Journal of Universal Computer Science, vol. 14, no. 20 (2008), 3370-3388
submitted: 23/6/08, accepted: 24/8/08, appeared: 28/11/08 © J.UCS

Custom Specializers in Object-Oriented Lisp

Jim Newton
(Cadence Design Systems
Mozartstrasse 2
D-85622 Feldkirchen, Germany
`jimka@cadence.com`)

Christophe Rhodes
(Department of Computing
Goldsmiths, University of London
New Cross, London, SE14 6NW, United Kingdom
`c.rhodes@gold.ac.uk`)

Abstract: We describe in this paper the implementation and use of custom specializers in two current dialects of Lisp: SKILL and Common Lisp. We motivate the need for such specializers by appealing to clarity of expression, referring to experience in existing industrial applications. We discuss the implementation details of such user-defined specializers in both dialects of Lisp, detailing open problems with those implementations, and we sketch ideas for solving them.
Key Words: Specializer – Generic Function – Metaobject Protocol – Lisp
Category: D.1, D.3.3

1 Introduction

Lisp has a venerable history of object-oriented programming; at one point in time, early in the history of object-orientation, Flavors [Moo86] and New Flavors, Common Objects, Object Lisp and Common Loops [BKK$^+$86] all coexisted. The Common Lisp Object System (CLOS) was incorporated into the language in June 1988 [Ste90, Chapter 26], and when the ANSI Common Lisp standard [PC94] was formalized in 1995, Common Lisp became the first ANSI-standardized programming language with support for object-oriented programming.

In the object systems in the Lisps under discussion in this paper, method specializers have the function of determining whether a particular method is applicable to a set of function arguments or not; method qualifiers determine the function of the method within the effective method (from method combination) if the method is applicable at all.

The repertoire of specializers in a given language dialect is typically limited in some way: in the SKILL® [Bar90, Pet93] dialect of Lisp (which we will introduce in Section 2.1), only classes are allowed as specializers by default, matching instances of that class; in Common Lisp, classes and `eql` specializers (matching a single object by identity) are allowed by default, though the CLOS

Metaobject Protocol (MOP) allows for extensibility in principle, as it specifies a `mop:specializer` metaobject class.

1.1 Custom Specializers

It is sometimes the case that applications require dispatch on objects whose behaviour is not separated by class structure; the dispatch may be influenced by the global application state, or by the values of slots in the objects, or other such factors. In object systems where the specializer metaobject class is not extensible, there is then an impedance mismatch between the expression of the functionality and its implementation, and it is this impedance mismatch that we address by allowing the user to define subclasses of the specializer class.

By giving the user this option, we aim to provide a means to improve locality and clarity of the implementation of a particular solution to a problem, by allowing direct expression rather than manual reimplementation of dispatch machinery to distinguish between things that happen to be instances of the same Lisp class (or where the class of the object is not relevant for dispatch). The provision of this option does not lead to any loss of efficiency for the user of standard generic functions and specializers, and many of the implementation techniques for an efficient implementation of Lisp object systems [KR93] can be applied to our custom specializers.

This paper discusses the use and implementation of metaobject protocols to allow the user to take advantage of the ability to define subclasses of the specializer class; after introducing some background and discussing related work in the next section, we present a worked example in Section 3 to attempt to motivate the definition and use of such specializer metaobject classes. We discuss implementation issues regarding both SKILL and Common Lisp in Section 4, and conclude in Section 5.

2 Background

2.1 The SKILL Programming Language

The users of Cadence Design Systems' custom Integrated Circuit (IC) tools use the SKILL programming language extensively. Programmers write applications that customize the look and feel of the graphical system, automate the design process by reducing the amount of repetitive work the design engineer must do, and perform time-consuming, tedious verification checks. Other types of programs include automatic layout generation tools that quickly produce parameterizable layouts that are correct by design. The language has an optional C-style syntax with many engineer-friendly shortcuts, making it easy for non-programmers to write simple scripts to help in their daily work.

The same language is also a Lisp system having the basic features one would expect: a Read-Eval-Print Loop (REPL), a debugger, garbage collection, lexical and dynamic scoping, macros, and anonymous functions. As with most Lisp systems, the language can be extended through adding functions to the run-time environment.

The SKILL language has a built-in object system called the SKILL++ Object System or simply SKILL++. SKILL++ is based on CLOS, but provides only a subset of the capabilities; missing are features such as: multiple dispatch, multiple inheritance, method combination, method qualifiers, equivalence specializers, and a Metaobject Protocol. Instead, it provides single dispatch, single inheritance, analogues to Common Lisp's `call-next-method` and `next-method-p`, class and method redefinition, explicit environment objects, and a per-method choice between lexical and dynamic scoping. Also important to note is that while the language is interpreted by a proprietary virtual machine, the method dispatch mechanism in particular is implemented in a high performance compiled language; consequently, generic function calls are as fast as normal function calls.

It should be stressed that, although SKILL is a special-purpose language environment and exists primarily within proprietary applications, it has a wide user base, as a substantial fraction of the world's IC design software is provided by Cadence Design Systems; many of the chips in today's consumer devices have been simulated or designed within a SKILL-based system. Thus, there is considerable potential benefit in learning from language design experience, both to improve SKILL itself and to make language innovations developed for SKILL environments available to Common Lisp users.

2.2 Common Lisp

CLOS was developed in conjunction with the design of a Metaobject Protocol (MOP), described in *The Art of the Metaobject Protocol* (AMOP) [KdRB91]. Common Lisp as standardized only includes a very small portion of this Metaobject Protocol (for instance, a recommendation to use `mop:slot-value-using-class` in `slot-value`; some introspective functionality such as `find-method`; and arguably a little ability for intercession in `compute-applicable-methods`, though in fact the standard does not require that `compute-applicable-methods` be called as part of generic function dispatch), and so to customize the behaviour of the object system in Common Lisp it is necessary to go beyond the standard language.

Many Common Lisp implementations support some of the MOP, to varying extents; a survey from a few years ago [BdL00] revealed many aspects of MOP support as being incomplete, even at the coarse level of specified classes and generic functions being unimplemented. More recently, the Closer[1] project has

[1] `http://common-lisp.net/project/closer/`

provided both a set of test cases for implementations of the Metaobject Protocol – which has encouraged some implementations to enhance their support for it[2] – and a compatibility layer to provide an environment as close as possible to that described in AMOP in major implementations of Common Lisp.

2.3 Related Work

The issue of dispatch customization in Common Lisp has arisen before; for example, predicate dispatching in Common Lisp has been discussed in [Uck01]. In that work, the predicate was not restricted at all, and the solution presented involved extending method qualifiers (arbitrary predicates not being associated with any particular argument, and methods being distinguished from each other only on the basis of qualifiers and specializers). Portability difficulties with this approach were noted at the time, and would likely still be present today; for example, some implementations will only accept non-standard qualifiers if the generic function has a non-standard method combination. Strictly, `define-method-combination` will signal errors if methods are placed in the same method group having the same specializers (even if the intent is to use qualifiers to influence method applicability): qualifiers in Common Lisp are meant to affect method combination rather than method selection.

Predicate dispatch in other languages has also been investigated; a system has been presented and implemented for Java [Mil04], wherein the predicates affecting dispatch are restricted to a set that can be reasoned over, and for which ambiguities are forbidden in the selection of the most specific method. We prefer to leave such policy decisions to the users of the system, at least while the capabilities and expressiveness are being explored: if it turns out that restricting specializers to express a limited set of predicates is acceptable, that can be enforced at a later stage.

At this time, we make no attempt to implement a specific predicate dispatch mechanism in either Skill or Common Lisp, but rather aim to provide a framework that is both sufficiently general to express predicate dispatch and straightforward to use, allowing issues of determinism, portability and performance to be explored and addressed by users.

3 Using Custom Specializers: a Worked Example

The following excerpts are from a code walker expressed using custom specializers. The code walker examines code written in a particular Lisp dialect and reports unbound and unused variables. For purposes of simplicity of presentation,

[2] At the time of writing, the MOP implementation of Steel Bank Common Lisp [N^{+}00] fails none of tests in the Closer MOP suite.

```
(defgeneric walk (expr env call-stack)
  (:generic-function-class sop-cons-generic-function))
```

Figure 1: Code walker generic function definition.

the illustrated implementation uses a Common Lisp-like syntax, with SKILL-like semantics in one or two respects noted below; the differences between the presented syntax and SKILL code are minor and typographical in nature (such as the use of `@` instead of `:` to denote keywords).

The goal of this illustration is to give an example of a solution that is more parsimonious when the language supports describing actions on wider ranges of data, rather than to convince that a particular type of specializer (such as the `cons` specializer used here) itself is a good idea. As with any pedagogical example, the same application could be written in many different ways without great loss of clarity.

The form in Figure 1 defines the generic function `walk` as an instance of the generic function metaclass named `sop-cons-generic-function`, which is assumed to already exist. This generic function metaclass is named for 'specializer-oriented programming', admitting `cons` specializers as well as the regular `class` and `eql` specializers. We discuss the implementation issues of this metaclass in Section 4.1.

The implementation of `walk` we present here contains four conceptual parts:

– a recursion engine that includes a termination condition and error handling;

– code to recognize variable references and mark bindings as *used*;

– code to ignore all irrelevant forms encountered during the recursion;

– code to handle special forms.

We begin by implementing the first three parts using standard CLOS functionality; the part to handle special forms is then implemented using a non-standard subclass of `mop:specializer`.

3.1 Code Walker Framework

The main engine of the code walker (Figure 2) starts at a top level expression. If the expression is a list, it calls itself recursively on the elements of the list – with a few notable exceptions. Some of the necessary exceptions can be handled by equivalence specializers such as `(eql t)` and `(eql nil)`. Lisp special forms, such as `(quote ...)` and `(lambda ...)` forms, cannot be described by equivalence specializers but can be with `cons` specializers.

```
(defmethod walk ((expr list) env call-stack)
  (let ((call-stack (cons expr call-stack)))
    (walk (car expr) env call-stack)
    (walk (cdr expr) env call-stack)))

(defmethod walk ((expr (eql nil)) env call-stack)
  nil)

(defmethod walk ((expr t) env call-stack)
  (format t "invalid expression ~A: ~A: ~A~%"
          (class-name (class-of expr)) expr call-stack))
```

Figure 2: Recursion engine and termination condition

Next is the traversal engine based on the class specializer `list` and the termination condition based on an equivalence specializer `(eql nil)`. Thus the engine keeps traversing the lists until they are exhausted. There is also a method specializing on class `t` that will be called if something is encountered which the code walker cannot otherwise handle. The job of the methods that follow will be to assure that everything that occurs in the traversal is handled by an appropriate method and that the `"invalid expression"` message never gets printed.

```
(defmethod walk ((var symbol) env call-stack)
  (if-let (binding (find-binding env var))
          (setf (used binding) t)
          (format t "unbound: ~A: ~A~%" var call-stack)))


(defmethod walk ((expr string) env call-stack)
  nil)
(defmethod walk ((expr number) env call-stack)
  nil)
(defmethod walk ((expr (eql t)) env call-stack)
  nil)
```

Figure 3: Checking the bindings of symbols, and ignoring other atoms.

When a symbol is encountered the first method in Figure 3 is applicable. A check is made to see whether the variable is bound in the environment[3]. If so, the `used` slot of the binding object it set to true, to note that the binding is used. If the variable is unbound, then a diagnostic message is emitted, informing

[3] The implementation of the `find-binding` function is omitted. It returns a *binding* object by searching for a named variable in a given *environment* object. Such a binding object has an accessor named `used` to hold a boolean, indicating whether the binding is used or not.

the user of where the reference to an unbound variable is made.

The three lower methods in Figure 3 show how certain types of self-evaluating atoms such as strings, numbers, and the symbol `t` are simply ignored when searching for variable references. A full implementation of this would ignore all atoms that cannot name variables; in this restricted Common Lisp-like language, we assume that those objects are instances of either `string` or `number`.

3.2 Special Forms

We now implement some of the special forms. Note that `quote` and `lambda` themselves are not special forms; they are simply symbols that evaluate as any other symbol – if one of these symbols is encountered in a context where it is used as a variable, the code walker must treat it as such. This means we cannot write a method for `walk` specializing on `(eql quote)`[4]. However, lists for evaluation whose first elements are `quote` or `lambda` are special and must be intercepted before the walker reaches the `quote` and `lambda` symbols themselves.

The `cons` specializer provides a mechanism for making a method applicable for such a list. Figure 4 implements methods for handling `quote` and `lambda` forms. The first method is applicable if its first argument is a list whose first element is the symbol `quote`. Since an evaluator would simply return the second element of this special form unevaluated, there can be no variable references inside it; so the code walker simply returns `nil`.

The second method handles `lambda` forms by creating new bindings as indicated by the lambda list and walking the body of the `lambda` with those bindings in place. After the code walker returns from walking the lambda body we can report if any of the new bindings were not referenced by the walked code.[5]

This implementation of `walk` is a simplified version of a walker for SKILL that is used in production; we have elided many details of the full version. For example, rather than printing diagnostics, the walker communicates with the environment, allowing the offending forms to be highlighted in the editor; additionally, the walker supports a much broader range of the SKILL language semantics, including ignorable and global variables, assignment, macro expansion and more special forms. The user-defined `cons` specializer presented here allows us to have a single generic function, `walk`, whose methods specialize on all of the different types of forms that must be handled differently.

[4] Note that unlike in Common Lisp, here the argument of the `eql` specializer is unevaluated; `(eql quote)` is correct, rather than `(eql 'quote)`. We discuss this further in Section 4.2.

[5] The implementations of the functions `derive-bindings-from-ll` and `make-env` are omitted for this illustration as they do not aid in understanding extensible specializers. The `derive-bindings-from-ll` function returns a list of *binding* objects from a lambda list. The `make-env` function allocates a new *environment* referencing the given list of binding objects, and also referencing the given parent environment.

```
(defmethod walk ((form (cons (eql quote))) env call-stack)
  nil)

(defmethod walk ((form (cons (eql lambda))) env call-stack)
  (destructuring-bind (lambda lambda-list &rest body) form
    (let ((bindings (derive-bindings-from-ll lambda-list)))
      (dolist (form body)
        (walk form (make-env bindings env) (cons form call-stack)))
      (dolist (bind bindings)
        (unless (used bind)
          (format t "unused: ~A: ~A~%" var call-stack))))))
```

Figure 4: Handling the `(quote ...)` and `(lambda ...)` special forms.

As an example of perhaps a potentially generally useful specializer type, consider a specializer corresponding to a pattern, similar to those found in Prolog [ISO95] unification or pattern-matching in the ML family of languages [MTHM97]. Using the mechanisms presented in this paper, it is possible to have the dispatch over patterns optimized as is expected in those languages, while still retaining the customary run-time extensibility of Lisp, by lazily compiling the dispatch (using algorithms such as those in [LFM01]) and invalidating the compiled code if methods are added or removed to the pattern-matching generic function.

It should be noted that there is an inefficiency in implementing this pattern-matching dispatch using generic function dispatch; where in ML the binding of pattern variables and dispatch is interleaved, the protocols for method dispatch and invocation enforce a separation, which means that the method function itself must destructure its arguments separately from the dispatch, checking for applicability of the method. This inefficiency is fundamental to the generic function invocation protocol, and not a result of our implementations of generalized specializers discussed below.

An application that, we believe, would benefit from a protocol for defining specializers for which there is no corresponding hierarchy is an Emacs-like text editor, where 'minor modes' can affect the functionality of keystrokes and editor function calls. For instance, in the Climacs text editor [RSM05], minor modes are currently implemented by the creation of anonymous classes with a combination of superclasses corresponding to the currently-active modes, whereas it should be simpler to express this as a dispatch on aspects of the current editor state.

4 Implementation Details

4.1 Skill, SKILL++ and VCLOS

To address the limitations of SKILL++ (see Section 2.1) a new object system for SKILL was needed, to provide more of the features of CLOS. The new object system was required to be able to interface to programs written in the existing SKILL++ system, and allow object-oriented techniques to be used on existing systems whose object models are not changeable, while also being extensible for the types of problems faced in application programming for IC development.

Neither VCAD (an organizational department within Cadence Design Systems) nor VCAD's customers have write access to the SKILL implementation, and so the language itself cannot be changed: the object-oriented extension must be provided as a loadable SKILL application. From its Lisp heritage, SKILL can be altered in this way so that the extension seems native to the SKILL programmer and invisible to the end-user.

4.1.1 VCLOS and its Metaobject Protocol

The resulting system, VCAD CL-like Object System (VCLOS), was developed over several years; the major difference from CLOS and its Metaobject Protocol [KdRB91] is that more importance is given to the `mop:specializer` metaobject class, rather than having most of the dispatch functionality of generic functions be computed from the `class` of arguments.

The VCLOS Metaobject Protocol implemented is then similar to the CLOS MOP, with the following points to note:

– the `ClosClassSpecializer` and `ClosEqvSpecializer` classes are both subclasses of `ClosSpecializer`, while users are encouraged to define their own subclasses of `ClosSpecializer` by the provision of a protocol for using them in computation of the effective method (described further below);

– in VCLOS, `ClosComputeApplicableMethodsUsingSpecializers` takes the place of `mop:compute-applicable-methods-using-classes` in the standard AMOP generic function invokation protocol;

– a good CLOS implementation will memoize the results of `mop:compute-applicable-methods-using-classes` if possible, with a key based on the classes of the arguments (see [KR93] for some details). VCLOS supports memoization based on specializer names, computed using `ClosComputeSpecializerNames`.

In order to use a user-defined specializer class, the user must define a subclass of `ClosSpecGenericFunction`, the generic function subclass following the protocols for extensible specializers. The protocol defined on `ClosSpecGenericFunction` allows for the user to specify how to put specializers in precedence order through defining methods on Metaobject Protocol functions: `ClosAvailableSpecializers` and `ClosCmpLikeSpecializers`. The method on `ClosAvailableSpecializers` applicable to a particular generic function class must return a list of specializer class names, from most specific to least specific; methods on `ClosCmpLikeSpecializers` must decide which of two specializers of the same class (assumed both applicable to the same generic function argument) is more specific.

Among the other Metaobject Protocol functions needing methods defined for user-defined specializers to work are `ClosArgMatchesSpecializerP`, a function of a specializer and an arbitrary object, which returns true if a specializer corresponds to a type of which the given object is a member, and `ClosGetClassPrecedenceList` (which should perhaps have been called `ClosGetSpecializerPrecedenceList`), that for a given specializer computes a linearization of its less-specific specializers.

The treatment mapping specializer surface syntax such as `(cons quote)` to specializer metaobjects is performed by generic functions `ClosMatchesSpecializerSyntaxP` and `ClosSetSpecializerData`. Note that in this respect the SKILL protocol and the extension to the Common Lisp Metaobject Protocol described in appendix A differ in the approach taken, as in Common Lisp the specializer syntax is sensitive to the lexical environment.

The SKILL implementation of VCLOS provides memoization, keyed on the specializers of the arguments, to the effective method, allowing the elision of calls to `ClosComputeApplicableMethodsUsingSpecializers`, in a similar way to the CLOS MOP protocol around `mop:compute-applicable-methods-using-classes`. There is still overhead involved in computing appropriate specializers corresponding to the arguments, relative to the baseline of computing an argument's class, but this memoization can significantly reduce the overhead of using a non-standard specializer.

4.2 Common Lisp and the Metaobject Protocol

Much of the work in implementing custom specializers in SKILL was of course taken up by providing a suitably rich object system such that customizations can meaningfully be made: essentially, taking a single-dispatch, single-inheritance object system as found in SKILL++ and implementing on top of it a multiple-dispatch, multiple-inheritance system with a Metaobject Protocol. By contrast, in Common Lisp (with the *de facto* standard MOP) we already have most of the framework for the implementation of custom specializers; for basic operation, we only require a few non-standard operators.

The Lisp-like language we have used for our examples, and the actual implementation of the specializer metaobject class in SKILL, share one important difference in detail from Common Lisp. In Common Lisp's `defmethod` macro, the `eql` specializer specifies not a specialization on a following literal, but instead a specialization on the value of a form in the lexical environment of the method definition.

This detail implies that there must be an operator, similar to `mop:make-method-lambda`, which is capable of converting surface syntax such as `(eql foo)` into code which constructs an `mop:eql-specializer` metaobject at the time when the `defmethod` is executed. Of course, we could restrict the use of the lexical environment to the standardized `eql` specializer, but since it is possible to support culturally-compatible use of the lexical environment through a relatively straightforward backward-compatible extension to the CLOS Metaobject Protocol (see appendix A), we choose to do so, defining our new operator as `make-method-specializers-form`. For convenience, we also suggest `parse-specializer-using-class` and `unparse-specializer-using-class` to consume and produce user-friendly representations of specializers, for use in `find-method` and printed representations of methods.

4.2.1 VCLOS implementation in SBCL

We have in addition implemented a version of the SKILL and VCLOS Metaobject Protocol described in Section 4.1.1 above, and used it to run the `walk` example from Section 3. The implementation of the VCLOS protocol in SBCL's MOP is by no means complete and certainly not industrial-strength; however, even the simple implementation, presented in appendix B, raises some issues.

Firstly, initial explorations revealed that current Common Lisp implementations have only partial support for subclassing `mop:specializer`; most implementations will allow defining the subclass, but very few recognize such a subclass as a valid specializer. In the implementation for SBCL, we had to alter a number of places in the CLOS implementation where the assumption had been made that a specializer was either a `class` or an `eql-specializer`. However, there were no bootstrapping issues introduced by this relaxation, because none of the implementation of CLOS itself requires the use of any new subclasses of the extensible specializer metaclass.

Secondly, since the system needs to call, as part of the discriminating function, a new function `compute-applicable-methods-using-specializers` instead of the usual `compute-applicable-methods-using-classes` (and we need to be calling `specializer-of` rather than `class-of` on the generic function arguments), we must override `mop:compute-discriminating-function` for our generic function class. This in turn means that we need to interpret or compile the result of `mop:compute-effective-method` ourselves, which is not a straightforward procedure, as suitable definitions for `call-method` and `make-method` need to be provided; `mop:compute-effective-method` returns a form, not something that is directly executable.

Additionally, we need to provide an implementation of `compute-applicable-methods`, as well as the new protocol function `compute-applicable-methods-using-specializers`, because the new methods must call our protocol function `specializer-applicable-p` (for determining whether an argument matches a specializer). An implementation is not difficult in principle, but tedious and error-prone; because of limited resources we have instead provided a method that considers only the first required argument to a generic function, leaving the implementation of the multiple-dispatch aspect for further work.

While the presence of user-defined specializers makes it harder to reason about the cacheability of effective methods or lists of applicable methods, there are still points in the protocol discussed above that would allow a value to be computed once and reused for efficiency; our current implementation in Common Lisp does not take advantage of these.

5 Conclusions and Future Work

We have presented the implementation and use of custom specializers both in a Lisp dialect where that functionality is used, and also in Common Lisp, a language with a standardized core and *de facto* standard Metaobject Protocol.

The implementation in SKILL is complete and used in production: the implementation is fully functional, has an extensive suite of unit tests, and is part of live design projects. Much time has been spent on optimization and refactoring for performance and readability of the code, but of course much more work in this area could be done.

The functionality for the user to define their own specializer classes has been available in SBCL since May 2007; in practice the design space seems to be too general for easy exploration: having to reimplement the entirety of `compute-applicable-methods` and `mop:compute-applicable-methods-using-classes` is excessive. Our 'toy' implementation of the VCLOS protocols should be refined and extended, so that users can experiment with their specializer classes without having to reimplement complicated protocol functions.

In particular, it is important to take advantage of the various points in the protocol where memoization can be used (in the calculation of the effective method, for instance), so that the run-time overhead from use of user-defined specializers is as low as possible. Doing this would allow us to compare the efficiency of the protocol implementation in SKILL and Common Lisp, and to identify further points for optimization if necessary.

One thing missing from the Metaobject Protocol for Common Lisp (including our extension) is a general case for something that SBCL in particular takes advantage of: in SBCL, a method definition with a standard specializer will inform the method body (by inserting a declaration) that the corresponding element in the method function arguments is of a relevant type. There is at present no way of communicating this information for an arbitrary user-defined specializer.

Acknowledgments

We thank the reviewers for their detailed comments on drafts of this paper. SKILL® is a registered trademark of Cadence Design Systems, Inc.

References

[Bar90] Barnes, T. J.: SKILL: A CAD system extension language; In *DAC '90*, pages 266–271. ACM, 1990.

[BdL00] Bradshaw, T. and de Lacaze, R.: A Survey of Current CLOS MOP Implementations; In *Japan Lisp Users Group Meeting*, 2000.

[BKK$^+$86] Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F.: Common Loops: Merging Lisp and Object-Oriented Programming; In *OOPSLA'86 Proceedings*, pages 17–29, 1986.

[ISO95] The ISO Prolog Standard; ISO/IEC 13211-1:1995, Internatonal Standards Organization, 1995.

[KdRB91] Kiczales, G., des Rivières, J., and Bobrow, D. G.: *The Art of the Metaobject Protocol* MIT Press, 1991.

[KR93] Kiczales, G. and Rodriguez Jr., L. H.: Efficient method dispatch in PCL; In Paepcke, A., editor, *Object-Oriented Programming: the CLOS Perspective*, pages 335–348. MIT Press, Cambridge, Mass., 1993.

[LFM01] Le Fessant, F. and Maranget, L.: Optimizing Pattern Matching; In *ICFP'01 Proceedings*, pages 26–37, 2001.

[Mil04] Milstein, T.: Practical Predicate Dispatch; In *OOPSLA '04*, pages 345–364. ACM, 2004.

[Moo86] Moon, D.: Object Oriented Programming with *Flavors*; In *OOPSLA'86 Proceedings*, pages 1–8, 1986.

[MTHM97] Milner, R., Tofte, M., Harper, R., and MacQueen, D.: *The Definition of Standard ML* MIT Press, Revised edition, 1997.

[N$^+$00] Newman, W. H. et al.: SBCL User Manual; `http://www.sbcl.org/manual/`, 2000.

[PC94] Pitman, K. and Chapman, K., editors *Information Technology – Programming Language – Common Lisp* Number 226–1994 in INCITS. ANSI, 1994.

[Pet93] Petrus, E. S.: SKILL: a Lisp based extension language; *Lisp Pointers*, VI(3):71–79, 1993.

[RSM05] Rhodes, C., Strandh, R., and Mastenbrook, B.: Syntax Analysis in the Climacs Text Editor; In *International Lisp Conference Proceedings*, 2005.

[Ste90] Steele, G. L., Jr: *Common Lisp: The Language* Digital Press, second edition, 1990.

[Uck01] Ucko, A. M.: Predicate dispatching in the Common Lisp Object System; Technical Report AITR-2001-006, MIT AI Lab, Cambridge, MA, 2001 MEng thesis.

A Common Lisp extension to the MOP

The CLOS Metaobject Protocol requires little extension to support everything discussed in this paper. On a fundamental level, in fact, no new operators are required, though even full implementations of the CLOS MOP may need careful attention to remove any assumptions that only predefined metaclasses can act as specializers. For convenient use of the standardized operators `defmethod` and `find-method` we propose an analogue to `mop:make-method-lambda` and operators to parse and unparse parameter specializer names; in the description that follows, we assume that the reader is familiar with the CLOS MOP [KdRB91].

In order to emulate the specific specializer handling present in VCLOS, an overriding implementation of `compute-applicable-methods` and `mop:compute-applicable-methods-using-classes` would be necessary. However, for any particular strategy for dealing with the method applicability and ordering computation, such an implementation need only be written once; once written, the CLOS user would be free to implement specializers using the defined protocol.

A.1 Dictionary

Generic Function `parse-specializer-using-class`

Syntax:

`parse-specializer-using-class` *generic-function specializer-name*

This generic function returns an instance of `mop:specializer`, representing the specializer named by *specializer-name* in the context of *generic-function*.

Primary Method `parse-specializer-using-class` (*gf* `standard-generic-function`) (*name* `t`)

This method applies the standard parsing rules for consistency with the specified behaviour of `find-method`.

Generic Function `unparse-specializer-using-class`

Syntax:

`unparse-specializer-using-class` *generic-function specializer*

This generic function returns the name of *specializer* for generic functions with class the same as *generic-function*

Primary Method `unparse-specializer-using-class` (*gf* `standard-generic-function`) (*specializer* `specializer`)

This method applies the standard unparsing rules for consistency with the specified behaviour of `find-method`.

Generic Function `make-method-specializers-form`

Syntax:

`make-method-specializers-form` *generic-function method specializer-names env*

This function is called with (maybe uninitialized, as with the analogous arguments to `mop:make-method-lambda`) *generic-function* and *method*, and a list of specializer names (being the parameter specializer names from a `defmethod` form, or the symbol `t` if unsupplied), and returns a form that evaluates to a list of specializer objects in the lexical environment of the `defmethod` form.

Primary Method `make-method-specializers-form` (*gf* `standard-generic-function`) (*method* `standard-method`) *names env*

This method implements the standard behaviour for parameter specializer names.

B Example in Common Lisp

We present here the walker example discussed in Section 3, but this time with all the necessary support code to run in an unmodified SBCL (which incorporates the proposals described in Appendix A). As discussed in Section 4.2.1, the support code is far from complete, as it supports only a small subset of the VCLOS protocols; we hope, however, that the issues raised in Section 4.2.1 are clear from this unpolished implementation.

B.1 Base VCLOS Protocol

Firstly, we define a generic function class to represent generic functions obeying the restricted VCLOS protocol:

```
(defclass vclos-generic-function (standard-generic-function) ()
  (:metaclass sb-mop:funcallable-standard-class))
```

We then define generic functions for the VCLOS protocols, with default methods for the base class:

```
(defgeneric specializer-of (object generic-function)
  (:method (o (gf vclos-generic-function)) (class-of o)))
(defgeneric available-specializers (generic-function)
  (:method ((gf vclos-generic-function))
    (list (find-class 'sb-mop:eql-specializer) (find-class 'class))))
(defgeneric specializer-precedence-list (specializer)
  (:method ((c class)) (sb-mop:class-precedence-list c))
  (:method ((e sb-mop:eql-specializer))
    (let ((class (class-of (sb-mop:eql-specializer-object e))))
      (cons e (specializer-precedence-list class)))))
```

```
(defgeneric cmp-like-specializers
    (specializer1 specializer2 generic-function)
  (:method ((spec1 sb-mop:specializer) (spec2 sb-mop:specializer) gf)
    (member spec2 (specializer-precedence-list spec1))))
(defgeneric specializer-applicable-p (specializer object)
  (:method ((c class) object)
    (typep object c))
  (:method ((e sb-mop:eql-specializer) object)
    (eql (sb-mop:eql-specializer-object e) object)))
```

In order to make our VCLOS protocol available to be used when calling instances of our generic function class, we must define methods for the standard metaobject protocol functions relating to generic function invocation. Below, we present the minimum necessary to achieve the necessary functionality; a more polished implementation would support multiple specialized arguments, cacheing of applicable methods where possible through the use of `mop:compute-applicable-methods-using-specializers`, cacheing of effective methods within `mop:compute-discriminating-function`, and other functionality besides.

```
(defmethod compute-applicable-methods ((gf vclos-generic-function) args)
  (let ((arg (car args))
        (methods))
    (dolist (m (sb-mop:generic-function-methods gf))
      (let ((mspec (car (sb-mop:method-specializers m))))
        (when (specializer-applicable-p mspec arg)
          (push m methods))))
    (let ((available (available-specializers gf)))
      (flet ((sorter (a b)
               (if (eq (class-of a) (class-of b))
                   (cmp-like-specializers a b gf)
                   (member b (member a available :test #'typep)
                           :test #'typep)))
             (key (m)
               (car (sb-mop:method-specializers m))))
        (setq methods (sort methods #'sorter :key #'key))))
    methods))

(defmethod sb-mop:compute-discriminating-function
    ((gf vclos-generic-function))
  (lambda (arg &rest args)
    (let ((specializer (specializer-of arg gf)))
      (multiple-value-bind (methods cachep)
          (values nil nil) ; C-A-M-U-S
        (unless cachep
          (setq methods (compute-applicable-methods gf (list arg))))
        (let* ((mc (sb-mop:generic-function-method-combination gf))
               (emf (sb-mop:compute-effective-method gf mc methods)))
          (interpret-effective-method emf (cons arg args)))))))
```

```
(defun interpret-effective-method (effective-method args)
  (eval `(locally
          (declare (sb-ext:disable-package-locks
                     call-method make-method))
          (macrolet ((call-method (method next-methods &rest cm-args)
                       `(apply (sb-mop:method-function ,method)
                               ',',args (list ,@next-methods) ',cm-args))
                      (make-method (form)
                       `(make-instance 'standard-method
                         :function (lambda (a nm &rest cm-a)
                                     ,form))))
            (declare (sb-ext:enable-package-locks
                       call-method make-method))
            ,effective-method))))
```

B.2 Implementing Cons Specializers

We are now in a position to implement support for a generic function class that can handle `cons` specializers for the first argument, in addition to `class` and `eql` specializers. Firstly, the metaclass definitions for the generic function and the specializer:

```
(defclass cons-generic-function (vclos-generic-function) ()
  (:metaclass sb-mop:funcallable-standard-class))
(defclass cons-specializer (sb-mop:specializer)
  ((inner :initarg :inner :reader inner)
   (direct-methods :initform nil)))
```

We must provide overriding methods for some of the VCLOS protocol functions, in order to support the new specializer within the VCLOS protocol:

```
(defmethod available-specializers ((gf cons-generic-function))
  (mapcar #'find-class '(sb-mop:eql-specializer cons-specializer class)))
(defmethod specializer-of (o (gf cons-generic-function))
  (if (consp o)
      (intern-cons-specializer (class-of (car o)))
      (call-next-method)))
(defmethod specializer-applicable-p ((spec cons-specializer) o)
  (and (consp o) (specializer-applicable-p (inner spec) (car o))))
```

We must also implement methods for some standard MOP functions. In addition to these two, a polished implementation would support `specializer-direct-methods` and `specializer-direct-generic-functions`.

```
(defmethod sb-mop:add-direct-method
    ((specializer cons-specializer) method)
  (pushnew method (slot-value specializer 'direct-methods)))
(defmethod sb-mop:remove-direct-method
    ((specializer cons-specializer) method)
  (setf (slot-value specializer 'direct-methods)
        (remove method (slot-value specializer 'direct-methods))))
```

Finally, we define a method on our operator `make-method-specializers-form` (described in appendix A), to enable `defmethod` forms to expand into code creating instances of our new specializer class.

```
(let ((table (make-hash-table)))
  (defun intern-cons-specializer (inner)
    (or (gethash inner table)
        (setf (gethash inner table)
              (make-instance 'cons-specializer :inner inner)))))
(defun make-specializer-form (name)
  (if (atom name)
      `(find-class ',name)
      (ecase (car name)
        ((eql) `(sb-mop:intern-eql-specializer ,(cadr name)))
        ((cons) `(intern-cons-specializer
                  ,(make-specializer-form (cadr name)))))))
(defmethod sb-pcl:make-method-specializers-form
    ((gf cons-generic-function) method names env)
  `(list ,@(loop for name in names
                 collect (make-specializer-form name))))
```

B.3 The Walker

We are now able to use our new generic function and specializer classes to implement the code walker of Section 3. Firstly, we define a few utilities:

```
(defclass binding ()
  ((used :initform nil :accessor used)))
(defun derive-bindings-from-ll (lambda-list)
  (mapcar (lambda (n) (cons n (make-instance 'binding))) lambda-list))
(defun make-env (bindings env)
  (append bindings env))
(defun find-binding (env var)
  (cdr (assoc var env)))
```

and then the `walk` generic function and methods

```
(defgeneric walk (expr env call-stack)
  (:generic-function-class cons-generic-function))

(defmethod walk ((expr list) env call-stack)
  (let ((cs (cons expr call-stack)))
    (walk (car expr) env cs)
    (walk (cdr expr) env cs)))
(defmethod walk ((expr (eql nil)) env call-stack)
  nil)
(defmethod walk ((expr t) env call-stack)
  (format t "~&invalid expression ~A: ~A: ~A~%"
          (class-name (class-of expr)) expr call-stack))
(defmethod walk ((expr (cons (eql 'quote))) env call-stack)
  nil)
```

```
(defmethod walk ((var symbol) env call-stack)
  (let ((binding (find-binding env var)))
    (if binding
        (setf (used binding) t)
        (format t "~&unbound: ~A: ~A~%" var call-stack))))
(defmethod walk ((form (cons (eql 'lambda))) env call-stack)
  (destructuring-bind (lambda lambda-list &rest body) form
    (let ((bindings (derive-bindings-from-ll lambda-list)))
      (dolist (form body)
        (walk form (make-env bindings env) (cons form call-stack)))
      (dolist (bind bindings)
        (unless (used (cdr bind))
          (format t "~&unused: ~A: ~A~%" (car bind) call-stack))))))
```

Some test forms for this walker are below; respectively, they should return silently, print a diagnostic about an unused variable `x`, and print a diagnostic about an unbound variable `x`.

```
(walk '((lambda (x) x) nil) nil nil)
(walk '((lambda (x) 'x) nil) nil nil)
(walk '((lambda () x)) nil nil)
```