watershed extracts the different basins within the image. Watershed algorithms produce a segmented image where each component is labeled with a different tag.

Climb provides a implementation of Meyer's flooding algorithm [3]. Components are initialized at the "lowest" region of the image (the darkest ones). Then, each component is extended until it collides with another one, following the image's topography.

## 2.4 Mathematical Morphology

Mathematical Morphology is heavily used in Image Processing. It defines a set of operators that can be cumulated in order to extract specific details from images.

Mathematical Morphology algorithms are based on the application of a structuring element to every points of an image, gathering information using basic operators (*e.g.* logical operators `and`, `or`, *etc.*) [6].

*Erosion and Dilation.* Applied on a binary image, these algorithms respectively erodes and dilate the `True` areas in the picture. Combining these operators leads to new algorithms:

**Opening** Apply an erosion then a dilation. This removes the small details from the image and opens the `True` shapes.

**Closing** Apply a dilation then an erosion. This closes the `True` shapes.

Mathematical Morphology can also be applied on grayscale images, leading to news algorithms:

**TopHat** Sharpen details

**Laplacian operator** Extract edges

**Hit-Or-Miss** Detect specific patterns

# 3. COMPOSITION TOOLS

Composing algorithms like the ones described in the previous section is interesting for producing more complex image processing chains. Therefore, in addition to the built-in algorithms that Climb already provides, a composition infrastructure is available.

## 3.1 Chaining Operators

### *3.1.1 The $ operator*

The most essential form of composition is the sequential one: algorithms are applied to an image, one after the other.

Programming a chain of algorithms by hand requires a lot of variables for intermediate storage of temporary images. This renders the code cumbersome to read an maintain.

To facilitate the writing of such a chain, we therefore provide a chaining operator which automates the plugging of an algorithm's output to the input of the next one. Such an operator would be written as follows in traditional Lisp syntax:

```
(image-save
  (algo2
    (algo1
      (image-load "image.png")
      param1)
    param2)
  "output.png")
```

Unfortunately, this syntax is not optimal. The algorithms must be read in reverse order, and arguments end up far away from the function to which they are passed.

Inspired by Clojure's Trush operator [3] and the JQuery[4] (->) chaining process, Climb provides the `$` macro which allows to chain operations in a much clearer form.

[3] `http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/->`
[4] `http://api.jquery.com/jQuery/`

```
$((image-load "image.png")
  (algo1 param1)
  (algo2 param2)
  (image-save "output.png"))
```

### 3.1.2 Flow Control

In order to ease the writing of a processing chain, the `$` macro is equipped with several other flow control operators.

- The `//` operator splits the chain into several independent subchains that may be executed in parallel.
- The quote modifier (`'`) allows to execute a function outside the actual processing chain. This is useful for doing side effects (see the example below).
- The `$1`, `$2`, *etc.* variables store the intermediate results from the previous executions, allowing to use them explicitly as arguments in the subsequent calls in the chain.
- The `#` modifier may be used to disrupt the implicit chaining and let the programmer use the `$1`, *etc.* variables explicitly.

These operators allow for powerful chaining schemes, as illustrated below:

```
($ (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"))))
```

## 3.2 Morphers

As we saw previously, many Image Processing algorithms result in images of a different type than the original image (for instance the threshold of a grayscale image is a Boolean one). What's more, several algorithms can only be applied on images of a specific type (for instance, watershed may only be applied to grayscale images).

Programming a chain of algorithms by hand requires a lot of explicit conversion from one type to another. This renders the code cumbersome to read an maintain.

To facilitate the writing of such a chain, we therefore provide an extension to the morpher concept introduced in Olena with SCOOP2 [1] generalized to any object with a defined communication protocol. Morphers are wrappers created around an object to modify how it is viewed from the outside world.

Two main categories of morphers are implemented: *value morphers* and *content morphers*.

*Value morpher.* A value morpher operates a dynamic translation from one value type to another. For instance, an RGB image can be used as a grayscale one when seen through a morpher that returns the pixels intensity instead of their original color. It is therefore possible to apply a watershed algorithm on a colored image in a transparent fashion, without actually transforming the original image into a grayscale one.

*Content morpher.* A content morpher operates a dynamic translation from one image structure to another. For instance, a small image can be used as a bigger one when seen through a morpher that returns a default value (*e.g.* black), when accessing coordinates outside the original image domain. It is therefore possible to apply an algorithm built for images with a specific size (*e.g.* power of two), without having to extend the original image.

Possible uses for content morphers include *restriction* (parts of the structures being ignored), *addition* (multiple structures being combined) and *reordering* (structures order being modified).

# 4. EXTENSIBILITY

The pixels of an image are usually represented aligned on a regular 2D grid. In this case, the term "pixel" can be interpreted in two ways, the position and the value, which we need to clearly separate. Besides, digital images are not necessarily represented on a 2D grid. Values can be placed on hexagonal grids, 3D grids, graphs *etc.* In order to be sufficiently generic, we hereby define an image as a function from a position (a *site*) to a value: $img(site) = value$.

A truly generic algorithm should not depend on the underlying structure of the image. Instead it should rely on higher level concepts such as neighborhoods, iterators, *etc.* More precisely, we have identified 3 different levels of genericity, as explained below.

## 4.1 Genericity on values

Genericity on values is based on the CLOS dispatch. Using multimethods, generic operators like comparison and addition are built for each value type, such as RGB and Grayscale.

## 4.2 Genericity on structures

The notion of "site" provides an abstract description for any kind of position within an image. For instance, a site might be an n-uplet for an N-d image, a node within a graph, *etc.*

Climb also provides a way to represent the regions of an image through the site-set object. This object is used in order to browse a set of coordinates and becomes particularly handy when dealing with neighborhoods. For instance, the set of nearest neighbors, for a given site, would be a list of coordinates around a point for an N-d image, or the connected nodes in a graph.

## 4.3 Genericity on implementations

Both of the kinds of genericity described previously allow algorithm implementers to produce a single program that works with any data type. Some algorithms, however, may be implemented in different ways when additional information on the underlying image structure is available. For instance, iterators may be implemented more efficiently when we know image contents are stored in an array. We can also distinguish the cases where a 2D image is stored as a matrix or a 1D array of pixels *etc.*

Climb provides a property description and filtering system that gives a finer control over the algorithm selection. This is done by assigning properties such as `:dimension` = `:1D`, `:2D`, *etc.* to images. When defining functions with the `defalgo` macro, these properties are handled by the dispatch system. The appropriate version of a function is selected at runtime, depending on the properties implemented by the processed image.

# 5. CONCLUSION

Climb is a generic Image Processing library written in Common Lisp. It is currently architected as two layers targeting two different kinds of users.

- For an image processing practitioner, Climb provides a set of built-in algorithms, composition tools such as the chaining operator and morphers to simplify the matching of processed data and algorithms IO.
- For an algorithm implementer, Climb provides a very high-level domain model articulated around three levels of abstraction. Genericity on values (RGB, Grayscale *etc.*), genericity on structures (sites, site sets) and genericity on implementations thanks to properties.

The level of abstraction provided by the library makes it possible to implement algorithms without any prior knowledge on the images to which they will be applied. Conversely supporting a new image types should not have any impact on the already implemented algorithms.

From our experience, it appears that Common Lisp is very well suited to highly abstract designs. The level of genericity attained in Climb is made considerably easier by Lisp's reflexivity and CLOS' extensibility. Thanks to the macro system, this level of abstraction can still be accessed in a relatively simple way by providing domain-specific extensions (*e.g.* the chaining operators).

Although the current implementation is considered to be reasonably stable, Climb is still a very young project and should be regarded as a prototype. In particular, nothing has been done in terms of performance yet, as our main focus was to design a very generic and expressive API. Future work will focus on:

**Genericity** By providing new data types like graphs, and enhancing the current property system.

**Usability** By extending the `$` macro with support for automatic insertion of required morphers and offering a GUI for visual programming of complex Image Processing chains.

**Performance** By improving the state of the current implementation, exploring property-based algorithm optimization and using the compile-time facilities offered by the language.

Climb is primarily meant to be an experimental platform for exploring various generic paradigms from the dynamic languages perspective. In the future however, we also hope to reach a level of usability that will trigger interest amongst Image Processing practitioners.

# 6. REFERENCES


[1] Th. Géraud and R. Levillain. Semantics-driven genericity: A sequel to the static C++ object-oriented programming paradigm (SCOOP 2). In *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL)*, Paphos, Cyprus, July 2008.

[2] 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.

[3] F. Meyer. *Un algorithme optimal pour la ligne de partage des eaux*, pages 847–857. AFCET, 1991.

[4] N. Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man and Cybernetics*, 9(1):62–66, Jan. 1979.

[5] J. Sauvola and M. Pietikäinen. Adaptive document image binarization. *Pattern Recognition*, 33(2):225–236, 2000.

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

# An iterative method to solve overdetermined systems of nonlinear equations applied to the restitution of planimetric measurements.

Giovanni Anzani
Università degli Studi di Firenze - Dipartimento di Architettura: Disegno, Storia, Progetto
Via S. Niccolo', 89/a-95 - 50125 Firenze (FI)
Mob.: +39/339/7056663
giovanni.anzani@unifi.it

## ABSTRACT

In the first part of this article, we describe the conditions for the analytical solution of overdetermined systems of nonlinear equations in the case of circumferences. The application of the theory in the development of AutoLISP procedures, described in the second part of this article, allows to evaluate and solve some problems in the field of computerized restitution of planimetric measurements. This application considerably CAD graphics extensively, allowing you to appreciate the interoperability between AutoCAD and AutoLISP.

## Categories and Subject Descriptors
Programming Languages: AutoLISP

## General Terms
Algorithms, Measurement, Experimentation.

## INTRODUCTION

This contribution concerns the realization of an algorithm in AutoLISP, adapted to determine analytically the position of a point, knowing its distances from a series of points with known coordinates, and then using AutoCAD to draw the data of the problem and the identified point. The command for AutoCAD that we have designed finds its concrete use as part of the architectural surveys. The following contribution will deal at first with the analytical approach of the problem[1], followed by a description of procedures and commands implemented by AutoLISP.

## ANALYTICAL STATEMENT OF THE PROBLEM

Given the equations of n circumferences (1.1), that one can derive given a set of known points (such as centers of these circumferences) and their distances from an unknown point (such as radii of these circles), we want to determine with the best approximation the unknown point (such as their point of intersection).

$$(1.1)\quad \begin{cases} f_1(x,y) = x^2 + y^2 + a_1 \cdot x + b_1 \cdot y + c_1 \\ f_2(x,y) = x^2 + y^2 + a_2 \cdot x + b_2 \cdot y + c_2 \\ \cdots \\ f_n(x,y) = x^2 + y^2 + a_n \cdot x + b_n \cdot y + c_n \end{cases}$$

We write the Jacobian matrix (1.2), whose rows are the partial derivatives of the equations of the circumferences:

$$(1.2)\qquad J_{f(x,y)} = \begin{bmatrix} 2\cdot x + a_1 & 2\cdot y + b_1 \\ 2\cdot x + a_2 & 2\cdot y + b_2 \\ \cdots & \cdots \\ 2\cdot x + a_n & 2\cdot y + b_n \end{bmatrix} \qquad where:$$

$$\begin{cases} f_{1\,x}^{'} = 2\cdot x + a_1 \\ f_{2\,x}^{'} = 2\cdot x + a_2 \\ \cdots \\ f_{n\,x}^{'} = 2\cdot x + a_n \end{cases} \qquad and: \begin{cases} f_{1\,y}^{'} = 2\cdot y + b_1 \\ f_{2\,y}^{'} = 2\cdot y + b_2 \\ \cdots \\ f_{1\,y}^{'} = 2\cdot y + b_n \end{cases}$$

Given an initial point $P_{(0)}$ as a solution from which you wish to start the search of the solution, and $P_{(1)}$ as a point that you want to determine in order to optimize the solution of the problem as to the point $P_{(0)}$, you can consider the vector $P_{(1)}$ - $P_{(0)}$ written in (1.3).

$$(1.3)\qquad P_{(1)} - P_{(0)} = \begin{bmatrix} x_1 - x_0 \\ y_1 - y_0 \end{bmatrix} \qquad where:$$

$$P_{(0)} = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \qquad and: \quad P_{(1)} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

Substitute the coordinates of the point $P_{(0)}$ inside the equations (1.1) of the circumferences and in the jacobian matrix (1.2), thus obtaining respectively:

[1] Many thanks to Prof. Giuseppe Conti for his valuable advices during the definition and the revision of this paper.

$$(1.4)\quad \begin{cases} f_1(P_{(0)}) = x_0^2 + y_0^2 + a_1 \cdot x_0 + b_1 \cdot y_0 + c_1 \\ f_2(P_{(0)}) = x_0^2 + y_0^2 + a_2 \cdot x_0 + b_2 \cdot y_0 + c_2 \\ \cdots \\ f_n(P_{(0)}) = x_0^2 + y_0^2 + a_n \cdot x_0 + b_n \cdot y_0 + c_n \end{cases}$$

$$(1.5)\quad J_{f(P_{(0)})} = \begin{bmatrix} 2 \cdot x_0 + a_1 & 2 \cdot y_0 + b_1 \\ 2 \cdot x_0 + a_2 & 2 \cdot y_0 + b_2 \\ \cdots & \cdots \\ 2 \cdot x_0 + a_n & 2 \cdot y_0 + b_n \end{bmatrix}$$

Multiply the rows of matrix (1.5) by the column of the vector (1.3), obtaining the following vector (1.6):

$$(1.6)\quad \begin{bmatrix} (2 \cdot x_0 + a_1)(x_1 - x_0) + (2 \cdot y_0 + b_1)(y_1 - y_0) \\ (2 \cdot x_0 + a_2)(x_1 - x_0) + (2 \cdot y_0 + b_2)(y_1 - y_0) \\ \vdots \\ (2 \cdot x_0 + a_n)(x_1 - x_0) + (2 \cdot y_0 + b_n)(y_1 - y_0) \end{bmatrix}$$

Add the rows of (1.4) to the rows of (1.6), then set them equal to 0, obtaining the following system of equations (1.7).

$$(1.7)\quad \begin{cases} \left(x_0^2 + y_0^2 + a_1 \cdot x_0 + b_1 \cdot y_0 + c_1\right) + \\ \quad +(2 \cdot x_0 + a_1)(x_1 - x_0) + \\ \quad +(2 \cdot y_0 + b_1)(y_1 - y_0) \;=\; 0 \\ \left(x_0^2 + y_0^2 + a_2 \cdot x_0 + b_2 \cdot y_0 + c_2\right) + \\ \quad +(2 \cdot x_0 + a_2)(x_1 - x_0) + \\ \quad +(2 \cdot y_0 + b_2)(y_1 - y_0) \;=\; 0 \\ \vdots \\ \left(x_0^2 + y_0^2 + a_n \cdot x_0 + b_n \cdot y_0 + c_n\right) + \\ \quad +(2 \cdot x_0 + a_n)(x_1 - x_0) + \\ \quad +(2 \cdot y_0 + b_n)(y_1 - y_0) \;=\; 0 \end{cases}$$

Carrying out the calculations, from the equations in (1.7) we get the equivalent (1.8) following system:

$$(1.8)\quad \begin{cases} (2 \cdot x_0 + a_1) \cdot x_1 + (2 \cdot y_0 + b_1) \cdot y_1 = x_0^2 + y_0^2 - c_1 \\ (2 \cdot x_0 + a_2) \cdot x_1 + (2 \cdot y_0 + b_2) \cdot y_1 = x_0^2 + y_0^2 - c_2 \\ \vdots \\ (2 \cdot x_0 + a_n) \cdot x_1 + (2 \cdot y_0 + b_n) \cdot y_1 = x_0^2 + y_0^2 - c_n \end{cases}$$

Now let's write the system of equations (1.8) in matrix form highlighting a matrix $A_{nx2}$ of the coefficients, a vector $X_{2x1}$ of unknowns and a vector $B_{nx1}$ of the known terms. We will indicate the matrix equation that we have obtained (1.9a) and (1.9b) in the following way :

$$(1.9a)\qquad A_{n\times 2} \cdot X_{2\times 1} = B_{n\times 1}$$

$$(1.9b)\quad \begin{bmatrix} 2 \cdot x_0 + a_1 & 2 \cdot y_0 + b_1 \\ 2 \cdot x_0 + a_2 & 2 \cdot y_0 + b_2 \\ \vdots & \vdots \\ 2 \cdot x_0 + a_n & 2 \cdot y_0 + b_n \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} x_0^2 + y_0^2 - c_1 \\ x_0^2 + y_0^2 - c_2 \\ \vdots \\ x_0^2 + y_0^2 - c_n \end{bmatrix}$$

Determine (1.10) the transposed matrix $A^{T}_{nx2}$ of the matrix $A_{nx2}$ of the coefficients expressed in (1.9):

$$(1.10)\quad A^T_{n\times 2} = \begin{bmatrix} 2 \cdot x_0 + a_1 & 2 \cdot x_0 + a_2 & \cdots & 2 \cdot x_0 + a_n \\ 2 \cdot y_0 + b_1 & 2 \cdot y_0 + b_2 & \cdots & 2 \cdot y_0 + b_n \end{bmatrix}$$

Multiply both members of the matrix equation (1.9) by the transposed matrix $A^{T}_{nx2}$ present in (1.10), having as a result a new matrix equation (1.11) that we indicate in the following way:

$$(1.11)\quad H_{2\times 2} \cdot X_{2\times 1} = K_{2\times 1}$$

$$where: \begin{cases} H_{2\times 2} = A^T_{2\times n} \cdot A_{n\times 2} \\ K_{2\times 1} = A^T_{2\times n} \cdot B_{n\times 1} \end{cases}$$

If we properly carry out the calculations expressed in matrix form in (1.11), we will obtain the matrix $H_{2x2}$ of the coefficients and the vector $K_{2x1}$ of the known terms, the solution of such a system will provide us with the coordinates of the Point $P_{(1)}$ with a best approximation as regards the point $P_{(0)}$ from which we started. Let's consider (1.12) and (1.13) for convenience:

$$(1.12)\quad H_{2\times 2} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \qquad where: \begin{cases} h_{21} = h_{12} \end{cases}$$

$$(1.13)\quad K_{2\times 1} = \begin{bmatrix} k_{11} \\ k_{12} \end{bmatrix}$$

With simple steps, although laborious, we obtain the elements of the two matrices whose final values are shown below (1.14) and (1.15):

$$
(1.14)\quad \begin{cases} h_{11} = 4\cdot n\cdot x_0^2 + 4\cdot x_0\cdot\left(\sum_{i=1}^{n} a_i\right)+\left(\sum_{i=1}^{n} a_i^2\right) \\ h_{12} = 4\cdot n\cdot x_0\cdot y_0 + 2\cdot y_0\cdot\left(\sum_{i=1}^{n} a_i\right)+ \\ \quad +2\cdot x_0\cdot\left(\sum_{i=1}^{n} b_i\right)+\left(\sum_{i=1}^{n} a_i\cdot b_i\right) \\ h_{22} = 4\cdot n\cdot y_0^2 + 4\cdot y_0\cdot\left(\sum_{i=1}^{n} b_i\right)+\left(\sum_{i=1}^{n} b_i^2\right) \end{cases}
$$

$$
(1.15)\quad \begin{cases} k_{11} = \left[2\cdot n\cdot x_0+\left(\sum_{i=1}^{n} a_i\right)\right]\cdot\left(x_0^2+y_0^2\right)+ \\ \quad -\left(\sum_{i=1}^{n} a_i\cdot c_i\right)-2\cdot x_0\cdot\left(\sum_{i=1}^{n} c_i\right) \\ k_{12} = \left[2\cdot n\cdot y_0+\left(\sum_{i=1}^{n} b_i\right)\right]\cdot\left(x_0^2+y_0^2\right)+ \\ \quad -\left(\sum_{i=1}^{n} b_i\cdot c_i\right)-2\cdot y_0\cdot\left(\sum_{i=1}^{n} c_i\right) \end{cases}
$$

By means of an iterative method, we continue searching for a point $P_{(n)}$ better than the previous point $P_{(n-1)}$ until the distance between $P_{(n)}$ and $P_{(n-1)}$ converges to less than some predetermined value that is given as a function of the radii of the starting circumferences. This value, for the application at issue, could for example be set as one-millionth of the smaller radius between the radii of the circumferences provided at the beginning.

As a further test, for each point $P_{(n)}$ obtained during the cycle, we have to determine, through the standard square deviation, the distance of the point itself from the provided circumferences, in order to verify the continuous optimization of the searched point[2]. This check is required because, contrary to what one might expect, the algorithm we have realized does not always converge towards the final point. In fact, starting from the initial point, at first, it sometimes seems to move away from the solution of the absolute minimum, by means of the iteration.

Concerning the choice of the initial point $P_{(0)}$, a clarification is required: in certain cases the iterative algorithm used to determine the point $P_{(n)}$ may not converge towards the point of absolute minimum, but towards a point of relative minimum. For this reason we will have to repeat this procedure identifying in a suitable way a set of Points $P_{(0)}$ from which we could derive a corresponding series of points $P_{(n)}$. Once you have verified for each of them, through the standard square deviation, the distance of the point from the provided circumferences, you will choose the one that seems to have the minimum value.

[2] For similar problems concerning the method of approximation by the standard deviation, see R. Corazzi, G. Conti, *Il segreto della Cupola del Brunelleschi a Firenze*, Angelo Pontecorboli Editore, Firenze 2011; where a similar method has been developed by the author and Prof. Giuseppe Conti, in order to obtain the best analytical description of various profiles of the dome.

## PROCEDURES AND COMMANDS REALIZED

The algorithms conversion in AutoLISP, referring to the analytical problem described above, required the definition of 29 functions and three commands. The algorithms that we have realized have been grouped, for convenience, into 5 main areas, whose description and list of the algorithms contained will be given in the following treatment.

Setting procedures of the AutoCAD variables. In managing the interoperability between AutoCAD and AutoLISP, that is to say when you need to transform calculations in drawings, it is necessary to manage the configuration of the design environment, making the setting of some environment variables: of the "*clayer*" or better the current "*layer*" where it is possible to insert the graphical entities that will be drawn, of the "*object snap*" active during the drawing, of the display mode "*pdmode*" of the selected points as input by the operator and the display mode "*cmdecho*" of the requests of input, displayed on the command line, by the AutoCAD commands executed automatically by the procedures in AutoLISP.

| | | |
|---|---|---|
| (LSET "layer") | → | "layer" |
| (LSETP) | → | "previous layer" |
| (VSET-OS-0) | → | 0 |
| (VSET-OS-P) | → | 47 |
| (VSET-CE-0) | → | 0 |
| (VSET-CE-P) | → | 1 |
| (VSET-PM-34) | → | 34 |
| (VSET-PM-P) | → | 0 |
| (VSET-3i "8") | → | (0 0 "8") |
| (VSET-3f) | → | ("8" 47 1) |

Design procedures and data input. For better interoperability between AutoCAD and AutoLISP, some procedures have been defined specifically optimized to draw the following graphic entities: *"pline"*, *"spline"*, *"circle"*, *"point"*. The main advantage of the choice consists in the possibility of devising procedures that, besides the specific task of the creation of the graphical entity, could control the incoming and outgoing setting of the environment variables mentioned in the previous paragraph. Further advantages consist in the possibility of creating graphical functions able to receive as input the input lists of the points to be processed without solution of continuity and in return as output of the name of the drawn graphical entity.

| | | |
|---|---|---|
| (DPLINE | '((11 12) (21 22) (31 32)) "layer" T) | |
| → | <Entity name: 7ffff606c10> | |
| (DSPLINE | '((11 12) (21 22) (31 32)) "layer" T) | |
| → | < Entity name: 7ffff606c20> | |

(DCIRCLE '(11 12) 4.0 "layer")
→ < Entity name: 7ffff606c40>

(DPOINT '(11 12) "layer")
→ < Entity name: 7ffff606c50>

In this group a procedure for the collection of input data has been inserted, such as the positions of the known points and the respective distances from the point to be determined. This procedure has got as a feedback graphic of the data gradually inserted the drawing of a circumference having as its center the known point and as a radius the corresponding distances given. In this way the operator can provide the data while he is visualizing at the same time the corresponding design and any numerical values given.

(get_nP_nr) → Number of known points: 3

→ Specify the 1st point: (15.9089 5.28831)

→ Specify the 1st measure: 11.9402

→ Specify the 2d point: (34.4962 4.87184)

→ Specify the 2d measure: 14.1044

→ Specify the 3d point: (11.7316 29.05)

→ Specify the 3d measure: 19.7646

→ (((15.9089 5.28831) (34.4962 4.87184) (11.7316 29.05)) (11.9402 14.1044 19.7646) )

Procedures on matrices. Using AutoLISP you can execute operations on matrices defining them as lists of lists; based on this assumption some procedures have been realized to perform: the standard/ horizontal/vertical transposition of a matrix, the extraction/deletion/addition of a row vector or column row from a matrix, the division by a given value of some row vector elements the determination of the solution of a matrix equation according to the method of Gaussian elimination.

(MA-TRA-S '((11 12 13) (21 22 23) (31 32 33)))
→ ((11 21 31) (12 22 32) (13 23 33))

(MA-TRA-O '((11 12 13) (21 22 23) (31 32 33)))
→ ((31 32 33) (21 22 23) (11 12 13))

(MA-TRA-V '((11 12 13) (21 22 23) (31 32 33)))
→ ((13 12 11) (23 22 21) (33 32 31))

(MA-GET-R '((11 12 13) (21 22 23) (31 32 33)) 1)
→ ((21 22 23))

(MA-GET-C '((11 12 13) (21 22 23) (31 32 33)) 1)
→ ((12) (22) (32))

(MA-DEL-R '((11 12 13) (21 22 23) (31 32 33)) 1)
→ ((11 12 13) (31 32 33))

(MA-DEL-C '((11 12 13) (21 22 23) (31 32 33)) 1)
→ ((11 13) (21 23) (31 33))

(MA-APP-C '((11 12) (21 22) (31 32)) '((13) (23) (33)))
→ ((11 12 13) (21 22 23) (31 32 33))

(MA-DIV-R 2.0 '(11 12 13))
→ (5.5 6.0 6.5)

(MA-GAUSS '((1 3)(5 4)) '((1)(2)))
→ ((0.181818) (0.272727))

Specific procedures of calculation. In this group we find four calculation functions specifically designed for the problem we are discussing about. They are able to determine the coefficients of the equations of a circumference as a function of the coordinates of its centre and the extent of its radius, to calculate the standard deviation of the distance of a generic point of a series of circumferences defined respectively by their centres and by their radii, to calculate the summations shown in (1.14) and (1.15), to determine the coefficients (1.12) and (1.13) of the matrices of the matrix equation (1.11) and calculate their solution.

(coeff_cer_Cr '(11.0 12.0) 4.0)
→ (-22.0 -24.0 249.0)

(calc_sqm_dis '((11 12) (21 22) (31 32)) '(1 2 3) '(5 4))
→ 1808.41

(make_summ '((11 12) (21 22) (31 32)) '(1 2 3))
→ (-126 6092 -132 6608 3161 6344 -167002 -173324 3)

(solve '(5 4) '(-126 6092 -132 6608 3161 6344 -167002 -173324 3))
→ (132.033 -89.2333)

Commands. Using the procedures described above, it was possible to implement a command that, given the incoming data of the problem, is able to iteratively determine the solution, the position of the searched point and to transform in a graphical form the result thus obtained. This command can draw circumferences of input and some lines connecting the centres of these circumferences with the point determined as a solution.
In order to verify the reliability of the iteration algorithm, two more test commands were implemented that graphically display the progress of the calculation by making full use of the interoperability between AutoLISP and AutoCAD.

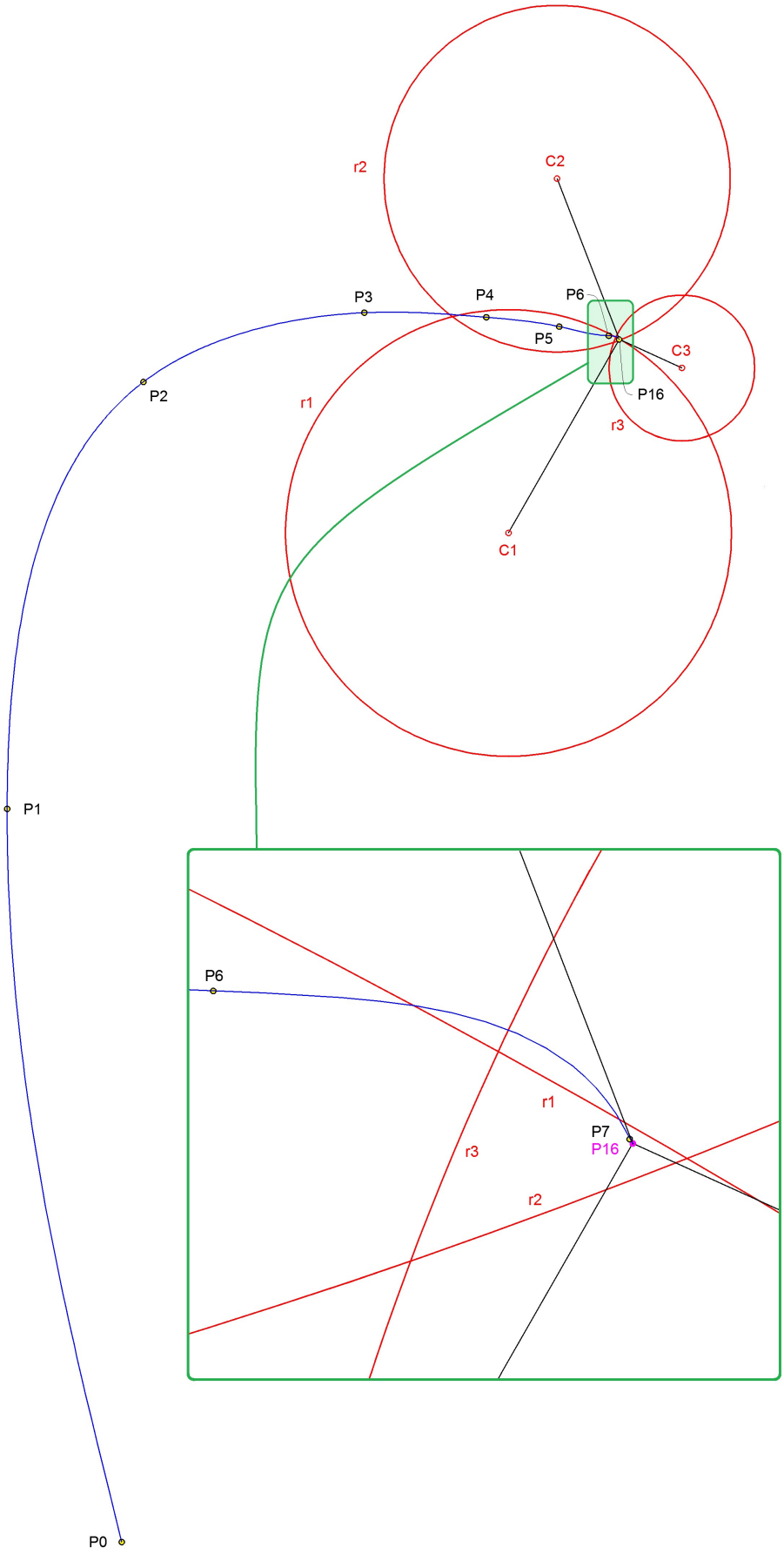
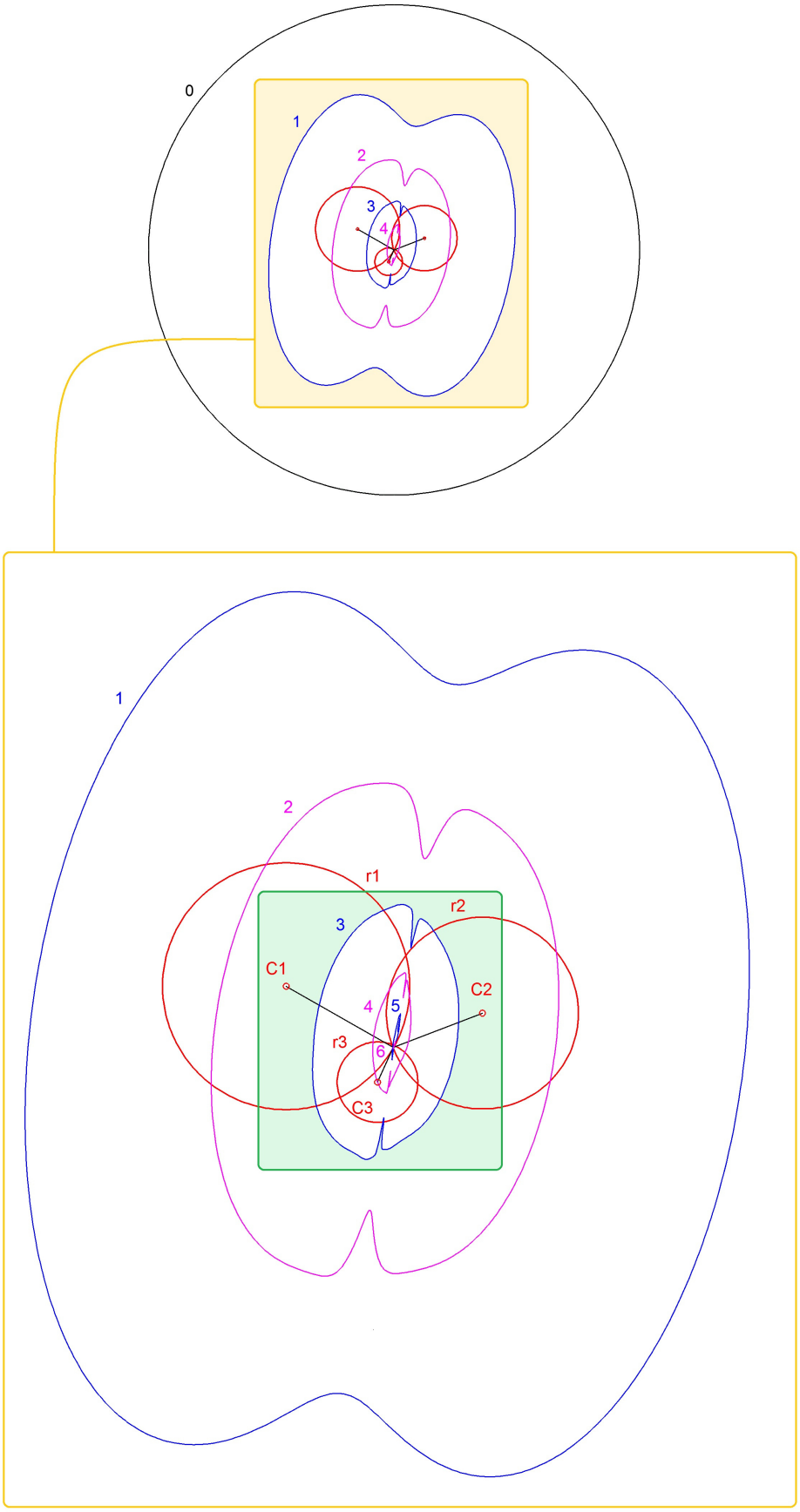



**Figure 1**

The first test command displays through a spline (blue in the figure) that the iterative calculation from a generic starting point P0 gets, passing by intermediate points P1, P2, P3, P4, P5, P6, P7, P16 to the end point determined as a solution; in the figure, we have considered as initial data of 3 circumferences, respectively of the centre C1, C2, C3 and radius r2 r1 r3. In the example it is possible to observe, that the path from the point P0 to the point P16 is not the shortest (the segment P16 P0) and that each point determined during the iteration is closer to the end point P16 than the previous one. In the enlargement of the detail (green in figure), you can see how the process, already at the seventh step of the iteration, is approaching the end point P16 so as not to allow a distinction between intermediate points from the P8 and P11.




**Figures 2 and 3**

In the two figures that follow, in which to the second test command is applied, we considered as starting data the same 3 circumferences of Figure 1. Through a series of concentric splines, numbered from 1 to 16 (blue and magenta colored alternately in the two figures), we can visualize the convergence towards the end point P16. The determined solutions have, as starting points, a set of points placed on the circumference 0 (black in the figure) having as its centre the end point P16 and a sufficiently large radius chosen at will. The usefulness of this procedure consists in checking the speed by which, upon the variation of the selected starting point, the iteration can determine the final solution; the graphical output of such a command also allows to check if solutions of relative minimum on which the algorithm converges do exist.

**Figure 3**

In the overall view (above) we can only distinguish the first 3 iterations; in a first enlargement (below) it is possible to distinguish the first 5 iterations.

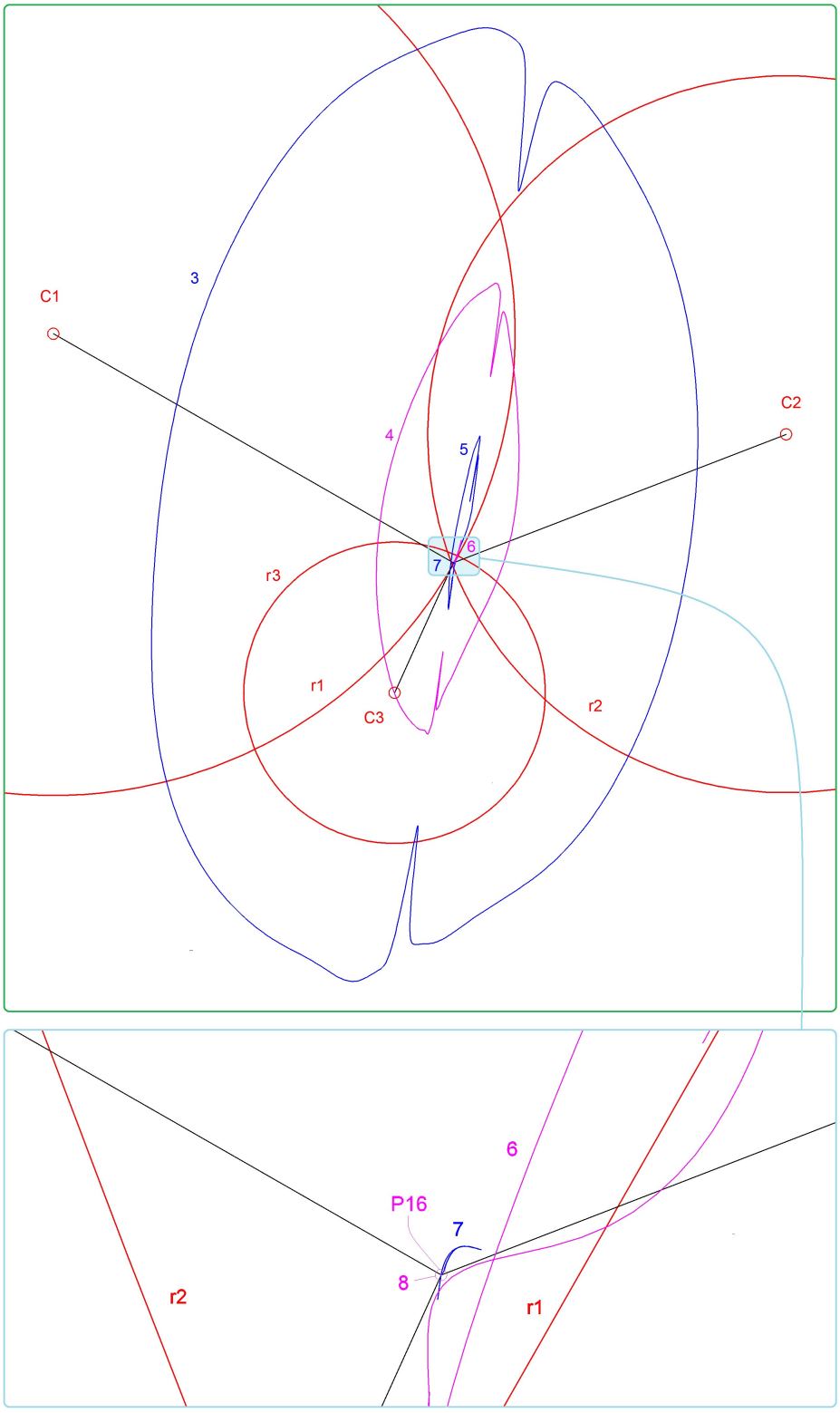



**Figure 4**

In the second enlargement (top) it is possible to distinguish the iterations 3, 4 and 5 and the area of intersection between the 3 circumferences assigned in which the end point P16 falls, and finally in a very big enlargement (below) it is possible to distinguish the iteration 7 and the place where the iteration from 8 to 15, now virtually coincident with the position of the end point P16, are settled.

## REFERENCES


[1] Agostini A. 1942. *Topografia e disegno topografico.Vol II and vol III.* Editore Ulrico Hoepli, Milano

[2] Agosto M. 1993. *AutoLISP. Corso base per utenti non programmatori.* Tecniche Nuove, Milano.

[3] Bousfield T. 1999. *AutoCAD AutoLISP.Guida pratica.* Tecniche Nuove, Milano. (original edition: Bousfield T. 1998. *A practical Guide to AutoCAD Autolisp.* Addison Wesley Longman Limited, England.)

[4] Cunietti M. 1959. *Corso teorico pratico sulle misure.* Edizioni libreria cortina, Milano

[5] Docci M. and Maestri D. 1984. *Il rilevamento architettonico. Storia metodi e disegno.* Editori Laterza, Bari.

[6] Gini C. and Pompilj G. 1976. *Metodologia statistica: integrazione e comparazione dei dati.* in: L. Berzolari (a cura di), *Enciclopedia delle matematiche elementari e complementi Vol. III Parte 3$^{a}$* , Hoepli, Milano.

[7] Krawczyk R. J. 2009. *The Codewriting Workbook. Creating computational Architecture in AutoLISP.* Princeton architectural press. New York

[8] Togores Fernandez R. and Otero Gonzales C. 2003. *Programacion en AutoCAD con Visual LISP.* Mc Graw Hill, Madrid.

[9] Valenti G. M. 2001. *Un ambiente virtuale per la progettazione, la simulazione e il collaudo di insiemi di misure di rilievo strutturati in forma reticolare.* in: Migliari R. (a cura di). *Frontiere del rilievo. Dalla matita alle scansioni 3D.* Gangemi Editore, Roma.

[10] Ventsel E. S. 1983, *Teoria delle probabilità*, Edizioni Mir, Mosca

# Session II

# Java interop with ABCL, a practical example

Alessio Stalla
ManyDesigns s.r.l.

alessiostalla@gmail.com

## ABSTRACT
In this paper, we give an overview of the Java interoperability features in the Armed Bear Common Lisp implementation, and we present an example of the application of some of those features to a real-world software project.

## Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *Classes and objects, Inheritance, Frameworks.*

## General Terms
Languages, Experimentation.

## Keywords
JVM, ABCL, Lisp, Java, interoperability.

## 1. INTRODUCTION
The Java Virtual Machine (JVM) [1] is a software platform that is widely used on servers, mobile devices, and, to a minor extent, desktops. Originally designed to host the now popular Java language [2], a mostly-static single-inheritance OO language in the C-like family, thanks to its dynamic features (which Java only partially benefits from), highly-optimized Just-In-Time compiler and garbage collectors, huge standard library and vast assortment of third-party libraries, frameworks and components, both commercial and open-source, the JVM nowadays hosts many static and dynamic languages, such as Scala[3], Groovy[4], JRuby[5], including several in the Lisp family: at least two Common Lisp implementations (ABCL[6] and CLForJava[7]), several Scheme implementations including the popular Kawa [8], and Clojure [9], a new Lisp dialect which has grown to respectable popularity.

We will focus in particular on the Armed Bear Common Lisp implementation (ABCL, [6]). We will give an overview of the facilities provided by ABCL to integrate Lisp and Java in a single software project. We will then show a proof-of-concept integration with a real-world application. We will also draw comparisons with Groovy, a popular scripting language for the Java platform.

## 2. ABCL
Armed Bear Common Lisp is an implementation of the Common Lisp standard targeting the Java Virtual Machine. ABCL is written in a combination of Java and Lisp and features both an interpreter and a compiler to JVM bytecode, limited debugging capabilities, SLIME integration, and facilities for Java interop which will be detailed in the following sections.

ABCL was originally developed by Peter Graves starting from 2002, as part of the J editor [10]. In 2008, Erik Huelsmann became the new maintainer of the project, which was subsequently moved from SourceForge to common-lisp.net and separated from the J editor. Over the years Huelsmann, with the help of other people, brought ABCL to be a stable, mature Common Lisp implementation with very few deviations from the standard, with recent release 1.0.1 failing only ~20 tests out of ~21700 in the ANSI compliance test suite by P. F. Dietz [11]. ABCL 1.0.0 was announced at the European Common Lisp Meeting 2011 in Amsterdam.

## 3. JAVA INTEROP OVERVIEW

### 3.1 Calling Java from Lisp
The JVM provides facilities for introspecting classes and objects, dynamically accessing fields and calling methods in arbitrary objects. Collectively these facilities are known as the Reflection API [12].

ABCL's Java FFI (Foreign Function Interface) is built upon the reflection API. Conceptually this makes it independent from any compiler support, although some optimizations are only possible by specially handling certain forms in the compiler (ABCL implements only some of the possible optimizations, which are out of the scope of this paper). Additionally, such reliance on the reflection API makes the Java FFI quite different from other Lisp FFIs (such as the popular CFFI [13], which allows to interface Lisp with C/C++ on a variety of platforms): whereas generally FFIs require the user to declare in advance the foreign data types and function signatures accessed by Lisp code, ABCL offers the option to dynamically select methods and fields giving the minimum amount of information, at the cost of a higher runtime cost per foreign call or field access. When performance is important, it is still possible to provide all the necessary type information to eliminate dynamic dispatch, although, as previously observed, there is still room for improvement in the compiler to further optimize certain calls.

Without going into pointless details, we illustrate with an example a typical use of the Java FFI (taken from the Dynaspring library):

```
(defun default-resource-loader ()
  (if *bean-definition-reader*
      (jcall "getResourceLoader"
             *bean-definition-reader*)
      (jnew "org...DefaultResourceLoader")))
```

ABCL's companion contrib project provides a library called JSS, originally developed by Alan Ruttenberg, which further simplifies the syntax for calling into Java. JSS is out of the scope of this paper, but we encourage every interested party to delve into the topic.

## 3.2 Calling Lisp from Java

ABCL is itself partly implemented in Java, to the extent that most fundamental Lisp types are defined by a corresponding Java class. It is thus possible to directly manipulate Lisp objects in Java, mimicking the style one would use in Lisp. Naturally, Java not being Lisp, such style is definitely more verbose than its Lisp counterpart. Additionally, direct manipulation of certain complex objects – for example structures and standard-objects – requires knowledge of their internal representation, and is therefore hard, low level, and too much dependent on implementation details.

Fortunately, ABCL provides a high-level interface to invoke Lisp from Java and perform common tasks. This interface follows a specification known as the Java Scripting Interface or JSR-223 [14] which makes ABCL a first-class citizen among scripting languages on the Java platform. The JSR-223 API is such that, for simple enough tasks (loading or compiling a file, calling a function, accessing a variable, etc.), no compile-time dependency on ABCL is required. This API also automatically performs useful conversions (from Java to Lisp and back) whenever appropriate.

Anyway, for certain Lisp constructs and idioms that do not exist in Java (such as binding special variables or condition handlers), some manual work is left to the programmer. For example, the following Java code executes a function with `*print-circle*` bound to `T`:

```
LispThread t = LispThread.currentThread();
SpecialBindingsMark m = t.markSpecialBindings();
t.bindSpecial(Symbol.PRINT_CIRCLE, Symbol.T);
try {
  return function.execute(args);
} finally {
  t.resetSpecialBindings(mark);
}
```

In our experience, in real-world projects it is preferable to use a mixed style of high-level JSR-223 calls and lower-level direct access to Lisp objects, depending on the task.

## 3.3 Beyond FFI: extending Java in Lisp

A feature which is, to our knowledge, unique of the ABCL FFI (as compared to other FFIs available on Common Lisp implementations), is the ability to integrate with the host platform to the point of extending Java classes with Lisp code. This feature actually comes in two flavours, detailed below.

### *3.3.1 Implementing interfaces*

With ABCL, it is possible to implement one or more Java interfaces using Lisp functions. In the JVM, an interface is a special kind of abstract class that defines a contract in the form of a set of methods that subclasses must implement. Contrarily to regular classes, multiple inheritance of interfaces is possible. Interfaces are commonly used to abstract all kinds of behavior, and especially when dealing with event listeners, visitor objects, callbacks and the like. The advent of some advanced techniques (e.g., certain popular forms of Aspect-Oriented Programming as provided by the Spring Framework [15]), as well as the integration of the so-called Project Lambda [16] in the forthcoming Java 8 specification (aiming to provide syntax sugar for anonymous function-like constructs), have made and will continue to make interfaces a key feature of the Java language and a principal point of integration with many libraries and frameworks. ABCL uses a native feature of the Java Reflection API, called dynamic proxies [17], to allow the implementation of interfaces in several ways, all revolving around the use of Lisp functions as the implementations of Java methods.

### *3.3.2 Extending classes*

Although programming to interfaces is considered a best practice in Java, it's not always possible to achieve integration using interfaces alone. Some libraries or frameworks mandate the user to extend certain third-party classes, for example to inherit important built-in functionality, or to customize and extend those libraries themselves, or simply due to poor design. Also, with the introduction of annotations in Java 5 [18], a system for attaching metadata to various program elements, it has become common practice to use classes as holders of various types of configuration information, for example in web frameworks such as Stripes[19] or other types of libraries (ORMs like Hibernate[20], application frameworks like Spring[21], etc.). ABCL tackles the aforementioned problem by giving the user the ability to extend Java classes in Lisp, principally by providing the implementation of methods with Lisp functions, but also, if required, by adding fields and placing annotation metadata. This feature, known as runtime-class, has been only recently re-added to ABCL (it had been lost for quite a long time), and at the time of writing it is still new and experimental.

The runtime-class feature will be demonstrated, along with other aspects of the FFI, in the example which is the subject of the next section.

# 4. A PRACTICAL EXAMPLE

We will examine a very simple example of the use of ABCL as the extension language of an existing real-world Java web application. The application was chosen taking into consideration the author's knowledge of it and the fact that it already has provisions for being customized via scripts written in Groovy, a popular dynamic language for the JVM which has easy Java integration, down to source compatibility, as a core feature. We believe that a comparison with Groovy will show Lisp's strong points as an extension language.

## 4.1 ManyDesigns Portofino

The target of our example is ManyDesigns Portofino 4, the flagship product of ManyDesigns s.r.l. It is a model-driven application framework tailored at building database-driven web applications running on the JVM, with extensibility as one of the core goals. At the time of writing, ManyDesigns Portofino 4 is being developed internally and sold commercially to a few selected clients, but it will eventually be released as open source.

For the purpose of this paper, it suffices to say that Portofino 4 applications are organized as a tree of pages of various types. This tree gives form to the URLs exposed by the application and to its navigation bar, and corresponds to a tree of directories physically stored on the file system. Each directory contains a couple of configuration files and, most importantly, a .groovy file defining a class. That class determines the type of the page (SCRUD over a database table, Chart page, Text page, etc.) and is also the main point of extension to customize the page. For example, a standard crud page will have at an action.groovy file like:

```
[imports elided]
class foo extends CrudAction {}
```

Each action can respond to HTTP requests via handler methods with the signature

```
public Resolution method()
```

if the request contains a GET or POST parameter X and a method with that signature and named X exists in the action, it will be called and the resulting Resolution object will be used by the Stripes web framework, on which Portofino 4 is built, to determine the next view to show to the user.

Via annotations, access to action methods can be restricted to only users with certain privileges, for example:

```
@RequiresPermissions(level = AccessLevel.VIEW)
public Resolution method() { ... }
```

Annotations can also be used to expose a given method as a user-visible button on the page. The same method can back up more than one button in different areas of the page. For example:

```
@Button(list = "crud-edit", key = "my.button")
public Resolution method() { ... }
```

Buttons may be guarded, which causes them to be disabled or hidden if their preconditions are not met, but this feature will not be interested by our little example.

## 4.2 Integrating Lisp in Portofino

Standing the premises outlined in the previous section, it appears clearly that a Lisp integrated in Portofino 4 needs to be able to extend Java classes so we can use them as custom actions, and to annotate them so we can control access rights and buttons.

Even though the file-based nature of the framework suits Groovy better than Lisp, where a somewhat more image-based style is generally preferred, we won't touch this aspect. We speculate that migrating to a more Lisp-friendly style of development and interaction would not be particularly complex, but to keep the example simple we have decided to supersede about it.

We will omit for brevity the incantations needed to add ABCL as a dependency to Portofino using Maven. Once ABCL is added as a library, we can start patching the application to replace Groovy with Lisp (a possible improvement over this basic example would be having Groovy, Lisp, and possibly other languages coexist in the same application). We will add a method to load a Lisp file defining a custom action class to the already existing ScriptingUtil class, which has a few utility methods for loading Groovy code:

```
public static final AbclScriptEngine ABCL =
(AbclScriptEngine) new
AbclScriptEngineFactory().getScriptEngine();

public static Class<?> getLispClass(File
storageDirFile, String id) throws IOException {
  File scriptFile =
    getLispScriptFile(storageDirFile, id);
  if(!scriptFile.exists()) {
    return null;
  }
  FileReader fr = new FileReader(scriptFile);
  try {
    Object result = ABCL.eval(fr);
    return (Class) result;
  } catch(ScriptException e) {
    throw new RuntimeException(e);
  } finally {
    IOUtils.closeQuietly(fr);
  }
}
```

Then, replacing the few calls to getGroovyClass with calls to getLispClass, we've done most of the necessary work. For various reasons, Portofino needs to check whether a certain action class is user-defined or not. Groovy classes all implement an interface, GroovyObject, so the system can test for it to detect classes written in Groovy. ABCL doesn't add any marker interface by itself, so we'll have to define our own (say, an empty LispAction interface) and remember to use it when we define our action classes. Also, since ABCL's startup time is noticeable, we can add a little piece of code – not shown here – to load it at startup rather than at the first invocation of an action.

We are now ready to replace our Groovy action. We'll take as an example a modified version of one of the actions that are part of Portofino's built-in ticket tracker example application: a class that customizes the projects CRUD page to add computed values for the project's create and update date, and to add a shortcut button to immediately show the form for creating a new ticket for that project. It is important to note that objects manipulated by CRUD pages are Java maps, for which Groovy has built-in syntax to access their contents.

```
import ...

@SupportsPermissions(...)
@RequiresPermissions(level = AccessLevel.VIEW)
class _1 extends CrudAction {

  boolean createValidate(object) {
    object.created_on = new Date();
    return true;
  }

  boolean editValidate(object) {
    object.updated_on = new Date();
    return true;
  }

  @Button(key = "project.issue.create",
          list = "crud-read",
          order = 100D)
  Resolution createNewTicket() {
    return new RedirectResolution
      (dispatch.originalPath + "/issues")
      .addParameter("create");
  }
}
```

Directly translating it to Lisp, we get:

```
(java:jnew-runtime-class
 "_1"
 :superclass "com...CrudAction"
 :interfaces '("com...LispAction")
 :annotations `("com...RequiresPermissions"
                ("com...SupportsPermissions"
                 ("value" :value ,(list ...))))
 :methods
 `(("createValidate" :boolean ("java.lang.Object")
    ,(lambda (this obj)
       (jcall
         "put" obj
         "created_on" (jnew "java.util.Date"))
       +true+))
   ("createNewTicket" "org...Resolution" nil
    ,(lambda (this)
       (jcall "addParameter"
              (jnew "org...RedirectResolution"
```

```
                    ...)
                "create"))
    :annotations
    `(("com...Button"
       ("list" :value "crud-read")
       ("key" :value "project.issue.create")
       ("order" :value 100.0))))))
```

That's quite more verbose than its Groovy counterpart, but with the help of some helper functions and a little macrology we can obtain:

```
(defaction "_1" (:crud)
  ("createValidate" :boolean ("java.lang.Object")
   (lambda (this obj)
     (setf (prop obj "created_on")
           (jnew "java.util.Date"))
     +true+))
  (:action "createNewTicket"
   (lambda (this)
     (with-parameters ("create")
       (redirect-to ...)))
   :buttons ((:list "crud-read"
              :key  "project.issue.create"
              :order 100.0))))
```

which is even shorter than in Groovy, and hides from the user many of the implementation details that are apparent from the Groovy class we saw earlier.

## 5. CONCLUSIONS

We have summarized the possibilities of interaction with Java in the ABCL implementation. We have also seen that with little effort it is possible to use Lisp as an extension language in a Java application, similarly to other languages on the JVM which have been expressly designed for that purpose, like Groovy. While the literal translation of a Java or Groovy class to Lisp is verbose and not idiomatic, it is possible to hide most of the boilerplate behind a simple DSL, so that the resulting code is often more compact and readable than the original.

## 6. REFERENCES


[1] Lindholm, T. and Yellin, F. *The Java™ Virtual Machine Specification*, second edition. http://java.sun.com/docs/books/jvms

[2] Gosling, J., Joy, B., Steele, G., and Bracha, G. *The Java Language Specification*, Third Edition. http://java.sun.com/docs/books/jls/

[3] Odersky, M. 2011. *The Scala Language Specification*. http://www.scala-lang.org/docu/files

[4] http://groovy.codehaus.org/

[5] http://jruby.org/

[6] http://common-lisp.net/project/armedbear/

[7] http://clforjava.org/

[8] http://www.gnu.org/software/kawa/

[9] http://clojure.org/

[10] http://armedbear-j.sourceforge.net/

[11] http://common-lisp.net/project/ansi-test/

[12] http://docs.oracle.com/javase/tutorial/reflect/

[13] http://common-lisp.net/project/cffi/

[14] http://www.jcp.org/en/jsr/detail?id=223

[15] http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/aop.html

[16] http://openjdk.java.net/projects/lambda/

[17] http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html

[18] http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html

[19] http://www.stripesframework.org/display/stripes/Home

[20] http://www.hibernate.org/

[21] http://www.springsource.org/

# Embedded probabilistic programming in Clojure

Nils Bertschinger
Max-Planck Institute for Mathematics in the Sciences
Leipzig, Germany
bertschi@mis.mpg.de

## ABSTRACT

Probabilistic programming is a powerful tool to specify probabilistic models directly in terms of a computer program. Here, I present a library that allows to embed probabilistic computations into Clojure. Automatic tracking of dependencies between probabilistic choice points enables an efficient way to sample from the distribution corresponding to a probabilistic program.

## 1. INTRODUCTION

Nowadays, many problems of artificial intelligence are formulated as probabilistic inference. Also the machine learning community makes heavy use of probabilistic models [2]. Thanks to the steady increase of computing power and algorithmic advances it is now feasible to apply such models to real world data from various domains. Many different types of inference algorithms are available, but most of them fall into just a few well-studied and understood classes, such as Gibbs sampling, Variational Bayes etc. Even if such standard algorithms are used, still a lot of hand-crafting is required to turn a probabilistic model into code.

Probabilistic programming tries to remedy this problem, by providing specialized programming languages for expressing probabilistic models. The inference engine is hidden from the user and can handle a large class of models in a generic way. WinBUGS [5] is a popular example of a language for specifying Bayesian networks which are then solved via Gibbs sampling. Instead of designing a custom domain specific language (DSL) for probabilistic computations, it is also possible to extent existing languages with probabilistic semantics. This has been done mainly for logic and functional programming languages [3, 7]. Especially, in the case of functional programming languages, rather lightweight embeddings have been developed [4, 12]. Furthermore, the structure of the embedding is well understood in terms of the probability monad [10].

Here, I present a shallow embedding of probabilistic programming into Clojure, a modern Lisp dialect running on the JVM. The implementation is based on the bher compiler [12], but does not require a transformational compilation step. Instead, it utilizes the dynamic features of the host language for a seamless and efficient embedding. As the bher compiler, my library uses the general Metropolis-Hastings algorithm (Sec. 2.1) to draw samples from a probabilistic program. In contrast to previous implementations, dependencies between the probabilistic choice points encountered during a run of the program are tracked (Sec. 3). This allows to speed up sampling by exploiting conditional independence relations. Sec. 4 illustrates the resulting efficiency on an example application. Finally, Sec. 5 concludes with some general comments on the development of this library.



## 2. PROBABILISTIC PROGRAMMING

A functional program consists of a sequence of functions which are applied to some input values in order to compute a desired output. Each function in the program can be understood as a mathematical function, i.e. it produces the same output when given the same input values. A function with this property, that the output can only depend on its inputs, is called "pure". The semantics of a functional program is thus given by a mapping from inputs to outputs. Probabilistic programs extend the notion of a function to probabilistic functions which map inputs to a distribution over their outputs. A probabilistic program thus specifies a (conditional) probability distribution instead of a deterministic function.

Probabilistic functions are not pure[1], in the sense that they could return different values – drawn from a fixed distribution – for the same input. Thus, a run of a probabilistic program gives rise to a computation tree where each node represents a basic random choice, such as flipping a coin. The children correspond to the possible outcomes of the choice and the edges of the tree are weighted with the probability of the corresponding choice (see Fig. 1 for a simple example). Each path through the computation tree corresponds to a possible realization of the probabilistic program and will be called a *trace* in the following. The probability of such a trace is simply the product of all edge weights along the path.

Similar trees also arise from non-deterministic operations which have often been embedded into functional languages [1]. The main difference is that probabilistic choices are weighted and the probabilities of different choices add up in the final result. Therefore, the common practice to implement non-determinism, i.e. following just one possible

[1]In particular, the semantics of a probabilistic program changes when memoization is introduced.

trace and back-tracking if no solution could be found, is not suitable to evaluate a probabilistic program. Instead, either all possible traces have to be considered or an approximate scheme to sample from the probability distribution corresponding to the program is necessary.

## 2.1 Metropolis-Hastings sampling

Metropolis-Hastings is a general method to draw samples from a probability distribution. Since it only needs to evaluate ratios of probabilities, it is particularly useful if the probabilities can only be calculated up to normalization. Metropolis-Hastings is a Markov-Chain Monte-Carlo (MCMC) method which, instead of directly sampling from the desired distribution $p(x)$, constructs a Markov chain $p(x'|x)$ with a unique invariant distribution $p(x)$ (see [2] for details). Assume that the current sample is $x$ and $p(x)$ can be evaluated up to normalization. Then, a proposal distribution $q(x'|x)$ is used to generate a new candidate sample. $x'$ is then accepted as a new sample with probability

$$\min\left\{\frac{p(x')q(x|x')}{p(x)q(x'|x)}, 1\right\}.$$

Otherwise the old sample $x$ is retained. It is well known that this defines a Markov chain with invariant distribution $p(x)$. The efficiency of this algorithm depends crucially on the proposal distribution. Ideally the proposed values should be from a distribution close to the desired one, but at the same time it should be easy to sample from.

In the case of a probabilistic program, each sample corresponds to a trace through the computation tree. A global store of all probabilistic choices is used to keep track of the current trace. Thus, the probabilistic program is purified and deterministically evaluates to the particular trace through the computation tree that is recorded in the global store. From this, a new proposed trace is constructed using the steps illustrated in Fig. 2:

- Select a random choice $c$ (with value $v$) from the old trace.
- Propose a new value $v'$ for this random choice using a local proposal distribution $q_c(v'|v)$.

Using the global store of all random choices, a new trace is obtained by re-running the probabilistic program with a new store reflecting the proposed value. Pseudo-code for this algorithm can be found in [12]. Unfortunately, they do not fully specify how the so-called forward and backward probabilities $q(x'|x)$ and $q(x|x')$ are obtained. Here, care needs to be taken that choice points, which only occur in the new (old) trace but not in the other trace, are accounted for in the forward (backward) probability.

An important point is how choice points are identified between the different traces. If they would be considered as unrelated, each trace would draw completely new choice points and the method boils down to rejection sampling. The change becomes more local the more choice points can be identified between the two traces. In [12] it is argued that choice points should be identified according to their structural position in the program. Currently, this is not yet implemented and instead the user has to explicitly tag each choice point. With dynamically bound variables it should be possible to pass along information about the structural position of each choice point. Implementing a naming scheme based on this idea is left for future work.

## 2.2 Memoization and conditioning

Memoization of choice points, as introduced in [3], can be implemented by simply changing the tag of a memoized choice point to reflect its type and arguments which should be memoized. This way, the same random choice is fetched from the global store if the memoized choice point is called again with the same arguments. The form (**memo** (<basic choice point><parameters>) <optional arguments>) is provided to memoize a basic choice point on its parameters and possibly further identifying arguments.

Conditioning can be implemented in different ways. The most general form allows to condition on any predicate and can be achieved by invalidating the trace, i.e. set its probability to zero, if the predicate evaluates to false. Unfortunately, this is a rather inefficient way to implement conditioning since an invalid trace is always rejected and thus a form of rejection sampling is obtained. In the common case of conditioning a basic random choice on a single, specified value, a better implementation is possible. In this case, the choice point is fixed at the conditioning value and deterministically reweights the trace by the probability of this value.

# 3. NETWORKS OF CHOICE POINTS

Whenever a new proposal is evaluated, the whole program is run again in order to compute the updated trace. In contrast, a hand-crafted sampling algorithm, e.g. for a graphical model, only recomputes those choice points which are neighbors of the node where a change is proposed. Especially, in the common case of sparsely connected models with many conditional independence relations, this leads to huge performance gains. Quite often, each sampling step can be computed in constant time, independent of the total number of choice points.

Similar observations have lead some researchers away from probabilistic functional programming. Instead, they have developed object-oriented frameworks [9, 6] which represent each choice point as an object. The program is then used to connect such objects and imperatively defines a model consisting of interconnected choice points. This model can then be solved either by exact methods based on belief propagation or approximate methods such as Markov-Chain Monte Carlo. This approach has the further advantage that subclassing allows the user to provide custom choice points and control the proposal distributions that are to be used. Unfortunately, the direct relation between the program and the model is lost and it usually not possible to specify models with a changing topology (e.g. Fig. 1). Thus, the object-oriented implementation is more of a library for Bayesian networks (or Markov random fields) than a language extension for embedded probabilistic programming[2].

Here, I propose an implementation which keeps track of the dependencies between probabilistic choices. Then only the actual dependents of a choice points need to be recomputed if a new value is proposed. This approach is inspired by reactive programming which allows to set variables via spreadsheet-like formulas. If a variable changes, all depen-

[2]A better embedding can be achieved if language constructs are overloaded for choice point objects. This works rather nicely for standard operators on numbers, list, etc. but looks somewhat clumsy for special forms, e.g. if. Furthermore, a small additional overhead is introduced for every operation whereas, ideally, the deterministic parts of the program should be able to run at full speed.

**A:**

```
(let [c1 (flip 0.5)
      c2 (flip (if c1 0.8 0.4))
      c3 (if (= c1 c2)
           (flip 0.3)
           true)]
  (and c1 c3))
```

**B:**

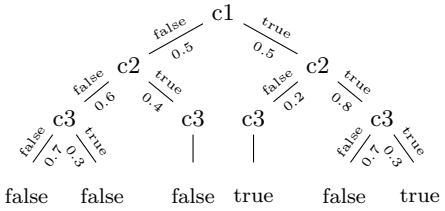



**Figure 1: A simple probabilistic program and its computation tree. A: The probabilistic program in Clojuresque pseudo-code: flip is a basic probabilistic function which returns true with the specified probability and false otherwise. B: The computation tree corresponding to the program on the left. Note that the number of random choices depends on the outcome of previous choices.**

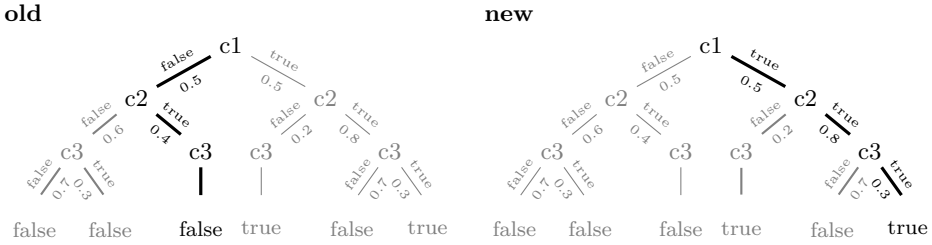



**Figure 2: Example of proposing a new trace, by changing the value of the first random choice $c1$. The value of choice $c2$ can be reused, but has to be reweighted with the new probability. The random choice at $c3$ is created anew when moving from the old to the new trace.**

dent formulas are then updated automatically. This approach has for example been implemented in Common Lisp (Cells [11]) and Python (Trellis) in order to simplify the specification of user interfaces. In both implementations, dependencies between variables are tracked automatically and dynamically changed if different conditional branches are followed.

In order to do this, choice points are specified explicitly and a dependency with another choice point is established whenever their value is accessed, with (**gv** <choice point>). Using this approach the example from Fig. 1 can be written as follows:

```
(let [c1 (flip-cp :c1 0.5)
      c2 (flip-cp :c2
           (if (gv c1) 0.8 0.4))
      c3 (det-cp :c3
           (if (= (gv c1) (gv c2))
             (gv (flip-cp :c3-a 0.3))
             true))]
  (det-cp :result
    (and (gv c1) (gv c3))))
```

Thus, the programmer can control which parts of the program are recomputed in case of a proposal. For this to work, the value of a choice point is not extracted in the binding form, but explicitly accessed with **gv**. In order to ensure that all parts of the program which use values of probabilistic choices can be recomputed, some additional deterministic choice points have to be introduced which re-evaluate their body whenever any choice point they depend on is changed. Overall, the program can be written in a style that is quite close to a full DSL for probabilistic programming. The programmer just has to take care that choice points are always accessed with **gv** and only inside other choice points[3].

[3]Clojure is dynamically typed and therefore the compiler cannot enforce that these rules are obeyed. Nevertheless, the restrictions on probabilistic programs are bearable and unproblematic in practical programs.

Overall, the resulting programming style is somewhat intermediate between a functional probabilistic programming language and the object-oriented style mentioned above. In the latter approach, objects are used to represent choice points and dependencies are explicitly constructed between them via methods that take parent choice points as arguments. Thus, the program is not a specification of the probabilistic model itself, but is used to construct it as a static network of choice points. Here, choice points are also represented as a special data structure, but as in a functional probabilistic programming the program itself is the probabilistic model. The explicit choice points are merely introduced in order to speed up recomputations necessary during Metropolis-Hastings sampling. In addition, having an explicit representation of choice points enables extensibility, as in the object oriented approach. The macro **def-prob-cp** allows to add custom choice points which support additional probability distributions or use specialized proposal distributions.

# 4. EXAMPLE

To illustrate the power of probabilistic programming Fig. 3 shows how a Gaussian mixture model with a Dirichlet prior for the mixture components can be implemented. The program closely follows the structure of the statistical model:

1. Draw the component weights from a Dirichlet prior.
2. Assign each data point to one of the components.
3. Draw a mean for the corresponding component model from a Gaussian prior.

**A:**

```
(defn mixture-memo [comp-labels data]
  (let [comp-weights (dirichlet-cp :weights (for [_ comp-labels] 10.0))]
    (doseq [[idx point] (indexed data)]
      (let [comp (discrete-cp [:comp idx] (zipmap comp-labels (gv comp-weights)))
            comp-mu (memo [:mu idx] (gaussian-cp :mu 0.0 10.0) (gv comp))]
        (cond-data (gaussian-cp [:obs idx] (gv comp-mu) 1.0) point)))
    (det-cp :mixture
      [(into {} (for [comp comp-labels]
                  [comp (gv (memo [:mu comp] (gaussian-cp :mu 0.0 10.0) comp))]))
       (gv comp-weights)])))
```

**B:**

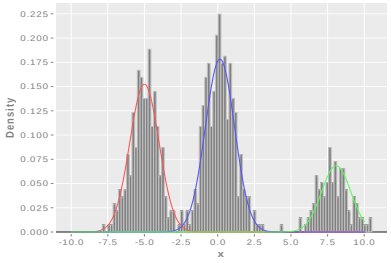



**Figure 3: A: Probabilistic Clojure program implementing a Gaussian mixture model. B: Sample from the posterior overlaid on a histogram of the data set.**

4. Condition the Gaussian component model on the observed data point.

Note, how memoization is used to share parameters of the component models between different data points assigned to the same mixture component. Furthermore, the example illustrates that conditioning is a side-effects and acts like an assignment.

The following form call to a library function

```
(metropolis-hastings-sampling
  (fn [] (mixture-memo [:a :b :c] data)))
```

and returns a sequence of samples from the posterior distribution of the model with three mixture components, labeled **:a, :b** and **:c**, when conditioned on a vector of observed **data** points.

Panel **B** of Fig. 3 shows a single sample from the posterior overlaid on a histogram of the data points. Here, a data set consisting of 750 points drawn from a mixture of three Gaussians has been used. Since each Metropolis-Hastings step only needs to recompute the choice points which are effected by the proposal, the program scales to considerably larger data sets. Preliminary tests show that it is slightly faster than the Python library PyMC [8] which also implements MCMC sampling.

## 5. CONCLUSIONS

Probabilistic programming is a powerful tool to specify probabilistic models. Many different types of models, e.g. mixture models, regression models etc., can be implemented with ease and conciseness. Thus, even though the performance falls short of hand-crafted algorithms, it is rather useful for rapid prototyping of probabilistic models. The presented Clojure library in addition demonstrates that Lisp is well suited as a host language for embedded probabilistic programming. The code base running the example from Sec. 4 is rather compact with just below 1000 LoC. This is possible due to a unique combination of features, namely dynamic variables, macros and immutable hash-maps. Especially the immutable data structures allow for an easy and efficient implementation of Metropolis-Hastings sampling: Reverting to the old trace is always possible without creating a copy of the global store first. Thus, they serve as efficient difference data structures which had to be handcrafted in [7] for this purpose.

Furthermore, the dynamic nature of Clojure was of great help in developing the library. During development I implemented different prototypes, which helped to clarify unforeseen corner cases making the tracking of dependencies somewhat tricky. Overall, Lisp is still a great language for rapid prototyping and the presented library now allows to explore different types of probabilistic models in Lisp itself.

## 6. REFERENCES


[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2 edition, 1996.

[2] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[3] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: A language for generative models. In *Uncertainty in Artificial Intelligence*, pages 220–229, 2008.

[4] O. Kiselyov and C. Shan. Embedded probabilistic programming. In W. Taha, editor, *IFIP working conference on domain-specific languages*, LNCS 5658, pages 360–384. Springer, 2009.

[5] D. J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. Winbugs – a bayesian modelling framework: concepts, structure, and extensibility. *Statistics and Computing*, 10:325–337, 2000.

[6] A. McCallum, K. Schultz, and S. Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances on Neural Information Processing Systems (NIPS)*, 2009.

[7] B. Milch and S. Russell. General-purpose mcmc inference over relational structures. In *22nd Conference on Uncertainty in Artificial Intelligence*, pages 349–358, 2006.

[8] A. Patil, D. Huard, and C. J. Fonnesbeck. Pymc: Bayesian stochastic modelling in python. *Journal of Statistical Software*, 35(4):1–81, 2010.

[9] A. Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics, 2009.

[10] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 154–165, New York, NY, USA, 2002. ACM.

[11] K. Tilton. The cells manifesto. http://smuglispweeny.blogspot.com/2008/02/cells-manifesto.html, 2008.

[12] D. Wingate, A. Stuhlmueller, and N. D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. *Proceedings of the 14th international conference on Artificial Intelligence and Statistics*, 2011.

# A little history of metaprogramming and reflection

**Pascal Costanza**
pc@p-cos.net | Intel| Belgium

"The interpreter for a computer language is just another program." - Hal Abelson

Whenever a program P written in a programming language L is executed on a computer, it is actually executed by another program M running on the same computer whose explicit purpose is to execute such programs written in L. Although it is possible to write programs that are executed directly by the computer itself, this is actually rare. Most of the time there is some, or even many, intermediate levels of programs whose sole purpose is to execute other programs written in languages that are easier to understand by programmers. Such intermediate levels are called interpreters or compilers, or sometimes just language processors.

A very powerful idea is to enable programmers to tell such a language processor how to process a given program in a specific way. Most programming languages actually provide such influence on the underlying language processor, for example by passing options to the processor, by adding declarations or annotations to a program, or by calling special functionality that can be provided only in an API provided by the language processor itself.

The programming language Lisp is unique in that it gives the illusion that the language processor is actually implemented in Lisp itself. This gives programmers the power to adapt the language processor by writing programs in the same programming language - Lisp - and with the same ease as any other regular programs. This ability to program the language processor from within itself is called reflection.

I will give an overview of the history of Lisp and reflection, together with some important concepts that have emerged during the decades and that are relevent in modern Lisp dialects today: Macros, objects and metaobject protocols.

# Session III

# Scheme in Industrial Automation

## [Extended Abstract]

Marco Benelli
mbenelli@yahoo.com

## ABSTRACT

This paper provides a description of a success story in using Scheme in production environment, in the field of industrial automation. It describe the migration of an application based on legacy technologies like Java applets and CGIs written in C, to a modern web application, through a set of smooth steps that have incrementally improved the product and the workflow. The components that have realized this improvement are a set of declarative languages for configuration and customizations and a web framework. The Scheme implementation Gambit-C, has showed itself very well suited for both these tasks, thanks to the scheme's nature and gambit's own extensions.

## Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.2.12 [**Software Engineering**]: Interoperability; H.3.5 [**Information Storage and Retrieval**]: On-line Information Services

## Keywords

scheme, gambit, functional programming, domain specific languages, web

## 1. INTRODUCTION

The limited resources typically found in SCADA systems seem to be an obstacle to the development of modern web-based Human Machine Interfaces. Since it's well known that "Hardware is Cheap, Programmers are Expensive", most manufacturers choose to upgrade the hardware and rely on the usual tools for web development: java, .net, php, python, ruby and so on. Nevertheless, the cost is not the only drawback of an hardware upgrade: space and power consumption are also important factors. Furthermore, in mass production, even a little update in the hardware can be quite expensive.

This paper describe how a legacy system has been improved to a better web interface, and how Scheme has made the transaction as smooth as possible.

## 2. REQUIREMENTS

The starting point was a system that already has a web interface. The more interactive parts of interface (data plotting and plant synopsys), however, were realized through Java applets. The rest of application was powered by CGIs (written in C) that handled data retrival and session management.

The system was developed for an ARM-powered board with about 200 Mhz of cpu frequency and 128 MB of RAM, and used for supervisoring wastewater plants, but it was intended to be the base of more generic applications and to be ported on other architectures[1]. An another important requirement was the easy of customization by users without programming experience. Furthermore, some customers asked to have the supervisor on proprietary devices, that doesn't give any option for installing custom software.

## 3. TOOLS

The first step has been the development of a declarative language to let users (domain experts) to generate interfaces from specification. This strategy was already implemented with java and XML, but it had some defects, and, using applets, it brokes the web paradigms, and require the presence of JRE on client machines. So it has been decided to get rid of applets, and adopt a more modern AJAX interface. Since the task was basically a source to source transformation, Lisp resulted the most natural choice. Then the cgi layer was replaced by a more flexible application, so a dynamic language with good performance was required. Considering future developments, portability was also a desiderable features, expecially because we want to be able to support other platforms in the futures: this constraints forced us to choose a language that was compilable to C.

There is a good choice of Common Lisp and Scheme compiler that satisfy these requirements. Scheme has been chosen because of its functional features [1], valuable in XML traversing[2]; Gambit[2] has been chosen because of its extensions like green threads, ffi, extended ports. A lot of scheme libraries are used: SRFIs, SSAX, irregex; most of which are

[1]The resulting system is actually used in production on ARM, x86 and SH-2 processors
[2]http://www.iro.umontreal.ca/~gambit/

already been ported and optimized for Gambit. No module system like Snow [3] or Black Hole [4] have been used: Gambit's namespaces and separate compilation was good enough for our needs. The macro system used is the Gambit's built-in define-macro (similar to Common Lisp's defmacro), the syntax-case expander has been initially used, but the overhead in size of object files was excessive.

## 4. DOMAIS SPECIFIC LANGUAGES

The pages more used were the synopsis applets: they showed an image of the plant, with a clickable icon or text field for each physical device (pumps, switches, sensors) and a visual feedback of their state (disabed, on, off, alarm). All the configuration were read from an XML file generated by a propietary editor, while runtime data were retrived from database.

To get rid of Java applets, a Domain Specific Language has been written. This language takes as input the XML generated by the editor, and produce a set of HTML, CSS and Javascripts files that realize the same applet's interface. The language makes heavy use of SSAX-SXML. In this way, it has been possible to have a new interface without any changes neither in existing resources, neither in workflow.

Building the XML configuration for the synopsis required the user to manually insert device informations (type, measure units, range and so on). This workflow was slow and bug-prone. To improve it, a new language has been developed, that takes as input an HTML map (created with common graphicals editor like Illustrator, Gimp...) with the positions of items, and a CVS file built from a spreadsheet, containing all the devices's configurations.

This workflow has been proved to be very effective and flexible enough to handle all the requirements emerged in new projects (new devices, different visual feedbacks) with little or no modification to the transformer.

A similar language has been developed for creating applications composed of only static HTML pages and AJAX communication. This tecnique has made possible the development of web interface on proprietary systems required by customers, with a minimal effort.

## 5. WEB SERVER AND APPLICATIONS

Given the constraints of host machines, the web interface was originally written as C CGIs. This strategy has been proved to be quite inflexible: basically the main CGI was an interface to database that return query results to the client as simple text. There is another CGI that handles authentication and sessions. The lack of a resident process forced the CGIs to continuously storing and reading data to database; furthermore, all this management has been written from scratch, due to the difficulties in porting usual tools on some of the target machines. Some experiment have been made to use Python but they have shown poor performance. And on some machine (ie SH-2), porting Python is not a trivial task.

[3]http://snow.iro.umontreal.ca/

[4]https://github.com/pereckerdal/blackhole

The solution to this tradeoff between flexibility and performance has been found in the Klio web server, a component of the open source Klio tools [5], a set of tools written in Gambit Scheme. The idea of replacing the web server is born after a long experience in using a simple Scheme web server for test and debug. The web server was a slighty modified version of the one present in Gambit distibution.

The Klio web server comes from completing the implementation by adding missing features to make it HTTP/1.1 compliant: persistent connection, caching, chunked data and so on. While it support continuation-based interaction[4], they are not been (yet) used due to the heavy use of AJAX communication between client and server. On the other hand, first-class continuations have been helpful in interaction on some acquisition systems, that use exclusively unix signal for interprocess communications. The time required for these developments has been surprisingly short, and the result has been more solid and performant than expected.

Thanks to Gambit's compilation model, the whole application can be compiled in a single executable and runned on platforms that don't support dynamic loading of libraries. On the other hand, the possibility of mixing compiled and interpreted code has been proved very useful for rapid prototyping and testing. The interaction between a multithreaded Scheme code with a C library (sqlite), has also worked quite well, using the sqlite bindings provided by Klio tools, based on a functional interface[3]. The old C cgi continued to works, thanks to Gambit process ports, so the new interface has been tested and benchmarked against the old one, showing better performances and reliability.

## 6. CONCLUSIONS AND FUTURE WORKS

This experience has confirmed the qualities of Lisp in creating source to source compilers and server applications.

In particular, Gambit Scheme has proved itself being in a sweet-spot between simplicity, performance, reliability: it has been ported without difficulties on quite uncommon processors, beating legacy C code in performance, and never showing a problem in several month of continuous running, despite the minimal quantity of test and debug required.

## 7. REFERENCES


[1] W. Clinger. Proper tail recursion and space efficieny. *Proceedings of the 1998 ACM Conference on Programming Languages Design and Implementation*, June 1998.

[2] O. Kiselyov. A better xml parsing through functional programming. *Fourth International Symposium on Practical Aspects of Declative Languages (PADL '02)*, January 2002.

[3] O. Kiselyov. Towards the best collection api (extended abstract). *Lightweight Languages 2033 (LL3) workshop*, November 2003.

[4] C. Queinnec. The influence of browsers on evaluators or, continuation to program web servers. *ICFP 2000 - International Conference on Functional Programming*, September 2000.


[5]http://mbenelli.github.com/klio

# Algorithm Engineering with Clojure

Gunnar Völkel
Institute of Theoretical Computer Science
University of Ulm
gunnar.voelkel@uni-ulm.de

Johann M. Kraus
Research Group Bioinformatics & Systems Biology
Institute of Neural Information Processing
University of Ulm
johann.kraus@uni-ulm.de

Hans A. Kestler
Research Group Bioinformatics & Systems Biology
Institute of Neural Information Processing
University of Ulm
hans.kestler@uni-ulm.de

## ABSTRACT

In this paper we present our tools to support Algorithm Engineering with Clojure. These tools support the two steps of Algorithm Engineering: implementation/development and experimental evaluation of algorithms. Based on function definition interception with means of the Clojure language we describe tracing and timing methods which can be set up for a specified set of functions without altering source code. Using a domain specific language (DSL) for experiment description and a related execution method, we can facilitate the experiment evaluation step. Furthermore, the experimental configuration as DSL enables us to exchange parts of the algorithm easily later.

## Categories and Subject Descriptors

D.3.2 [**Applicative (functional) languages**]: Clojure
; D.2.5 [**Testing and Debugging**]: Tracing
; D.2.5 [**Testing and Debugging**]: Debugging aids
; D.2.8 [**Metrics/Measurement**]: Performance measures

## General Terms

Algorithms,Experimentation,Languages

## Keywords

Clojure, Algorithm Engineering,tracing, timing, domain specific language, macros

## 1. INTRODUCTION

Clojure [3] is a relatively new addition to the Lisp family. Clojure runs on the Java Virtual Machine (JVM). Clojure has a strong functional orientation and was designed as general-purpose language. One major goal of Clojure is to facilitate developing multi-threaded (concurrent) software. Therefore, it features built-in immutable data structures, called "persistent data structures". "The Joy of Clojure" [1] can be consulted for a more detailed discussion of Clojure's features.

We chose Clojure because of its concurrency promises, as in our field of interest, namely combinatorial optimization algorithms, parallelization of algorithms is often applied. Also being able to use first-class functions facilitates the implementation of meta-heuristics applied to the different optimization problems that we envision. Finally, we liked the option to be able to use existing Java libraries when needed.

*Algorithm Engineering* is a method for algorithmic research. In Sanders [8] the main process of Algorithm Engineering is described "as a cycle consisting of algorithm design, analysis, implementation and experimental evaluation".

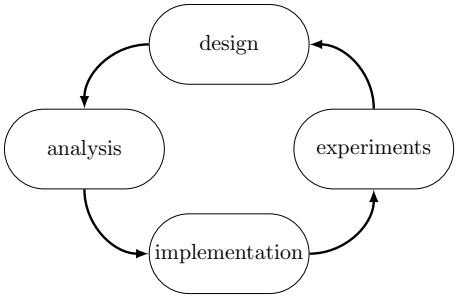



**Figure 1: Algorithm Engineering (simplified)**

In this paper we present tools that facilitate the implementation and experimental evaluation step of Algorithm Engineering with Clojure. For the implementation we provide a general method to add code to existing function implementations for the tracing or timing of functions. This is implemented in Clojure only. No external tools like profilers are needed. The method can be used for additional enhancements as well. For the experimental evaluation step we provide a configuration library and an execution method. The configuration library allows to specify experiments via a domain specific language (DSL). We implemented an execution method to run these experiment configurations. The execution method has a built-in progress report, remaining runtime estimation and automatic persistence of result data.

Our main goal is to support Algorithm Engineering in Clojure. By providing debug facilities, we can support the de-

velopment of algorithms. For easing the evaluation of algorithms we need means to define and to perform experiments as well as to analyze their results.

*Algorithm Development.* Having a strong functional orientation, Clojure organizes its code in functions (standalone ones and implementations of protocols). Hence, we are interested in analyzing the calls to these functions: What parameters were passed to a function? What did the function return? With which parameters were the forms of the function invoked? What did these invocations return? Did they call other functions that we analyze? Monitoring function invocations this way supports finding programming errors more quickly. In addition to that, capturing the runtime of function invocations enables us to detect performance bottlenecks. Finally, we need a general way to inspect Clojure data for displaying the parameters and results of function invocations.

*Algorithm Evaluation.* An Experiment Definition Language (EDL) specifies the structure of an experimental setup by defining the variable parts of an experiment like constant parameters, strategy functions or filenames of input data files that shall be used in the experiments. The EDL acts as a Meta-DSL whose documents generate the Experiment Setup Language (ESL) that can be used to define a concrete experimental setup. With the ESL we can specify a setup that defines a series of experiments with different parameters.

We also wanted to be able to execute such an experimental setup. Generating experiments from our setup scenarios should be automatically possible with two different methods: a Cartesian product like method and a sampling method. Furthermore, the execution should be performed in an execution environment that already provides data persistence for experiment results, a progress report along with a remaining runtime estimation. We also wanted to be able to utilize a given number of cores on a multi-core processor system to run the experiments in parallel.

This paper is organized as follows. In Section 2 we review existing tools for Clojure assisting the implementation and experiment evaluation step. The development tools are described in Section 3. Section 4 contains the description of the experiment evaluation tools.

# 2. EXISTING TOOLS

According to Clojure's official IRC channel a common debug technique is best described by the following listing.

```
(println "data␣=" data)
```

That means many Clojure developers print out data at the points where they suspect an error. A less frequent suggestion is the `(break)` macro from Swank Clojure[1] or the `(debug-repl)`[2] macro from which it is derived. Both start an interactive REPL in the context where the macro was specified. That means one can access all the local bindings to inspect their values and evaluated function calls with them.

When using Eclipse with the Clojure development plug-in Counterclockwise[3] breakpoints are available and also a stepping through the code inspecting the values of local bindings is possible. For profiling the runtime of functions also no Clojure specific tools are available.

To our best knowledge currently no Clojure tools exist to perform experimental algorithm evaluation based on a domain specific language configuration.

# 3. DEVELOPMENT TOOLS

Our development tools are based on the ability to intercept Clojure function definitions during compilation. Built on that we wanted to add code during the compilation to the function implementation for tracing function calls or measuring function runtimes. In the following we will describe our function interception method and its application for tracing and timing.

## 3.1 Intercepting Function Definitions

Most named functions in Clojure are defined via the `defn` macro. Intercepting a function definition is done by replacing Clojure's `defn` macro with the `defn-intercept` macro. For the technical description of function interception the following assumption is made: `defn-intercept` is used to define the function we want to intercept. We will present more convenient configuration options for interception later.

Assuming we have a fixed function *`intercept-fn`*, we can define `defn-intercept` like in the following listing.

```
(defmacro defn-intercept [fsymb, & fdecl]
  (let [{:keys [meta-map, body-list]}
          (process-defn-decl fdecl),
        modified-bodies
          (map intercept-fn body-list)]
    `(~defn ~fsymb ~meta-map
       ~@modified-bodies)))
```

The function `process-defn-decl` gathers the given meta data and the specified list of implementation bodies like `defn` would do. Then we apply our intercept function to all implementation bodies. Finally, we return a form that defines the function with the altered implementations via `defn`.

Now that we know how to intercept our function definitions, it seems inconvenient to be forced to replace `defn` with `defn-intercept` for every function we want to intercept. Furthermore, we do not want to be forced to create a version of `defn-intercept` for every interception use case (having its own interception function). Hence, `debug.intercept` namespace implements a registry that maps function symbols to a list of interception functions which shall be applied to the related function. These interception functions are applied as a composition where the first one is applied first. We implemented a macro that generates an interception setup macro for a given interception function `f` that can be used after the `ns` statement of a Clojure file. This setup macro

[1] Swank Clojure is a server that allows SLIME (Superior Lisp Interaction Mode for Emacs) to connect to Clojure projects.

[2] from George Jahad https://github.com/GeorgeJahad/debug-repl

[3] http://code.google.com/p/counterclockwise/

is invoked with the symbols of the functions that shall be intercepted with the function `f`. Assume we define an interception setup macro `trace-setup` in namespace `trace` with

```
(ns trace)
(defn trace-intercept-fn
  [fns, fsymb, params-body] ...)
(create-setup
  trace-setup, trace-intercept-fn)
```

then we can use it in a namespace to intercept a function like the following listing demonstrates.

```
(ns example (:use trace))
(trace-setup square)
(defn square [x] (* x x))
```

This works since `trace-setup` replaces `clojure.core/defn` with `debug.intercept/defn-intercept` for the namespace it is invoked in. Our macro `debug.intercept/defn-intercept` is then called for every `defn` in that namespace and intercepts registered functions but defines unregistered ones normally.[4] The replacement is done similar to the following listing.

```
(ns-unmap *ns* 'defn)
(refer 'debug.intercept
  :only [defn-intercept]
  :rename {defn-intercept defn})
```

With this method of specifying interceptions we do not need to change the function definitions, but we still need to alter a source file. We can improve this by enabling the usage of an external configuration file. Such a configuration file looks like the following listing provided there are functions `square` and `sqrt` within the namespace `example`.

```
(enable-intercept true)
(trace example/square example/sqrt)
```

The configuration file consists of Clojure forms. The first form can be used to enable function interception which is disabled by default.

The following example scenario shows how to set up function interception via an external configuration using the widespread Clojure development tool *Leiningen* [2]. Leiningen is a command line tool whose first parameter determines the task to execute, e.g. `lein repl` launches a Clojure REPL within the context of the current project if there is a configuration file called `project.clj` in the current working directory. In this configuration file Leiningen allows to specify code that is run before anything else is done within the project, e.g. before the above REPL task evaluates Clojure forms. We can use that configuration to set up our interception with the project configuration like follows.

```
(defproject Examples 0.1
  :project-init
  (do (use 'debug.intercept)
    (setup-global-interception
      "config.clj" '[debug.trace/trace])))
```

In this case our configuration file is *config.clj* and we support a `trace` configuration form within that file. The function `setup-global-interception` loads the given configuration commands (in this case `trace` from namespace `debug.trace` ) so that they can be used in the configuration file. The configuration file is run and finally the root binding to Clojure's `defn` is changed to our `defn-intercept`.

```
(alter-var-root #'clojure.core/defn
  (constantly (var-get #'defn-intercept)))
```

Every time we start a REPL with Leiningen using this setup the functions specified in the configuration file will be intercepted as soon as they are loaded, e.g. when `(use 'example)` is executed, the function `square` will be compiled with trace support. Although this scenario is written for Leiningen, something similar can be set up for every other development environment by using `setup-global-interception` during REPL initialization.

We can go even further and call `setup-global-interception` in the main method of a jar file of a Clojure project. Provided that only the Clojure file containing this main method is compiled and all other Clojure files are included as sources the interception configuration can be changed before a run of the jar file in order to choose different functions to trace.

## 3.2 Tracing Functions

We can now use function interception to implement tracing functionality for functions. Given a set $\mathcal{F}$ of configured functions that shall be traced, the main task of tracing is to create a tree of all function invocations of the functions in $\mathcal{F}$. A node in this tree represents a function invocation of function $f \in \mathcal{F}$ and the children of this node are direct function calls of any function $g \in \mathcal{F}$ invoked by $f$ or indirect function calls starting in $f$ where any number of functions $h_1, h_2, \ldots \notin \mathcal{F}$ may be invoked until $g$ is executed. Each node in the tree has at least information about the invoked function, the parameters used in this particular invocation and the return value of this invocation (or the exception that was thrown). From now on we will call this tree *trace tree*.

For using the function interception we need to write a function with the following signature.

```
(defn trace-intercept-fn
  [fns, fsymb, params-body] ...)
```

This means we get the namespace, name, signature and implementation forms of the function $f \in \mathcal{F}$ that is intercepted. Within `trace-intercept-fn` we can examine and change the implementation forms. The return value of `trace-intercept-fn` will be the implementation of $f$.

When talking about intercepting a function $f \in \mathcal{F}$, we denote the implementation forms of $f$ as $I(f)$ and the return value of the interception function, the implementation that will be used for $f$ instead of $I(f)$, as $I'(f)$.

To enable the building of the trace tree via code addition by the interception function, we have to use Clojure's ability to rebind values of variables via `binding` and its mutable reference type `atom`. Our `trace-intercept-fn` constructs $I'(f) = \Delta_{\text{result}} \circ \Delta_{\text{node}}(I(f))$. Function $\Delta_{\text{node}}$ adds code for three tasks: (1) create an invocation node with information about the given parameters, (2) add the node to the current parent node (if any), and (3) rebind the current parent node

[4] Using the registry is the difference to the previous example for `defn-intercept`.

to this node and execute $I(f)$.

In case of the first invocation, the node is added to a root node collection. Function $\Delta_{\text{result}}$ gets the altered code from $\Delta_{\text{node}}$ and adds code for capturing the result (or the thrown exception) in the invocation node. `binding` is used to rebind the parent node for the execution of $I(f)$. Adding children to the parent node requires a mutable children collection wherefore we use a `vector` within an `atom`. The root node collection is implemented like that as well. The added code contains an `if` form that activates tracing only when specified. We can perform a trace with the `with-trace` macro, e.g. `(with-trace (square 3))`, which initializes the root node collection atom with an empty `vector`, activates tracing, executes the given forms and displays the call tree afterwards. If there are multiple independent calls given in the `with-trace` form, e.g. `(with-trace (+ (square 3)( square 4)))`, we will have a collection of call trees.

Now, we are able to create a tree of function calls. With function interception we can do much better since we can alter the implementation of the intercepted function $f$. In fact we traverse the implementation forms $I(f)$ and wrap every form into code that creates a node that stores with which parameters the form was invoked and what it returned. When doing this we distinguish between forms that are macros and the ones that are functions. For macros we add a special node that contains only one child – the node for the expansion of the macro. In the case of core macros like `let`, `for` or other known macros the expansion node can be skipped. This is implemented via a multimethod that has a default implementation for unknown macros and one implementation for each known macro. With the nodes for the implementation forms of a function when tracing, we can also see what caused the function result.

## 3.3 Timing Functions

Next we want to use function interception for determining the runtimes of configured functions. That means we want to measure the runtimes of these functions and calculate their minimum, average and maximum runtime. Therefore we use a similar implementation approach like the one for tracing. The *timing tree* differs from the tracing tree in that a node is a summary of multiple function calls, i.e. when a function $f$ would have multiple child nodes of $g$ in the tracing tree then it has only one child node of $g$ that contains the timing data. A function $g$ that is called from different other functions $f_1, f_2, \ldots$ has multiple nodes in the timing tree (Figure 2): one node for $g$ as child of each of the $f_1, f_2, \ldots$ nodes. Hence, the tree contains information about the context where the time of a function $g$ is spent. A total summary for each function can be determined from the tree. A node in the timing tree contains the following information: invocation count, minimum, average and maximum runtime.

The added code for timing a function $f$ determines the current timestamp before and after the execution of $I(f)$ and then updates the node for $f$ in the current parent node with the measured duration of $I(f)$. Similar to tracing, the timing functionality is only activated for invocations that are surrounded by a `with-timing` macro but then for all specified functions in the whole call tree resulting from that invocation.

The main advantage of this timing method is that we can limit the functions to investigate in advance so that we still get fast running times of the program. Compared to Clojure's built-in `time` macro which only measures the total runtime of the given form, our timing approach can measure every specified function individually.

## 3.4 Displaying Data

The previous sections explained how to gather tracing and timing data. Now, this section explains briefly how to view that data. We implemented a generic inspection function `debug.inspect/inspect` that is able to display all standard Clojure data types. Additional data types can be added by implementing a multimethod or implementing a certain protocol or Java interface. The `inspect` function will open a Swing user interface which consists of a `JXTreeTable` from the SwingX project [7] which renders the given data. We display a tree in the first column so that composite data can be displayed briefly but when needed the tree node can be expanded to view the details. The second column is used to display the type of the data corresponding to the node or its value.

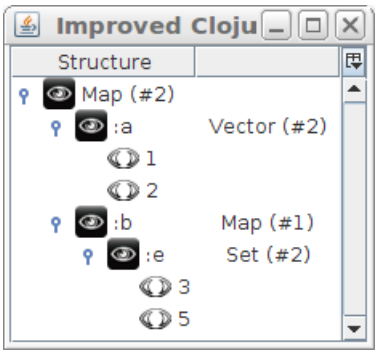



**Figure 3: Screenshot of the inspection GUI. This example shows a map containing a vector and another map which contains a set. The indentation visualizes the containment relation. The map contains a vector with key `:a` that contains two integers.**

We defined a protocol `Inspectable` which has the method `attribute-map`. This method is supposed to return a hash map that maps attribute names (or keywords) to their values. Data that satisfies this protocol can be automatically viewed via `inspect`. For Clojure's `deftype` we have defined an interception that automatically adds an implementation of `Inspectable` to a configured `deftype`.

# 4. EXPERIMENT EXECUTION

Algorithms can have many parameters like number parameters, but also functions that implement different strategies used in the algorithm, e.g. different heuristic functions for a greedy algorithm. We want to set up experiments to evaluate our algorithm in a descriptive form. So, for our experiments we developed a domain specific language (DSL) in Clojure that allows such a descriptive experiment configuration.

Corresponding to that DSL we implemented a runtime environment that takes a configuration instance and executes all

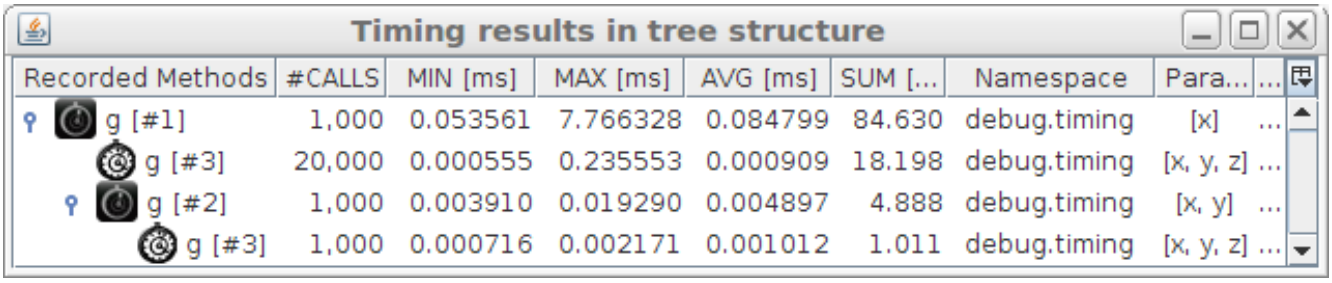



**Figure 2: Screenshot of the timing GUI displaying a function $g$ with different arity implementations that call each other. The implementation of function $g$ with 3 parameters is invoked 20,000 times from the one with one parameter. All calls sum up to a total runtime of about 18 milliseconds. The same implementation is also called from the implementation with 2 parameters.**

experiments defined in it. The runtime environment includes automatic persistence for the result data of the experiments and a progress report that also estimates the remaining runtime.

Finally, there is a function that takes a description on how to build Incanter[5] [4] datasets from the experiment result data and creates these datasets incrementally[6] to be able to handle a large number of experiments and their result data. Generating an Incanter dataset facilitates visualization of the results since Incanter provides easy access to many types of charts from the JFreeChart [6] library and basic manipulation operations for its datasets.

## 4.1 Configuration

The experiment configuration is generic to make it applicable to any algorithm written in Clojure. In order to be able to configure algorithms the configuration workflow has two steps (Figure 4). First we have to define the *experiment definition* for our specific algorithm with the *Experiment Definition Language* (EDL) we implemented. Each algorithm we want to evaluate needs only one experiment definition file. This experiment definition will generate an algorithm specific *Experiment Setup Language* (ESL). In the second step we need to describe a concrete *experimental setup* via the Experiment Setup Language. The experimental setup can then be used to execute a series of experiments.

### 4.1.1 Experiment Definition Language

The Experiment Definition Language consists of two definition statements: a configuration definition and a parameter definition. The following listing shows a configuration definition with two parameters.

```
(CONFIG HEURISTIC :heuristic
  (PARAM ALPHA :alpha :value :float)
  (PARAM EVAL-FN :eval-fn :function [x]))
```

The semantics of the configuration are `(CONFIG <name> <id> <param-list>)` where `<name>` will be the macro name of the configuration in the generated ESL, `<id>` is the configuration type id and `<param-list>` is a list of parameter definitions. The semantics of the parameter definition are

[5]Incanter is a Clojure library that implements an R-like statistical environment.
[6]The persisted result data is read incrementally.

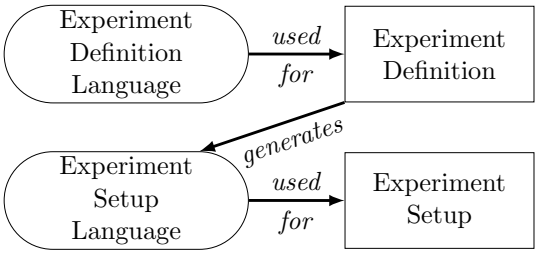



**Figure 4: Configuration workflow.** An Experiment Definition is written in terms of the Experiment Definition Language. The Experiment Setup Language is generated from the Experiment Definition. An Experiment Setup is defined with the Experiment Setup Language.

`(PARAM <name> <id> <type> <type-spec>)` where `<name>` will be the macro name of the parameter in the generated ESL, `<id>` is an internal id and `<type>` specifies which *type* of values the parameter can have. The following types are defined: `:value`, `:function` and `:config`. The *type specification* `<type-spec>` is optional and has a different semantic and different values depending on which type the parameter has.

For the type `:value` the type specification can have the values `:string`, `:float`, `:integer`, `:boolean` and `:keyword`. Then the parameter declaration in ESL expects string values, floating point number values, integer values, Boolean values or Clojure keywords, respectively. The type `:function` defines that the parameter declaration in ESL expects function symbols and the type specification can contain a signature these functions must satisfy. The signature can only be checked if the functions were defined via the library's `defn-meta` macro. The check is limited to the arity – the parameter names are only for documentation purpose. The type `:config` defines a parameter that is another configuration and the type specification can contain a configuration type id which allows only configuration instances with this configuration type id for the parameter. The associated checks for the type specification `<type-spec>` will be checked by the ESL when defining an experimental setup.

The ability to have parameters that are configurations themselves allows individual structuring of the configuration as suited. Though, for the execution of the experimental setup

we need a known configuration. Therefore the following configuration is defined:

```
(CONFIG EXPERIMENT-SETUP :experiment-setup
  (PARAM REPETITION-COUNT :repetition-count
    :value :integer)
  (PARAM INSTANCES :instances
    :config :instance-description)
  (PARAM PARAMETERS :parameters
    :config :experiment-parameters)
  (PARAM OUTPUT :output
    :config :output-files)
  (PARAM? GENERATION-MODE :generation-mode
    :value :keyword :cartesian)
  (PARAM? SAMPLE-COUNT :sample-count
    :value :integer))
```

The `EXPERIMENT-SETUP` configuration contains a parameter `PARAMETERS`. The values of this parameter are the algorithm specific configuration instances which contain the parameters for the algorithm. This means that an algorithm specific configuration has to be defined similar to the following:

```
(CONFIG ALGORITHM
  [:algorithm :experiment-parameters]
  <parameter-definitions>)
```

In this case `:algorithm` is the configuration type id and `:experiment-parameters` is a configuration category id. As you may notice the used configuration category id is the same keyword as the type specification in the `PARAMETERS` parameter from `EXPERIMENT-SETUP`. For configuration instances that are passed to a parameter in ESL either the configuration type id or the configuration category id must match. `<parameter-definitions>` contains all parameters that are passed to the algorithm.

### 4.1.2 Experiment Setup Language

Having defined our experiment we can use the generated Experiment Setup Language. For every configuration definition in EDL (e.g. `ALGORITHM`) two ESL words are generated: the configuration definition word (`ALGORITHM`) and the configuration inheritance word (`ALGORITHM<--`). The configuration definition word (`ALGORITHM`) is used to define an instance of the corresponding configuration. The configuration inheritance word (`ALGORITHM<--`) is used to derive a new configuration instance from another one by taking all the specified parameter values from the other one and overwriting only those in the `ALGORITHM<--` form, like `( ALGORITHM<-- "derived""base"<parameters>)`. For every parameter definition in EDL (e.g. `ALPHA`) four different ESL words are generated. The first two words are used to specify values for a parameter either as `(ALPHA 1.0, 2.0, 4.0)` or as Clojure collection, e.g. `(ALPHA* [1.0, 2.0, 4.0])` or even `(ALPHA* (range 0.5 1.0 0.1))`. The other two words `ALPHA-???` and `ALPHA-???*` specify that sampling is applied to the values of this parameter. They are used like the ones without `-???`. The experiment execution will apply parameter sampling to the parameters marked with `-???` only if the optional parameter `GENERATION-MODE` is given in the `EXPERIMENT-SETUP` configuration instance. The value of this parameter is either `:cartesian` (no sampling) or `:latin-hypercube-sampling`.

Now that we know the words of our generated Experiment Setup Language we use them to define concrete experimental setups that can be executed. Assume we have our previous `HEURISTIC` configuration and defined our main configuration as follows:

```
(CONFIG ALGORITHM
  [:algorithm :experiment-parameters]
  (PARAM USE-HEURISTIC :use-heuristic
    :config :heuristic))
```

The following listing shows an example of an experimental setup.

```
(HEURISTIC "Std"
  (ALPHA 1.0 3.0)
  (EVAL-FN clojure.core/identity))

(HEURISTIC "Greedy"
  (ALPHA* (range 1.0 2.0 0.1))
  (EVAL-FN example/greedy))

(ALGORITHM "Std+Greedy"
  (USE-HEURISTIC "Std", "Greedy"))

(EXPERIMENT-SETUP
  (REPETITION-COUNT 1)
  (PARAMETERS "Std+Greedy")
  (INSTANCES <instance-list>)
  (OUTPUT <output-file-definition>))
```

The previous example defines two different `HEURISTIC` configuration instances that are used in the same `ALGORITHM` configuration. So the algorithm will be run with both instances. From a semantic point of view it would make more sense to define two `ALGORITHM` instances one for each `HEURISTIC` instance. However, both approaches are possible. The `EXPERIMENT` instance specifies a repetition count of one for every experiment on every instance, a list of instances and a definition of where to store the result data of the experiments.

Since parameters of type `:config` only need a configuration type or category id of the configuration instance that is used, we can easily organize our configurations in different files provided we make sure that all are loaded. We have a function implemented which loads all configuration files from a file system directory, recursively if needed. The configurations will be linked together according to parameter type and configuration instance id before the execution is started.

## 4.2 Execution

First we have a look at the structure of the experiments defined by a given experimental setup. The experiments that will be executed are derived from the experimental setup. Remember that an experimental setup has a list of parameter values for each parameter (sometimes with only one element). An experiment needs exactly one value for each parameter. There are two methods of deriving the experiments. The first is to determine something like a Cartesian product on the configuration tree which is the experimental setup. If the configuration instance that is specified in `PARAMETERS` has no tree structure, then a cartesian product will be used. In case of a tree structure one configuation instance is selected for each configuration parame-

ter[7]. Metaphorically speaking, one child node (configuration instance) is selected for each inner node (configuration instance with configuration parameter) in the tree. Finally, for all the resulting trees a cartesian product of their parameters is calculated. In our previous example that would create a series of two experiments with the *"Std"* heuristic and a series of ten experiments with the *"Greedy"* heuristic. The second method is to use parameter sampling. Currently, we have implemented *Latin Hypercube sampling* [5] but other sampling methods can be added. When sampling is used the parameter values of the parameters specified with the suffix `-???` or `-???*` will be sampled. The sampling method first calculates all configurations determined by the parameters that are not sampled via the Cartesian method. For each of the resulting configurations sampling is then applied for the selected parameters. A sample count has to be given in the `EXPERIMENT` configuration.

Now that we have described how the experiments are generated, we need to explain the whole execution process. The Cartesian method or the Latin Hypercube sampling creates experiments from the experimental setup. For each of these experiments we run the algorithm on all the given instances as many times as specified in the repetition count value of the `EXPERIMENT` configuration. The execution function that implements the whole experiment execution has a similar definition as the following.

```
(defn execute-experiment [experiment-name,
    instance-exec-fn] ...)
```

The `experiment-name` specifies which experimental setup is used and the `instance-exec-fn` function must have the following signature.

```
(defn execute-algo [instance, parameters])
```

The parameter `instance` contains an instance description that is used to load or create the desired problem instance. The `parameters` contain the experiment parameters whose creation we described before. The function `execute-algo` is supposed to set up and run the algorithm which we are evaluating. The result value of this function has to be the result data of the algorithm which is automatically stored by our experiment library. The file locations to store the result data are specified via the `OUTPUT` parameter of the `EXPERIMENT` configuration.

The function `execute-experiment` has an option to specify the number of experiments that shall be executed in parallel. Hence, we use a Clojure agent for progress reports after each finished experiment run on an instance. The progress report includes a built-in estimation of the remaining time that will be need to finish all experiments.

## 4.3 Result analysis

Since we can use Incanter datasets for data manipulation (grouping, selection, min/avg/max determination, ...) and creating charts easily, we provide a simple way to create those datasets from the stored result data. We define a dataset with a vector containing an id for the dataset and the functions that create its columns. Each of these functions receives the result data and can create one or more columns by returning a hash map where the keys are the column names and the values are the column values. These dataset definitions can be passed to our `create-experiment-datasets` function which then incrementally loads the data from the result files and creates the rows of the datasets – one row in each dataset per result data object. The incremental approach allows us to work on large result files provided our result datasets fit into memory. The dataset creation can also be performed in parallel which is interesting especially if the result data consists of compressed result data objects.

[7]Parameter with type `:config`

# 5. CONCLUSION

We presented tools that support Algorithm Engineering with Clojure. In particular, we support the algorithm development and the experimental algorithm evaluation step.

For the algorithm development step we have implemented tools based on function definition interception. Function definition interception is possible due to Clojure's macros that allow inspection and changing of code at compile time. Function definition interception can be done using three different scopes: *per function* by using an interception macro instead of `defn`, *per namespace* by using a setup macro with a set of function names to intercept at the beginning of the namespace, and *globally* using a configuration file for the whole project by executing a setup function in a central initialization, e.g. `:project-init` as in Leiningen's `defproject`. For a set of specified functions the tracing allows to inspect the call tree consisting of these functions. Furthermore, that call tree contains also trace nodes for the expressions implementing the specified functions. The timing functionality we presented allows to measure the runtime of the specified functions and builds a tree of timing nodes that represents the call hierarchy, but also summarizes calls to the same function that have the same parent node. The timing nodes contain information about invocation count and min/avg/max runtimes. To view the collected data a generic Swing control was implemented that renders the data in a tree table.

For the experimental algorithm evaluation we implemented a Experiment Definition Language (EDL) which is used to describe an experiment for a given algorithm and then generates an Experiment Setup Language (ESL) for that algorithm. The ESL is then used to defined particular experimental setups (i.e. series of experiments) that can be run with an execution function we implemented. That execution function has a built-in progress report with remaining time estimation and stores experiment result data automatically into files given in the experimental setup. We implemented a function that creates Incanter datasets from the result data when given a definition of these datasets. A dataset is defined in terms of a collection of functions that create the column data. Incanter datasets can then be used to create charts or to analyze the results with the Incanter library.

In our practical algorithm development our tracing and timing tools were very useful. In fact, this practical algorithm engineering aspect started the whole project. There is also the idea to use function definition interception for wrapping a `(debug-repl)` around specified functions for analysis which is not implemented, yet. The experiment configuration via DSL was very useful for our past experimental

evaluations. Especially, if heuristic functions exist that can have different implementations, the configuration via DSL allows us to configure a new heuristic function in the ESL.

# 6. REFERENCES


[1] M. Fogus and C. Houser. *The Joy of Clojure*. Manning Publications Co, 2011.

[2] P. Hagelberg. Leiningen. https://github.com/technomancy/leiningen, 2012.

[3] R. Hickey. Clojure. http://clojure.org, 2012.

[4] D. E. Liebke. Incanter. http://incanter.org, 2012.

[5] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 42:55–61, February 2000.

[6] Object Refinery Ltd. JFreeChart. http://www.jfree.org/jfreechart, 2012.

[7] Oracle. SwingX. http://swinglabs.org, 2012.

[8] P. Sanders. Algorithm Engineering – An Attempt at a Definition. In *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2009.

# Session IV

# Object enumeration

**Irène A. Durand**
idurand@labri.fr
LaBRI, CNRS, Université de Bordeaux, Talence, France

## ABSTRACT
We address the concrete problem of enumerating sets of discrete objects. Enumerating sets of objects is essential when the sets are infinite or too large to be computed in extenso. We give an abstract data type for the concept of an *enumerator* of objects. We give a Lisp implementation of many useful general enumerators and some more particular ones in the framework of terms and term automata.

## Categories and Subject Descriptors
D.1.1 [**Software**]: Programming Techniques, Applicative (Functional) Programming; F.1.1 [**Theory of Computation**]: Models of Computation, Automata

## Keywords
Enumeration, Term, Term automata, Lisp, graphs

## 1. INTRODUCTION
Search problems with large answers raise algorithmic questions in many domains of computer science. Typical examples for such tasks are database queries extraction problems for large data collections like the web, and answers to constraint satisfaction problems [14].

The general question is how to deal with search problems with large answer sets. The first approach which has found much interest in the last five years is answer enumeration [9]. The objective here is to allow a user to quickly inspect some answers in order to avoid the computation of all answers. So the problem of enumerating objects is well-motivated.

Another case where enumeration is essential is when we are dealing with *infinite* or too large sets of objects which can not be computed in extenso.

An *enumerator* object can be used to produce, **one by one**, the values $s_n$ of a sequence $(s_n)_{\mathbb{N}}$.

In this paper, we do not address the whole search problem but we propose basic tools for enumeration which could be used in the search framework.

```
enum Suit {
   Diamonds = 5,
   Hearts,
   Clubs = 4,
   Spades
};
```

**Figure 1: Enumerator in `C++`**

We can make a parallel with enumerative combinatorics [11] which provides functions for counting objects. Combining *counting functions* for basic objects with adequate operators, one can obtain new counting functions for more complicated object.

Here we want to provide *enumerators* for enumerating objects and adequate operators for combining enumerators for basic objects in order to obtain enumerators for more complicated ones.

Our enumerators must not be confused with what is called enumerator in `C++` or `C#` which is just a finite sequence of named integers as shown in Figure 1.

Some programming languages (`Java`, `Lisp`) provide libraries for enumerating simple objects such as lists, arrays, strings, hash tables but no operation to combine these simple enumerators in order to build enumerators for more complex user defined objects. The Sage [12] software is a free open-source mathematics software system licensed under the General Public License. It combines the power of many existing open-source packages into a common Python-based interface. It implements enumeration for complex objects like *graphs*, *posets*, etc.

However, Sage does not handle terms and term automata which are the objects we are currently interested in. Also, we would like a Lisp implementation rather than a Python one.

The `SERIES Lisp` package by Richard C. Waters[10] also deals with finite or infinite sequences. However, these series are not enumerators in the sense that a series or a finite consecutive part of it is treated as a whole. There is no explicit cursor that moves along the elements of a series and which is accessible to the user. However, such cursors must exist in the implementation otherwise an operation like computing the cartesian product of two series would not be possible.

Many features provided by our enumeration package are essentially the same as the ones provided by the SERIES package. The essential difference is that an enumerator points to a current element of its underlying sequence. A series is a functional object while an enumerator is a state machine (a non functional object). In other words, there may be several enumerators, pointing to different elements of the same (possibly virtual) sequence while there is just one series for a given sequence.

Our package also gives an object-oriented version of these concepts while the SERIES package does not use CLOS (Common Lisp Object System). This gives object-oriented extensibility that we do not have with the SERIES package. We use this extensibility when we define enumerators for enumerating terms (see Section 6.2) or enumerators for enumerating the accessible states of an automaton on a given term as presented in Section 6.3.

The infinite lists of Haskell[1] offer the same kind of possibilities as the SERIES package.

The ENUMERATION [8] package in extensions of Common Lisp is closer to what we would like to have as it provides a notion of position in the sequence. However, only basic enumerators can be created and there are no operations to derive new enumerators from existing ones.

XProc [15] is an XML pipeline language, for describing operations to be performed on XML documents. It is a language for building pipelines. A pipeline processes XML documents. A process is composed of various operations. These operations are performed sequentially. Enumerators and pipelines are related in the sense that both output a sequence of data: the enumerated objects in the enumerator case and an XML stream in the case of an XProc pipeline. A pipeline accepts as input one or more XML streams and outputs an XML stream. The pipeline performs some operations on the input streams to create the output stream. Pipelines can be combined to create new ones like enumerators can be combined to create new ones. However, as for series, the pipeline operations are linear in the sense that an input stream cannot be restarted and considered several times.

The purpose of this article is to define an abstract data type for the concept of an enumerator of the objects of a set, to provide a Lisp implementation for many useful general enumerators and to show some applications in the framework of terms and term automata.

## 2. THE ENUMERATOR ABSTRACT DATA TYPE

From a mathematical point of view, an *enumerator* is a state machine whose states are the elements of a sequence

$$(E_n)_{n\in\mathbb{N}}$$

of elements $e_0, e_1, \ldots$ of a given type. An enumerator can be finite

$$(E_n)_{n\in[0,N]}, N \in \mathbb{N}$$

or infinite

$$(E_n)_{n\in\mathbb{N}}.$$

The main operation applicable to an enumerator $E$ is

`call-enumerator`$(E)$

which returns two values: $(e, true)$ if there exists a next element $e$ in the sequence, $(false, false)$ otherwise. This operation is destructive as it changes the internal state of the enumerator.

On othe words, the first call returns $(e_0, true)$, the second call $(e_1, true)$ and so on. If the sequence of length $n$, the $n$-th call returns $(e_{n-1}, true)$ and all the subsequent ones $(false, false)$.

An enumerator $E$ can be reinitialized with a call to

`init-enumerator`$(E)$

in order to start the enumeration from the beginning again. For convenience, `init-enumerator` will return the enumerator. This operation is **destructive** as it changes the internal state of the enumerator back to the initial state.

The operation

`copy-enumerator`$(E)$

returns a reinitialized copy of a given enumerator $E$.

```
(defclass abstract-enumerator () ())

(defgeneric init-enumerator (enumerator)
  (:documentation
   "reinitializes and returns ENUMERATOR"))

(defgeneric copy-enumerator (enumerator)
  (:documentation
   "returns a reinitialized
    copy of ENUMERATOR"))

(defgeneric next-element-p (enumerator)
  (:documentation
   "returns NIL if there is no next
    element, a non NIL value otherwise"))

(defgeneric next-element (enumerator)
  (:documentation
   "returns the next element,
    moves to the following one"))

(defgeneric call-enumerator (enumerator)
  (:documentation
   "if there is a next element e,
    returns e and T
    and moves to the next element;
    otherwise returns NIL and NIL"))
```

**Figure 2: API for the `enumerator` abstract data type**

The `call-enumerator` operation is implemented for all enumerators using the two lower level operations `next-element-p (enumerator)` and `next-element (enumerator)`. This is shown in Figure 3.

The `init-enumerator` operation is implemented for all enumerators as shown in Figure] 4. But it is meant to be completed by secondary methods (`:after` methods most often) according to the internal definition of a concrete enumerator. See for instance,

`make-*-enumerator*`$(\ldots)$

```
(defmethod call-enumerator
    ((e abstract-enumerator))
  (if (next-element-p e)
      (values (next-element e) t)
      (values nil nil)))
```

**Figure 3: Implementation of `call-enumerator`**

```
(defmethod init-enumerator
   ((e abstract-enumerator))
  e)
```

**Figure 4: Implementation of `init-enumerator`**

operations to create specific enumerators relying or not on other enumerators.

The Application Program Interface is given in Figure 2.

# 3. SIMPLE ENUMERATORS

## 3.1 Enumerator of the elements of a list

The elements of the sequence are the elements of the list. The order is the order of the elements of the list. An example of use of such enumerator is given in Figure 5.

The class `list-enumerator` shown in Figure 6 represents this kind of enumerator. The slot `initial-list` keeps a pointer to the beginning of the initial list while the slot `current-list` points to the rest of the list of elements to be enumerated.

The operations of the `API` are easily implemented as shown in Figure 7.

To create such a list enumerator we provide the

`make-list-enumerator`(`l &optional circ`)

function. With the optional parameter `circ` set to `T`, we get an infinite enumerator which circularly enumerates the objects of `l` as shown in Figure 8.

An example of use of a circular list enumerator is given in Figure 9.

## 3.2 Enumeration of a sequence defined by a linear induction

Often a sequence $(s_n)_{n\in\mathbb{N}}$ is defined recursively with an initial value $v$ and a function $f$ which computes the next element from the previous one:

$$\begin{cases} s_0 &= v \\ s_{n+1} &= f(s_n) \quad \text{if } n > 0 \end{cases}$$

EXAMPLE 3.1. *For instance, with the initial value $v = 1$ and*

$$\begin{cases} f: & \mathbb{N} \to \mathbb{N} \\ & x \mapsto 2 + x \end{cases}$$

*we obtain the sequence of odd natural integers.*

$$1, 3, 5, \ldots$$

```
ENUM> (setf
        *e*
        (make-list-enumerator
         '(1 2 3)))
#<LIST-ENUMERATOR {1004711A71}>
ENUM> (call-enumerator *e*)
1
T
ENUM> (call-enumerator *e*)
2
T
ENUM> (init-enumerator *e*)
#<LIST-ENUMERATOR {1005AF15F1}>
ENUM> (call-enumerator *e*)
1
T
ENUM> (call-enumerator *e*)
2
T
ENUM> (call-enumerator *e*)
3
T
ENUM> (call-enumerator *e*)
NIL
NIL
ENUM> (call-enumerator *e*)
NIL
NIL
```

**Figure 5: Example of a list enumerator**

```
(defclass list-enumerator
      (abstract-enumerator)
  ((initial-list :type list
                 :initarg :initial-list
                 :reader initial-list)
   (current-list :type list
                 :initarg :current-list
                 :accessor current-list))
  (:documentation
   "enumerators of the
    elements of a list"))
```

**Figure 6: Class for list enumerators**

```
(defmethod init-enumerator :after
    ((e list-enumerator))
  (setf (current-list e) (initial-list e)))

(defmethod copy-enumerator
    ((le list-enumerator))
  (make-list-enumerator
   (initial-list le)))

(defmethod next-element-p
    ((le list-enumerator))
  (current-list le))

(defmethod next-element
    ((le list-enumerator))
  (pop (current-list le)))
```

**Figure 7: Operations for `list-enumerator`**

```
(defun ncirc (l) (nconc l l))
(defun circ (l) (ncirc (copy-list l)))

(defun make-list-enumerator
    (l &optional (circ nil))
  (when circ
    (setf l (circ l)))
  (make-instance
    'list-enumerator
    :initial-list l :current-list l))
```

**Figure 8: Circular list enumerator**

```
ENUM> (setf
       *e*
       (make-list-enumerator '(1 2) t))
#<LIST-ENUMERATOR {1004742B71}>
ENUM> (call-enumerator *e*)
1
T
ENUM> (call-enumerator *e*)
2
T
ENUM> (call-enumerator *e*)
1
T
```

**Figure 9: Example of a circular list enumerator**

This kind of enumerator is implemented by the class

`inductive-enumerator`

shown in Figure 10. The slot `init-value` holds the initial value $v$ while the slot `current-value` holds the next element to be enumerated. Initially, the current value is the initial value. The function $f$ is stored in the slot `fun` brought over by inheritance with the `fun-mixin` class.

The rest of the implementation is straightforward and given in Figure 11.

A variant of an inductive enumerator which has an additional function $g$ to be applied after the function $f$ which performs the inductive step, can be easily defined. $g$ is often a modulo kind of function. Part of the implementation is given in Figure 12. `after` methods are used for `init-enumerator` and `next-element` in order to apply the modulo function.

```
(defclass fun-mixin ()
  ((fun :initarg :fun :reader fun)))

(defclass inductive-enumerator
      (abstract-enumerator fun-mixin)
  ((init-value
    :initarg :init-value
    :accessor init-value)
   (current-value
    :initarg :current-value
    :accessor current-value)))
```

**Figure 10: Inductive enumerators**

```
(defmethod init-enumerator
    ((e inductive-enumerator))
  (setf (current-value e)
        (init-value e))
  e)

(defmethod next-element-p
    ((e inductive-enumerator))
  t)

(defmethod next-element
    ((e inductive-enumerator))
  (setf
   (current-value e)
   (funcall
    (fun e)
    (current-value e))))

(defun make-inductive-enumerator
    (init-value fun
     &optional current-value)
  (unless current-value
    (setf current-value init-value))
   (make-instance
    'inductive-enumerator
    :fun fun
    :init-value init-value
    :current-value current-value))
```

**Figure 11: Code for inductive enumerator**

```
(defclass mod-inductive-enumerator
   (inductive-enumerator)
  ((mod-fun :initarg :mod-fun
            :reader mod-fun)))

(defmethod init-enumerator :after
    ((e mod-inductive-enumerator))
  (setf (current-value e)
        (funcall
         (mod-fun e)
         (current-value e))))

(defmethod next-element :after
    ((e mod-inductive-enumerator))
  (setf (current-value e)
        (funcall
         (mod-fun e)
         (current-value e))))

(defun make-mod-inductive-enumerator
    (initial-value fun mod-fun
     &optional current-value)
  (unless current-value
    (setf current-value initial-value))
  (make-instance
   'mod-inductive-enumerator
   :fun fun
   :mod-fun mod-fun
   :initial-value initial-value
   :current-value current-value))
```

**Figure 12: Modulo inductive enumerator**

```
ENUM> (setf
       *e*
       (make-mod-inductive-enumerator
        0 #'1+ (lambda (x) (mod x 2))))
#<MOD-INDUCTIVE-ENUMERATOR {1002A7E1A1}>
ENUM> (call-enumerator *e*)
0
ENUM> (call-enumerator *e*)
1
ENUM> (call-enumerator *e*)
0
ENUM> (call-enumerator *e*)
1
```

**Figure 13: Example of a modulo inductive enumerator**

EXAMPLE 3.2. *With such an enumerator one can easily generate an infinite sequence of the form*

$$0, 1, \ldots n-1, 0, 1, \ldots$$

*as shown in Figure 13 with* $n = 2$*.*

# 4. ENUMERATOR RELYING ON OTHERS

We denote by $\mathbb{E}$ the set of enumerators. Enumerators may be combined to obtain new enumerators. For instance, if we have two finite enumerators $E_1$ and $E_2$, enumerating respectively $e_0^1, e_1^1 \ldots$ and $e_0^2, e_1^2 \ldots$, we may want an enumerator for the concatenation of the two sequences or for pairs of elements of the two sequences (parallel product, cartesian product), etc. In this section, we will define operators applying to enumerators and yielding new enumerators.

DEFINITION 1. *An* operator *is an application*

$$\left\{ \begin{array}{rcl} \mathbb{E}^n & \rightarrow & \mathbb{E} \\ E_1, \ldots E_n & \mapsto & E \end{array} \right.$$

An operator will be implemented as Lisp function taking as input at least one enumerator and returning a new enumerator.

DEFINITION 2. *We say that an enumerator* $E'$ relies *on an enumerator* $E$ *if the enumeration of the objects of* $E'$ *requires the enumeration of some objects of* $E$*. Conversely, we say that* $E$ underlies $E'$.

An enumerator may rely on zero, one, two or more enumerators.

Figure 14 shows the operations on enumerators yielding relying enumerators.

## 4.1 Abstract classes for relying enumerators

The abstract class

`relying-enumerator`

contains the enumerators which rely on at least one enumerator. The

`underlying-enumerators`$(E)$

operation returns the list of enumerators underlying $E$. Initializing such an enumerator requires at least initializing each of the underlying enumerators. This is described in Figure 15.

```
(defclass relying-enumerator
      (abstract-enumerator) ()
  (:documentation
   "enumerators relying on
    at least one enumerator"))

(defgeneric underlying-enumerators
    (relying-enumerator)
  (:documentation
   "enumerators on which
    RELYING-ENUMERATOR relies"))

(defmethod init-enumerator :after
    ((e relying-enumerator))
  (mapc #'init-enumerator
        (underlying-enumerators e)))
```

**Figure 15: Relying enumerators**

```
(defclass unary-relying-enumerator
      (relying-enumerator)
  ((enum
    :type abstract-enumerator
    :initarg :enum
    :reader enum))
  (:documentation
   "enumerator relying on
    one enumerator"))

(defclass nary-relying-enumerator
      (relying-enumerator)
  ((enums :type list :initarg :enums
          :reader underlying-enumerators))
  (:documentation
   "enumerator relying more than two
    enumerators"))
```

**Figure 16: Relying on one, two or more enumerators**

We shall treat the special cases where an enumerator has exactly one underlying enumerator with the class

`unary-relying-enumerator`

and the general case when there is at least one underlying enumerator with the class

`nary-relying-enumerator`.

These classes are depicted in Figure 16.

## 4.2 Concrete classes for relying enumerators

### *4.2.1 Mapping operators*

Suppose that we have an enumerator $E$ which enumerates

$$e_0, e_1, \ldots$$

and a function $f$ such that every $e_i$ is in the domain of $f$. We would like an enumerator $E' =$ `funcall`$(f, E)$ which enumerates

$$f(e_0), f(e_1), \ldots$$

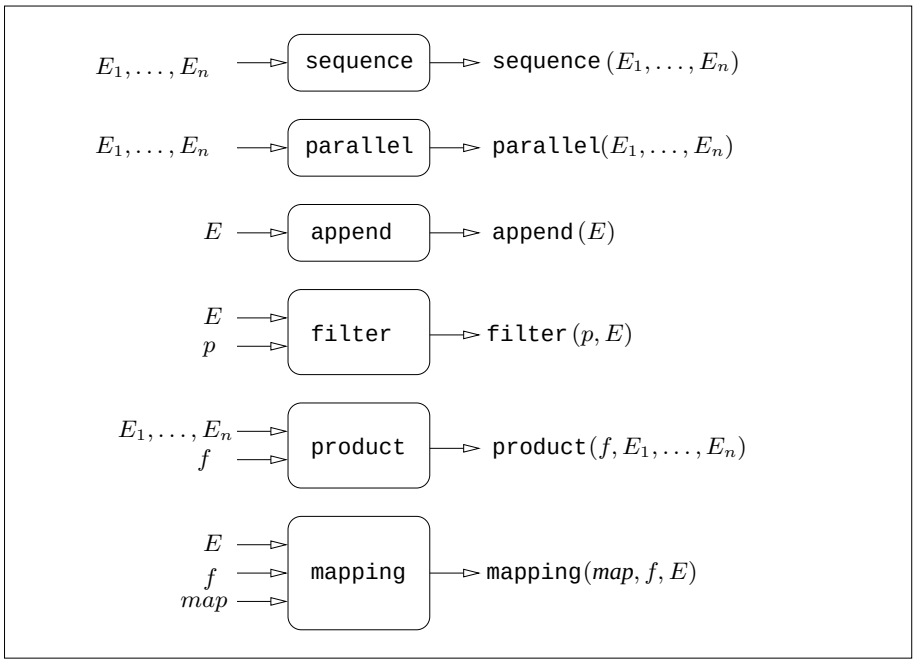



**Figure 14: Basic relying enumerators**

$f$ being a parameter, $f(e_i)$ can be computed using

$$(\texttt{funcall}\ f\ e_i)$$

Similarly, suppose that we have an enumerator $E$ which enumerates the objects

$$l^0, l^1, \ldots$$

such that each $l^i$ is a list $(e_1^i, e_2^i, \ldots, e_{k_i}^i)$ and $f$ is a function taking a variable number of arguments. We would like an enumerator $E' = \texttt{apply}(f, E)$ which enumerates

$$f(e_1^0, e_2^0, \ldots, e_{k_0}^0),$$
$$f(e_1^1, e_2^1, \ldots, e_{k_1}^1), \ldots$$

$f$ being a parameter and

$$l^i = (e_1^i, e_2^i, \ldots, e_{k_i}^i)$$

a list,

$$f(e_1^i, e_2^i, \ldots, e_{k_i}^i)$$

can be computed using $(\texttt{apply}\ f\ l^i)$.

These two kind of enumerators may be obtained through a single *mapping* operator parameterized by a *map function*:

$$\texttt{mapping}(\textit{map}, f, E)$$

The *map* function will often be either

$$\texttt{funcall} \text{ or } \texttt{apply}.$$

The class

```
mapping-enumerator
```

```
(defclass mapping-enumerator
      (unary-relying-enumerator fun-mixin)
  ((map-fun :initarg :map-fun
            :reader map-fun)))

(defun make-mapping-enumerator
    (fun map-fun e)
  (make-instance
   'mapping-enumerator
   :fun fun :map-fun map-fun
   :enum (copy-enumerator e)))

(defmethod next-element-p
    ((e mapping-enumerator))
  (next-element-p (enum e)))

(defmethod next-element
    ((e mapping-enumerator))
  (funcall
   (map-fun e)
   (fun e)
   (next-element (enum e))))
```

**Figure 17: Code for mapping enumerator**

```
(defun make-funcall-enumerator (fun e)
  (make-mapping-enumerator
   fun #'funcall e))

(defun make-apply-enumerator (fun e)
  (make-mapping-enumerator
   fun #'apply e))
```

**Figure 18: `funcall` and `apply` enumerators**

```
(make-funcall-enumerator
 (lambda (x) (* x x))
 (make-inductive-enumerator 0 #'1+))
```

**Figure 19: Squares of natural integers**

corresponds to enumerators obtained with the `mapping` operator and is presented in Figure 17. The $f$ function is stored in the `fun` slot of `fun-mixin`.

The two kinds of enumerators described at the beginning of this subsection are a special case of a mapping enumerator, one using `funcall` as the `map-function` and the other using `apply`. The corresponding code is in Figure 18.

EXAMPLE 4.1. *If we want an enumerator of the squares of the natural integers,*

$$0, 1, 4, 9, 16, \ldots$$

*we may write the expression of Figure 19.*

The Lisp `reduce` function could also be used as a `map-fun`.

### *4.2.2 Sequence enumerator*

Given a sequence of $n$ enumerators $E_1, E_2, \ldots, E_n$, each $E_i$ enumerating

$$e_0^i, e_1^i, \ldots, e_{k_i}^i,$$

we would like an enumerator

$$E = \texttt{sequence}(E_1, E_2, \ldots, E_n)$$

which enumerates first all elements of $E_1$, then all elements of $E_2$ and so on, *i.e.*

$$\begin{array}{l} e_0^1, e_1^1, \ldots, e_{k_1}^1, \\ e_0^2, e_1^2, \ldots, e_{k_2}^2, \\ \ldots \\ e_0^n, e_1^n, \ldots, e_{k_n}^n, \end{array}$$

The `sequence`$(E_1, E_2, \ldots, E_n)$ is finite if and only if every $E_i$ is finite.

EXAMPLE 4.2. *For instance, with $n = 2$, $E_1$ enumerating*

$$a, b, c$$

*and $E_2$ enumerating*

$$1, 2, \ldots$$

*then*

$$\texttt{sequence}(E_1, E_2)$$

```
(defclass sequence-enumerator
      (nary-relying-enumerator)
  ((remaining-enumerators
    :type list
    :initarg :remaining-enumerators
    :accessor remaining-enumerators))
  (:documentation
   "enumerates sequentially the objects
    of each enumerator in
    (underlying-enumerators e)"))

(defmethod move-to-non-empty-sequence
    ((e sequence-enumerator))
  (loop
    while (remaining-enumerators e)
    until
    (next-element-p
     (first (remaining-enumerators e)))
    do (pop (remaining-enumerators e))))

(defmethod init-enumerator :after
   ((e sequence-enumerator))
 (setf (remaining-enumerators e)
       (underlying-enumerators e))
 (move-to-non-empty-sequence e)))

(defun make-sequence-enumerator (enums)
 (assert (not (null enums)))
 (setf enums
       (mapcar #'copy-enumerator enums))
 (make-instance
  'sequence-enumerator
  :underlying-enumerators enums
  :remaining-enumerators enums))
```

**Figure 20: Sequence enumerators**

*enumerates*

$$a, b, c, 1, 2, \ldots$$

The `sequence-enumerator` class implements this kind of sequential enumerator. As such enumerators rely on several enumerators, the class is derived from the class

`nary-relying-enumerator`.

The slot

`remaining-enumerators`

is `NIL` or points to the non empty sublist of underlying enumerators not yet enumerated. The first one is the one being currently enumerated. See Figure 20. The `move-to-non-empty-sequence` operation moves forward the pointer on the remaining underlying enumerators until it reaches the end or until the first enumerator has a next element. This operation is called at initialization and after each call to `next-element`.

The `next-element` operation consists of applying

`next-element`

to the first of the remaining underlying enumerators. When there is no more remaining enumerator, the sequence enumerator has no

```
(defmethod next-element-p
    ((e sequence-enumerator))
  (let ((re (remaining-enumerators e)))
    (and re (next-element-p
             (first re)))))

(defmethod next-element
    ((e sequence-enumerator))
  (prog1
      (next-element
       (first
        (remaining-enumerators e)))
    (move-to-non-empty-sequence e)))
```

**Figure 21: Code for sequence enumerator**

```
(defclass parallel-enumerator
      (nary-relying-enumerator) ())

(defmethod next-element-p
    ((e parallel-enumerator))
  (every #'next-element-p
         (underlying-enumerators e)))

(defmethod next-element
    ((e parallel-enumerator))
  (loop
    for enumerator
      in (underlying-enumerators e)
    collect (next-element
                 enumerator)))
```

**Figure 22: Parallel enumerator**

next element. The rest of code for sequence enumerators is given in Figure 21.

### *4.2.3 Parallel enumerator*

Given a sequence of $n$ enumerators $E_1, E_2, \ldots, E_n$ each enumerating $e_0^i, e_1^i, \ldots, e_{k_i}^i$, we would like an enumerator

$$E = \texttt{parallel}(E_1, \ldots, E_n)$$

which enumerates the tuples

$$\begin{array}{l}(e_0^1, e_0^2, \ldots, e_0^n)\\(e_1^1, e_1^2, \ldots, e_1^n)\\\ldots\\(e_k^1, e_k^2, \ldots, e_k^n)\end{array}$$

where $k = min(k_1, \ldots, k_n)$. The enumerator

$$\texttt{parallel}(E_1, \ldots, E_n)$$

is infinite only if all $E_i$ are infinite.

The parallel enumerator is also easy to implement. The code is given in Figure 22.

# 5. PRODUCT OF ENUMERATORS

## 5.1 Introduction to the problem

Suppose that we have $n$ enumerators $E_1, E_2, \ldots, E_n$ each enumerator $E_i$ enumerating

$$e_0^i, e_1^i, \ldots$$

We would like an enumerator $E = X(E_1, \ldots, E_n)$ enumerating the tuples of the cartesian product of the list of values enumerated by each of the $E_i$ *i.e.*

$$\begin{array}{l}(e_0^1, e_0^2, \ldots, e_0^{n-1}, e_0^n)\\(e_0^1, e_0^2, \ldots, e_0^{n-1}, e_1^n)\\\ldots\\(e_0^1, e_0^2, \ldots, e_0^{n-1}, e_{k_n}^n)\\(e_0^1, e_0^2, \ldots, e_1^{n-1}, e_0^n)\\(e_0^1, e_0^2, \ldots, e_1^{n-1}, e_1^n)\\\ldots\\(e_0^1, e_0^2, \ldots, e_1^{n-1}, e_{k_n}^n)\\\ldots\\(e_0^1, e_0^2, \ldots, e_{k_{n-1}}^{n-1}, e_1^n)\\\ldots\\(e_0^1, e_0^2, \ldots, e_{k_{n-1}}^{n-1}, e_{k_n}^n)\\\ldots\\(e_{k_1}^1, e_{k_2}^2, \ldots, e_{k_{n-1}}^{n-1}, e_{k_n}^n)\end{array}$$

$X(E_1, \ldots, E_n)$ is finite if and only if all $E_i$s are finite. If $E_i$ is infinite then just one element of each of the preceding enumerators $E_1, \ldots, E_{i-1}$ will be enumerated.

To have all the values of all the enumerators potentially enumerated,

$$E_2, \ldots, E_n$$

should all be finite. If $E_1$ is finite the resulting enumerator will be finite, infinite otherwise[1].

EXAMPLE 5.1. *For instance, with $n = 2$, $E_1$ enumerating*

$$1, 2, \ldots$$

*and $E_2$ enumerating*

$$a, b, c$$

*then $X(E_1, E)$ enumerates*

$$(1, a), (1, b), (1, c), (2, a), (2, b), (2, c), \ldots$$

It is no more difficult and it is more general to create an enumerator $E = \texttt{product}(f, E_1, \ldots, E_n)$ enumerating the values of $f$

[1] In the case where, all the enumerators are infinite, we could enumerate the tuples in a diagonal way but we have not treated this case so far because we did not need it right away.

```
(defclass enum-res ()
  ((object :accessor enum-object)
   (found :accessor enum-found))
  (:documentation
   "to store the result of
    call-enumerator"))

(defmethod set-enum-res
    ((res enum-res)
     (e abstract-enumerator))
     (multiple-value-bind (object found)
         (call-enumerator e)
    (setf (enum-found res) found)
    (if found
        (setf (enum-object res) object)
        (slot-makunbound res 'object))))

(defun make-enum-res ()
  (make-instance 'enum-res))
```

**Figure 23: Storing the values returned by `call-enumerator`**

applied to the tuples of the cartesian product *i.e.*

$$
\begin{array}{l}
f(e_0^1, e_0^2, \ldots, e_0^{n-1}, e_0^n) \\
f(e_0^1, e_0^2, \ldots, e_0^{n-1}, e_1^n) \\
\ldots \\
f(e_0^1, e_0^2, \ldots, e_0^{n-1}, e_{k_n}^n) \\
f(e_0^1, e_0^2, \ldots, e_1^{n-1}, e_0^n) \\
f(e_0^1, e_0^2, \ldots, e_1^{n-1}, e_1^n) \\
\ldots \\
f(e_0^1, e_0^2, \ldots, e_1^{n-1}, e_{k_n}^n) \\
\ldots \\
f(e_0^1, e_0^2, \ldots, e_{k_{n-1}}^{n-1}, e_1^n) \\
\ldots \\
f(e_0^1, e_0^2, \ldots, e_{k_{n-1}}^{n-1}, e_{k_n}^n) \\
\ldots \\
f(e_{k_1}^1, e_{k_2}^2, \ldots, e_{k_{n-1}}^{n-1}, e_{k_n}^n)
\end{array}
$$

Then $X(E_1, \ldots, E_n)$ is a particular case of

$$\texttt{product}(f, E_1, \ldots, E_n)$$

taking $f = \texttt{list}$.

While enumerating the values of $E_i$, we must remember the values of $E_1, \ldots, E_{i-1}$. For this purpose, we create a kind of enumerator with a one element memory `memo-enumerator` which remembers the result of the latest call (see Section 5.2).

The class `enum-res` shown in Figure 23 is a class of object to hold the return values of an enumerator. The operation

```
set-enum-res (enum-res enumerator)
```

sets such object with the result of the call to the enumerator.

## 5.2 One-memory enumerator

We define the class of `memo-enumerator` enumerators which will be used when we need the same value of an enumerator several times; this is the case for the product for example. It is a unary relying enumerator having an `enum-res` slot to hold the previous result.

```
(defclass memo-enumerator
      (unary-relying-enumerator)
  ((enum-res :accessor enum-res
             :initform (make-enum-res)))
  (:documentation
   "enumerator with one memory"))

(defmethod make-memo-enumerator
    ((e abstract-enumerator))
  (init-enumerator
   (make-instance
    'memo-enumerator
    :enum (copy-enumerator e))))

(defmethod set-memo-res
    ((e memo-enumerator))
  (set-enum-res
   (enum-res e) (enum e)))

(defmethod init-enumerator :after
    ((e memo-enumerator))
  (set-memo-res e))

(defmethod enum-found
    ((e memo-enumerator))
  (enum-found (enum-res e)))

(defmethod enum-object
    ((e memo-enumerator))
  (enum-object (enum-res e)))
```

**Figure 24: Code for `memo-enumerator`**

## 5.3 Product enumerator

A product enumerator $\texttt{product}(f, E_1, \ldots, E_n)$ is defined as an `nary-relying-enumerator` and a `fun-mixin`.

The code is a little bit more complicated than it was for the previous kinds of enumerators because some enumerators must be reinitialized. The list of underlying enumerators is mapped to a vector and each underlying enumerator is encapsulated into a

```
memo-enumerator
```

so that its value can be asked for several times. This is shown in Figure 25.

This kind of operator cannot be implemented in the `XProc` framework but can be implemented in a functional style in the `SERIES` package.

Enumerating pairs or tuples is easily implemented using the general `enumerator-product` as shown by Figure 26.

## 5.4 A append enumerator

Sometimes, we may have an enumerator enumerating lists $l^0, l^1, \ldots$ with each $l^i = (l_0^i, l_1^i, \ldots, l_{k_i}^i)$ but we would like an enumerator enumerating the elements of the concatenation of the lists avoiding the computation the lists in extenso and their concatenation *i.e.*, so

```
(defclass enumerator-product
      (nary-relying-enumerator fun-mixin)
  ())

(defmethod make-enumerator-product
    (fun (enums (eql nil)))
  (make-empty-enumerator))

(defmethod make-enumerator-product
    (fun (enums list))
  (let ((v (map
            'vector
            #'make-memo-enumerator
            enums)))
    (if (every #'enum-found v)
        (make-instance
         'enumerator-product
         :underlying-enumerators v
         :fun fun)
        (make-empty-enumerator))))

(defmethod enum-i
    ((e enumerator-product) (i integer))
  (aref (underlying-enumerators e) i))

(defmethod next-element-p
    ((e enumerator-product))
  (enum-found (enum-i e 0)))

(defmethod next-element
    ((e enumerator-product))
  (let ((enums (underlying-enumerators e)))
    (prog1
        (apply
         (fun e)
         (map 'list
              (lambda (ei)
                (enum-object ei)) enums))
      (let ((index (1- (length enums))))
        (set-memo-res (enum-i e index))
        (loop ;; until we can start again
          until (enum-found
                 (enum-i e index))
          until (zerop index)
          do (init-enumerator
              (enum-i e index))
          do (set-memo-res
              (enum-i e (decf index)))))))))
```

**Figure 25: Code for product enumerator**

```
(defun make-enumerator-nil
    (&optional (n 1))
  (make-list-enumerator
   (make-list n)))

(defun make-enumerator-cons
    (enum1 enum2)
  (make-enumerator-product
   #'cons
   (list enum1 enum2)))

(defun make-enumerator-list
    (&rest enums)
  (if enums
      (make-enumerator-product
       #'list enums)
      (make-enumerator-nil)))
```

**Figure 26: Enumerator for pairs and lists**

an enumerator enumerating

$$l_0^0, l_1^0, \ldots, l_{k_0}^0, l_0^1, l_1^1, \ldots, l_{k_1}^1, \ldots$$
$$l_0^i, l_1^i, \ldots, l_{k_i}^i, \ldots$$

The `append-enumerator` inherits from a unary relying enumerator and has a slot `next-elements` containing the non empty sublist of the current underlying enumerator still to be enumerated. When all the elements of `next-elements` have been output, calls must be made to the remaining underlying enumerators until one returns a non empty list of elements or until there is no more underlying enumerator to call.

## 5.5 Summary of the general enumerators

All the classes corresponding the the enumerators described in this section are shown in Figure 28.

## 5.6 Filtering enumerator

Given an enumerator $E$ and a predicate $p$, it is easy to implement an enumerator

$$\texttt{filter}(p, E)$$

which enumerates the subsequence of elements of $E$ which satisfy $p$ like a `remove-if-not` on a `Lisp sequence`. Such type of enumerator is a unary relying enumerator.

The code is relatively easy. The underlying enumerator is encapsulated into a `memo-enumerator` enumerator and called until an element satisfying $p$ is found or until it has no more element. The code is given in Figure 29.

EXAMPLE 5.2. *Let $E$ be an enumerator of the natural integers; The enumerator $\texttt{filter}_{primep}$ with `primep` a predicate testing primality enumerates the set of primes $2, 3, 5 \ldots$,.*

# 6. APPLICATIONS

The previously defined general enumerators have been used in Autowrite[2] [5] for different purposes. Autowrite is a program entirely

[2] `http://dept-info.labri.fr/~idurand/autowrite/`

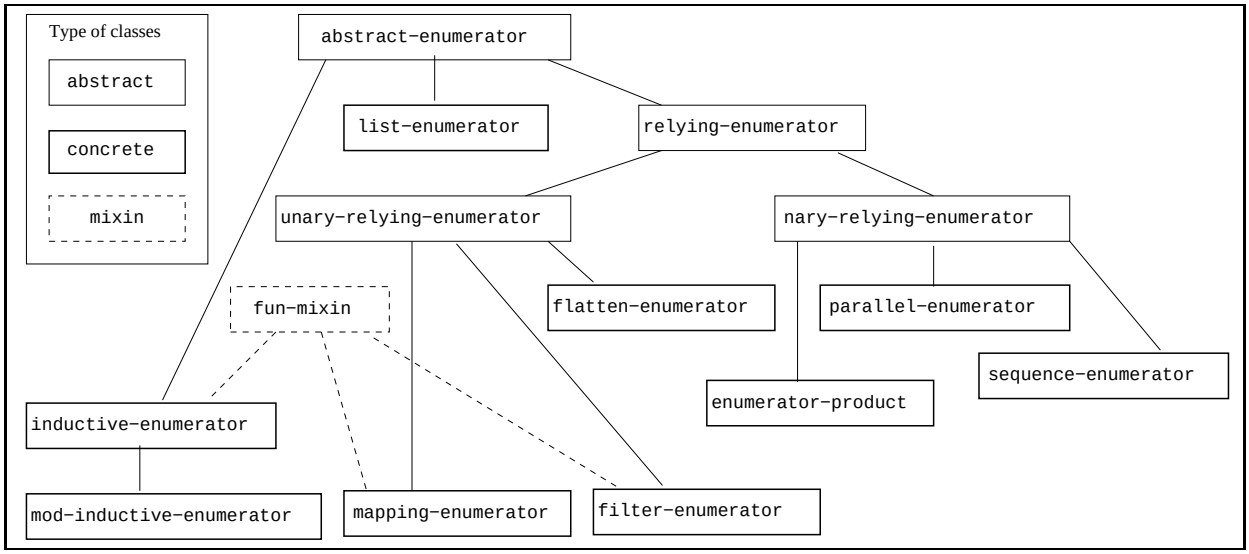



**Figure 28: Class hierarchy for enumerators**

```
(defclass append-enumerator
      (unary-relying-enumerator)
  ((next-elements
    :initform nil
    :accessor next-elements))
  (:documentation
   "the unary underlying operator
    must enumerate lists"))

(defmethod make-append-enumerator
    ((e abstract-enumerator))
  (init-enumerator
   (make-instance
    'append-enumerator
    :enum (copy-enumerator e))))

(defmethod skip-to-next-non-null
    ((e append-enumerator))
  (loop
    until (next-elements e)
    while (next-element-p (enum e))
    do (setf (next-elements e)
             (next-element (enum e)))))

(defmethod init-enumerator :after
    ((e append-enumerator))
  (skip-to-next-non-null e))

(defmethod next-element-p
    ((e append-enumerator))
  (next-elements e))

(defmethod next-element
    ((e append-enumerator))
  (prog1
      (pop (next-elements e))
    (skip-to-next-non-null e)))
```

**Figure 27: Code for append-enumerator**

```
defclass filter-enumerator
      (unary-relying-enumerator fun-mixin)
  ())

(defmethod skip-to-next
    ((e filter-enumerator))
  (loop
    while (enum-found (enum e))
    until (funcall
            (fun e)
            (enum-object (enum e)))
    do (set-memo-res (enum e))))

(defmethod init-enumerator :after
    ((e filter-enumerator))
  (skip-to-next e))

(defmethod next-element-p
    ((e filter-enumerator))
  (enum-found (enum e)))

(defmethod next-element
    ((e filter-enumerator))
  (prog1
      (enum-object (enum e))
    (set-memo-res (enum e))
    (skip-to-next e)))

(defmethod make-filter-enumerator
    ((e abstract-enumerator) filter-fun)
  (init-enumerator
   (make-instance
    'filter-enumerator
    :enum (make-memo-enumerator e)
    :fun filter-fun)))
```

**Figure 29: Filtering enumerator**

written in Common Lisp which deals with terms, term rewriting systems and term automata. Its specificity is *fly-automata* whose transition function is represented by a function. In this setting, the automata may be infinite: they may have an infinite countable signature and/or an infinite number of states.

## 6.1 Term automata

We recall some basic definitions concerning terms and term automata. Much more information can be found in the online book [2]. We consider a finite signature $\mathcal{F}$ (set of symbols with a fixed arity) and $\mathcal{T}(\mathcal{F})$ the set of (ground) terms built upon $\mathcal{F}$.

EXAMPLE 6.1. *Let $\mathcal{F}$ be a signature containing the symbols*

$$\{a, b, add_{a\_b}, rel_{a\_b}, rel_{b\_a}, \oplus\}$$

*with*

$$\begin{array}{|l|}\hline \mathsf{arity}(a) = \mathsf{arity}(b) = 0 \quad \mathsf{arity}(\oplus) = 2 \\ \mathsf{arity}(add_{a\_b}) = \mathsf{arity}(rel_{a\_b}) = \mathsf{arity}(rel_{b\_a}) = 1 \\ \hline \end{array}$$

*We shall see in Section 6.5 that this signature is suitable to write terms representing graphs of clique-width at most* 2*.*

EXAMPLE 6.2. *$t_1, t_2, t_3$ and $t_4$ are terms built with the signature $\mathcal{F}$ of Example 6.1.*

$$\begin{array}{|lcl|}\hline t_1 & = & \oplus(a, b) \\ t_2 & = & add_{a\_b}(\oplus(a, \oplus(a, b))) \\ t_3 & = & add_{a\_b}(\oplus(add_{a\_b}(\oplus(a, b)), add_{a\_b}(\oplus(a, b)))) \\ t_4 & = & add_{a\_b}(\oplus(a, rel_{a\_b}(add_{a\_b}(\oplus(a, b))))) \\ \hline \end{array}$$

*We shall see in Table 1, Section 6.4, their associated graphs.*

DEFINITION 3. *A (finite bottom-up)* term automaton *is a quadruple $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ consisting of a finite signature $\mathcal{F}$, a finite set $Q$ of states, disjoint from $\mathcal{F}$, a subset $Q_f \subseteq Q$ of final states, and a set of transitions rules $\Delta$. Every transition is of the form*

$$g(q_1, \ldots, q_n) \to q$$

*with $g \in \mathcal{F}$, $\mathsf{arity}(g) = n$ and $q_1, \ldots, , q_n, q \in Q$.*

Term automata recognize *regular* term languages [13]. The class of regular term languages is closed by the boolean operations union, intersection, complementation on languages which have their counterpart on automata. For all details on terms, term languages and term automata, the reader should refer to [2].

## 6.2 Enumerating terms

A term in Autowrite is roughly represented by its root label and the list of its arguments which are terms. A constant term has an empty list of arguments.

Given a function $h$ which given a root label returns a list of root labels, we can easily construct a (term) enumerator of terms with all possibilities of labels. The code is shown Figure 31. The structure of the term enumerator follows the recursive structure of the term as shown by Figure 32.

```
(defclass term (abstract-term)
  ((root :initarg :root :accessor root)
   (arg :initform nil
        :initarg :arg :accessor arg)
```

**Figure 30: Term class**

```
(defmethod make-term-enumerator
    ((term term) h)
  (make-funcall-enumerator
   (lambda (tuple)
     (build-term
      (first tuple) (cdr tuple)))
   (make-enumerator-cons
    (make-list-enumerator
     (funcall h (root term)))
    (apply
     #'make-enumerator-list
     (mapcar
      (lambda (arg)
       (make-term-enumerator arg h))
      (arg term))))))
```

**Figure 31: Term enumerator**

EXAMPLE 6.3. *Given the term $t = g(g(a, a), a)$, the function $h$ such that $h(a) = \{a, b\}$ and $h(g) = \{+, *\}$, the call to*

$$\texttt{make-term-enumerator}(t, h)$$

*yields an enumerator enumerating the terms*

$$\begin{array}{llll} +(+(a,a),a), & +(+(a,a),b), & +(+(a,b),a), & +(+(a,b),b), \\ +(+(b,a),a), & +(+(b,a),b), & +(+(b,b),a), & +(+(b,b),b), \\ +(*(a,a),a), & +(*(a,a),b), & +(*(a,b),a), & +(*(a,b),b), \\ +(*(b,a),a), & +(*(b,a),b), & +(*(b,b),a), & +(*(b,b),b), \\ *(+(a,a),a), & *(+(a,a),b), & *(+(a,b),a), & *(+(a,b),b), \\ *(+(b,a),a), & *(+(b,a),b), & *(+(b,b),a), & *(+(b,b),b), \\ *(*(a,a),a), & *(*(a,a),b), & *(*(a,b),a), & *(*(a,b),b), \\ *(*(b,a),a), & *(*(b,a),b), & *(*(b,b),a), & *(*(b,b),b) \end{array}$$

*Although finite, the enumerated set is exponentially large with regards to the input term and this is why it is interesting (if not necessary) to enumerate rather than to compute the set in extenso.*

## 6.3 Enumerating accessible states

Previously, in Autowrite, given an non deterministic term automaton and a term, the only possibility of computing the accessible states (the target) from this term, was to compute all of them at the same time. Using enumerators, it is now possible to enumerate these accessible states one by one. Like a term enumerator, a target enumerator follows the recursive structure of the term. When the term is a constant term ($a$ for instance), it suffices to create a list enumerator with all the states returned by the non deterministic transition function $a \to \{q_1, \ldots, , q_k\}$. This is shown in Figure 33.

For the non constant case

$$t = g(t_1, \ldots, t_n),$$

it is more complicated as shown in Figure 34. We compute the enumerators

$$E_1, \ldots, E_n$$

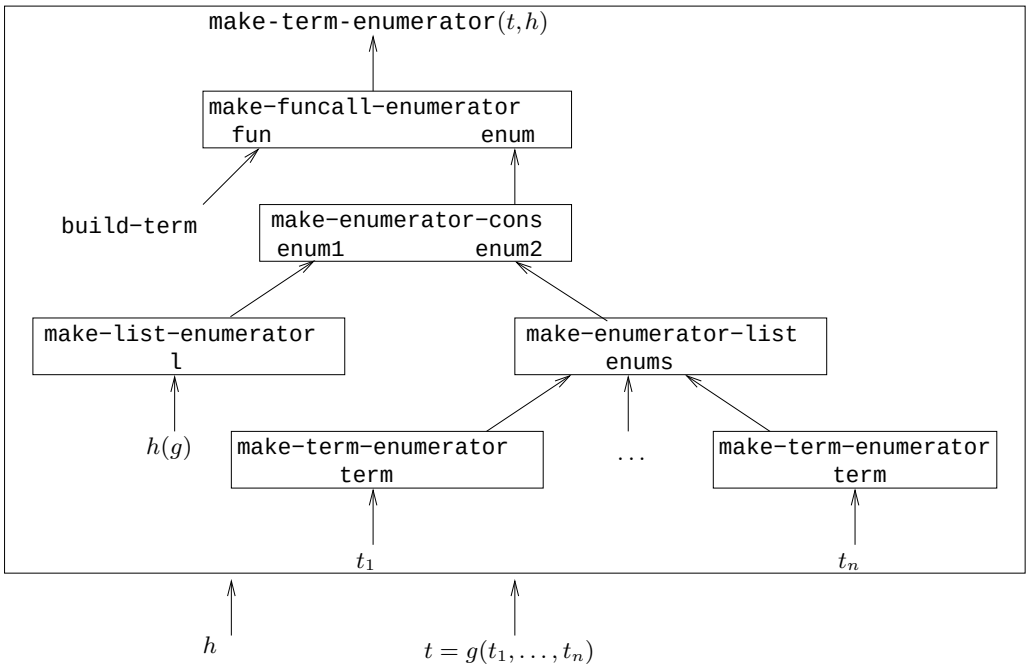



**Figure 32: Structure of a term enumerator**

```
(defmethod target-enumerator-ra
    ((root abstract-symbol) (arg (eql nil))
     (transitions abstract-transitions))
  (make-list-enumerator
   (apply-transition-function
    root nil transitions)))
```

**Figure 33: Target-enumerator for a constant**

for the term arguments $t_1, \ldots, t_n$, with `mapcar`, then the enumerator for

$$\texttt{product}(\texttt{list}, E_1, \ldots, E_n)$$

the cartesian product of $E_1, \ldots, E_n$. On each tuple of states

$$(q_1^i, \ldots, q_n^i)$$

enumerated by $X(E_1, \ldots, E_n)$ (and the root symbol $g$), we must apply the transition function of the automaton

```
(apply-transition-function
 root states transitions)
```

which returns a list of states $\{p_1^i, \ldots, p_{k_i}^i\}$. This is done with a funcall enumerator which enumerates the lists

$$\{p_1^1, \ldots, p_{k_1}^1\}$$
$$\{p_1^2, \ldots, p_{k_2}^2\}$$
$$\ldots$$

The

`make-enumerator-append`

is needed to obtain the states one at a time.

$$p_1^1, \ldots, p_{k_1}^1, p_1^2, \ldots, p_{k_2}^2, \ldots$$

```
(defmethod target-enumerator-ra
  ((root abstract-symbol) (arg list)
   (transitions abstract-transitions))
(let*
    ((targets
      (apply
       #'make-enumerator-list
       (mapcar
        (lambda (arg)
          (target-enumerator-ra
           (root arg)
           (arg arg)
           transitions))
        arg)))
     (target
      (make-append-enumerator
         (make-funcall-enumerator
          (lambda (states)
           (apply-transition-function
            root states transitions))
          targets))))
  target))
```

**Figure 34: Target-enumerator**

Note that this technique does not remove the possible duplicates in this list.

EXAMPLE 6.4. *An example of execution of a target enumerator is given in Figure 35 to compare with the computation of all the states of the target at one time with* `compute-target`.

## 6.4 Applications to graphs

```
AUTOGRAPH> *t*
add_a_b(oplus(a,oplus(a,b)))
AUTOGRAPH> *f*
fly-asm(2-COLORING-2)
AUTOGRAPH> (setf
           *e*
           (target-enumerator *t* *f*))
#<GENERAL::APPEND-ENUMERATOR {1004995D91}>
AUTOGRAPH> (call-enumerator *e*)
!<a:1 b:0>
T
AUTOGRAPH> (call-enumerator *e*)
!<a:0 b:1>
T
AUTOGRAPH> (call-enumerator *e*)
NIL
NIL
AUTOGRAPH> (compute-target *t* *f*)
{!<a:1 b:0> !<a:0 b:1>}
```

**Figure 35: Enumeration versus computation**

.

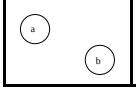
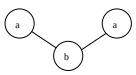
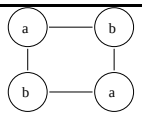
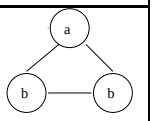
| $t_1$ | $t_2$ | $t_3$ | $t_4$ |
|---|---|---|---|
|  |  |  |  |

**Table 1: Graphs corresponding to the terms of Example 6.2**

In [3, 4], we have shown how graphs of bounded clique-width can be represented by terms and how some graph properties can be verified by term automata on terms representing graphs. We recall here this representation of graphs by terms to make the paper be self-contained.

## 6.5 Term representation of graphs of bounded clique-width

DEFINITION 4. *Let $\mathcal{L}$ be a finite set of vertex labels and consider graphs $G$ such that each vertex $v \in \mathcal{V}_G$ has a label $label(v) \in \mathcal{L}$. The operations on graphs are $\oplus$, the union of disjoint graphs, the unary edge addition $add_{a\_b}$ that adds the missing edges between every vertex labeled $a$ to every vertex labeled $b$, the unary relabeling $rel_{a\_b}$ that renames $a$ to $b$ (with $a \neq b$ in both cases). A constant term $a$ denotes a graph with a single vertex labeled by $a$ and no edge.*

*Let $\mathcal{F}_{\mathcal{L}}$ be the set of these operations and constants.*

*Every term $t \in \mathcal{T}(\mathcal{F}_{\mathcal{L}})$ defines a graph $G(t)$ whose vertices are the leaves of the term $t$. Note that, because of the relabeling operations, the labels of the vertices in the graph $G(t)$ may differ from the ones specified in the leaves of the term. A graph has* clique-width *at most $k$ if it is defined by some $t \in \mathcal{T}(\mathcal{F}_{\mathcal{L}})$ with $|\mathcal{L}| \leq k$.*

EXAMPLE 6.5. *For $\mathcal{L} = \{a, b\}$, the corresponding signature has already be presented in Example 6.1. The graphs corresponding to the terms defined in Example 6.2 are depicted in Table 1.*

```
AUTOGRAPH> *t2*
add_a_b(oplus(a,oplus(a,b)))
AUTOGRAPH> (setf
           *e*
           (make-color-term-enumerator
            *t2* 2))
#<ENUM::MAPPING-ENUMERATOR {10046F7671}>
AUTOGRAPH> (call-enumerator *e*)
add_a_b(oplus(a~0,oplus(a~0,b~0)))
T
AUTOGRAPH> (call-enumerator *e*)
add_a_b(oplus(a~0,oplus(a~0,b~1)))
T
AUTOGRAPH> (call-enumerator *e*)
add_a_b(oplus(a~0,oplus(a~1,b~0)))
T
```

**Figure 36: Color term enumerator**

In this setting, we have implemented automata to verify graph properties, like

- connectivity,
- acyclicity,
- $k$-colorability,
- acyclic-colorability,
- etc.

This collection of automata is maintained in the Autograph [7] system which depends on Autowrite. Most of the time, these automata are huge and represented as fly-automata [6]. In this framework, we have used enumerators for two purposes.

The first one as described in Section 6.3, is when we have a non deterministic automaton and we want to enumerate the accessible states. As soon as a final state is enumerated, we know that the term (the graph) is recognized by the automaton (satisfies the property corresponding to the automaton) and we may stop the enumeration.

The second one is, given a graph property, to enumerate graphs (terms) until we find one which satisfies the property (*i.e.* which is recognized by the corresponding automaton). For instance, given a term representing a graph, we want to enumerate terms representing vertex-colored versions of the same graph. Colors are represented by integers. A constant term `a` (which corresponds to a single vertex) is written `a~i` when colored with color `i`. Using `make-term-enumerator`, it is easy to write a

```
make-color-term-enumerator(term,k)
```

function which returns an enumerator of the colored versions of the term using at most `k` colors. An example of use is shown by Figure 36.

So far this has mainly be used to enumerate colored versions of a given graph until a graph coloring (no two adjacent vertices have the same color) is discovered. We hope to help solving some conjectures from graph theory.

## 7. CONCLUSION

We presented the enumerator concept, proposed an implementation of all purpose enumerators and shown how they can be used in the context of terms and term automata. The work presented in this paper is brand new. No performance tests have been performed so far. The code should be improved for more efficiency. More applications could be foreseen in particular in the search domain[3] as mentioned in the introduction.


## Acknowledgments

The author would like to thank the referees for their constructive remarks which helped improving the quality of the paper and the code.


## 8. REFERENCES


[1] Al. *The Haskell Programming Language*. `http://www.haskell.org`.

[2] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2002. Draft, available from `http://tata.gforge.inria.fr`.

[3] B. Courcelle and I. Durand. Verifying monadic second order graph properties with tree automata. In *Proceedings of the 3rd European Lisp Symposium*, pages 7–21, May 2010.

[4] B. Courcelle and I. Durand. Automata for the verification of monadic second-order graph properties. to appear in Journal of Applied Logics, 2011.

[5] I. Durand. Autowrite: A tool for term rewrite systems and tree automata. In *Proceedings of the Workshop on Rewriting Strategies*, pages 5–14, Aachen, June 2004.

[6] I. Durand. Implementing huge term automata. In *Proceedings of the 4th European Lisp Symposium*, pages 17–27, Hamburg, Germany, March 2011.

[7] I. Durand. Autograph. Common Lisp package, since 2010.

[8] *Enumeration Package for Common Lisp*, 2012. http://common-lisp.net/project/cl-enumeration/.

[9] A. E. Group. Enum : Algorithms and complexity for answer enumeration (ANR project). 2007-2011.

[10] I. Guy L. Steele, Thinking Machines. *Common Lisp the Language, 2nd edition*. Digital Press, 1990.

[11] R. P. Stanley. *Enumerative combinatorics*. Cambridge University Press; 2nd edition, 2000.

[12] W. Stein et al. *Sage Mathematics Software (Version x.y.z)*. The Sage Development Team. `http://www.sagemath.org`.

[13] J. Thatcher and J. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2:57–81, 1968.

[14] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego, 1993.

[15] *XProc: An XML Pipeline Language*. http://www.w3.org/TR/xproc/.


[3] A proposal for an ANR project has been submitted on January 5th, 2012 which includes propositions to work in this direction.

# Doplus
# the high-level, Lispy, extensible iteration construct

Alessio Stalla
ManyDesigns s.r.l.
alessiostalla@gmail.com

## ABSTRACT
In this paper, we briefly present a novel iteration macro for Common Lisp.

## Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features—*control structures*.

## General Terms
Languages, Experimentation.

## Keywords
Lisp, macro, iteration.

## 1. INTRODUCTION
There are three general-purpose, widely used iteration constructs in Common Lisp. Of these, two are part of the Common Lisp standard itself (DO/DO* and LOOP) and one is a third-party library (Iterate).

The author was unsatisfied with all the three of them. DO, a glorified C for, is too low level. LOOP, with its PASCAL-like embedded DSL, does not mix well with the rest of the language (:collect can't be used in the body of a let form, for example), it's not SLIME-friendly, and it's not extensible in a portable, easy way. Iterate is mostly fine, except for a not-so-tiny detail: it is implemented using a code walker that has freedom to move pieces of user code around. This breaks macrolet and creates some potentially unexpected effects when a form lexically following another is moved back before the preceding one.

So, in pure Lisp hacker spirit, the author rolled his own and began marketing it all around the world as the best thing since sliced bread. This very paper is a prime example of the aforementioned phenomenon.

## 2. DOPLUS
DO+ (doplus) is an iteration macro for Common Lisp. It is conceptually a high-level DO in that it retains from DO the separation of iteration control forms from the body of the loop itself. In other words, a doplus form has the following shape:

```
(do+ (clause*) form*)
```

We will not go into unnecessary details here because the basic features of doplus are the same as LOOP and Iterate. Clauses comprise the usual paraphernalia: numeric iteration, walking over sequences, accumulators, generators, etc., as well as user-defined ones. Inside the body, a few macros can be used to control iteration: collect, terminate and skip, which basically do the obvious thing suggested by their names. This is a more or less random example:

```
(do+ ((for x (in '(4 5 6 7)))
      (for y (from 3 :to 10))
      (for z (being (+ x y)))
      (stop-when (> z 11)))
  (if (oddp x)
    (collect (list x z))
    (collect (list y z))))
```

Besides that, doplus has some unique features. Some of them are detailed below.

### 2.1 Atomic updates
In doplus, contrarily to other loop macros, variables are updated atomically, i.e. each clause that is executed at the beginning or at the end of an iteration sees consistent values for the variables, independently of the order of the clauses in the head. An example can make it clearer. Consider the following two iterate forms:

```
(iter
 (for k :in '(a b c d e))
 (for x :in-vector #(1 2 3 4))
 (finally (return (list x k))))
=> (4 E)
(iter
 (for x :in-vector #(1 2 3 4))
 (for k :in '(a b c d e))
 (finally (return (list x k))))
=> (4 D)
```

They only change in the order of the two for clauses, yet their return values differ. LOOP behaves the same. This doesn't happen with doplus; the following two forms return the same value:

```
(do+ ((for x (across #(1 2 3 4) :index index))
      (for k (in '(a b c d e)))
      (returning (list x k :index index))))
=> (4 D :INDEX 3)
(do+ ((for k (in '(a b c d e)))
      (for x (across #(1 2 3 4) :index index))
      (returning (list x k :index index))))
=> (4 D :INDEX 3)
```

That happens because as soon as the index goes out of the vector's bounds, the loop is terminated and any updates remain confined in the atomic section, without being visible to the other forms, such as those computing the return values.

## 2.2 Iteration over arbitrary sequences

The built-in clause `IN` can iterate over arbitrary sequences - lists, vectors, as well as user-defined ones in implementations that support extensible sequences [1]. On those implementations, the native sequence iterator facility is used; on the others, a port of SBCL's iterator implementation is used instead.

There are also specialized clauses for lists and vectors that, besides being slightly more efficient, provide additional features that do not make sense for arbitrary sequence types, such as the ability to bind a loop variable to the successive CDRs of a list, or to the index of a vector, which is being iterated over.

## 2.3 Nested loops

Nested doplus forms can interact in certain ways with outer doplus forms. In Iterate, it is possible for an inner loop to execute a piece of code as if it was part of an outer loop[2]. Doplus does not employ a code walker and thus limits these kinds of interactions to some selected cases. In particular, it is possible to refer by name to an outer accumulator or generator, and, if an outer loop is given a name, skip and terminate macros can refer to that name and behave accordingly. For example:

```
(do+ ((for x (in (list 1 2 3)))
      (accumulating-to result)
      (stop-when (> (length result) 10))
      (returning result))
  (do+ ((for y (in (list 'a 'b))))
    (collect (list x y) :into result)))

=> ((1 A) (1 B) (2 A) (2 B) (3 A) (3 B)
    (4 A) (4 B) (5 A) (5 B) (6 A) (6 B))

(do+ (outer-loop ;;<-- name of the loop
      (for x (in (list 1 2 3))))
  (print x)
  (do+ ((for k (to 3)))
    (when (> (+ x k) 5)
      (terminate outer-loop))))
```

# 3. EXTENSIONS AND IMPLEMENTATION NOTES

Extending doplus amounts to writing regular Lisp macros that in most cases expand to a list of clauses as the user would write them in the head of a doplus form. For example, the code below is adapted from the definition a clause built in doplus:

```
(defclause in-vector
  (vector &key (index (gensym "INDEX")))
  "Loops across a vector."
  (let ((tmp-var (gensym "VECTOR")))
    `((with (,tmp-var ,vector) (,index 0))
      (declaring
       (type (integer
              0
              ,(1- array-total-size-limit))
             ,index))
      (for ,index
           (from 0 :to (1- (length ,tmp-var))
                   :by +1))
      (for ,*iteration-variable*
           (being (aref ,tmp-var ,index))))))
```

Here we can observe the only two peculiarities one can encounter when writing doplus extensions. First, *iteration-variable*, which is bound by the `FOR` macro to the variable (or lambda list) provided by the user as the first argument to `FOR`. Access to `*iteration-variable*` is necessary when writing extensions to `FOR`. Second, the defclause macro, which is equivalent to defmacro, except that it also records the macro name as a known clause for documentation purposes.

Regarding the implementation, there is a notable aspect, in our opinion, in the use of macros to implement iteration clauses. These macros have the peculiarity that they do not return Lisp code - rather, they return doplus "code", that is, instances of structures that instruct the doplus macro on how to generate code[1]. Effectively, the head of a doplus clause is written in a Lisp-based DSL that adopts the CL macro system for its own purposes.

## 3.1 Obtaining doplus

The doplus project is hosted at http://code.google.com/p/tapulli/. Doplus is also available through Quicklisp.

# 4. REFERENCES


[1] C. Rhodes. 2007. *User-extensible sequences in Common Lisp.* ILC 2007 Proceedings, http://doc.gold.ac.uk/~mas01cr/papers/ilc2007/sequences-20070301.pdf

[2] J. Amsterdam, 1989 and L. Oliveira, 2006. *The Iterate Manual.* http://common-lisp.net/project/iterate/doc/Named-Blocks.html


[1] Although technically structures are valid Lisp forms, since they evaluate to themselves, we emphasize the fact that doplus clause structures are used as the building blocks of a mini-language understood by the doplus macro.

# Lightening Talks

## Using Clojure in Linguistic Computing

***Zoltan Varju***

zoltan.varju@gmail.com | Weblib LLC | Hungary

**Richard Littauer**

richard.littauer@gmail.com | Saarland University, Computational Linguistics Department | Germany

**Peteris Ernis**

peteris.ernis@gmail.com | Selwyn College, University of Cambridge | United Kingdom

Due to recent developments in the humanities and the social sciences more and more scholars are turning towards computing. Although there are accessible and well designed toolkits to get into computing, taking the step from surface knowledge of programming to tackling non-trivial issues is difficult and time intensive. Even those researchers with some computing background may face problems when they want to deepen their knowledge and start serious projects, especially when participating with a team of programmers is not an option. We think using Clojure for linguistic computing answers these concerns. Here, we examine the existing resources and explain why Clojure can help in both educating and researching. We also present two mini case studies as examples of Clojure's interoperability and use.

## QueryFS, a virtual filesystem based on queries, and related tools

**Mikhail Raskin**

raskin@mccme.ru | Independent University of Moscow | Russian Federation

Modern hardware and software allow users to store and transmit huge data collections. Applications can rely on filesystem (or database) interface for most tasks. Unfortunately, indexing and searching these data collections has to be done using specialized tools with limited interoperability with existing software. This paper describes a tool providing a unified POSIX filesystem interface (using FUSE) to the results of search queries. The search queries themselves may be expressed using high-level languages, including SQL and specialized Common Lisp API.

# Using Clojure in Linguistic Computing

Zoltán Varjú
Weblib LLC
Kaposhegy u. 13
7400 Kaposvar, Hungary
zoltan.varju@weblib.com

Richard Littauer
Saarland University
Computational Linguistics
Department
66041 Saarbrücken, Germany
richard.littauer@gmail.com

Peteris Erins
University of Cambridge
Selwyn College
Cambridge, United Kingdom
peteris.erins@gmail.com

## ABSTRACT

Due to recent developments in the humanities and the social sciences more and more scholars are turning towards computing. Although there are accessible and well designed toolkits to get into computing, taking the step from surface knowledge of programming to tackling non-trivial issues is difficult and time intensive. Even those researchers with some computing background may face problems when they want to deepen their knowledge and start serious projects, especially when participating with a team of programmers is not an option. We think using Clojure for linguistic computing answers these concerns. Here, we examine the existing resources and explain why Clojure can help in both educating and researching. We also present two mini case studies as examples of Clojure's interoperability and use.

## Categories and Subject Descriptors

A.m [**General Literature**]: Miscellaneous; I.7.m [**Computing Methodologies**]: Document and Text Processing—*Miscellaneous*

## General Terms

Theory

## Keywords

Clojure, linguistic computing

## 1. INTRODUCTION

Computational methods are standard tools in the natural sciences, and there is an increasing demand for easy to use tools in the emerging fields of digital humanities, linguistic computing, computational social sciences and data journalism. In these fields, the practice of scientific investigation relies on computational methods. Analyzing data and running simulations are inherent parts of current scientific work that require programming skills.

Although there is a plethora of high quality open source projects, the average researcher must become a polyglot and use various programming languages to achieve their goals. This results, far too often, in a long and steep learning curve, slowly progressing research projects, and bad programming practices. There is no royal road to computer science. Those who want to step into emerging new fields do not have to become computer scientists, but they have to acquire the basics in order to participate in interdisciplinary projects and conduct their own research.

The Clojure language is ideal for the above mentioned scenario for several reasons. The theoretical basis of functional programming can be related to the humanities curriculum. The clear and accessible Clojure development environment encourages good software engineering practice. Clojure, as a JVM language, and with its unique mechanism for parallelism, is a natural choice in the age of the data deluge.

## 2. EXISTING RESOURCES

The most popular natural language processing (NLP) tool is the Python Natural Language Toolkit (`http://www.nltk.org/`) (NLTK). It contains a wide range of algorithms, both statistical and rule based. It was designed for multidisciplinary instruction[3] and it is used at institutions all over the world. Today, NLTK is not only an academic toolkit; it is commonly used in the industry too. The freely available NLTK book[4] and popular titles from big publishers[18] paired with its wide range of algorithms make NLTK an outstanding NLP tool.

The Stanford CoreNLP(`http://nlp.stanford.edu/software/corenlp.shtml`) and the Apache OpenNLP(`http://incubator.apache.org/opennlp/index.html`) projects are the two most reliable and advanced Java-based NLP tools. Although their quality is unquestionable, using them require advanced knowledge of the Java language and object oriented programming methodology. The lack of good tutorials and comprehensive documentation mean a barrier to non-programmers.

SNLTK(`http://www.snltk.org/`) is the most comprehensive Lisp-based NLP library. It is designed for educational purposes, building on the rich ecosystem around Racket and Scheme. SNLTK inspired us as its authors emphasize the rich Lisp literature on the foundations computational Linguistics[8].

Prolog once was a prominent language among computational linguists. Logic programming is ideal for expressing foun-

dational concepts like automata theory and one can easily implement basic grammar formalisms in Prolog. A rich literature[6, 5] developed around the language over the years but the decline of symbolic AI made Prolog less popular, leaving it unaccessible for a wider audience.

In the last decade, Haskell has emerged as a new tool for those who are interested in applying computational methods to logic and semantics. Titles like [10] and [20] laid down foundations. Natural Language Processing for the Working Programmer[9] is a work-in progress open book and a proof of concept of the usage of Haskell for statistical NLP.

The R statistical programming language is very popular among corpus linguists, psycholinguists and social scientists. Packages like tm[12] and zipfR[11] are not only valuable educational and research tools, but can be used in applied research too. Despite its merits, it is criticized because of its steep learning curve and inherent problems with parallelism and scalability[15].

# 3. MOTIVATIONS FOR CHOOSING CLOJURE

We believe that Clojure is a good choice as a second programming language for scholars and students in the humanities and social sciences. At present, we would like to test our assumptions through mini case studies.

Norvig in his seminal Paradigms of Artificial Intelligence Programming[17] lists eight features that make Lisp different and the natural choice for artificial intelligence programming: built-in support for lists, automatic storage management, dynamic typing, first-class functions, uniform syntax, interactive environment, extensibility, and history.

Clojure not only supports list, but other common data structures. Every popular IDE has Clojure support and emacs with Slime(`http://common-lisp.net/project/slime/`) or VimClojure(`http://www.vim.org/scripts/script.php?script_id=2501`) provide users with a high quality development environment. Java interoperability means more extensibility with the abundance of Java libraries. We have to add to the list the open source community built around the language, which made tools like the ClojureScript(`https://github.com/clojure/clojurescript`) and the ClojureScriptOne(`http://clojurescriptone.com/`) web development environment and the pallet tool for cloud computing(`http://palletops.com/`) which reduces the learning curve as one get things done in a language.

As an extensible language, Clojure is multiparadigmatic. Type checking is optional, and core.logic implements miniKanren, a Prolog-like logic programming library.

## 3.1 Pedagogical considerations

Although the curriculum is changing, some elementary logic is usually part of the humanities education, at least in philosophy and linguistics programs. Social scientists have to acquire statistics and get used to a basic level of mathematical rigour during their studies. Given this background and some exposure to the fundamental concepts of programming they are in a very good position to learn 21st century computing skills that can be useful in academic research and even in the job market.

However the above mentioned type of people have to face serious problems when they want to deepen their knowledge. Although we have well-written accessible books on computational linguistics they are either outdated or one should learn at least two programming languages in parallel to get things done.

## 3.2 Scientific considerations

Computing is going through a big change[13]. The available data grows exponentially; "today's big data is tomorrow's mid-size" as the saying goes[19]. New technologies, and even new professions like data scientist have emerged in the last few years.

As the Google $n$-gram viewer(`http://books.google.com/ngrams`) project shows we can have access to unimaginable amount of data[16]. To work effectively and be able to collaborate with colleagues from other fields, humanities researchers need a tool that is smoothly scalable.

While Linguistics is not traditionally a data-intensive science, with the increasing size of linguistic databases, it is nevertheless entering the era of the fourth paradigm[14], where managing and accessing data is as integral a part of the scientific process as collecting it using computational methods. These unprecedented possibilities urge linguists to build a cyber-infrastructure[2] and rethink their theories[1] in the light of data.

Using a tool, like Clojure, that addresses these issues could foster scholarly work of a new type, and especially interdisciplinary work.

# 4. MINI CASE STUDIES

We started a project blog where we are showing off possible uses of Clojure and we are trying to get feedback from Clojure enthusiast and members of the humanities computing community. We found two types of interested individuals; the first is using computational methods as a tool during their analysis, the second is interested in using the computational tools to express their ideas.

## 4.1 Java interoperability

The first group sees programming as a tool that helps analysis. This does not mean that they compromise quality for usability. This group can benefit from the interoperability of Clojure with high quality Java libraries like OpenNLP. A corpus linguist interested in the distribution of various parts of speech elements can use a POS tagger to automatically tag collections of raw electronic texts. As a common software engineering practice, one can use a tool as a kind of black box without any expert knowledge about its internal mechanism.

OpenNLP can be integrated into a Clojure project via Leiningen. One can easily define the necessary tools to POS tag sentences with a few line of code.

```
(ns hello-nlp.core
    (use opennlp.nlp)
    (use opennlp.treebank)
    (use opennlp.tools.filters)
    (use (incanter core charts)))

(def get-sentences
    (make-sentence-detector "models/en-sent.bin"))

(def tokenize
    (make-tokenizer "models/en-token.bin"))

(def pos-tag
    (make-pos-tagger "models/en-pos-maxent.bin"))
```

There is no need for expert knowledge to get started with counting parts of speech. Thinking about linguistic data structures in terms of lists (or sets and hash-maps) is a natural choice, as using functions on these structures is close to linguistic formalism. Counting parts of speech in a text can be expressed in an easy to read manner.

```
(defn tag-sent [sent]
    (pos-tag (tokenize sent)))
(def pos-austen
    (map pos-tag (map tokenize (get-sentences austen))))

(pos-filter determiners #"^DT")
(pos-filter prepositions #"^IN")

(def preps
    (reduce + (map count (map prepositions pos-austen))))
(def dets
    (reduce + (map count (map determiners pos-austen))))
(def nps
    (reduce + (map count (map nouns pos-austen))))
(def vps
    (reduce + (map count (map verbs pos-austen))))
(def stats
    [nps vps dets preps])

(view (bar-chart ["np" "vp" "dts" "preps" ] stats))
```

The output graph can be seen in Fig. 1.

Collecting and presenting word frequency distributions is a tedious task. Linguists often use pipelines of various tools, one for collecting and cleaning up texts and another for analyzing and report findings. This works requires using wrappers of third party tools like part of speech taggers. The aforementioned process is called ”software carpentry”, which describes the pragmatic approach of those who see software as a means, not an end. But as we have described in Section 3.2, the scale of scientific data is changing and traditional tools for processing it are becoming insufficient. While the product of ”carpentry” can be used as prototypes, we can gain in productivity by using interoperable tools.

In the future, the ability to work together with computer scientists will be more important. Product cycles are getting shorter and shorter, the border of prototyping and production is disappearing and parallelism is becoming common.

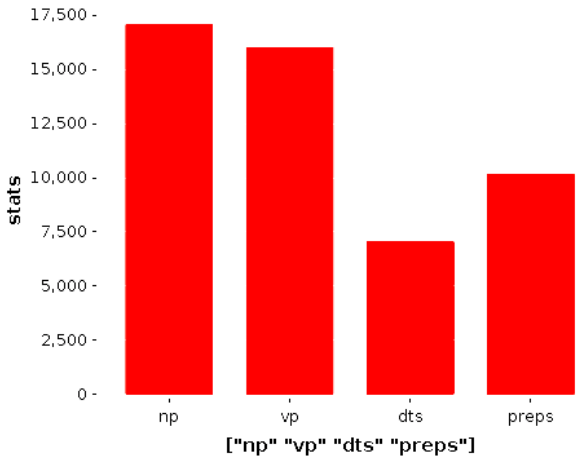



**Figure 1: Word tag frequencies**

With Clojure, one can use Java or Clojure libraries developed by the project team or foundnd in a public open source repository. Upon completing the work, the end result should ready for integration.

## 4.2 Logic programming

Logic programming is a programming paradigm that uses mathematical logic for writing programs. Logic programs are classically written as sets of inference rules and a query over the rule database. The execution of a logic program is an attempt to constructively prove the query using a built-in proof-search method.

Many concepts in computational linguistics, such as automata and grammars, can be described declaratively. Logic programming languages can be used to turn their definitions into computations. Linguists can write recognizers, parsers and generators without implementing a search algorithm.

Prolog has been the dominant logic programming language in linguistics and beyond. Its use has diminished over the years; however, the premise of logic programming still stands. The recent language Kanren is an embedded logic programming environment in Scheme. A version of the library called miniKanren[7] has been ported to Clojure in the form of the core.logic library.

Below we implement an automata in core.logic that accepts lists consisting of multiple copies of the letter $a$, as seen in Fig. 2.

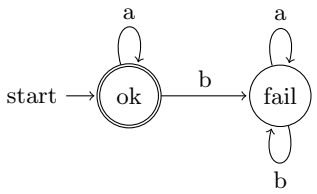



**Figure 2: Example automata**

```
(defrel start q)
(fact start 'ok)


(defrel transition from via to)
(facts transition [['ok 'a 'ok]
                   ['ok 'b 'fail]
                   ['fail 'a 'fail]
                   ['fail 'b 'fail]])

(defrel accepting q)
(fact accepting 'ok)

(defn recognize
  ([input]
     (fresh [q0]
            (start q0)
            (recognize q0 input)))
  ([q input]
     (matche [input]
        (['()]
           (accepting q))
        ([[i . nput]]
           (fresh [qto]
                  (transition q i qto)
                  (recognize qto nput))))))
```

We can use the automata to recognize strings in the language, or even to generate them.

```
(run* [q] (recognize '(a a a)))
;; => (_.0) where "_0" implies success (run* [q] (recognize '(a b a)))
;; => ()

(run 3 [q] (recognize q))
;; => (() (a) (a a))
```

Embedding in a functional language permits the use of logic programming just when appropriate. That is an advantage over Prolog, which does not feature functional constructs.

# 5. FUTURE WORK

Our primary goal is to provide more mini-case studies and get more feedback from a wider audience. Ultimately, we would like to combine the case studies into a toolkit that can be used for linguistics education and research.

# 6. REFERENCES


[1] Bender, E. M., and Good, J. A grand challenge for linguistics: Scaling up and integrating models. *White paper contributed to NSF's SBE 2020: Future Research in the Social, Behavioral and Economic Sciences initiative 1*, 1 (June 2010), 1–1.

[2] Bender, E. M., and Langendoen, D. T. Computational linguistics in support of linguistic theory. *Linguistic Issues in Language Technology 3*, 2 (February 2010), 0–0.

[3] Bird, S., Klein, E., Loper, E., , and Baldridge, J. Multidisciplinary instruction with the natural language toolkit. In *Proceedings of the Third Workshop on Issues in Teaching Computational Linguistics* (June 2008), Third Workshop on Issues in Teaching Computational Linguistics, pp. 62–70.

[4] Bird, S., Klein, E., and Loper, E. *Natural Language Processing with Python*. O'Reilly Media, Sebastopol, CA, 2009.

[5] Blackburn, P., and Bos, J. *Representation and Inference for Natural Language: A First Course in Computational Semantics*. Center for the Study of Language and Information, Palo Alto, CA, 2005.

[6] Blackburn, P., Bos, J., and Striegnitz, K. *Learn Prolog Now*. College Publications, London, UK, 2006.

[7] Byrd, W. E. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. PhD thesis, Indiana University, August 2009.

[8] Cavar, D., Gulan, T., Kero, D., Pehar, F., and Valerjev, P. The scheme natural language toolkit (snltk). In *Proceedings of the 4th European Lisp Symposium* (April 2011), European Lisp Symposium, pp. 58–61.

[9] de Kok, D., and Brouwer, H. *Natural Language Processing for the Working Programmer*. `http://nlpwp.org/`, Groningen, NL, 2011.

[10] Doets, K., and van Eijck, J. *The Haskell Road to Logic, Maths and Programming*. College Publications, London, UK, 2004.

[11] Evert, S., and Broni, M. zipfr: Word frequency distributions in R. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics* (2007).

[12] Feinerer, I., Hornik, K., and Meyer, D. Text mining infrastructure in R. *Journal of Statistical Software 25*, 1 (March 2008), 1–54.

[13] Halvey, A., Norvig, P., and Pereira, F. The unreasonable effectiveness of data. *IEEE Intelligent Systems 24*, 2 (March/April 2009), 8–12.

[14] Hey, T., Tansley, S., and Tolle, K., Eds. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Corporation, US, 2009.

[15] Ihaka, R., and Lang, D. T. Back to the future: Lisp as a base for a statistical computing system. *Compstat 2008 24*, 1-1 (August 2008), 1–1.

[16] Michel, J.-B., Shen, Y. K., Aiden, A. P., Veres, A., Gray, M. K., Pickett, J. P., Clancy, D. H. D., Norvig, P., Orwant, J., Pinker, S., Nowak, M. A., Aiden, E. L., and Team, G. B. Quantitative analysis of culture using millions of digitized books. *Science 331*, 176 (January 2011), 176–182.

[17] Norvig, P. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, Waltham, MA, 1991.

[18] Russell, M. A. *Mining the Social Web: Analyzing Data from Facebook, Twitter, LinkedIn, and Other Social Media Sites*. O'Reilly Media, Sebastopol, CA, 2011.

[19] Slocum, M. *Big Data Now*. O'Reilly Media, Sebastopol, CA, 2011.

[20] van Eijck, J., and Unger, C. *Computational Semantics with Functional Programming*. Cambridge University Press, Cambridge, UK, 2010.

# QueryFS, a virtual filesystem based on queries, and related tools.

## Abstract.

Michael A. Raskin*
Independent University of Moscow
115162 Moscow
Bolshoy Vlasyevskiy 11
Russia
raskin@mccme.ru

## ABSTRACT

Modern hardware and software allow users to store and transmit huge data collections. Applications can rely on filesystem (or database) interface for most tasks. Unfortunately, indexing and searching these data collections has to be done using specialized tools with limited interoperability with existing software. This paper describes a tool providing a unified POSIX filesystem interface (using FUSE) to the results of search queries. The search queries themselves may be expressed using high-level languages, including SQL and specialized Common Lisp API.

## Keywords

FUSE, filesystems, search, virtual directories, domain-specific languages

## 1. INTRODUCTION

Modern filesystems allow users to store large volumes of data. When the data has some special structure, an SQL database may better suited for the task. In both cases there are many software packages implementing the same interface. Applications use the same interface to access multiple storage implementations; and many applications developed before a technology improvement becomes available still benefit from it. For example, SBCL has no need to know about RAID0 to get improved write speed. Neither it needs to know about SSH to read source from server using SSHFS-FUSE.

There are also many tools to find data in the storage. Some of them traverse all the storage to find the needed piece of information, some create and maintain indices, some expect user to explicitly add the data into indexed area.

*This work was partially supported by RFBR grant 10-01-93109-NTsNIL_a

Unfortunately, making these tools interact with unsuspecting applications is often hard and the query language may have limited expressive power.

This paper describes how QueryFS project tries to solve the problems of using query results in applications unaware of any special API, saving queries for future use and expressing complicated conditions with queries.

## 2. EXISTING PROJECTS

To refine the goals and to give general information about previous work in this area this section contains a list of some projects or products working on similar problems.

The problem of finding a file in the storage is probably as old as the very notion of file. Even modern systems have utilities which can be traced to the very time when the word “file” got a meaning related to computers. POSIX requires utility called “find” to be present. If we only consider interaction with programs following the original Unix design principles, “find” satisfies all the conditions outlined in the previous section; its command line can be easily saved to a text file, it generates output convenient to feed to programs with command-line interface, and it allows arbitrary logical combinations of basic conditions in queries. Unfortunately, modern GUI programs will not let user easily feed “find” output to a file selection dialog.

Another old example is feeding of search results to the UI element intended for directory view in many file managers. In current versions of Gnome Nautilus, Windows Explorer or MacOS X Finder user can save such a search query and interact with it as if it was a folder. The main problem is that applications unaware of this feature cannot use such directories. Even *WinFS* project by Microsoft was going to require applications to use special API to access such search folders.

Inability of some applications to access virtual directories and use plain text file lists can be mitigated by using *FUSE*. It allows mounting special filesystems and processing of the filesystem operation in the userspace.

For example, *beaglefs*, uses indices created by Beagle desktop search system. User can mount a directory filled with

symbolic links to all files matching a Beagle query by a single invocation of "beaglefs" command, which makes use of search results in other applications trivial. Saving queries is as easy as creating a shell script. Unfortunately, the expressive power of Beagle queries is relatively limited.

Some of the filesystems emphasize user-entered metadata. For example, *tagfs* and *movemetafs* support marking each file with tags (arbitrary strings) instead of building file hierarchies. User can then go into a virtual directory which contains only the files having all the tags from some list. The full path of the directory can easily be saved as a symbolic link. Unfortunately, even file size cannot be taken into account in such queries.

The *libferris* project (together with *ferris-fuse*) provides means to access many different types of metadata found inside common file types. *libferris* on its own requires use of special API or utilities to access the data, but allows complicated queries in query languages like XPath and SQL. The FUSE filesystem, *ferris-fuse* only allows browsing the data. Another project, *BaseX*, uses XQuery language and has GUI and command-line tools for browsing indexed data. Currently, *BaseX* lacks *FUSE* support.

The *RelFS* project has its focus on representing SQL queries as directories. A *RelFS* filesystem can store files and symbolic links like an ordinary filesystem. It also allows going into a directory with name starting with "#" symbol, which is interpreted as an SQL query. Running "find" on such a directory returns approximately the same result as running the SQL query put into directory name. *RelFS* uses SQL query language, allows queries to return complicated directory trees, and allows saving queries as symbolic links. Unfortunately, *RelFS* queries process only files and symbolic links stored on the *RelFS* filesystem itself, and storing large files on *RelFS* causes performance problems.

Two projects with the most radical goals, *dbfs* and *Hypocampus*, store all the files inside the DB and have no hierarchical structure. All available ways to access files create special SQL queries.

# 3. WISHLIST FOR QUERYFS

*QueryFS* project started as an attempt to reimplement *RelFS* and remove some of its weaknesses.

It is obvious that a *FUSE* filesystem will never beat a well-optimised kernel filesystem in storing large files, so *QueryFS* is not supposed to carry files of any significant size. Whenever such files should appear in earch results, symbolic links will be used.

This decision rules out maintaining up-to-date indices by monitoring access to the filesystem itself. Fortunately, there are a lot of other projects dedicated to indexing of data (e.g. aforementioned Beagle and BaseX). Some of them even use some kind of filesystem event notifications supported by recent operating system kernels to update information in real-time. That means that ease of adding an interface to such a "foreign" index is much more important than non-trivial indexing implemented inside *QueryFS*.

As the queries should be easy to use from other applications unaware of *QueryFS*, they should be represented as directories. Actually, all the content of a *QueryFS* instance is generated this way. Some of the queries may provide access to internals of the filesystem, e.g. allow loading new queries by writing to a special file.

One of the design goals is allowing user to save queries. *QueryFS* takes an extreme position by making it hard to use a query without saving it. Neither the core nor example plugins and queries support this. By default, *QueryFS* expects path to a directory having subdirectories "results" (future mountpoint), "queries" (user queries) and "plugins" (non-core code, expected to define ways of parsing queries in different format). Loading queries from files also eliminates syntactical problems related to fitting complex expressions into a command or even filesystem paths (*RelFS* and *tagfs* actually do put queries into filesystem paths).

# 4. PROVIDED INTERFACE

*QueryFS* tries to make adding support for a new query type as easy as possible. To make query parsers easily replaceable, *QueryFS* is split into core code (FUSE interaction, path management, basic plugin and query management), plugins (query parsers and helper functions for queries) and queries (generators of filesystem contents).

In general, lifecycle of *QueryFS* instance looks like the following.

When *QueryFS* is launched, it loads plugins. Plugins can register query parsers. Afterwards queries are loaded. Query parser corresponding to a query file currently depends only on the file extension. The parser receives the query and returns source code which describes the resulting layout. Layout is described in declarative terms, all path processing is done in the core code. This code is labeled with the query file name (without extension) and saved. After all queries are parsed, all the generated layout code is compiled and executed as needed to answer filesystem requests. Execution results can be cached for a short amount of time (mainly to handle cases like "ls -l" command), but these caches are invalidated when a file or a directory is created or removed.

# 5. IMPLEMENTATION

*QueryFS* is written Common Lisp, because parsing queries have to be translated into the main implementation language and it is way simpler with Lisp language family. Some parts of *QueryFS* and *CL-FUSE* currently require *Steel Bank Common Lisp* to run.

The lowest two levels written in Lisp are a wrapper around *FUSE* library implemented using *CFFI* and a more Lisp-like API for implementing *FUSE* filesystem without constant use of *CFFI*.

Next abstraction level allows defining filesystem layout in a declarative syntax without reimplementing path processing each time. It is made easier by the fact that Lisp programs are represented by trees in the most explicit way. There are two ways to write such a description. User can specify a literal tree structure with attributes in nodes; Lisp macrosystem allowed to create a simpler syntax for creating

"standard" nodes. For example, `(mk-file "README" "This is QueryFS")` and `‘(:TYPE :FILE :NAME "README" :CONTENTS ,(LAMBDA () "This is QueryFS") :WRITER NIL :REMOVER NIL)` mean the same: a file named "README" with text "This is QueryFS" should be created in the directory described by containing expression, it should not be writeable or removable. Operations currently used in *QueryFS* plugins are: "mk-file", "mk-dir", "mk-symlink" for describe filesystem contents, "mk-creator" for describing entry addition and "mk-pair-generator" for easier generation of filesystem structure based on information retrieved from external sources or computed at run time. The first three operations just accept expressions that will be evaluated to retrieve their names and contents. For files there can be extra expressions to handle file modification or removal. "mk-creator" accepts expression that need to be evaluated to create a file or directory entry in containing directory.

The last operation, "mk-pair-generator", requires an expression returning list of contents and an expression with a free parameter which can give details about each entry. The first expression returns a list of lists, where first entry of each list is entry name and the rest should be used when evaluating entry details.

There is also a more general operation, "mk-generator", which allows use of independent content lister and name parser. This allows special features like allowing to access HTTP URLs by accessing "http/www.example.org/path/to/file". Currently, no *QueryFS* plugin is able to generate code using this feature.

Next level is *QueryFS* itself. As described in the previous section, all the functionality of its core is related to handling of queries and plugins.

Loading plugins is done in a very straightforward way. There are some checks allowing to specify either full path or just the file name (if it is in the plugin directory); but basically it is one "load" call wrapped in error handling. Some of the query parsing code in the core is present only to be used by plugins. More specifically, there are two macros, "def-query-parser" and "def-linear-query-parser" to generate code that can be used in any plugin.

The first macro, "def-query-parser", acts in a way similar to Lisp function definition syntax. It simply defines a function with the specified body and registers it as query parser for specified type of queries. The second one assumes that query can be parsed by reading first "word" in Lisp sense and looking for it in a list of actions. It generates an invocation of the first macro and additional code to do the matching. This approach allows reusing the parser components in plugins.

Plugins are loaded as is and can select their own namespace to use. They are supposed to use the same namespace as the core *QueryFS* code.

To describe query grammar for a parser, one can use "Parser Expression Grammars". *Esrap-PEG*, a wrapper around existing *Esrap* packrat parser supporting stadard "PEG" syntax, was developed for the needs of *QueryFS*.

Loading queries is only a bit more complicated. Each query is processed by one of a few query parsers; it also gets loaded into its own namespace.

Currently, the most polished plugin is SQL2. It provides a syntax based on bash and SQL to represent results of SQL queries as directories.

```
 transient master_password "" setter "::password"
master_password

for p in "select username || '@' || service, password from
username is not null and ${master_password} <> ''" encrypt-
$(with-file $name do on-read $p[1]; done)

mkdir "by-service" do

  on-create-dir name "insert into passwords (service) value

  grouped-for srv in "select distinct service from password
  ${master_password} <> ''" do on-create-file name "insert
  (service, username) values (${srv[0]}, ${name})"

  with-file "::remove::" do on-read "" on-write data "delet
  passwords where service = ${srv[0]}" done

  for un in "select username, password from passwords where
  ${srv[0]} and username is not null" encrypt-by $master_pa
  with-file $name do on-read $un[1] on-write data "update p
  password = ${data} where username = ${name} and service =
  on-remove "delete from passwords where username = ${name}
  = ${srv[0]}" done) done done
```

# 6. STABILITY AND SECURITY

*QueryFS* uses *CL-FUSE* functionality to catch errors in run time. So a query with a mistake should not easily take down the entire filesystem instance. On the other hand, both malicious query and malicious plugin amount to arbitrary code running with user priviliges, so untrusted plugins and queries should not be run.

As the queries are processed with plugins, plugins may limit code generation to exclude unsafe function calls. Unfortunately, doing this well requires developing a security model that can allow loading "safe" external libraries and has correct definition of "safe". For queries this may be worked around if plugins wrap the library calls they consider safe. Anyway, if a query uses SQL and it is supposed to issue "DELETE FROM" sometimes, it can just drop the database unless a powerful query analyzer is used. Security model for plugins is an even more complicated task, because they can do whatever queries can, but they are also the natural place to put wrapper over foreign code; and even well-intentioned third-party code may be useful to a malicious plugin if such third-party code can write to a file.

# 7. REUSABLE LIBRARIES WRITTEN

Regardless of your opinion about *QueryFS* project, you may find one of its library useful.

## 7.1 CL-Fuse

A wrapper around *FUSE* libraries for Common Lisp.

## 7.2 Esrap-PEG

A library to support standard programming language independent "PEG" syntax for parser generation.

# 8. FUTURE PLANS

One of the long-term ideas of *RelFS* project was storing metadata in a SQLite database in the directory containing relevant data. For example, a USB HDD could contain a picture archive and a database with metadata for all the pictures. User could attach the database to *QueryFS* instance and select files by metadata.

Another feature *QueryFS* could eventually support is helping user to manage DB schema for metadata if user wants to create metadata manually or using scripts. Currently setting up the tables has to be done manually.

A "SPARQL" plugin for *QueryFS* would allow convenient means of experimentation with NoSQL metadata storage.

From the point of view of concrete applications, creating a convenient schema for storing email would be a good demonstration. Currently I read all my email using special *QueryFS* queries, but there is much space for improvement to make search more convenient.

# 9. REFERENCES


[1] FUSE project, `http://fuse.sf.net`

[2] RelFS project, `http://relfs.sf.net`

[3] Holupirek, Grün, Scholl: BaseX and DeepFS — Joint Storage for Filesystem and Database. EDBT 2009 (Demo Track), March 2009. `http://www.inf.uni-konstanz.de/dbis/publications/download/joint_storage.pdf`

[4] libferris project, `http://libferris.com`

[5] Gorter, O.: Database File System An Alternative to Hierarchy Based File Systems. `http://tech.inhelsinki.nl/dbfs/dbfs-screen.pdf`