

Breakout Group Proposal: Predicate Dispatching

Jim Newton, Cadence Design Systems
Pascal Costanza, Vrije Universiteit Brussel

The current generic dispatch implementation in CLOS allows a generic function to specialize its parameters on classes or on single objects; moreover, the arbitration is done at runtime: When the generic function is called, CLOS sorts the applicable methods by order of their specificity and invokes a dynamically calculated effective method. This effective method calls the most specific method only after stashing away the remaining methods for further use later.

The checks for applicability can be understood as predicates. Cf. the following example:

```
(defmethod print-person ((p person))
  ...)
```

```
(defmethod print-person ((p (eql *joe*)))
  ...)
```

The method is applicable if a concrete argument *P* is indeed of class *PERSON* and the second method is applicable if it is EQL to **JOE**. The respective predicates, `(person-p` and `joe-p :-)` are `(lambda (obj) (typep obj 'person))` and `(lambda (obj) (eql obj *joe*))`.

There have been several approaches to conceptualizing a more general notion of predicate dispatching. The idea is to not only allow one to use the *TYPEP* and *EQL* predicates, but instead to generalize this to arbitrary predicates. For example, the following should do the obvious:

```
(defmethod print-person ((p (and (eql *joe*) (< 10:00 (time) 18:00))))
  ...)
```

This means, the method should only be applicable when the parameter is **JOE**, and only between 10:00 and 18:00.

As another example, we take a look at the following "predicate-less" definition of factorial that *almost* works.

```
;; 0! = 1
(defmethod ! ((n (eql 0)))
  1)
```

```
;; n! = n * (n-1)!
(defmethod ! ((n number))
  (* n (! (1- n))))
```

However, it fails when *n* is negative or a non-integer. If we had predicate dispatching, we could retrogressively repair the factorial function without having to resort to explicitly editing the existing partial implementation (which might be impossible in some systems).

```
;; (-n)! = -(n!)
(defmethod ! ((n (? minusp)))
  (- (! (- n))))
```

```
:: factorial of a float is factorial of the next lower integer times the float  
(defmethod ! ((n (? floatp)))  
  (* n (! (truncate n))))
```

Note that this example shows the delicate problem of determining the right order of applicable methods. Clearly, the class NUMBER is more specific than the class FLOAT and the predicate MINUSP. However, which is the most specific method when calling (! -3.2)?

Existing approaches for predicate dispatching don't provide general solutions to this problem. They most notably fail to deal with overlapping predicates such as FLOAT and MINUSP. Another example of two overlapping predicates is (< 10:00 (time) 18:00) and (< 14:00 (time) 22:00). Neither of the two predicates is clearly more specific than the other. As yet, existing approaches either restrict the predicate language to be used so that there is always a guaranteed order of predicates, or else they ignore the problem and allow for ambiguous dispatching.

A good idea may be to let the generic function decide on the order of predicates, or to reject predicates that it cannot deal with. In this way, predicate dispatching could be tuned to domain-specific needs. However, this also seems to require changes to the metaobject protocol and it is doubtful that it can be done without performance losses.

The goal of this breakout group is to discuss this idea, envision applications which could benefit from a good implementation, play around with a few examples, and ultimately propose a design for an extension of CLOS and/or its MOP that makes predicate dispatching practical (or else better understand why that goal is ultimately not achievable).

See <http://c2.com/cgi/wiki?PredicateDispatching> for an overview of predicate dispatching. See <ftp://publications.ai.mit.edu/ai-publications/2001/AITR-2001-006.pdf> for an existing CLOS-based approach. It's useful to have a good understanding of the CLOS MOP. See the book "The Art of the Metaobject Protocol" by Kiczales, des Rivieres and Bobrow for details.