Filtered Dispatch

Pascal Costanza Charlotte Herzeel
Jorge Vallejos Theo D'Hondt

Programming Technology Lab
Vrije Universiteit Brussel
B-1050 Brussels, Belgium
pascal.costanza | charlotte.herzeel | jorge.vallejos | tjdhondt @vub.ac.be

ABSTRACT

Predicate dispatching is a generalized form of dynamic dispatch, which has strong limitations when arbitrary predicates of the underlying base language are used. Unlike classes, which enforce subset relationships between their sets of instances, arbitrary predicates generally do not designate subsets of each other, so methods whose applicability is based on predicates cannot be ordered according to their specificity in the general case. This paper introduces a decidable but expressive alternative mechanism called filtered dispatch that adds a simple preprocessing step before the actual method dispatch is performed and thus enables the use of arbitrary predicates for selecting and applying methods.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*Object-oriented Programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

Method dispatch, predicate dispatch, generic functions

1. INTRODUCTION

A central contribution of object-oriented programming is ad-hoc polymorphism via dynamic method dispatch, which enables behavioral variations based on typically one *receiver* argument in object-centric languages, or potentially multiple arguments in languages based on generic functions. In object-centric languages, methods and their overriding relationships are defined along the inheritance chains of classes or objects, but even in the case of generic functions, method selection and combination is driven by the inheritance hierarchies in which the received arguments are involved.

Dynamic dispatch based on generic functions was originally introduced in EL1 [28], and has subsequently found its way into a number of other programming language, including Common Lisp [5], Dylan [23], MultiJava [8], and Fortress [1], to name a few. The common feature in these systems is that a generic function can select and apply methods based on the classes of the received arguments, allowing users to define new methods for new classes and thus specialize generic functions for their own purposes.

```
(defmethod fac ((n number))
  (* n (fac (- n 1))))

(defmethod fac ((n (eql 0)))
  1)
```

Figure 1: Factorial as a generic function.

There exist several extensions and variations of this basic approach, like the inclusion of aspect-style advice and metaobject protocols for influencing the exact semantics of method dispatch. A common extension is to specialize methods not only on classes, but also on single objects, such that specific objects can have their own behavior that deviates from the general behavior of their classes. For example, in the implementation of the factorial function in Fig. 1 implemented using Common Lisp, there are two methods defined, one on the class `number` computing the general case for the factorial function, and the other on the concrete number object 0, which simply returns 1.[1]

Since `eql` is a predicate in Common Lisp for object comparison (similar to `==`, or so, in other languages), a typical question that arises is whether this can be generalized to a dispatch mechanism based on other, arbitrary predicates. Indeed, it could be envisioned that the code above could be defined as in Fig. 2, using Common Lisp's `typep` predicate for checking whether an object is an instance of a certain type or class.[2]

However, the generalization to full predicate dispatch has its limitations because predicate implications cannot be decided in general. Consider the example in Fig. 3, assuming that `primep` tests for prime numbers, `oddp` for odd numbers and `evenp` for even numbers. If we call, say, `(print-number-property 2)`, the methods for prime and even numbers are applicable, but we cannot determine which method is the "most specific" one, because prime numbers are not a subset of even numbers and vice versa. This example illustrates why inheritance hierarchies are so useful: Subclassing

[1]Like in several other dynamic object-oriented languages, numbers are considered objects in Common Lisp.
[2]In Common Lisp, predicates usually end in `p`, hence `typep`.

```
(defmethod fac ((n (typep 'number)))
  (* n (fac (- n 1))))

(defmethod fac ((n (eql 0)))
  1)
```

Figure 2: Factorial as if by predicate dispatch.

guarantees an unambiguous specialization relationship that can be exploited in dynamic dispatch, due to the fact that a class designates a set of instances that is always a subset of the sets of instances designated by any of the class's superclasses. The inclusion of *eql specializers* in Common Lisp (also called *singleton specializers* in Dylan or some Scheme object systems) does not introduce any ambiguity here: Objects are just always considered more specific than their respective classes, because the set of a single object is always a subset of the set of instances designated by its class.

Nevertheless, there exist a number of systems that provide predicate dispatch [7, 14, 19, 27]. They differ in details of their design, but they all follow basically the same approach for resolving ambiguities when comparing predicates: The set of predicates that can be used for the purpose of method dispatch is restricted to a well-chosen subset which is not Turing complete and can thus be statically analyzed. This leads to a viable approach, but can be limiting in some circumstances, since users cannot extend predicate dispatch with their own arbitrary predicates in a straightforward way. In [7, 14], tests can be added which can contain arbitrary boolean expressions from the underlying programming language, but they are treated as blackboxes and the overriding relationship between two syntactically different expressions is considered ambiguous. This is unsatisfactory because a function like `print-number-property` cannot be easily implemented in this case.

This paper introduces a decidable but powerful generalized dispatch mechanisms based on a separate filtering / preprocessing step for arguments received by *filtered generic functions*. Such filters map arguments to representatives of equivalence classes, which are then used in place of the original arguments for method selection and combination. The thus invoked methods, however, can operate on the original arguments. We discuss use cases, an integration into the Common Lisp Object System (CLOS), some implementation details, related work, and ideas for future work.

2. TRADITIONAL DYNAMIC DISPATCH

2.1 Object-centric dispatch

In traditional object-centric systems, method invocation is triggered by messages being sent to objects where the objects then decide which method to execute based on an object- or class-specific mapping from message signatures to actual methods (aka method tables). Typically, such mappings are fixed for specific objects, which means that the dynamic state of a running system cannot (easily) influence the dispatch mechanism for a particular object anymore.

One solution is to use forwarding, which means that an object that receives a message forwards it to another object, based on some arbitrary criteria. A popular example for that approach is the State pattern [15], which enables separation

```
(defmethod print-number-property ((n (primep)))
  (print "This is a prime number."))

(defmethod print-number-property ((n (oddp)))
  (print "This is an odd number."))

(defmethod print-number-property ((n (evenp)))
  (print "This is an even number."))
```

Figure 3: Ambiguous predicate method specificity.

of method definitions according to the state of a particular (receiver) object. For example, Fig. 4 shows a diagram for a use of the State pattern (from [15]): An object representing a TCP connection may behave differently according to whether the connection is in the state `established`, `listen` or `closed`. Here, the state is stored as a field in the TCP connection object and actually refers to the object that contains the methods appropriate for the current state. In general, messages can be forwarded to arbitrary objects based on arbitrary dynamic conditions.

A drawback of message forwarding is that it introduces object identity problems: The current `self` or `this` reference is not the original receiver of the message anymore for which the current method is being executed. This is typically referred to as the object schizophrenia problem [22]. There are a number of suggestions to solve certain aspects of that problem, for example by rebinding `self` or `this` to the original receiver in delegation-based languages [18, 25], or by grouping delegating objects in *split objects* and letting them share a common object identity [4]. However, the core problem that it is not straightforward to unambiguously *refer* to a single object anymore remains: A programmer has to ensure that the right object in a delegation chain is being referred to, and even in split objects, the correct *role* of an object has to be selected in the general case.

Smalltalk and CLOS, among others, provide ways to change the class of an object, and thus its method table: Smalltalk's `become:` replaces an object `o1` with another `o2`, such that all references to `o1` will refer to `o2` afterwards while keeping the identity of `o1`. CLOS's `change-class` directly changes the class of an object without affecting its object identity either. In principle, those operators can be used to implement State-like patterns. However, it is generally advised not to use such operators: Smalltalk's `become:` does not check for compatible layouts of the objects involved, so can actually lead to (delayed) system crashes if used incorrectly [24]. The Common Lisp specification also explicitly advises against the use of `change-class` [2]: When a method is selected based on the class of a particular object, changing the class of that object while the method is executed may have "undefined results", which is due to optimizations that a Common Lisp compiler is allowed to perform. The same holds for Smalltalk's `become:` for slightly different reasons: If an object instance is changed via `become:`, it is not clear whether currently executing methods bodies see the old or the new object [24].

Gilgul's *referent assignment* operator [9] can be regarded as a cleaned-up version of Smalltalk's `become:`. Gilgul guarantees that object layouts are compatible and provides means to gracefully wait for, or return early from, methods that are executing on objects to be replaced by other objects. How-

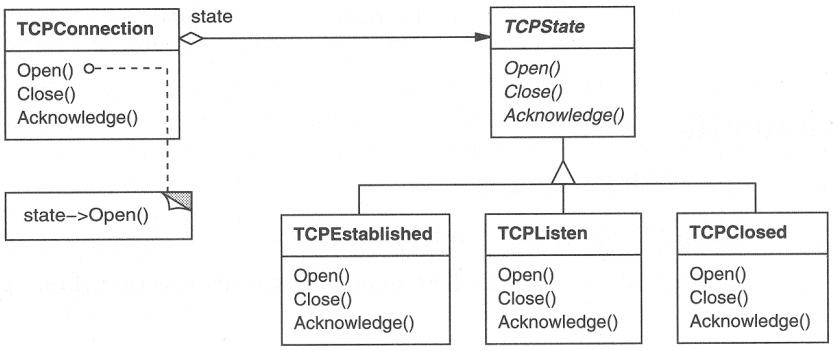

Figure 4: A use of the State pattern (from [15]).

```
(defmethod fac ((n (> 0)))
  (* n (fac (- n 1))))

(defmethod fac ((n (eql 0)))
  1)

(defmethod fac ((n (< 0)))
  (error "Fac not defined for negative numbers."))
```

Figure 5: Fixed factorial as if by predicate dispatch.

ever, while this solves the technical issues, it turns out that typical idioms in Gilgul use forwarding between the replaced and the original version of an object in order to enable incremental adaptations of methods, which effectively means that we get the same object schizophrenia problems again as in forwarding and delegation.

2.2 Generic functions

Recall the factorial example from Fig. 1. It states that the factorial function has a general case for all numbers, and a specific base case for the number 0. However, this is not correct: The factorial function is actually not defined for negative numbers, but this case is not covered in the definition in Fig. 1. Calling that version of factorial with a negative number will actually iterate until the memory is filled with large negative numbers, and then probably just crashes. It would be good if we could define factorial similar to Fig. 5 to avoid such a problem. Alas, when generalized, this leads exactly to the problems of predicate dispatch discussed in the introduction to this paper. The reason is that it is not possible to restrict method specializers to user-defined sets of instances, and that subclassing is not enough to deal with such a situation.

3. FILTERED DISPATCH

In this paper, we introduce a new dispatch mechanism called *filtered dispatch*. Filtered dispatch is based on generic functions and extends them with a filtering step where the arguments received by a generic function are mapped to other values based on user-defined mapping functions. Those filtered values are then used to perform the actual selection and execution of applicable methods. Nevertheless, the methods see the original objects as received by the generic function, and not the filtered ones.

We first introduce filtered dispatch and filtered functions using a few examples to illustrate their expressive power, and then discuss the syntax and sematics of filtered functions in more detail, followed by a discussion of (some) implementation aspects.

3.1 Filtered functions by example

3.1.1 Factorial

In order to be able to use filtered functions, we need to provide filter functions that map received arguments to values that we actually want to base our dispatch on. For the factorial function, we want to distinguish between negative and positive numbers, and the number zero. We therefore define a function `sign` that makes this distinction for us.

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

We can now define a filtered function `fac` and specify that it uses that filter.

```
(define-filtered-function fac (n)
  (:filters (:sign (function sign))))
```

This definition is much like CLOS's `defgeneric` for "announcing" generic functions: It names a generic function (here `fac`) and gives it a signature (here `(n)`, stating that it takes exactly one parameter `n`). Additionally, it defines a filter with the name `:sign` and specifies that the above defined function `sign` is to be used for that filter.
We can now define methods for the filtered function `fac`:

```
(defmethod fac :filter :sign ((n (eql 'pos)))
  (* n (fac (- n 1))))

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

(defmethod fac :filter :sign ((n (eql 'neg)))
  (error "Fac not defined for negative numbers."))
```

We use the qualifiers `:filter :sign` in the method definitions to indicate that we indeed want to use the `:sign` filter

for method selection. We then use eql specializers to ensure that the method definitions are applicable for the three different cases that the `sign` function yields.

Methods defined on filtered functions will always see the original arguments, not the filtered ones. Therefore, the definitions for `fac` are correct: The first method will be called if the argument is a positive number and computes the general case of the factorial function. The second method will be called if the argument is 0 and returns 1. The third method will be called if the argument is a negative number and signals an error.

The filter function `sign` will be called on any argument, so if `fac` is called with an argument that cannot be compared using `<`, `=` or `>`, `sign` will signal a type error at runtime. To prevent such errors, the filter specification can be guarded. For example, `fac` can also be defined like this:

```
(define-filtered-function fac (n)
  (:filters (:sign (when (integerp n)
                     (function sign)))))
```

In case the argument passed to `fac` is now not of type `integer`, methods associated with the filter `:sign` will not be considered for method selection and application. (Since there are no other kinds of methods defined for this function, this means that calling a function with a non-integer will still signal an error, now indicating that no applicable method was found. However, this is a qualitatively different error than the type error we get without the guard).

3.1.2 State pattern

Filtered functions can be used to dispatch methods based on the state of an argument passed to a filtered function, which enables expressing State-like patterns without object identity problems. Assume the following simple CLOS class is defined for implementing a stack.

```
(defclass stack ()
  ((contents :initform (make-array 10)
             :reader stack-contents)
   (index :initform 0
          :accessor stack-index)))
```

This class has two slots: the `contents` referencing an array with 10 elements which can be read using the function `stack-contents`, and an `index` into that array which can be read and written using the accessor `stack-index`.

A stack has three different states: It can be empty or full, or anywhere in between (in 'normal' state). We can express this as a function that recognizes the state of a stack.

```
(defun stack-state (stack)
  (cond ((<= (stack-index stack) 0)
         'empty)

        ((>= (stack-index stack)
             (length (stack-contents stack)))
         'full)

        (t 'normal)))
```

According to the function `stack-state`, a stack is in state `empty` if its index is less than or equal to 0, in state `full` if its index is greater than or equal to the length of the `contents` array, and `normal` otherwise.

It is now straightforward to use `stack-state` in a filter named `:state` for the typical stack operations.

```
(define-filtered-function stack-push (stack value)
  (:filters (:state (function stack-state))))

(define-filtered-function stack-pop (stack)
  (:filters (:state (function stack-state))))

(define-filtered-function stack-emptyp (stack)
  (:filters (:state (function stack-state))))
```

The filtered function `stack-push` takes a stack and a value to be pushed to the stack as parameters, and the filtered functions `stack-pop` and `stack-emptyp` both take a stack as their only parameters. We can now group the behavior of a stack according to its different states.

Normal state.

In `stack-push`, the new value is assigned (`setf`) to the `contents` array of the stack, with a reference (`aref`) to the current `index`. That index is then incremented by 1 (`incf`), so that it points to the next entry in the array.

```
(defmethod stack-push (stack value)
  (setf (aref (stack-contents stack)
              (stack-index stack))
        value)
  (incf (stack-index stack)))
```

Accordingly, `stack-pop` first decrements that index (`decf`) before it reads the array reference (`aref`) of the `contents` array of the stack at that `index`.

```
(defmethod stack-pop (stack)
  (decf (stack-index stack))
  (aref (stack-contents stack)
        (stack-index stack)))
```

In normal state, a stack is never considered empty, so `stack-emptyp` can just return `nil`, which stands for the boolean false value in Common Lisp.

```
(defmethod stack-emptyp (stack)
  nil)
```

Empty state.

In empty state, only two operations deviate from the regular stack behavior: The function `stack-pop` has to signal an error to indicate that it cannot fetch further elements from the stack, and `stack-emptyp` actually has to return `t` as the boolean true value.

```
(defmethod stack-pop
  :filter :state ((stack (eql 'empty)))
  (error "Stack is empty."))

(defmethod stack-emptyp
  :filter :state ((stack (eql 'empty)))
  t)
```

Full state.

In full state, only one operation deviates from the regular stack behavior: The function `stack-push` has to signal an error to indicate that it cannot take any elements anymore.

```
(defmethod stack-push
  :filter :state ((stack (eql 'full)) value)
  (error "Stack is full."))
```

Notice how this way of specifying the stack behavior cleanly separates the definition of the several stack states from the behavior for the distinct states. Further note that in the code for the `normal` state, we actually have not mentioned the filter `:state` as a qualifier for the method definitions, because one can always define regular methods for filtered functions as well, and in this particular case, it would not make a difference for the overall semantics of the stack data abstraction whether we had used the `:state` filter for the `normal` state or not.

This version of a State-like idiom avoids any object identity problems: A particular stack always retains its identity, no matter what state it is in. Since the state function `stack-state` is automatically derived from the value a stack's `index` currently has, one does not have to worry about managing an explicit state with explicit state switches in the corresponding `push` and `pop` operations.

However, it is also possible to use an explicit state representation for filtered dispatch. Assume the stack data abstraction is varied as follows.

```
(defclass stack ()
  ((contents :initform (make-array 10)
             :reader stack-contents)
   (index :initform 0
          :accessor stack-index)
   (state :initform 'empty
          :accessor stack-state)))
```

The class `stack` now has an extra slot for representing the `state` of a stack, initialized to `empty`. We can therefore define the stack operations as follows using the reader function for that slot. (They are actually unchanged from the above versions!)

```
(define-filtered-function stack-push (stack value)
  (:filters (:state (function stack-state))))

(define-filtered-function stack-pop (stack)
  (:filters (:state (function stack-state))))

(define-filtered-function stack-emptyp (stack)
  (:filters (:state (function stack-state))))
```

We additionally need to define `:after` methods on `stack-push` and `stack-pop` to manage the state.

```
(defmethod stack-push :after (stack value)
  (if (>= (stack-index stack)
          (length (stack-contents stack)))
    (setf (stack-state stack) 'full)
    (setf (stack-state stack) 'normal)))

(defmethod stack-pop :after (stack)
  (if (<= (stack-index stack) 0)
    (setf (stack-state stack) 'empty)
    (setf (stack-state stack) 'normal)))
```

3.1.3 A simple Lisp interpreter

In the following, we sketch the code for a simple Lisp interpreter, implemented using filtered functions. The interpreter below is starkly simplified: For example, it does not implement lexical scoping or any other advanced features that one would expect from a modern Lisp dialect. The focus is rather on how the code for such an interpreter can be organized using filtered functions.[3]

The heart of a Lisp interpreter is the function `eval` that takes an *s-expression* as a parameter and evaluates it according to the semantics of the Lisp dialect at hand. An s-expression is either a symbol, denoting a variable whose binding can be looked up in an environment; a *cons cell* whose first element denotes an operator and whose remaining elements denote arguments for that operator; or any value other than a symbol or a cons cell (typically numbers, strings and other objects) that are just evaluated to themselves. In the case of a *cons cell*, the interpretation of the s-expression depends on the `first` element of the expression: It can evaluate to a function that should be applied to the arguments in the `rest` of the expression after they are evaluated themselves, or it can be one of the *special operators* which follow special evaluation rules for their arguments, like `quote`, `setq`, `lambda`, `if`, and so on.

We start our Lisp interpreter by defining a global variable `*environment*` building the environment for looking up variable bindings, which by default is an empty association list. (We use only one global environment to keep the presentation of the interpreter simple.) We also define the filtered function `eval` with one filter `:first` that is only used when the argument to `eval` is a cons cell (tested with `consp`) and filters out its `first` element.

```
(defvar *environment* '())

(define-filtered-function eval (form)
  (:filters (:first (when (consp form)
                      (function first)))))
```

By default, any form just evaluates to itself, unless it is a symbol, in which case it represents a variable whose value is looked up in the global environment.

```
(defmethod eval (form) form)

(defmethod eval ((form symbol))
  (lookup form *environment*))
```

If the form is a cons cell, we assume that it denotes a function application by default: All elements of the cons cell are evaluated by mapping `(function eval)` over them with `mapcar`. We then assume that the `first` element is a function and apply it to the other (`rest`) elements as arguments.

```
(defmethod eval ((form cons))
  (let ((values (mapcar (function eval) form)))
    (apply (first values) (rest values))))
```

In all the remaining cases we are dealing with special operators. We define the methods for handling special operators using both the `:first` filter and eql specializers to compare the thus filtered first element of a cons cell with a specific operator symbol, here `quote`, `setq`, `lambda` and `if`.

The operator `quote` returns the second element of the cons cell without evaluating it.

```
(defmethod eval :filter :first ((form (eql 'quote)))
  (second form))
```

[3] An alternative, more complete and correct variant of a Lisp interpreter was actually a major motivating example that led to our discovery of filtered functions [16].

The operator `setq` assumes that the second element denotes a variable binding, and that the third element evaluates to a value that should be assigned to that binding.

```
(defmethod eval :filter :first ((form (eql 'setq)))
  (associate (second form)
             (eval (third form))
             *environment*))
```

The operator `lambda` creates a function and assumes that the second element is a cons cell denoting a list of parameter names for that function, and that all remaining elements are forms to be evaluated in an environment extended with new bindings for the parameter names whenever the created function is applied to some concrete arguments.

```
(defmethod eval :filter :first ((form (eql 'lambda)))
  (lambda (&rest values)
    (let ((*environment* (extend *environment*
                                 (second form)
                                 values)))
      (loop for subform in (rest (rest form))
            for result = (eval subform)
            finally (return result)))))
```

The operator `if` evaluates the second element, based on its truth value either evaluates the third or the fourth element, and returns the value resulting from the latter evaluation.

```
(defmethod eval :filter :first ((form (eql 'if)))
  (if (eval (second form))
    (eval (third form))
    (eval (fourth form))))
```

The advantage of expressing an interpreter using filtered functions lies in the fact that each special operator can be defined separately from all the other ones. In contrast, typical expressions of Lisp interpreters require placing all cases for special operators in one large `cond` or nested `if` form. With filtered functions, it is now easy to extend an interpreter with new special operators without the necessity to touch the rest of the interpreter.

For example, we can add a `let` form to the above interpreter with one filtered method by expressing it in terms of the existing special operator `lambda`. The following definition assumes that the second element in the form to be evaluated contains a list of variable names paired with forms that will evaluate to the initialization values for those variables, and that all remaining elements are forms to be evaluated in an environment extended with new bindings for those properly initialized variables.

```
(defmethod eval :filter :first ((form (eql 'let)))
  (eval `((lambda ,(mapcar (function first)
                           (second form))
            ,@body)
          ,@(mapcar (function second)
                    (rest (rest form))))))
```

3.2 Filtered functions: syntax and semantics

In the following, we describe filtered functions as they are integrated in CLOS. They are thus based on CLOS generic functions, and largely integrate with the syntax and semantics of the latter. In fact, the essential change in semantics, apart from filtering arguments, is in the way different applicable methods are compared to determine their specificity: Methods associated with the same filters are compared using the standard CLOS rules, whereas methods from different filters are compared using the order in which the respective filter specifications appear in the `define-filtered-function` form. This enables programmers to specify how predicates are to be compared in a straightforward and intuitive way.

In the following, we first discuss the syntax and (informal) semantics of filtered functions, followed by the syntax and (informal) semantics of filtered methods.

3.2.1 Filtered functions

A filtered function is introduced with a `define-filtered-function` form. The general syntax looks like this:

`(define-filtered-function` *function-name parameters*
`[(:filters {(`*filter-name filter-expression*`)}*)])`

A filtered function has a *function-name* and takes *parameters*. Any *function-name* and *parameters* which would be valid for CLOS generic functions are also valid for filtered functions. A filtered function can specify zero or more filters, which consist of *filter-names* (typically symbols) and *filter-expressions*. If zero filters are specified, the `:filters` option can also be omitted.[4]

A *filter expression* is a Common Lisp expression which is evaluated in a lexical environment where the parameters of the filtered function are bound to the actual arguments received by the filtered function. A filter expression can have one of the following return values:

1. A single function which is used to filter the first argument as received by the filtered function in order to search for applicable methods.

2. A list of functions which are used to filter all required arguments as received by the filtered function in order to search for applicable methods. If there are fewer filter functions than required arguments, the remaining required arguments remain unfiltered for determining applicable methods. If there are more filter functions than required arguments, the excess filter functions are not used.

3. The boolean false value `nil` to indicate that methods associated with this filter should not be used.

4. The boolean true value `t` to indicate that methods associated with this filter should be used, but none of the arguments need to be filtered in order to search for applicable methods.

Case 2 and 3 are the essential cases: A filter expression must be able to indicate whether its corresponding methods are to be considered in method dispatch or not, and which actual filter functions to use. Case 1 and 4 are special cases that can also be expressed by returning a list of functions in case 2, either with just one element, or with a list containing identity functions only. However, especially case 1 – returning just one function – is a very common case due to the fact that single dispatch is very common, and thus cases 1 and 4 are added for convienence.

[4]Filtered functions also support all other standard generic functions options of CLOS, but they are not discussed here.

3.2.2 Filtered methods

A filtered method is introduced with a regular CLOS `defmethod` form. The general, slightly simplified syntax looks like this:[5]

`(defmethod` *function-name* [*qual*] *parameters form**`)`

qual = {*method-qualifier*}* [`:filter` *filter-name*]

A filtered method has a *function-name* to specify the filtered function which it should be associated with, as well as a (specialized) parameter list and a method body consisting of one or more forms. As in CLOS, a `defmethod` form has an optional list of *qualifiers*. The list of qualifiers can consist of the usual CLOS qualifiers, like `:before`, `:after` and `:around`. For filtered methods, the list of qualifiers optionally ends with the keyword `:filter` followed by the name of a filter which this method should be associated with. If there is no `:filter` keyword, the method is unfiltered. If there is no other qualifier, the method is a (filtered or unfiltered) *primary* method.

3.2.3 Filtered dispatch

When a filtered function is called with particular arguments, the following three steps are performed to determine the methods to be executed, just like in CLOS, but adapted to filtered dispatch.

1. **Selecting the Applicable Methods** For each filter specification, the filter expression is evaluated with the (required) arguments received by the filtered function. If a filter expression returns `nil`, none of the methods associated with this filter are considered applicable. Otherwise, the filter functions returned by the filter expression are applied to the arguments received by the filtered function, and the thus filtered arguments are then used to determine which of the methods associated with this filter are applicable according to the CLOS rules (either an argument is an instance of a class specializer, or the same as the object referenced by an eql specializer). Unfiltered methods (methods without a `:filter` qualifier) are considered as potentially applicable methods as well and are checked against the unfiltered arguments.

2. **Sorting the Applicable Methods** The applicable methods from the previous step are sorted to determine their specificity as follows: First, they are arranged into groups of methods that are associated with the same filters, with an optional additional group of unfiltered methods that are not associated with any filter. Second, within each group, methods are sorted according to the precedence order of the class or eql specializers following the normal CLOS rules. Finally, the resulting lists of sorted methods are appended to each other according to the order of the filter specifications as they appear in the `define-filtered-function` form: If a filter specification appears earlier to the left of the list of filter specifications, it is considered less specific than filter specifications that come afterwards to the right of the list. Unfiltered methods are considered less specific than all other methods in this step.

3. **Combining the Applicable Methods** The standard qualifiers for method combination from CLOS determine the order in which the sorted applicable methods are eventually executed: First, the most specific `:around` method is invoked, if any. An `:around` method may invoke `call-next-method` to execute the respective next specific `:around` method, if any. If there are no (further) `:around` methods, then all `:before` methods are executed, with the most specific `:before` method being executed first, followed by all next most specific `:before` methods. Next, the most specific primary method is invoked. A primary method may invoke `call-next-method` to execute the respective next specific primary method, if any. After return from the primary method(s), all `:after` methods are executed, with the least specific `:after` method being executed first, followed by all next least specific `:after` methods in order. Finally, execution returns to the remaining code to be executed in the `:around` methods, if any, following the respective invocations of `call-next-method`.

The above steps follow essentially the same rules as the corresponding rules for selection and application of methods in CLOS,[6] except that method specificity for filtered methods is determined according to the user-defined order of their corresponding filter specifications. This allows programs to determine an ordering between predicates, alleviating the issue of predicate dispatch that predicates cannot be automatically ordered for specificity. For example, the introductory example of methods defined for prime, odd and even numbers from Fig. 3 can now be defined as follows.

```
(define-filtered-function print-number-property (n)
  (:filters (:prime (when (primep n) t))
            (:odd   (when (oddp n)   t))
            (:even  (when (evenp n)  t))))

(defmethod print-number-property ((n number))
  (print "This is a number."))

(defmethod print-number-property
  :before :filter :prime (n)
  (print "This is a prime number."))

(defmethod print-number-property
  :before :filter :odd (n)
  (print "This is an odd number."))

(defmethod print-number-property
  :before :filter :even (n)
  (print "This is an even number."))
```

Calling `(print-number-property 2)` will now print *"This is an even number."*, *"This is a prime number."* and *"This is a number."* in that, unambiguously defined, order. Likewise, `(print-number-property 5)` will print *"This is an odd number."*, *"This is a prime number."* and *"This is a number."* in exactly that order.

3.3 Implementation details

We have implemented a first version of filtered functions on top of the CLOS metaobject protocol (CLOS MOP, [17]) to modify the dispatch algorithm according to the rules

[5]The full syntax of the CLOS `defmethod` form is supported.

[6]This includes exceptional situations, like invoking `call-next-method` when there are no next methods, which we do not discuss in this paper.

specified above. In addition, we have added a user-defined method combination to handle `:filter` specifications in `defmethod` forms and corresponding `standard` qualifiers `:before`, `:after` and `:around`. Unfortunately, the subprotocols of the CLOS MOP that specify generic function dispatch are the least well supported across different Common Lisp implementations: Currently, the only implementation where we have been able to successfully implement filtered functions by adhering to the CLOS MOP specification is SBCL. We have also been able to support LispWorks, but it was necessary to use implementation-dependent features for this purpose. In principle, arbitrary user-defined method combinations can be supported in SBCL, as long as they correctly recognize and handle the `:filter` qualifier. However, in LispWorks only our own simulation of the standard method combinaton is supported due to limitations in LispWorks's support for the CLOS MOP. We have tried to support other major Common Lisp implementations, but so far have not been successful in doing so. The fact that filtered functions can be implemented in SBCL shows that the CLOS MOP is expressive enough to support complex extensions and should be an encouragement for other implementations to extend their adherence to the CLOS MOP. We are nevertheless currently investigating more portable implementations of filtered functions.

We have not yet payed a lot of attention to efficiency concerns. However, we have taken advantage of some special cases that are supported by the CLOS MOP specification: When no filters are specified, or none of the specified filters are used in actual method definitions for a given filtered function, the default dispatch algorithm of CLOS is used unchanged and should thus have good performance characteristics. If filters are specified for a given filtered function, and exactly one filter is ever used in method definitions for that filtered function,[7] method dispatch should be reasonably efficient, since the only overhead incurred is the filtering step, after which the CLOS method selection and application algorithm of CLOS can again be used mostly unchanged: Only an additional `:around` method is added by default which "unfilters" the previously filtered arguments such that applied methods see the original arguments as originally received by the filtered function. Only in the case where methods are defined for several different filters, or in conjunction with unfiltered methods, our implementation has to resort to a more complex dispatch algorithm customized for filtered dispatch. We have not performed any benchmarks yet, and we expect that there is still a lot of room for improving the performance of our current implementation of filtered functions.

4. RELATED WORK

Classifiers in the language Kea are close to our approach: They allow dynamic classifications of objects to classes in the same designated group of mutually exclusive classes, based on associated predicates [20]. Kea classifiers require more effort than our approach since classes need to be defined to which instances can be mapped, while our approach allows mapping to representatives of equivalence classes, which can be instances of any already existing class and can be dispatched using eql specializers. Kea is also restricted in that it is a purely functional language without side effects, which means that classifications cannot vary over time, like in our stack example in Sect. 3.1.2.

Mode classes [26] enable dispatching on an explicit state of an object that can be modified afer each method defined for its class. Predicate classes [6] extend this idea by dispatching on computed states. Mode classes correspond to an explicit management of state, while predicate classes compute state implicitly. This means that mode classes and predicate classes are similar to the second and first variant of our stack example in Sect. 3.1.2 respectively. Predicate classes were a precursor to generalized predicate dispatch [14].

Specialization-oriented programming [21] extends generic function dispatch with custom specializers, similar to class and eql specializers. More specifically, it provides a new `cons` specializer that can further specify what the contents of a `cons` cell should specialize on. For example, the `eval` method for the special operator `if` in our interpreter in Sect. 3.1.3 could be defined as follows in their approach.

```
(defmethod eval ((form (cons (eql 'if))))
  (if (eval (second form))
    (eval (third form))
    (eval (fourth form))))
```

This method is specialized on `cons` cells whose first element is `eql` to the symbol `if`. That approach also defines a metaobject protocol for defining other kinds of user-defined specializers, which relies on ordering the specificity between different kinds of specializers explicitly. This was an important influence on our notion of filtered dispatch, where the order between filters is also specified explicitly, but at the base level rather than at the metalevel. The CLOS MOP has some restrictions which make it impossible to integrate such specialization oriented programming into CLOS, but [21] describes an extension of the CLOS MOP that sufficiently relaxes these restrictions.

Fickle [13] provides an object migration facility (similar to Smalltalk's `become:` and Gilgul's *referent assignment*) which can be used to implement State-like idioms. However, Fickle's static type system based on effects leads to a situation where the reclassification of one object implies the assumption that all instances of that object's class are potentially reclassified as well.

5. DISCUSSION AND FUTURE WORK

In this paper, we have introduced filtered dispatch, expressed as an extension of generic functions called filtered functions. Filtered functions allow mapping arguments to representatives of equivalence classes that are used in place of the original arguments to determine applicable methods. Methods are then sorted according to their specificity, taking the order into account in which filter specifications are given by the programmer. When methods are executed, they can operate on the original, unfiltered arguments, independent of whether they require filtering of arguments for method dispatch or not. The fact that the order in which filter specifications are given is taken into account when sorting applicable methods for specificity avoids the problems caused by potential ambiguities when comparing arbitrary predicates that do not designate instance subsets of each other.

Currently, filtered functions are implemented using the CLOS MOP and can be used in two different Common Lisp implementations. We have not considered efficiency issues

[7] This means there may not be any unfiltered methods either.

in detail yet, and have not explored more general implementation techniques. However, efficient implementation techniques for generalized predicate dispatch have been investigated in detail in the past [3, 7] and should be useful for implementing filtered functions as well.

We are currently investigating how to combine filtered functions with layered [12] and dynamically scoped functions [10]. We are also experimenting with using filters based on *special slots* [11]. Currently, our own context-oriented programming language extensions support context-dependent behavior on a per-class basis, but cannot restrict context-dependent behavior to single objects in a straightforward way. We hope that a combination of layered functions, dynamically scoped functions and/or special slots with filtered functions increases the flexibility of context-oriented programming in that regard.

In our own uses of filtered functions, we almost always use eql specializers as soon as we define filtered methods. This gives rise to a number of interesting research questions: To what extent can we drop classes from an object system and rely on filtered dispatch alone? How can we still keep the advantages of inheritance hierarchies and facilities like `call-next-method` for performing super calls then? What is the impact on efficiency of method dispatch, which is typically achieved by grouping methods along classes? What is the impact on understandability and maintainability of source code, especially when different filtered functions use different kinds of filters and different orderings for their filters? We plan to investigate these questions in more detail in the near future.

6. REFERENCES

[1] Eric Allen, J.J. Hallett, Victor Luchango, Sukyong Ryu, Guy L. Steele Jr. Modular Multiple Dispatch with Multiple Inheritance. *ACM Symposium on Applied Computing (SAC'07*. Seoul, Korea, March 2007. Proceedings, ACM Press.

[2] ANSI/INCITS X3.226-1994. *American National Standard for Information Systems - Programming Language - Common Lisp*, 1994.

[3] Jonathan Bachrach. Efficient Dispatch for Dylan. Presentation at Massachusetts Institute of Technology AI Lab, October 2000.

[4] Daniel Bardou and Christophe Dony. Split objects: a disciplined use of delegation within objects. *OOPSLA 1996 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Jose, California, USA, October 1996. Proceedings, ACM Press.

[5] Daniel Bobrow, Linda DeMichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales, David Moon. Common Lisp Object System Specification. Lisp and Symbolic Computation 1, 3-4 (January 1989), 245-394.

[6] Craig Chambers. Predicate Classes. *7th European Conference on Object-Oriented Programming (ECOOP'93)*, Kaiserslautern, Germany, July 1993. Proceedings, Springer LNCS.

[7] Craig Chambers and Weimin Chen. Efficient Multiple and Predicate Dispatching. *OOPSLA 1999 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Denver, Colorado, USA, November 1999. Proceedings, ACM Press.

[8] Curtis Clifton, Gary T. Leavens, Craig Chambers, Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Minneapolis, Minnesota, USA, October 2004. Proceedings, ACM Press.

[9] Pascal Costanza. Dynamic replacement of active objects in the Gilgul programming language. *First International IFIP/ACM Working Conference on Component Deployment (CD 2002)*, Berlin, Germany, June 2002. Proceedings, Springer LNCS.

[10] Pascal Costanza. Dynamically Scoped Functions as the Essence of AOP. ECOOP 2003 Workshop on Object-oriented Language Engineering for the Post-Java Era, Darmstadt, Germany, July 22, 2003. ACM Sigplan Notices 38, 8 (August 2003).

[11] Pascal Costanza. How to Make Lisp More Special. International Lisp Conference 2005, Stanford. Proceedings.

[12] Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-oriented Programming. *ACM Dynamic Languages Symposium 2005*. ACM Press.

[13] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, Paola Giannini. Fickle: Dynamic Object Re-classification. *15th European Conference on Object-Oriented Programming (ECOOP 2001)*, Budapest, Hungary, June 2001. Proceedings, LNCS.

[14] Michael Ernst, Craig Kaplan, Chraig Chambers. Predicate Dispatching: A Unified Theory of Dispatch. *12th European Conference on Object-Oriented Programming (ECOOP'98)*, Brussels, Belgium, July 1998. Proceedings, Springer LNCS.

[15] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[16] Charlotte Herzeel, Pascal Costanza, Theo D'Hondt. Reflection for the Masses. *Workshop on Self-sustaining Systems (S3) 2008*, Potsdam, Germany, May 2008. Proceedings, Springer LNCS (to be published).

[17] Gregor Kiczales, Jim Des Rivières, Daniel Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[18] Henry Lieberman. Using Prototypical Objects to Implement Shared Behaviour in Object Oriented Systems. *OOPSLA 1986 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Portlang, Oregon, USA, November 1986. Proceedings, ACM Press.

[19] Todd Millstein. Practical Predicate Dispatch. *OOPSLA 2004 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, British Columbia, Canada, October 2004. Proceedings, ACM Press.

[20] W.B. Mugridge, J.G. Hosking, J. Hamer. Multi-methods in a statically-typed programming language. *5th European Conference on Object-Oriented Programming (ECOOP'91)*, Geneva, Switzerland, July 1991. Proceedings, Springer LNCS.

[21] Jim Newton and Christophe Rhodes. Custom Specializers in Object-Oriented Lisp. *1st European Lisp Symposium (ELS 2008*, Bordeaux, France, May 22-23, 2008. Proceedings.

[22] K. Chandra Sekharaiah and D. Janaki Ram. Object Schizophrenia Problem in Object Role System Design.

Object-Oriented Information Systems, 8th International Conference (OOIS 2002, Montpellier, France, September 2002. Proceedings, Springer LNCS.

[23] Andrew Shalit. *The Dylan Reference Manual*. Addison-Wesley, 1996.

[24] David N. Smith. Dave's Smalltalk FAQ, 1995-96. Originally at `http://www.dnsmith.com/SmallFAQ/SmallFaq.html`, copy found at `http://udos.users.dolphinmap.net/archive/DavesSmalltalkFAQ.pdf`.

[25] David Ungar and Randall B. Smith. Self: The Power of Simplicity. *Lisp and Symbolic Computation*, 4(3), Kluwer Academic Publishers, June 1991.

[26] Antero Taivalsaari. Object-oriented programming with modes. *Journal of Object-Oriented Programming*, June 1993.

[27] Aaron Mark Ucko. *Predicate Dispatching in the Common Lisp Object System*. Master thesis, Massachusetts Institute of Technology, AI Technical Report 2001-006, June 2001.

[28] Ben Wegbreit. The Treatment of Data Types in EL1. *Communications of the ACM*, Vol. 17, No. 5, May 1974, ACM Press.