Property-based dispatch in functional languages

Christopher Chedeau

Technical Report n^o1116, 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, solves 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++, résoud ce problème en rajoutant des propriétés.

Nous présentons 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 – http://www.lrde.epita.fr/

Copying this document

Contents

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.

The usual characteristic used for dispatch is the type of the argument. Some languages also support equality with a constant but it is about it. While they have proven to be good to express a wide range of use cases, they do not properly fit our needs for image processing algorithms. We want to be able to express complex characteristics with all the possibilities of the programming language, not only class hierarchy and equality.

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 uses Common Lisp strengths. It provides better expressiveness thanks to lambda functions along with a handy syntax powered by macros. Finally, we make a round-up of different generic dispatch techniques in various programming languages in order to understand what are the common dispatch concerns.

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 describing other concepts of the library such as accumulators, functions, operators, site sets, windows and values. A listing of some of those properties 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

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**.

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 takes a window and an offset. It moves every point of the window with the offset. For example, Figure 2.2 shows a 4-connectivity window shifted by the point $(3, 3)$.

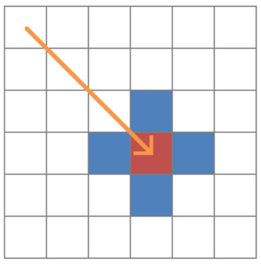

Figure 2.2: Shift Example

The shift operation has two distinct specializations based on the window argument properties as described in Figure 2.3. In order to shift a window with a point, you must be able to do addition between coordinates of points. The property support with the value regular express this hypothesis. Hence, all the specialization require the window to be on a regular support.

A window can be defined in multiple ways. The most common is to give a list of points stored in a site set. The window will be said to have a unique definition. When the window is defined a combination of several site sets, it will be said to have a multiple definition.

The specialization (1) only works on a unique definition and also requires the fixed size (the number of points must not change during the execution of the program). The specialization (2) works on multiple definitions and has no constraint on the size.

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

Figure 2.3: 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 chose 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 2.4. It is just a matter of writing the proper dispatch method and property checks. However, this is a repetitive and error-prone operation.

```
template <typename W> mln_regular(W)
shift_(trait::window::definition::unique, W& win, mln_dpsite(W)& dp)
{
        // Checks for non−valid property combination
        mlc_is(mln_trait_window_size(W), trait::window::size::fixed)::check();

        /* Specialized implementation (1) */
}

template <typename W> mln_regular(W)
shift_(trait::window::definition::multiple, W& win, mln_dpsite(W)& dp)
{
        /* Specialized implementation (2) */
}

template <typename W> mln_regular(W)
shift(Window<W>& win, mln_dpsite(W)& dp)
{
        // Checks for non−valid property combination
        mlc_is(mln_trait_window_support(W), trait::window::support::regular)::check();
        mlc_is_not(mln_trait_window_definition(W), trait::window::definition::any)::check();
        mlc_is_not(mln_trait_window_definition(W), trait::window::definition::varying)::check();

        // Dispatch on definition property
        return shift_(mln_trait_window_definition(W)(), exact(win), dp);
}
```

Figure 2.4: 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 crucial factor in a performance oriented library. In order to achieve a static dispatch in C++, some trade-offs are 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 are added in order to hide the complexity and provide a friendly syntax. It also increases 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

Peroperties in Lisp

The implementation of a static dispatch is a difficult task. The implementation in Olena is a feat of strength that extensively uses C++ type checking and meta-programing techniques to do it. 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 propose 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 2.3 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.

```lisp
(defalgo shift (
  (win
    :properties (
      :support :regular
      :definition :unique
      :size :fixed)
    window)
  (dp dpsite))
  ; Specialized implementation for definition::unique
)
(defalgo shift (
  (win
    :properties (
      :support :regular
      :definition :multiple)
    window)
  (dp dpsite))
  ; Specialized implementation for definition::multiple
)
```

Figure 3.1: Lisp Implementation

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

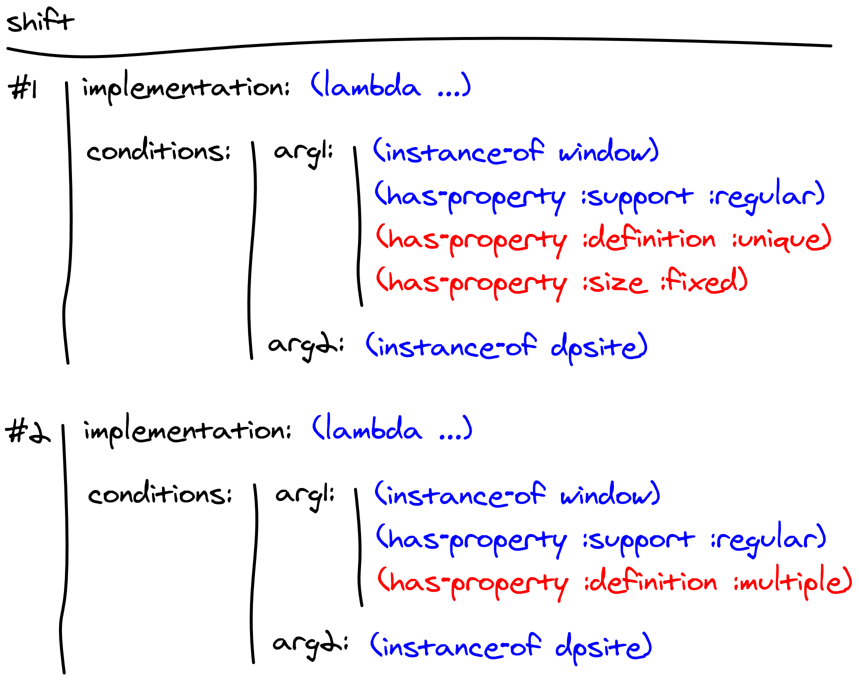
shift
#1
implementation: (lambda ...)
conditions:
arg1:
(instance-of window)
(has-property :support :regular)
(has-property :definition :unique)
(has-property :size :fixed)
arg2: (instance-of dpsite)
#2
implementation: (lambda ...)
conditions:
arg1:
(instance-of window)
(has-property :support :regular)
(has-property :definition :multiple)
arg2: (instance-of dpsite)

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:

$$F_0 = 0, \tag{3.1}$$

$$F_1 = 1, \tag{3.2}$$

$$F_n = F_{n-1} + F_{n-2} \tag{3.3}$$

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.

```
(defalgo fibo ((n
                (lambda (n)
                  (< n 2))))
  n)
```

```
(defun <2 (n) (< n 2))

(defalgo fibo ((n #'<2))
  n)
```

```
(defun is (a)
  (lambda (b)
    (eq a b)))

(defalgo fibo ((n (is 0)))
  0)
(defalgo fibo ((n (is 1)))
  1)
```

```
(defalgo fibo (n)
  (+ (fibo (- n 2)) (fibo (- n 1))))
```

Figure 3.3: Various ways to encode Fibonnaci 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 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, we wrote one in Javascript called Full Dispatch (Chedeau (2010b)). 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:

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

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

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

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

```python
@dispatch(inside(0, 1))
def fibo(n):
    return n

@dispatch(int)
def fibo(n):
    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 CLOS that implements the 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 of 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.

```
(defmethod fibo (n)
  (+ (fibo (− n 1)) (fibo (− n 2))))

(defmethod fibo ((n (eql 1)))
  n)

(defmethod fibo ((n (eql 0)))
  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 to 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.

```
(defun sign (n)
  (cond
    ((< n 0) 'neg)
    ((= n 0) 'zero)
    ((> n 0) 'pos)))

(defmethod factorial :filter :sign ((n (eql 'pos)))
  (* n (factorial (− n 1))))

(defmethod factorial :filter :sign ((n (eql 'zero)))
  1)

(defmethod factorial :filter :sign ((n (eql 'neg)))
  (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[1] 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 in order to specify the different specializations.

[1]All Javascript examples from now on will be written using CoffeeScript syntax to make it easier to understand.

```
contacts = [
  { name:'Jack', service:'Twitter', handle: '@jack' },
  { name:'Diane', service:'Email', address:'diane@gmail.com' },
  { name:'John', service:'Phone', number: '919−919−9191' }
]

getId = multimethod()
  .dispatch((contact) −> contact.service)
  .when('Twitter',
    (contact) −> contact.handle)
  .when('Email',
    (contact) −> contact.address)
  .default(
    (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.

```
quicksort [ ] = [ ]
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.

This reports concludes a serie of three reports where we implemented all the generic aspects that Olena provides for image processing taking advantage of Common Lisp dynamic aspects. In Chedeau (2010a), we introduced genericity on data structures. An image can not only be represented by a regular 2d grid but also as a 3d hexagonal grid or even as a graph. In Chedeau (2011), we have seen that values are also subject to genericity. It is common to see intensity values represented with 8-bit integer and floating point, Red-Green-Blue channels. Sometimes it is convenient to hold other value types such as booleans or even images. Eventually, this report completes the library including genericity on implementations.

Chapter 6

Bibliography

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

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

Chedeau, C. (2010b). 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.