

# Dispatch by property in a functional language

Christopher Chedeau

Technical Report *n°1116*, Décembre 2011  
revision 2187

Climb is a generic image processing library. A generic algorithm interface often requires several different specialized implementations. Olena, a C++ library, solved this using properties.

We present a way to dispatch a function call to the best specialized implementation using properties in a dynamic programming language: Common Lisp. Then, we introduce examples of algorithms and properties used in image processing.

Climb est une bibliothèque de traitement d'image générique. L'interface générique d'un algorithme nécessite souvent plusieurs implémentations différentes spécialisées. Olena, une bibliothèque C++, a résolu ce problème en rajoutant des propriétés.

Nous allons présenter un moyen de rediriger un appel de fonction vers la meilleure implémentation spécialisée grâce aux propriétés dans un langage de programmation dynamique: Common Lisp. Nous allons ensuite montrer des exemples d'algorithmes et de propriétés utilisées dans le domaine du traitement d'image.

## Keywords

Climb, Common Lisp, Dispatch, Properties, Image Processing



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

## Copying this document

Copyright © 2011 LRDE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just “Copying this document”, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Olena Properties</b>	<b>5</b>
2.1	Properties used for Dispatch . . . . .	5
2.2	C++ Implementation . . . . .	6
2.3	Analysis . . . . .	7
<b>3</b>	<b>Lispy Way</b>	<b>8</b>
3.1	Shift example . . . . .	8
3.2	Generalized Dispatch . . . . .	9
3.3	Analysis . . . . .	10
<b>4</b>	<b>Other Dispatch Implementations</b>	<b>11</b>
4.1	Javascript Full Dispatch . . . . .	11
4.2	Python Generic Dispatch . . . . .	11
4.3	MetaObject Protocol Dispatch . . . . .	12
4.4	MOP Filtered Dispatch . . . . .	13
4.5	Javascript Multimethod . . . . .	13
4.6	Haskell Function Pattern Matching . . . . .	14
<b>5</b>	<b>Conclusion</b>	<b>15</b>
<b>6</b>	<b>Bibliography</b>	<b>16</b>

# Chapter 1

## Introduction

It is often not possible to write one generic and optimized version of an algorithm that fits all inputs. Instead, we often write multiple specialized versions based on inputs characteristics. A dispatch technique is required to chose the best implementation. The implementation of the dispatch mechanism is closely tied to the programming language capabilities. This reports attempts to see how languages impacts dispatch implementation and philosophy.

In the first part, we explain how Olena represents inputs characteristics with properties and implements a compile-time dispatch algorithm using C++ metaprogramming techniques. Then, we propose an alternative dispatch algorithm that fits the Common Lisp mentality better, it provides better expressiveness along with a handy custom syntax. Finally, we make a round-up of different generic dispatch techniques in various programming languages in order to understand what are the common dispatch problematics.

## Chapter 2

# Olena Properties

Properties are used in order to write specialized implementations of a function. It is the case when you cannot write a single generic function that handles all the inputs. Specialized implementations are also written in order to improve performance for property combinations where there are known optimizations.

### 2.1 Properties used for Dispatch

In Olena, there are 37 properties with a total of 141 different values. Around half of them are about images, the other half describes other concepts of the library such as accumulator, function, operators, site set, window and value. We restrict our study to properties that are used for method dispatch. A listing is available in [Figure 2.1](#).

Data Type	Property Name	Property Values
image	category	any, morpher, primary
	dimension	any, one_d, three_d, two_d
	ext_domain	any, extendable, none, some
	ext_io	any, read_only, read_write
	kind	any, logic
	quant	any, low
	speed	any, fastest
	value_access	any, direct
	value_alignment	any, with_grid
	value_storage	any, disrupted, one_block, piecewise, singleton
	vw_io	any, read_write
	vw_set	any, uni
site_set	arity	any, multiple, unique
	nsites	any, known
value	nature	any, floating, scalar, vectorial
	quant	any, high, low
window	definition	any, multiple, unique, varying
	support	any, regular

Figure 2.1: Example of Olena Properties used for Dispatch

Properties in Olena can be seen as a combination of three parts:

**Data Type** Properties are specific to a data type.

**Property Name** The main goal of a property is to decide what version of the algorithm to use. It describes low-level implementation details of data types.

**Property Value** Unless the data type implementation has this property, the value is set to **any**. There can be several mutually exclusive values for a property.

## 2.2 C++ Implementation

Olena resolves the specialized implementation at compile time thanks to the SCOOP2 paradigm (Géraud and Levillain (2008)). Through an example, we are going to show the code required to write specializations of a method. The shift operation has two distinct specializations based on the window argument properties as described in Figure 2.2. The specialization (1) works with a rectangular support, a unique definition and a fixed size. The specialization (2) also works on regular support but requires multiple definitions and has no constraint on the size.

	definition				support		size	
	any	unique	multiple	varying	any	rectangular	any	fixed
Specialization (1)		✓				✓		✓
Specialization (2)			✓			✓	✓	✓

Figure 2.2: Specialization constraints for the shift algorithm

In order to implement this in C++, properties are separated into two categories.

**Discriminant Properties** The definition property can discriminate between the two specializations. This is useful to know as we can use the C++ dispatch mechanism based on this property to choose the right specialization.

**To-Check Properties** In order to have meaningful error message when there is no matching specialization, we need to check for all the combination of properties that are not valid specializations.

Once this categorization work is done, the implementation is straightforward as seen in Figure 2.3. It is just a matter of writing the proper dispatch method and property checks. However, this is a repetitive and error-prone operation.

```

1 template <typename W> mln_regular(W)
2 shift_(trait::window::definition::unique, W& win, mln_dpsite(W)& dp)
3 {
4     // Checks for non-valid property combination
5     mlc_is(mln_trait_window_size(W), trait::window::size::fixed)::check();
6
7     /* Specialized implementation (1) */
8 }
9
10 template <typename W> mln_regular(W)
11 shift_(trait::window::definition::multiple, W& win, mln_dpsite(W)& dp)
12 {
13     /* Specialized implementation (2) */
14 }
15
16 template <typename W> mln_regular(W)
17 shift_(Window<W>& win, mln_dpsite(W)& dp)
18 {
19     // Checks for non-valid property combination
20     mlc_is(mln_trait_window_support(W), trait::window::support::regular)::check();
21     mlc_is_not(mln_trait_window_definition(W), trait::window::definition::any)::check();
22     mlc_is_not(mln_trait_window_definition(W), trait::window::definition::varying)::check();
23
24     // Dispatch on definition property
25     return shift_(mln_trait_window_definition(W)(), exact(win), dp);
26 }

```

Figure 2.3: Shift specialization in Olena

## 2.3 Analysis

The dispatch by property implemented in Olena has one fundamental characteristic: it is done at compile time. It means that there is no associated runtime cost which is an extremely good feature in a performance oriented library. In order to achieve a static dispatch in C++, some trade-offs were made.

- The implementation of the framework required to make the dispatch work is not trivial, it adds a new layer to maintain to the project. Many macros were added in order to hide the complexity and provide a friendly syntax. It also adds a non-negligible compilation time.
- Because properties are divided into two groups and used in two different contexts, finding the list of property values accepted by a specialization requires some work.
- Adding new specializations may add new discriminant properties and therefore require to alter the prototype of previously written specializations. Adding new values for a property requires to add checks for all the specializations that uses the property. This makes property and specialization management not scalable.

## Chapter 3

# Lispy Way

The implementation of a static dispatch is an extremely difficult task. The implementation in Olena is a feat of strength that extensively uses C++ type checking and meta-programming techniques to do it. Common Lisp has no built-in facilities for static code analysis, it is hard to compete in the same field.

However, Common Lisp is a functional language with the ability for the library developer to sculpt the best syntax for the problem. With those features in mind, we are going to write a more generic and easy to implement dispatch system based on [Denuzière \(2011\)](#).

### 3.1 Shift example

Let us get back to the shift example. So far, we have been reading Figure 2.2 from a property perspective. We were focused on the characteristics of the properties in order to translate them in C++. But the intuitive way is to read the table line by line. We want to know for each specialization what properties it accepts. This is also what we expect to see in the code.

```
1 (defalgo shift ((win
2                 window
3                 (prop :support :regular)
4                 (prop :definition :unique)
5                 (prop :size :fixed))
6                 (dp
7                 dpsite))
8           ; Specialized implementation for definition::unique
9 )
10 (defalgo shift ((win
11                 window
12                 (prop :support :regular)
13                 (prop :definition :multiple))
14                 (dp
15                 dpsite))
16           ; Specialized implementation for definition::multiple
17 )
```

Figure 3.1: Lisp Implementation



Given this requirement, we can create a helper called `defalgo` that let us declare our specializations as seen in Figure 3.1. It has the same syntax as a usual Common Lisp method declaration but accepts a wider range of argument specializers, in this case, the `prop` specializer.

`defalgo` implementation is straightforward. It keeps an internal structure that contains the list of all the specializations of an algorithm. Each specialization has a list of tests to be applied on the arguments. The first specialization with arguments passing all the tests is going to be called. Figure 3.2 illustrates the structure generated by the `shift` example. Lines in red are tests that differ from both specializations.

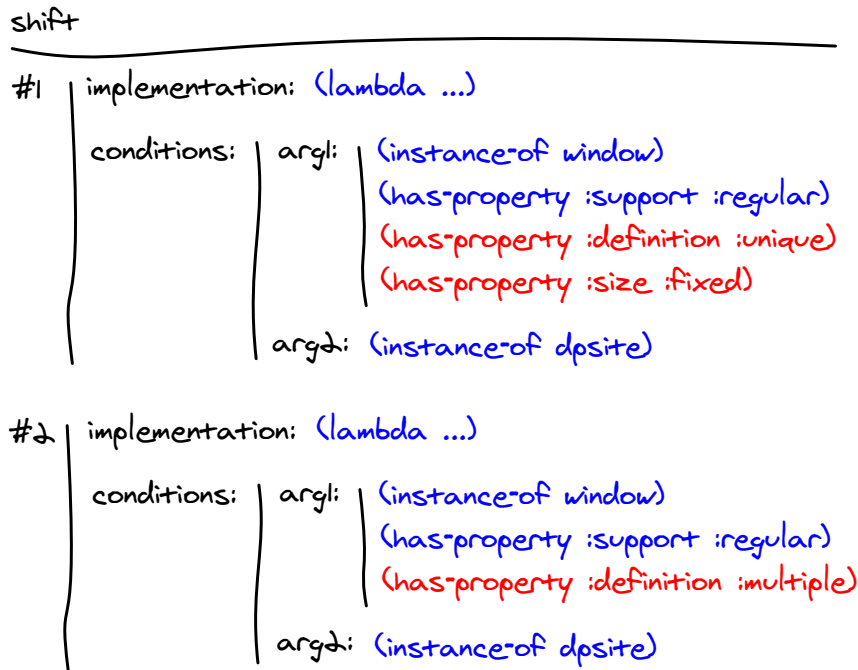


Figure 3.2: Shift Specializations

## 3.2 Generalized Dispatch

The strength of the method is the ability to use any lambda function as argument specializer. We are no longer constrained by a language feature such as C++ that imposes specializers to be types. In the `shift` example, we've been using two specializers: `instance-of` and `has-property`. Since we control `defalgo`, we can wrap an implicit `instance-of` around an argument specializer that is a class.

In order to present the different ways to specify an arbitrary argument specializer, we are going to use as an example the Fibonacci numbers. It is a sequence defined such as:

$$F_0 = 0, \quad (3.1)$$

$$F_1 = 1, \quad (3.2)$$

$$F_n = F_{n-1} + F_{n-2} \quad (3.3)$$

Figure 3.3 shows three ways to encode a specializer.

**A lambda expression** It takes one argument and returns a boolean telling whether the specialization is accepted or not.

**A reference to a specializer** It is often handy to define a generic specializer elsewhere and use a reference to it in the algorithm declaration.

**Any expression that evaluates to a specializer** The expression will be evaluated once and the result will be used as specializer.

1 (defalgo fibo ((n	1 (defun <2 (n) (< n 2))	1 (defun is (a)
2                   (lambda (n)	2	2   (lambda (b)
3                   (< n 2))))	3 (defalgo fibo ((n #'<2))	3   (eq a b)))
4    n)	4    n)	4
		5 (defalgo fibo ((n (is 0)))
		6   0)
		7 (defalgo fibo ((n (is 1)))
		8   1)
1 (defalgo fibo (n)		
2   (+ (fibo (- n 2)) (fibo (- n 1))))		

Figure 3.3: Various ways to encode Fibonacci special cases

### 3.3 Analysis

The generic dispatch provides a huge expressive power since it can accept any user function as argument specializer. However, this comes at both at a cost of performance and integration to common development chain.

- This generic dispatch has a runtime cost. Every time you call a function, you have to go through the list of all argument specializers until you find a matching specialization. Since it is ultra-generic, it is hard if not impossible to write an efficient caching mechanism for the dispatch lookup.
- One way to develop code is to keep a Common Lisp instance running and evaluate the current file every time we make a change. This does not generate the intended result. Instead of having the newest version available in the environment, the old version is still at the top of the specialization list and is being used.
- In the same vein, the order of the specialization declarations is important. You must always declare the most general version last, or it will shadow all the other specializations.

## Chapter 4

# Other Dispatch Implementations

In order to see why generic dispatch using Common Lisp is a great fit, we are going to compare it against other dispatch methodologies and implementation of generic dispatch in other languages.

### 4.1 Javascript Full Dispatch

Before writing the Common Lisp implementation of generic dispatch, I wrote one in Javascript called Full Dispatch ([Chedeau \(2010\)](#)). The implementation is very similar using functions as specializers. However, the syntax is not nearly as good as in Common Lisp. Javascript does not have a macro facility that let us transform the syntax into something intuitive.

As an example, we implement the Ackermann function in Figure 4.1 defined as following:

$$A(m, n) = n + 1, \text{ if } m = 0 \quad (4.1)$$

$$A(m, n) = A(m - 1, 1), \text{ if } m > 0 \text{ and } n = 0 \quad (4.2)$$

$$A(m, n) = A(m - 1, A(m, n - 1)), \text{ if } m > 0 \text{ and } n > 0 \quad (4.3)$$

```
1 ack = FullDispatch()
2 ack.add [Zero, null], (m, n) -> n + 1
3 ack.add [StrictPositive, Zero], (m, n) -> ack(m - 1, 1)
4 ack.add [StrictPositive, StrictPositive], (m, n) -> ack(m - 1, ack(m, n - 1))
```

Figure 4.1: Ackermann function using Javascript Full Dispatch

### 4.2 Python Generic Dispatch

Michael Axiak implemented a generic dispatch method in Python ([Axiak \(2010\)](#)). He managed to improve the syntax compared to the Javascript version using Python decorators. It looks better but there is a problem, the argument name and specializers are not next to each other. It makes it hard to visually find the link when there are more than one argument.

Figure 4.2 demonstrate how to implement Fibonacci Numbers using a dispatch based on Python decorators.

```

1 @dispatch(inside(0, 1))
2 def fibo(n):
3     return n
4
5 @dispatch(int)
6 def fibo(n):
7     return fibo(n - 1) + fibo(n - 2)

```

Figure 4.2: Fibonacci Numbers using a dispatch based on Python decorators

### 4.3 MetaObject Protocol Dispatch

Common Lisp is bundled with an object oriented layer called MetaObject Protocol ([Keene and Gerson \(1989\)](#)) (MOP). Along with all the class definition, it provides a traditional dispatch mechanism. The argument specializers are limited to class inheritance and exact value match through eql specializers.

The main conceptual difference between MOP dispatch and our generic dispatch is the fact that the matched function will be the most specialized specialization instead of the first specialization found that matches. By changing this behavior, it allows to easily implement arbitrary lambda as argument specializers and specialization on any amount of arguments.

A summary of the differences between our generic dispatch and MOP dispatch is available in Figure 4.3.

Type	defalgo	defmethod
Specialization Behaviors		
Specialization Selection	First found	Most specialized
Specialization on arbitrary amount of arguments	✓	×
Allowed Specializers		
Instance of Class	✓	✓
EQL Specializer	✓	✓
Arbitrary Lambda	✓	×

Figure 4.3: Differences with MOP

We took a lot inspiration from MOP in order to design the syntax of the generic dispatch. In Figure 4.4 you can see the implementation of Fibonacci Numbers using MOP. It only requires to rename defmethod by defalgo, eql by is and to reorder the declarations in order to fall back to the generic implementation of Figure 3.3.

```

1 (defmethod fibo (n)
2   (+ (fibo (- n 1)) (fibo (- n 2))))
3
4 (defmethod fibo ((n (eql 1)))
5   n)
6
7 (defmethod fibo ((n (eql 0)))
8   n)

```

Figure 4.4: Fibonnaci using MOP

## 4.4 MOP Filtered Dispatch

The idea behind Pascal Costanza Filtered Dispatch ([Costanza et al. \(2008\)](#)) is to enhance MOP dispatch mechanism with filters. Instead of doing an eql specializer directly on the argument, we are going to apply it on the argument filtered by a function. This way, the whole MOP framework with specialization selection is still working, while providing more expressiveness in the dispatch.

Figure 4.5 is an example of the factorial implementation. The eql specializer is used on the sign of the argument instead of the argument itself.

```

1 (defun sign (n)
2   (cond
3     ((< n 0) 'neg)
4     ((= n 0) 'zero)
5     ((> n 0) 'pos)))
6
7 (defmethod factorial :filter :sign ((n (eql 'pos)))
8   (* n (factorial (- n 1))))
9
10 (defmethod factorial :filter :sign ((n (eql 'zero)))
11   1)
12
13 (defmethod factorial :filter :sign ((n (eql 'neg)))
14   (error "Factorial_not_defined_for_negative_numbers."))

```

Figure 4.5: Factorial using MOP Filtered Dispatch

## 4.5 Javascript Multimethod

Kris Jordan implemented a similar idea in the multimethod library. An example is available in Figure 4.6<sup>1</sup> where the objective is to get the service identifier of the contact based on its service field.

To make a parallel with the previous dispatch, `dispatch` is the equivalent of a filter and `when` of an eql specializer. The implementation takes advantage of the method chaining design pattern ([Chedeau \(2011\)](#)) in order to specify the different specializations.

<sup>1</sup> All Javascript examples from now on will be written using CoffeeScript syntax to make it easier to understand.

```
1 contacts = [  
2   { name:'Jack', service:'Twitter', handle: '@jack' },  
3   { name:'Diane', service:'Email', address:'diane@gmail.com' },  
4   { name:'John', service:'Phone', number: '919-919-9191' }  
5 ]  
6  
7 getId = multimethod()  
8   .dispatch((contact) -> contact.service)  
9   .when('Twitter',  
10     (contact) -> contact.handle)  
11   .when('Email',  
12     (contact) -> contact.address)  
13   .default(  
14     (contact) -> contact.name)
```

Figure 4.6: Example use of multimethod.js library

## 4.6 Haskell Function Pattern Matching

Haskell provides pattern matching at function level. You can write multiple specializations, the arguments will be evaluated in order and the first specialization that matches will be used. One interesting aspect is the ability to destructure an argument into different variables within the specialization prototype.

Figure 4.7 shows how to write Quicksort algorithm using function pattern matching in Haskell.

```
1 quicksort [] = []  
2 quicksort (p:xs) = (quicksort (filter (< p) xs)) ++ [p] ++ (quicksort (filter (>= p) xs))
```

Figure 4.7: Example of Haskell Function Pattern Matching

## Chapter 5

# Conclusion

In this report, we first showed an implementation of static property dispatch in C++. It extensively uses C++ features in order to dispatch for a very specific need. Then we introduce a generic dispatch implementation that removes the compile-time constraint but which is able to express a huge range of dispatch behaviors. In particular, the previous dispatch is trivial to implement. Finally, we compare our generic dispatch to dispatch mechanisms from various languages and libraries both in term of syntax and expressiveness power.

In the end, there are as hundreds of ways to implement an expressive dispatch mechanism, many of them being trivial to implement, but none really stood out. We believe that it is because the use cases are limited. When writing an application or library, you don't often write multiple implementations of the same function that needs to be dispatched with complex rules. And in the case where you have to, you can use standard language features such as pattern matching, switch statement or even if statements to express that rule.

The conclusion is unfortunate, but even if the proposed general dispatch has a better expressive power and syntax than most of the other dispatch mechanisms reviewed, it solves a problem that does not appear in real applications.

## Chapter 6

# Bibliography

Axiak, M. (2010). Python generic dispatch. <http://mike.axiak.net/blog/2010/06/25/python-generic-dispatch/>.

Chedeau, C. (2010). Javascript - full dispatch (extended form of multimethod). <http://blog.vjeux.com/2010/javascript/javascript-full-dispatch-multimethod.html>.

Chedeau, C. (2011). Component trees and chaining operators in climb. Technical Report 1108, EPITA Research and Development Laboratory (LRDE).

Costanza, P., Herzeel, C., Vallejos, J., and D'Hondt, T. (2008). Filtered dispatch. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 4:1–4:10, New York, NY, USA. ACM.

Denuzière, L. (2011). Designing the user interface for a common lisp generic library. Technical report, EPITA Research and Development Laboratory (LRDE).

Géraud, Th. and Levillain, R. (2008). 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.

Keene, S. and Gerson, D. (1989). *Object-oriented programming in Common LISP: a programmer's guide to CLOS*. Addison-Wesley.