Generalizers: New Metaobjects for Generalized Dispatch

Christophe Rhodes
Department of Computing
Goldsmiths, University of London
London SE14 6NW
c.rhodes@gold.ac.uk

Jan Moringen
Universität Bielefeld
Technische Fakultät
33594 Bielefeld
jmoringe@techfak.uni-bielefeld.de

David Lichteblau
ZenRobotics Ltd
Vilhonkatu 5 A
FI-00100 Helsinki
david@lichteblau.com

ABSTRACT

This paper introduces a new metaobject, the generalizer, which complements the existing specializer metaobject. With the help of examples, we show that this metaobject allows for the efficient implementation of complex non-class-based dispatch within the framework of existing metaobject protocols. We present our modifications to the generic function invocation protocol from the *Art of the Metaobject Protocol*; in combination with previous work, this produces a fully-functional extension of the existing mechanism for method selection and combination, including support for method combination completely independent from method selection. We discuss our implementation, within the SBCL implementation of Common Lisp, and in that context compare the performance of the new protocol with the standard one, demonstrating that the new protocol can be tolerably efficient.

Report-No.: `http://eprints.gold.ac.uk/id/eprint/9924`

Categories and Subject Descriptors

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

General Terms

Languages, Design

Keywords

generic functions, specialization-oriented programming, method selection, method combination

1. INTRODUCTION

The revisions to the original Common Lisp language [16] included the detailed specification of an object system, known as the Common Lisp Object System (CLOS), which was eventually standardized as part of the ANSI Common Lisp standard [12]. The object system as presented to the standardization committee was formed of three chapters. The first two chapters covered programmer interface concepts and the functions in the programmer interface [17, Chapter 28] and were largely incorporated into the final standard; the third chapter, covering a Metaobject Protocol (MOP) for CLOS, was not.

Nevertheless, the CLOS MOP continued to be developed, and the version documented in [8] has proven to be a reasonably robust design. While many implementations have derived their implementations of CLOS from either the Closette illustrative implementation in [8], or the Portable Common Loops implementation of CLOS from Xerox Parc, there have been largely from-scratch reimplementations of CLOS (in CLISP[1] and CCL[2], at least) incorporating substantial fractions of the Metaobject Protocol as described.

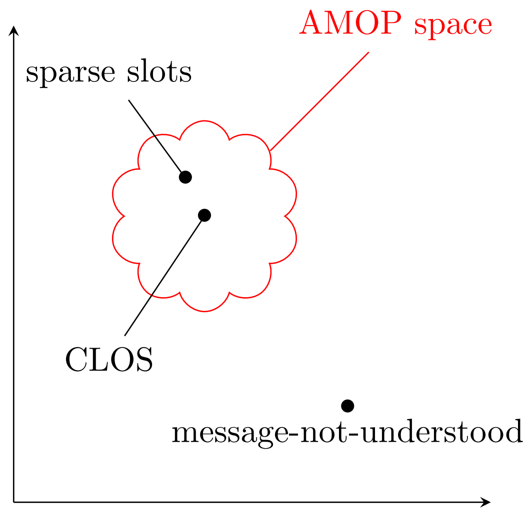

Figure 1: MOP Design Space

Although it has stood the test of time, the CLOS MOP is neither without issues (e.g. semantic problems with `make-method-lambda` [1]; useful functions such as `compute-effective-slot-definition-initargs` being missing from the standard) nor is it a complete framework for the metaprogrammer to implement all conceivable variations of object-oriented behaviour. While metaprogramming offers

[1]GNU CLISP, at `http://www.clisp.org/`
[2]Clozure Common Lisp, at `http://ccl.clozure.com/`

some possibilities for customization of the object system behaviour, those possibilities cannot extend arbitrarily in all directions (conceptually, if a given object system is a point in design space, then a MOP for that object system allows exploration of a region of design space around that point; see figure 1). In the case of the CLOS MOP, there is still an expectation that functionality is implemented with methods on generic functions, acting on objects with slots; it is not possible, for example, to transparently implement support for "message not understood" as in the message-passing paradigm, because the analogue of messages (generic functions) need to be defined before they are used.

Nevertheless, the MOP is flexible, and is used for a number of things, including: documentation generation (where introspection in the MOP is used to extract information from a running system[3]); object-relational mapping[4] and other approaches to object persistence [11]; alternative backing stores for slots (hash-tables [7] or symbols [3]); and programmatic construction of metaobjects, for example for interoperability with other language runtimes' object systems.

One area of functionality where there is scope for customization by the metaprogrammer is in the mechanics and semantics of method applicability and dispatch. While in principle AMOP allows customization of dispatch in various different ways (the metaprogrammer can define methods on protocol functions such as `compute-applicable-methods`, `compute-applicable-methods-using-classes`), for example, in practice implementation support for this was weak until relatively recently[5].

Another potential mechanism for customizing dispatch is implicit in the class structure defined by AMOP: standard specializer objects (instances of `class` and `eql-specializer`) are generalized instances of the `specializer` protocol class, and in principle there are no restrictions on the metaprogrammer constructing additional subclasses. Previous work [10] has explored the potential for customizing generic function dispatch using extended specializers, but there the metaprogrammer must override the entirety of the generic function invocation protocol (from `compute-discriminating-function` on down), leading to toy implementations and duplicated effort.

This paper introduces a protocol for efficient and controlled handling of new subclasses of `specializer`. In particular, it introduces the `generalizer` protocol class, which generalizes the return value of `class-of` in method applicability computation, and allows the metaprogrammer to hook into cacheing schemes to avoid needless recomputation of effective methods for sufficiently similar generic function arguments (See Figure 2).

The remaining sections in this paper can be read in any order. We give some motivating examples in section 2, including reimplementations of examples from previous work, as well as examples which are poorly supported by previous protocols. We describe the protocol itself in section 3, describing each protocol function in detail and, where applicable, relating it to existing protocol functions within the CLOS MOP. We survey related work in more detail in section 4, touching on work on customized dispatch schemes in other environments. Finally, we draw our conclusions from this work, and indicate directions for further development, in section 5; reading that section before the others indicates substantial trust in the authors' work.

[3]as in many of the systems surveyed at `https://sites.google.com/site/sabraonthehill/lisp-document-generation-apps`

[4]e.g. CLSQL, at `http://clsql.b9.com/`

[5]the Closer to MOP project, at `http://common-lisp.net/project/closer/`, attempts to harmonize the different implementations of the metaobject protocol in Common Lisp.

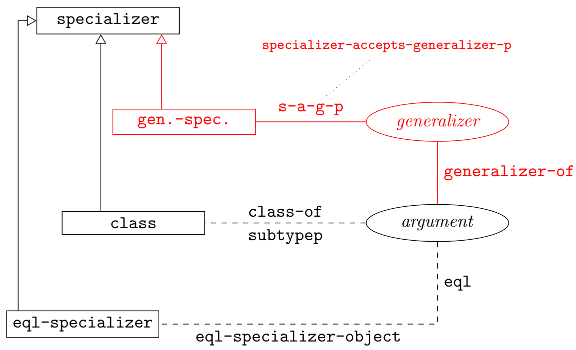

Figure 2: Dispatch Comparison

2. EXAMPLES

In this section, we present a number of examples of dispatch implemented using our protocol, which we describe in section 3. For reasons of space, the metaprogram code examples in this section do not include some of the necessary support code to run; complete implementations of each of these cases, along with the integration of this protocol into the SBCL implementation [13] of Common Lisp, are included in the authors' repository[6].

A note on terminology: we will attempt to distinguish between the user of an individual case of generalized dispatch (the "programmer"), the implementor of a particular case of generalized dispatch (the "metaprogrammer"), and the authors as the designers and implementors of our generalized dispatch protocol (the "metametaprogrammer", or more likely "we").

2.1 CONS specializers

One motivation for the use of generalized dispatch is in an extensible code walker: a new special form can be handled simply by writing an additional method on the walking generic function, seamlessly interoperating with all existing methods. In this use-case, dispatch is performed on the first element of lists. Semantically, we allow the programmer to specialize any argument of methods with a new kind of specializer, `cons-specializer`, which is applicable if and only if the corresponding object is a `cons` whose `car` is `eql` to the symbol associated with the `cons-specializer`; these specializers are more specific than the `cons` class, but less specific than an `eql-specializer` on any given `cons`.

The programmer code using these specializers is unchanged from [10]; the benefits of the protocol described here are: that the separation of concerns is complete – method selection is independent of method combination – and that

[6]the tag `els2014-submission` in `http://christophe.rhodes.io/git/specializable.git` corresponds to the code repository at the point of submitting this paper.

the protocol allows for efficient implementation where possible, even when method selection is customized. In an application such as walking source code, we would expect to encounter special forms (distinguished by particular atoms in the car position) multiple times, and hence to dispatch to the same effective method repeatedly. We discuss the efficiency aspects of the protocol in more detail in section 3.1.2; we present the metaprogrammer code to implement the cons-specializer below.

```
(defclass cons-specializer (specializer)
  ((%car :reader %car :initarg :car)))
(defclass cons-generalizer (generalizer)
  ((%car :reader %car :initarg :car)))
(defmethod generalizer-of-using-class
    ((gf cons-generic-function) arg)
  (typecase arg
    ((cons symbol)
     (make-instance 'cons-generalizer
                    :car (car arg)))
    (t (call-next-method))))
(defmethod generalizer-equal-hash-key
    ((gf cons-generic-function)
     (g cons-generalizer))
  (%car g))
(defmethod specializer-accepts-generalizer-p
    ((gf cons-generic-function)
     (s cons-specializer)
     (g cons-generalizer))
  (if (eql (%car s) (%car g))
      (values t t)
      (values nil t)))
(defmethod specializer-accepts-p
    ((s cons-specializer) o)
  (and (consp o) (eql (car o) (%car s))))
```

The code above shows a minimal use of our protocol. We have elided some support code for parsing and unparsing specializers, and for handling introspective functions such as finding generic functions for a given specializer. We have also elided methods on the protocol functions specializer< and same-specializer-p; for cons-specializer objects, specializer ordering is trivial, as only one cons-specializer (up to equality) can ever be applicable to any given argument. See section 2.3 for a case where specializer ordering is non-trivial.

As in [10], the programmer can use these specializers to implement a modular code walker, where they define one method per special operator. We show two of those methods below, in the context of a walker which checks for unused bindings and uses of unbound variables.

```
(defgeneric walk (form env stack)
  (:generic-function-class cons-generic-function))
(defmethod walk
    ((expr (cons lambda)) env call-stack)
  (let ((lambda-list (cadr expr))
        (body (cddr expr)))
    (with-checked-bindings
        ((bindings-from-ll lambda-list)
         env call-stack)
      (dolist (form body)
        (walk form env (cons form call-stack))))))
(defmethod walk
    ((expr (cons let)) env call-stack)
  (flet ((let-binding (x)
           (walk (cadr x) env
                 (cons (cadr x) call-stack))
           (cons (car x)
                 (make-instance 'binding))))
    (with-checked-bindings
        ((mapcar #'let-binding (cadr expr))
         env call-stack)
      (dolist (form (cddr expr))
        (walk form env (cons form call-stack))))))
```

Note that in this example there is no strict need for cons-specializer and cons-generalizer to be distinct classes. In standard generic function dispatch, the class functions both as the specializer for methods and as the generalizer for generic function arguments; we can think of the dispatch implemented by cons-specializer objects as providing for subclasses of the cons class distinguished by the car of the cons. This analogy also characterizes those use cases where the metaprogrammer could straightforwardly use filtered dispatch [2] to implement their dispatch semantics. We will see in section 2.3 an example of a case where filtered dispatch is incapable of straightforwardly expressing the dispatch, but first we present our implementation of the motivating case from [2].

2.2 SIGNUM specializers

Our second example of the implementation and use of generalized specializers is a reimplementation of one of the examples in [2]: specifically, the factorial function. Here, dispatch will be performed based on the signum of the argument, and again, at most one method with a signum specializer will be applicable to any given argument, which makes the structure of the specializer implementation very similar to the cons specializers in the previous section.

The metaprogrammer has chosen in the example below to compare signum values using =, which means that a method with specializer (signum 1) will be applicable to positive floating-point arguments (see the first method on specializer-accepts-generalizer-p and the method on specializer-accepts-p below). This leads to one subtle difference in behaviour compared to that of the cons specializers: in the case of signum specializers, the *next* method after any signum specializer can be different, depending on the class of the argument. This aspect of the dispatch is handled by the second method on specializer-accepts-generalizer-p below.

```
(defclass signum-specializer (specializer)
  ((%signum :reader %signum :initarg :signum)))
(defclass signum-generalizer (generalizer)
  ((%signum :reader %signum :initarg :signum)))
(defmethod generalizer-of-using-class
    ((gf signum-generic-function) (arg real))
  (make-instance 'signum-generalizer
                 :signum (signum arg)))
(defmethod generalizer-equal-hash-key
    ((gf signum-generic-function)
     (g signum-generalizer))
```

```
  (%signum g))
(defmethod specializer-accepts-generalizer-p
    ((gf signum-generic-function)
     (s signum-specializer)
     (g signum-generalizer))
  (if (= (%signum s) (%signum g))
      (values t t)
      (values nil t)))

(defmethod specializer-accepts-generalizer-p
    ((gf signum-generic-function)
     (s specializer)
     (g signum-generalizer))
  (specializer-accepts-generalizer-p
   gf s (class-of (%signum g))))

(defmethod specializer-accepts-p
    ((s signum-specializer) o)
  (and (realp o) (= (%signum s) (signum o))))
```

Given these definitions, and once again some more straightforward ones elided for reasons of space, the programmer can implement the factorial function as follows:

```
(defgeneric fact (n)
  (:generic-function-class signum-generic-function))
(defmethod fact ((n (signum 0))) 1)
(defmethod fact ((n (signum 1))) (* n (fact (1- n))))
```

The programmer does not need to include a method on `(signum -1)`, as the standard `no-applicable-method` protocol will automatically apply to negative real or non-real arguments.

2.3 Accept HTTP header specializers

In this section, we implement a non-trivial form of dispatch. The application in question is a web server, and specifically to allow the programmer to support RFC 2616 [5] content negotiation, of particular interest to publishers and consumers of REST-style Web APIs.

The basic mechanism in content negotiation is as follows: the web client sends an HTTP request with an `Accept` header, which is a string describing the media types it is willing to receive as a response to the request, along with numerical preferences. The web server compares these stated client preferences with the resources it has available to satisfy this request, and sends the best matching resource in its response.

For example, a graphical web browser might send an `Accept` header of `text/html,application/xml;q=0.9,*/*;q=0.8` for a request of a resource typed in to the URL bar. This should be interpreted as meaning that: if the server can provide content of type `text/html` (i.e. HTML) for that resource, then it should do so. Otherwise, if it can provide `application/xml` content (i.e. XML of any schema), then that should be provided; failing that, any other content type is acceptable.

In the case where there are static files on the filesystem, and the web server must merely select between them, there is not much more to say. However, it is not unusual for a web service to be backed by some other form of data, and responses computed and sent on the fly, and in these circumstances the web server must compute which of its known output formats it can use to satisfy the request before actually generating the best matching response. This can be modelled as one generic function responsible for generating the response, with methods corresponding to content-types – and the generic function must then perform method selection against the request's `Accept` header to compute the appropriate response.

The `accept-specializer` below implements this dispatch. It depends on a lazily-computed `tree` slot to represent the information in the accept header (generated by `parse-accept-string`), and a function `q` to compute the (defaulted) preference level for a given content-type and `tree`; then, method selection and ordering involves finding the `q` for each `accept-specializer`'s content type given the `tree`, and sorting them according to the preference level.

```
(defclass accept-specializer (specializer)
  ((media-type :initarg :media-type :reader media-type)))
(defclass accept-generalizer (generalizer)
  ((header :initarg :header :reader header)
   (tree)
   (next :initarg :next :reader next)))
(defmethod generalizer-equal-hash-key
    ((gf accept-generic-function)
     (g accept-generalizer))
  `(accept-generalizer ,(header g)))
(defmethod specializer-accepts-generalizer-p
    ((gf accept-generic-function)
     (s accept-specializer)
     (g accept-generalizer))
  (values (q (media-type s) (tree g)) t))
(defmethod specializer-accepts-generalizer-p
    ((gf accept-generic-function)
     (s specializer)
     (g accept-generalizer))
  (specializer-accepts-generalizer-p
   gf s (next g)))

(defmethod specializer<
    ((gf accept-generic-function)
     (s1 accept-specializer)
     (s2 accept-specializer)
     (g accept-generalizer))
  (let ((m1 (media-type s1))
        (m2 (media-type s2))
        (tree (tree g)))
    (cond
      ((string= m1 m2) '=)
      (t (let ((q1 (q m1 tree)))
               (q2 (q m2 tree))))
           (cond
             ((= q1 q2) '=)
             ((< q1 q2) '>)
             (t '<))))))
```

The metaprogrammer can then add support for objects representing client requests, such as instances of the `request` class in the Hunchentoot[7] web server, by translating these

[7]Hunchentoot is a web server written in Common Lisp, al-

into accept-generalizer instances. The code below implements this, by defining the computation of a generalizer object for a given request, and specifying how to compute whether the specializer accepts the given request object (q returns a number between 0 and 1 if any pattern in the tree matches the media type, and nil if the media type cannot be matched at all).

```
(defmethod generalizer-of-using-class
    ((gf accept-generic-function)
     (arg tbnl:request))
  (make-instance 'accept-generalizer
                 :header (tbnl:header-in :accept arg)
                 :next (call-next-method)))
(defmethod specializer-accepts-p
    ((s accept-specializer)
     (o tbnl:request))
  (let* ((accept (tbnl:header-in :accept o))
         (tree (parse-accept-string accept))
         (q (q (media-type s) tree)))
    (and q (> q 0))))
```

This dispatch cannot be implemented using filtered dispatch, except by generating anonymous classes with all the right mime-types as direct superclasses in dispatch order; the filter would generate

```
(ensure-class nil :direct-superclasses
 '(text/html image/webp ...))
```

and dispatch would operate using those anonymous classes. While this is possible to do, it is awkward to express content-type negotiation in this way, as it means that the dispatcher must know about the universe of mime-types that clients might declare that they accept, rather than merely the set of mime-types that a particular generic function is capable of serving; handling wildcards in accept strings is particularly awkward in the filtering paradigm.

Note that in this example, the method on specializer< involves a non-trivial ordering of methods based on the q values specified in the accept header (whereas in sections 2.1 and 2.2 only a single extended specializer could be applicable to any given argument).

Also note that the accept specializer protocol is straightforwardly extensible to other suitable objects; for example, one simple debugging aid is to define that an accept-specializer should be applicable to string objects. This can be done in a modular fashion (see the code below, which can be completely disconnected from the code for Hunchentoot request objects), and generalizes to dealing with multiple web server libraries, so that content-negotiation methods are applicable to each web server's request objects.

```
(defmethod generalizer-of-using-class
    ((gf accept-generic-function)
     (s string))
  (make-instance 'accept-generalizer
                 :header s
                 :next (call-next-method)))
```

lowing the user to write handler functions to compute responses to requests; http://weitz.de/hunchentoot/

```
(defmethod specializer-accepts-p
    ((s accept-specializer) (o string))
  (let* ((tree (parse-accept-string o))
         (q (q (media-type s) tree)))
    (and q (> q 0))))
```

The next slot in the accept-generalizer is used to deal with the case of methods specialized on the classes of objects as well as on the acceptable media types; there is a method on specializer-accepts-generalizer-p for specializers that are not of type accept-specializer which calls the generic function again with the next generalizer, so that methods specialized on the classes tbnl:request and string are treated as applicable to corresponding objects, though less specific than methods with accept-specializer specializations.

3. PROTOCOL

In section 2, we have seen a number of code fragments as partial implementations of particular non-standard method dispatch strategies, using generalizer metaobjects to mediate between the methods of the generic function and the actual arguments passed to it. In section 3.1, we go into more detail regarding these generalizer metaobjects, describing the generic function invocation protocol in full, and showing how this protocol allows a similar form of effective method cacheing as the standard one does. In section 3.2, we show the results of some simple performance measurements on our implementation of this protocol in the SBCL implementation [13] of Common Lisp to highlight the improvement that this protocol can bring over a naïve implementation of generalized dispatch, as well as to make the potential for further improvement clear.

3.1 Generalizer metaobjects

3.1.1 Generic function invocation

As in the standard generic function invocation protocol, the generic function's actual functionality is provided by a discriminating function. The functionality described in this protocol is implemented by having a distinct subclass of standard-generic-function, and a method on compute-discriminating-function which produces a custom discriminating function. The basic outline of the discriminating function is the same as the standard one: it must first compute the set of applicable methods given particular arguments; from that, it must compute the effective method by combining the methods appropriately according to the generic function's method combination; finally, it must call the effective method with the arguments.

Computing the set of applicable methods is done using a pair of functions: compute-applicable-methods, the standard metaobject function, and a new function compute-applicable-methods-using-generalizers. We define a custom method on compute-applicable-methods which tests the applicability of a particular specializer against a given argument using specializer-accepts-p, a new protocol function with default implementations on class and eql-specializer to implement the expected behaviour. To order the methods, as required by the protocol, we define a pairwise comparison operator specializer< which defines an ordering between specializers for a given generalizer argument (remembering that even in standard CLOS the ordering between `class` specializers can change depending on the actual class of the argument).

The new `compute-applicable-methods-using-generalizers` is the analogue of the MOP's `compute-applicable-methods-using-classes`. Instead of calling it with the `class-of` each argument, we compute the generalizers of each argument using the new function `generalizer-of-using-class` (where the `-using-class` refers to the class of the generic function rather than the class of the object), and call `compute-applicable-methods-using-generalizers` with the generic function and list of generalizers. As with `compute-applicable-methods-using-classes`, a secondary return value indicates whether the result of the function is definitive for that list of generalizers.

Thus, in generic function invocation, we first compute the generalizers of the arguments; we compute the ordered set of applicable methods, either from the generalizers or (if that is not definitive) from the arguments themselves; then the normal effective method computation and call can occur. Unfortunately, the nature of an effective method function is not specified, so we have to reach into implementation internals a little in order to call it, but otherwise the remainder of the generic function invocation protocol is unchanged from the standard one. In particular, method combination is completely unchanged; programmers can choose arbitrary method combinations, including user-defined long form combinations, for their generic functions involving generalized dispatch.

3.1.2 Effective method memoization

The potential efficiency benefit to having `generalizer` metaobjects lies in the use of `compute-applicable-methods-using-generalizers`. If a particular generalized specializer accepts a variety of objects (such as the `signum` specializer accepting all reals with a given sign, or the `accept` specializer accepting all HTTP requests with a particular `Accept` header), then there is the possibility of cacheing and reusing the results of the applicable and effective method computation. If the computation of the applicable method from `compute-applicable-methods-using-generalizers` is definitive, then the ordered set of applicable methods and the effective method can be cached.

One issue is what to use as the key for that cache. We cannot use the generalizers themselves, as two generalizers that should be considered equal for cache lookup will not compare as `equal` – and indeed even the standard generalizer, the `class`, cannot easily be used as we must be able to invalidate cache entries upon class redefinition. The issue of `class` generalizers we can solve as in [9] by using the `wrapper` of a class, which is distinct for each distinct (re)definition of a class; for arbitrary generalizers, however, there is *a priori* no good way of computing a suitable hash key automatically, so we allow the metaprogrammer to specify one by defining a method on `generalizer-equal-hash-key`, and combining the hash keys for all required arguments in a list to use as a key in an `equal` hash-table.

3.2 Performance

We have argued that the protocol presented here allows for expressive control of method dispatch while preserving the possibility of efficiency. In this section, we quantify the efficiency that the memoization protocol described in section 3.1.2 achieves, by comparing it both to the same protocol with no memoization, as well as with equivalent dispatch implementations in the context of methods with regular specializers (in an implementation similar to that in [9]), and with implementation in straightforward functions. We performed our benchmarks on a quad-core X-series ThinkPad with 8GB of RAM running Debian GNU/Linux, and took the mean of the 10 central samples of 20 runs, with the number of iterations per run chosen so as to take substantially over the clock resolution for the fastest case. Despite these precautions, we advise against reading too much into these numbers, which are best used as an order-of-magnitude estimate.

In the case of the `cons-specializer`, we benchmark the walker acting on a small but non-trivial form. The implementation strategies in the table below refer to: an implementation in a single function with a large `typecase` to dispatch between all the cases; the natural implementation in terms of a standard generic function with multiple methods (the method on `cons` having a slightly reduced `typecase` to dispatch on the first element, and other methods handling `symbol` and other atoms); and three separate cases using `cons-specializer` objects. As well as measuring the effect of memoization against the full invocation protocol, we can also introduce a special case: when only one argument participates in method selection (all the other required arguments only being specialized on `t`), we can avoid the construction of a list of hash keys and simply use the key from the single active generalizer directly.

implementation	time (µs/call)	overhead
function	3.17	
standard-gf/methods	3.6	+14%
cons-gf/one-arg-cache	7.4	+130%
cons-gf	15	+370%
cons-gf/no-cache	90	+2700%

The benchmarking results from this exercise are promising: in particular, the introduction of the effective method cache speeds up the use of generic specializers in this case by a factor of 6, and the one-argument special case by another factor of 2. For this workload, even the one-argument special case only gets to within a factor of 2-3 of the function and standard generic function implementations, but the overall picture is that the memoizability in the protocol does indeed drastically reduce the overhead compared with the full invocation.

For the `signum-specializer` case, we choose to benchmark the computation of 20!, because that is the largest factorial whose answer fits in SBCL's 63-bit fixnums – in an attempt to measure the worst case for generic dispatch, where the work done within the methods is as small as possible without being meaningless, and in particular does not cause heap allocation or garbage collection to obscure the picture.

implementation	time (µs/call)	overhead
function	0.6	
standard-gf/fixnum	1.2	+100%
signum-gf/one-arg-cache	7.5	+1100%
signum-gf	23	+3800%
signum-gf/no-cache	240	+41000%

The relative picture is similar to the cons-specializer case; including a cache saves a factor of 10 in this case, and another factor of 3 for the one-argument cache special case. The cost of the genericity of the protocol here is starker; even the one-argument cache is a factor of 6 slower than the standard generic-function implementation, and a further factor of 2 away from the implementation of factorial as a function. We discuss ways in which we expect to be able to improve performance in section 5.1.

We could allow the metaprogrammer to improve on the one-argument performance by constructing a specialized cache: for signum arguments of rational arguments, the logical cache structure is to index a three-element vector with (1+ signum). The current protocol does not provide a way of eliding the two generic function calls for the generic cache; we discuss possible approaches in section 5.

3.3 Full protocol

The protocol described in this paper is only part of a complete protocol for specializer and generalizer metaobjects. Our development of this protocol is as yet incomplete; the work described here augments that in [10], but is yet relatively untested – and additionally our recent experience of working with that earlier protocol suggests that there might be useful additions to the handling of specializer metaobjects, independent of the generalizer idea presented here.

4. RELATED WORK

The work presented here builds on specializer-oriented programming described in [10]. Approximately contemporaneously, filtered dispatch [2] was introduced to address some of the same use cases: filtered dispatch works by having a custom discriminating function which wraps the usual one, where the wrapping function augments the set of applicable methods with applicable methods from other (hidden) generic functions, one per filter group; this step is not memoized, and using eql methods to capture behaviours of equivalence classes means that it is hard to see how it could be. The methods are then combined using a custom method combination to mimic the standard one; in principle implementors of other method combinations could cater for filtered dispatch, but they would have to explicitly modify their method combinations. The Clojure programming language supports multimethods[8] with a variant of filtered dispatch as well as hierarchical and identity-based method selectors.

In context-oriented programming [6, 18], context dispatch occurs by maintaining the context state as an anonymous class with the superclasses representing all the currently active layers; this is then passed as a hidden argument to context-aware functions. The set of layers is known and under programmer control, as layers must be defined beforehand.

In some sense, all dispatch schemes are specializations of predicate dispatch [4]. The main problem with predicate dispatch is its expressiveness: with arbitrary predicates able to control dispatch, it is essentially impossible to perform any substantial precomputation, or even to automatically determine an ordering of methods given a set of arguments.

[8] http://clojure.org/multimethods

Even Clojure's restricted dispatch scheme provides an explicit operator for stating a preference order among methods, where here we provide an operator to order specializers; in filtered dispatch the programmer implicitly gives the system an order of precedence, through the lexical ordering of filter specification in a filtered function definition.

The Slate programming environment combines prototype-oriented programming with multiple dispatch [15]; in that context, the analogue of an argument's class (in Common Lisp) as a representation of the equivalence class of objects with the same behaviour is the tuple of roles and delegations: objects with the same roles and delegations tuple behave the same, much as objects with the same generalizer have the same behaviour in the protocol described in this paper.

The idea of generalization is of course not new, and arises in other contexts. Perhaps of particular interest is generalization in the context of partial evaluation; for example, [14] considers generalization in online partial evaluation, where sets of possible values are represented by a type system construct representing an upper bound. Exploring the relationship between generalizer metaobjects and approximation in type systems might yield strategies for automatically computing suitable generalizers and cache functions for a variety of forms of generalized dispatch.

5. CONCLUSIONS

In this paper, we have presented a new generalizer metaobject protocol allowing the metaprogrammer to implement in a straightforward manner metaobjects to implement custom method selection, rather than the standard method selection as standardized in Common Lisp. This protocol seamlessly interoperates with the rest of CLOS and Common Lisp in general; the programmer (the user of the custom specializer metaobjects) may without constraints use arbitrary method combination, intercede in effective method combination, or write custom method function implementations. The protocol is expressive, in that it handles forms of dispatch not possible in more restricted dispatch systems, while not suffering from the indeterminism present in predicate dispatch through the use of explicit ordering predicates.

The protocol is also reasonably efficient; the metaprogrammer can indicate that a particular effective method computation can be memoized, and under those circumstances much of the overhead is amortized (though there remains a substantial overhead compared with standard generic-function or regular function calls). We discuss how the efficiency could be improved below.

5.1 Future work

Although the protocol described in this paper allows for a more efficient implementation, as described in section 3.1.2, than computing the applicable and effective methods at each generic function call, the efficiency is still some way away from a baseline of the standard generic-function, let alone a standard function. Most of the invocation protocol is memoized, but there are still two full standard generic-function calls – generalizer-of-using-class and generalizer-equal-hash-key – per argument per call to a generic function with extended specializers, not to mention a hash table lookup.

For many applications, the additional flexibility afforded by generalized specializers might be worth the cost in efficiency, but it would still be worth investigating how much the overhead from generalized specializers can be reduced; one possible avenue for investigation is giving greater control over the cacheing strategy to the metaprogrammer.

As an example, consider the `signum-specializer`. The natural cache structure for a single argument generic function specializing on `signum` is probably a four-element vector, where the first three elements hold the effective methods for `signum` values of -1, 0, and 1, and the fourth holds the cached effective methods for everything else. This would make the invocation of such functions very fast for the (presumed) common case where the argument is in fact a real number. We hope to develop and show the effectiveness of an appropriate protocol to allow the metaprogrammer to construct and exploit such cacheing strategies, and (more speculatively) to implement the lookup of an effective method function in other ways.

We also aim to demonstrate support within this protocol for some particular cases of generalized specializers which seem to have widespread demand (in as much as any language extension can be said to be in "demand"). In particular, we have preliminary work towards supporting efficient dispatch over pattern specializers such as implemented in the Optima library[9], and over a prototype object system similar to that in Slate [15]. Our current source code for the work described in this paper can be seen in the git source code repository at `http://christophe.rhodes.io/git/specializable.git`, which will be updated with future developments.

Finally, after further experimentation (and, ideally, non-trivial use in production) if this protocol stands up to use as we hope, we aim to produce a standards-quality document so that other implementors of Common Lisp can, if they choose, independently reimplement the protocol, and so that users can use the protocol with confidence that the semantics will not change in a backwards-incompatible fashion.

5.2 Acknowledgments

We thank the anonymous reviewers for their helpful suggestions and comments on the submitted version of this paper. We also thank Lee Salzman, Pascal Costanza and Mikel Evins for helpful and informative discussions, and all the respondents to the first author's request for imaginative uses for generalized specializers.

6. REFERENCES

[1] Pascal Costanza and Charlotte Herzeel. make-method-lambda considered harmful. In *European Lisp Workshop*, 2008.

[2] Pascal Costanza, Charlotte Herzeel, Jorge Vallejos, and Theo D'Hondt. Filtered Dispatch. In *Dynamic Languages Symposium*. ACM, 2008.

[3] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of ContextL. In Roel Wuyts, editor, *Symposium on Dynamic Languages*, pages 1–10, 2005.

[4] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In Eric Jul, editor, *ECOOP 1998 – Object-Oriented Programming*, number 1445 in LNCS, pages 186–211. Springer, Berlin, 1998.

[5] R. Fielding, J. Gettys, J. Movil, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, June 1999.

[6] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[7] Gregor Kiczales, J. Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G. Bobrow. Metaobject protocols: Why we want them and what else they can do . In Andreas Paepke, editor, *Object-Oriented Programming: The CLOS Perspective*, pages 101–118. MIT Press, Cambridge, MA, 1993.

[8] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Mass., 1991.

[9] Gregor Kiczales and Luis Rodriguez. Efficient Method Dispatch in PCL. In *LISP and Functional Programming*, pages 99–105, Nice, 1990.

[10] J. Newton and C. Rhodes. Custom specializers in Object-oriented Lisp. *Journal of Universal Computer Science*, 14(20):3370–3388, 2008. Presented at *European Lisp Symposium*, Bordeaux, 2008.

[11] Andreas Paepke. PCLOS: A Flexible Implementation of CLOS Persistence. In Stein Gjessing and Kristen Nygaard, editors, *ECOOP '88 Conference on Object-Oriented Programming*, number 322 in LNCS, pages 374–389. Springer, Berlin, 1988.

[12] Kent Pitman and Kathy Chapman, editors. *Information Technology – Programming Language – Common Lisp*. Number 226–1994 in INCITS. ANSI, 1994.

[13] C. Rhodes. SBCL: A Sanely-Bootstrappable Common Lisp. In Robert Hirschfeld and Kim Rose, editors, *Self-Sustaining Systems*, number 5146 in LNCS, pages 74–86. Springer-Verlag, Berlin, 2008. Presented at *Workshop on Self-Sustaining Systems*, Potsdam, 2008.

[14] Erik Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford, California, USA, 1993.

[15] Lee Salzman and Jonathan Aldrich. Prototypes with Multiple Dispatch: An Expressive and Dynamic Object Model. In Andrew P. Black, editor, *ECOOP 2005 – Object-Oriented Programming*, number 3586 in LNCS, pages 312–336. Springer, Berlin, 2005.

[16] Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press, Newton, Mass., 1984.

[17] Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press, Newton, Mass., second edition, 1990.

[18] Jorge Vallejos, Sebastián González, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt, and Kim Mens. Predicated Generic Functions: Enabling Context-Dependent Method Dispatch. In *International Conference on Software Composition*, pages 66–81, 2010.

[9] `https://github.com/m2ym/optima`