Object Oriented Systems **3,** (1996) 31–49

An implementation of Telos in Common Lisp

RUSSELL BRADFORD

School of Mathematical Sciences, University of Bath, Claverton Down, Bath, BA2 7AY, UK

Telos (The EuLisp Object System) is a new Metaobject Protocol (MOP) designed as part of the developing EuLisp standard. Much simpler than CLOS, it is intended to implement as efficiently as possible those parts of object-oriented programming that are most commonly used, while still being flexible enough to allow more complicated models (such as method combination, multiple inheritance) in user-implementable code or libraries. Whereas Telos is part of the EuLisp standard, the MOP is more-or-less independent of the dialect of Lisp. This paper describes our experience of the implementation of the Telos MOP in Common Lisp (CLtL1) from scratch (no other object code, such as CLOS, is used). We then investigate the meta-programming of some the more advanced facilities available in CLOS using the Telos MOP.

Keywords: Metaobject protocol, Telos, EuLisp, MOP, CLOS

1. Introduction

EuLisp [1, 2] has several distinguishing features including threads, modules and notably the object system, Telos. Telos was designed after CLOS [3], and after the CLOS MOP [4, 5] and integrates directly as part of the language so there is no type versus class distinction, as in CLOS. Several parts of the EuLisp language use the object-oriented features in a natural manner (e.g. the exception handling mechanism). Some details of the Telos MOP are still under discussion, but its overall design is stable enough for there to have been several test implementations, including one as part of a EuLisp trial implementation [6], one in Scheme [7] and one in Common Lisp, the subject of this paper. By default the Telos MOP supports a system that is very much simpler than CLOS as many of the more complicated features such as method combination and multiple inheritance (MI) are missing. This is because the authors of Telos wanted a clear distinction between object level and metaobject level programming since this helps many optimizations and eliminates unnecessary overheads. For example, the programmable mechanism for slot access in a general class needed in a metaobject system can be simplified greatly when we have guarantees on the particular structure of a given class (e.g. if we know an instance is represented by a vector, we can use a simple vector reference to retrieve a value). This is often called a ‘don’t use, don’t lose’ philosophy: if a program does not require sophisticated facilities such as multiple inheritance or being able dynamically to change the class of an object there is no reason for it to pay the overhead of supporting them.

The Telos MOP is designed so that it is of at least equal power to CLOS, that is, any CLOS construct can be programmed in Telos as user code or can be loaded from libraries. One of the aims of this paper is to show that this is indeed true.

The structure of this paper is as follows: we give some background on object and metaobject systems including a brief outline of Telos, then move on to a discussion of the low-level support decisions to be made before a MOP can be begun to be written. Next we outline the method of bootstrapping Telos and we finish with a few examples that demonstrate the power of Telos by using metaobject programming to implement structures, multiple inheritance and method combination.

Fuller descriptions of Telos can be found in [8], [9] (also available as [1]), and a special edition of *Lisp and Symbolic Computation* devoted to EuLisp [2]. This paper describes the state of Telos as found in EuLisp draft 0.99; some details may have changed in later drafts.

2. Background: metaobject programming

The style of object programming as provided by languages such as C++ is fixed. That is to say, the protocol that determines which method to apply in any particular circumstance is given as part of the language definition. The language definition also dictates whether there is dispatch off a single argument or off multiple arguments of a generic function; whether there is single or multiple inheritance of behaviour of classes; how slots are accessed in instances of classes; the structure the classes themselves – these and many other parts are given and cannot be changed. The relaxation of these constraints is the purpose of *metaobject programming*, namely the programming of the object system itself.

With a metaobject system the programmer can choose the details of inheritance of class behaviour, can define new classes that have instances that live on disk instead of in memory (persistence), can define generic functions where the order of applicable methods is reversed, or even randomized each time the function is called. To be able to define such new behaviours we require a protocol to control the object protocol: such a thing is called a *metaobject protocol* (MOP).

A MOP generally consists of a collection of classes and functions, together with their call-graphs, that create and control the object protocol. For example, the class creation protocol (see Fig. 6) defines a collection of functions that create and initialize class objects, determines if and how properties of those classes are to be inherited from superclasses, and creates functions to read and write slots in instances. Similar protocols determine the structure and behaviour of generic functions and the other parts of the object system.

Of course, the best way to make a MOP flexible and easy to use is to make it object oriented, and the best candidate language is itself. Indeed the Telos MOP is defined and implemented in terms of itself, i.e. is metacircular. By its very nature, a MOP is *introspective* as it can access data structures that represent its own behaviour, including first-class classes, generic functions, methods and so on. These structures can be modified dynamically, e.g. by adding a new method to a generic function, so in this sense a MOP is *reflective.*

A major worry of using a MOP is the potential loss of efficiency over a ‘hard-wired’ object system. If we had to compute all behaviour at run-time the overhead would be unacceptable. Therefore a well-designed MOP will endeavour to precompute as much behaviour as possible as early as possible, preferably as early as compile-time (though probably at load-time). This means that the MOP must have a clear idea of what is a run-time requirement and what is part of the development system. Similarly, making a clear separation between object and metaobject programming will aid early computation though class-specific optimizations.

One of the first MOPs was developed on top of the Common Lisp object system (CLOS), a large and complicated language. A mostly-portable implementation of the object system, PCL, helped to make CLOS firstly a *de facto*, and now a *de jure* [3] standard for object programming in Common Lisp. With need for metaprogramming came the CLOS MOP – however this was severely constrained in its requirement to retrofit to CLOS. Some features (e.g. slot access) in CLOS have certain guaranteed behaviours, and to provide these behaviours compatibly in a MOP turns out to be computationally very expensive unless a great deal of work is done on optimization. The PCL implementation is exemplary in this regard, but this in turn has a cost in complexity of code. Furthermore, there is an overhead of overgenerality; some features (e.g. class redefinition) require mechanisms to be present even if those features are never used. This is an extra cost in both code size and run time unless there is sophisticated optimization present.

See [5] for extensive discussion on CLOS, and [4] for details on the CLOS MOP.

When EuLisp was designed there was the opportunity for a fresh start. No longer constrained by a preexisting object system, the aims of simplicity, efficiency and power were simultaneously achieved in the Telos MOP [8]. Although the default methods in the MOP do not implement the full range of CLOS behaviours, we illustrate in this paper that they can be implemented. Moreover, Telos has many good properties; much computation is trivially promoted to load-time, leading to increased efficiency before any optimization; classes that do not use sophisticated facilities need not be hamstrung by their presence; object and metaobject parts are clearly distinguishable; the core MOP is small, but is sufficiently expressive to allow more optimization if required.

3. Overview of Telos

Following the EuLisp design principle of combining simplicity with flexibility by the use of modules, the object protocol as set out in the definition is layered. The Level-0 layer is a simple, non-introspective object system. It has restricted power, perhaps roughly equivalent to that of classes in C++, with comparable scope for efficient compilation. Level-1 is a full-featured MOP.

The message design follows that of CLOS, namely generic functions with (multi) methods. Generic functions are created by the `defgeneric` form and the argument list of the generic may be specialized to indicate restrictions of the domains of the methods that may be added. For example,

```
(defgeneric plus ((x <number>) (y <number>)))
```

defines a generic function which only allows methods with domains specializing `<number>` $\times$ `<number>`. By convention, as in Dylan [10], class names are distinguished by < >.

A method is added to a generic function by `defmethod`. Note that the generic function must already exist before methods are added to it. This is in contrast to CLOS, where `defmethod` may itself cause a new generic function to be created.

A method on a generic function can access the behaviour of less specific methods by means of method combination. The default is *primary method combination*, which is achieved by `call-next-method`. In the body of a method this form passes control to the next most specific method for the given arguments and returns whatever that method returns. CLOS also allows *standard* method combination, see Section 10.

New classes at Level-0 are created using the `defclass` form (Fig. 1). This is largely equivalent to CL's `defstruct` or C's `struct` (i.e. a record-like construct that is essentially a vector with named accessors). At Level-0 there may be at most one superclass; extensions in Level-1 (see Section 9) allow more than one. All user-defined classes at Level-0 share a single metaclass, which means that all their

```
(defclass class-name (superclass*)
   (slot-description*)
   class-option*)

(defgeneric gf-name specialised-argument-list
   gf-option*)

(defmethod gf-name method-option* specialised-argument-list form*)
```

Fig. 1. Defining syntaxes.

instances share a common structure. The class options allow the programmer to name constructors and predicates for the class (and allow extended features at Level-1).

In Level-0 EuLisp there is intentionally no scope for metaobject programming, as it is intended to be small and efficiently compilable – only object programming is possible. Metaobjects appear at Level-1, with the introduction of `class-of`, an extended `defclass` and the full MOP. Additional allowable class options include the `class` option, which indicates the class of the newly defined class (i.e. the metaclass of the class instances).

At Level-1, `defgeneric` and `defmethod` allow extra initialization options. In `defgeneric` (Fig. 1) the `method-class` option determines the classes of the methods that can can be added, so that every added method must be a subclass of the declared method-class. In `defmethod` we can describe extra method options that may be required by more complicated classes of methods. Most importantly, there are numerous functions at Level-1 that define the functionality of the MOP in a manner similar to that described in [4]. See Section 6 for details.

Telos	Common Lisp/CLOS
first class classes	first class classes
first class generic functions and methods	first class generic functions and methods
multi methods	multi methods
classes are the type system	separate type system
metaobject protocol	metaobject protocol
	(not part of proposed ANSI standard)
single inheritance	multiple inheritance
primary method combination	standard method combination
class-based dispatch	class and instance-based dispatch
instance slots	class and instance slots
untyped slots	typed slots
	class and instance redefinition
	changing the class of an object

Fig. 2. Default Telos and CLOS features.

The default Telos protocol only defines a restricted set of those behaviours that appear in CLOS: see Fig. 2 for an overview.

4. Low-level decisions

We began with a bare Common Lisp [3] (principally AKCL), devoid of any support for object-oriented programming, so there were many low-level implementation decisions that needed to be resolved. In particular, we needed to determine the class and instance structure, resolve the integration of Telos classes and CL types and determine how to represent generic functions as Common Lisp functions.

The principal factor underlying all the decisions was that of simplicity; given the choice of a simple algorithm or data structure and a complicated one we always chose the former, except in cases where the former was overwhelmingly inefficient. Efficiency itself was low on the list of priorities, well below clarity of code. This precluded machine and Lisp-specific support in the underlying Lisp. This is not to say the resulting system is devastatingly slow – quite the converse is true, mostly due to the design of the Telos MOP.

4.1 *Object and classes*

We implemented classes and instances as CL structs (Fig. 3). The `primitive-class` structure is functionally equivalent to a vector of two slots, the first slot being read and set by the function `primitive-class-class`, the second by `primitive-class-slots`. (The `defstruct` form also names the function that should be used to print an instance as `primitive-print`.) The `class` slot contains the class of the object and the `slots` slot contains the slots of the object, typically as a simple vector (see [12] for the object structure in PCL). This encodes the class of an object directly, and provides a uniform structure for all Telos instances. Thus a call to `class-of` is generally a single slot look-up on a Telos instance. Telos classes that correspond to CL types, such as `integer`, are kept in a hash-table. Calls to generic functions potentially require calls to `class-of` to determine the classes of their arguments, so it is worth putting a little effort into speed for this one function.

An alternative strategy would be to have objects implemented directly as structures (i.e. class slots are directly `defstruct` slots). This has the advantage of one less indirection to access a slot value, but is more problematic in the dynamic creation of classes (`defstruct` being a macro in CL, i.e. not a form that should be used dynamically), and the need for support for low-level Telos operations such as `primitive-class-of` and `(setf primitive-class-of)` which are necessary for the eventual implementation of the high-level extension `change-class`.

```
(defstruct (primitive-class (:print-function primitive-print))
  class
  slots)
```

Fig. 3. Primitive class definition.

4.2 *Metaclasses*

Metaclasses (classes that have classes as instances) are distinguishable in Telos as being direct or indirect subclasses of the class `<class>`. The EuLisp Telos definition allows for a fixed metaclass structure, which has two valuable consequences: firstly the system implementor may assume that all system metaclasses have a known structure; secondly that the system user cannot define metaclasses with a differing structure. This allows increased efficiency in the MOP (those functions that access metaclasses can be normal functions instead of generic functions) and increased ease of implementation of the bootstrapping process. Initially, our implementation allowed variable metaclass structure but it was found that the generality obtained by doing this was not used. Further, the bootstrap was problematic as generic function accessors needed to be defined in order to implement generic functions. Variable structure metaclasses were deemed to be more trouble than they were worth and eventually this code was replaced by much simplified code for fixed structure metaclasses that is faster to execute, clearer to read and consequently easier to understand and maintain.

4.3 *The CL type hierarchy*

Common Lisp has a powerful resident type system, so we must address the question of the integration of the class system and the type system. This question does not arise in EuLisp as the type system *is* the class system. It would be useful to have the class hierarchy follow the CL type hierarchy as closely as possible, but the latter does not seem to be sufficiently well-defined. That is to say the standard allows a great deal of flexibility over (a) what types exist, (b) which are distinct types and (c) which are subtypes of which. There are many good reasons for leaving these type relationships undefined – an implementation may wish to add extra intermediate classes for efficiency reasons, or may define certain classes in terms of others – however, a class system cannot follow suit when defining a MOP.

Regarding (a), an implementation is free to implement simple types (e.g. `fixnum`) and range types `(fixnum 1 10)`, and composite types `(cons fixnum null)`. Several implementations, it appears, have only simple types.

For (b), it is implementation dependent whether, say, `long-float` and `double-float` are distinct as types. Many systems conflate several types; only relatively few types must be distinct to conform with the CL definition (see [3]).

As for (c), it is generally open whether any type is a subtype of any other type. Some subtype relations are specified (e.g. a CL `defstruct` that derives from another is a subtype of it), but many are left unspecified.

This variety of type hierarchies led us to implement a simplified class hierarchy that only identifies the major types; thus we have `<integer>` to cover all the types of integer and `<float>` to cover all the types of floats. It must be noted that in the proposed ANSI standard for CL much more of this area is fixed, but as we were working mostly with CLtL1 conformant implementations there was some variance. The CL classes we defined did not need much functionality, indeed only the existence of classes is required for generic function dispatch. Thus CL classes as implemented have just enough structure to identify themselves and little more. Further, these classes share a metaclass `<built-in-class>` which disallows any introspection.

There is the additional complication of the ‘multiple inheritance’ in the CL type hierarchy, where the EuLisp Level-1 standard only requires single inheritance. In particular, in CL `null` is a subtype of both `list` and `symbol`, and `vector` is a subtype of both `sequence` and `array`. To mimic this with subclassing would require the associated classes to have multiple direct superclasses. The problem with `null` does not occur in EuLisp as the value of `nil` is not a symbol, but is the unique instance of the class `<null>`, which is a subclass of `<list>` and is a sibling class of `<cons>`. Also, EuLisp has no problem with `vector` since there are no arrays in the kernel definition.

There are other occurrences of multiple inheritance (MI) in the CL type hierarchy but they are not a problem as they do not have corresponding classes in the simplified hierarchy mentioned above. Thus, although `simple-vector` is a subtype of the unrelated types `vector` and `simple-array` it is not a problem as we only have the classes `<array>` and its subclass `<vector>`. As all the CL type classes are implemented as ‘hard-wired’ classes in the bootstrap code, it is a simple matter to tweak the class precedence lists for the special class `null` to make it appear to have multiple direct superclasses. This provides ‘method multiple inheritance’ at very little cost. There are no slots to complicate matters.

There was no hardship in hand crafting CL classes rather than the use of the MOP later to create them, since a decision regarding generic functions more-or-less forced this approach, anyway.

4.4 *Generic functions*

One glaring difference from EuLisp in our implementation is in the treatment of generic functions. In the PCL implementation of CLOS there is a section of machine-dependent code which implements generic functions as callable objects. This sometimes requires changes to parts of the underlying Lisp. Rather than follow this somewhat involved route, we chose the simpler path of using the two-valued nature of CL to store the generic function object in the value cell of a symbol and the associated discriminating function in the function cell. The discriminating function is a piece of code that takes the arguments that were passed to the generic function and determines and calls the appropriate method.

This means the user has to be slightly careful in the use of generic functions as values since there is a potential confusion of the generic function object and the function that implements its despatch. In practice, though, the confusion of object versus function seems rarely to happen.

In the source of the code of the machine-dependent section of PCL its authors note that it is possible to implement generic functions portably by the use of a hash table to associate generic function functions with the generic function objects. This is the route that Telosis [7] takes, but this seems to be not particularly satisfactory due to the extra overhead in generic function call, plus additional problems in implementation and the difficulty of garbage collection of dead generic functions in the absence of weak pointers.

Another problem was where in the initial class hierarchy to place the class `<generic-function>`, due to the difference between funcallable objects (a CL class) and generic function objects (a Telos class). We decided to place it as a subclass of `<function>` to allow methods to be defined uniformly over all kinds of functions. The class of the normal CL functions was named `<standard-function>` and was placed as a sibling to `<generic-function>`, both as subclasses of `<function>`. This had the consequence that some CL classes had to be hand crafted before the MOP proper could be written.

4.5 *Miscellaneous*

Next, classes are implemented as first-class objects. By this we mean that they are normal values, not hidden in a `find-class` reference of a symbol. We do this as it simplifies the code and it seems more natural in a MOP; after all, in a MOP the classes are the objects of primary interest.

The initial class hierarchy as implemented is illustrated in Fig. 4, where `built-in-class [class]` indicates the class bound to the name `<built-in-class>`, which is an instance of the class `<class>`. In the figure, indentation indicates the subclass relationship.

The non-CL part of the class hierarchy follows directly from the class hierarchy of EuLisp; this is mainly the introspective part of the MOP and is required to implement the MOP as defined in EuLisp. The only difference is `<generic-function>`, which we have placed as a subclass to `<function>`, as mentioned above. The CL part of the hierarchy follows the simplified CL type hierarchy.

```
object [class]
  class [class]
    function-class [class]
    built-in-class [class]
  method [class]
  slot-description [class]
    local-slot-description [class]
  built-in-object [class]
    number [built-in-class]
      rational [built-in-class]
        integer [built-in-class]
        ratio [built-in-class]
      float [built-in-class]
      complex [built-in-class]
    package [built-in-class]
    readtable [built-in-class]
    function [built-in-class]
      generic-function [function-class]
      standard-function [built-in-class]
    symbol [built-in-class]
      null [built-in-class]
    ...                                ; more built-in classes
    sequence [built-in-class]
      list [built-in-class]
        null [built-in-class]
        cons [built-in-class]
      ...                              ; more sequence classes
    structure [class]
```

Fig. 4. Initial CL class hierarchy.

5. The bootstrap

The bootstrap code of Telos is approximately 1200 lines of fairly sparse Lisp code, much of which is straightforward function definitions such as `class-name` and the like, building by hand the basic parts of the system classes such as `<class>`, `<list>`, and definitions of supporting macros such as `defclass`.

In a similar vein to the code in [4], we start with functions that are low-level analogues of generic functions that will be defined later. We use these to help build the initial classes, but we use the same defining forms throughout: thus to add a method to a generic function we always use the general macro `defmethod`. Now this should call the general instance-creation function `make` to make the method and then `add-method` to attach it to the generic function, but in the bootstrap these generic functions have not yet been defined. So instead this macro expands to calls of the functions `make-method` and `stable-add-method` (Fig. 5). These functions can divert to `primitive-` forms which make use of knowledge of the fixed class structure of `<generic-function>` and `<method>`. Later, when `add-method` is defined as part of the MOP, its default method more-or-less repeats the code for `primitive-add-method`. This way of programming has several advantages: firstly it allows the uniform use of `defmethod` as a means to add methods both within and outside of the bootstrap; secondly it provides a minor optimization in general code as the normal case of adding methods is trapped out and passed to specialized code (as in `make-method` above) rather than passing through the general generic route of `add-method`; and thirdly it allows a grounding of the potentially infinite recursion in the MOP. This (and similar other parts of the code) is the point at which the reflective nature of the MOP is terminated.

```
(defun make-method (method-class domain fn initargs)
  (if (and (eq method-class <method>)      ; if this is a basic method,
           (listp domain)                   ; i.e., class <method> and
           (functionp fn)                   ; there are no initargs
           (null initargs))
      (primitive-make-method domain fn)     ; then use the primitive form
      (apply #'make                         ; else use the general make
             method-class
             :domain domain
             :function fn
             initargs)))

(defun stable-add-method (gf md)
  (if (and (eq (class-of gf) <generic-function>) ; if trying to add a basic
           (eq (class-of md) <method>))          ; method to a basic
                                                 ; generic
      (primitive-add-method gf md)               ; then use primitive form
      (add-method gf md)))                       ; else use general form
```

Fig. 5. Bootstrapping methods.

6. The MOP in the MOP

Once the basic system classes `<class>`, `<object>`, `<method>`, `<generic-function>`, and `<slot-description>` have been created, together with the CL classes (in particular `<list>` and `<null>`) we can proceed with the MOP itself. Instances of the class `<slot-description>` control how slots are accessed; each slot in a class has an associated slot description which typically contains the functions that read and write the values of that slot in instances of the class, together with other useful information.

Due to the careful coding of macros and 'stable' functions as outlined above, the MOP is written entirely as an application of itself.

```
compatible-superclasses-p cl direct-superclasses -> boolean
  compatible-superclass-p cl superclass -> boolean
compute-class-precedence-list cl direct-superclasses -> list(class)
compute-inherited-initargs cl direct-superclasses ->
  list(list(initarg))
compute-initargs cl direct-initargs inherited-initargs -> list(initarg)
compute-inherited-slot-descriptions cl direct-superclasses ->
  list(list(slotd))
compute-slot-descriptions cl slot-specs inherited-slotds -> list(slotd)
  either
  compute-defined-slot-description cl slot-spec -> slotd
    compute-defined-slot-description-class cl slot-spec -> slotd-class
  or
  compute-specialized-slot-description cl inherited-slotds slot-spec
    -> slotd
    compute-specialized-slot-description-class cl inherited-slotds
      slot-spec -> slotd-class
compute-and-ensure-slot-accessors cl effective-slotds inherited-slotds
  -> list(slotd)
  compute-slot-reader cl slotd effective-slotds -> function
  compute-slot-writer cl slotd effective-slotds -> function
  ensure-slot-reader cl slotd effective-slotds reader -> function
    compute-primitive-reader-using-slot-description slotd cl
      effective-slotds -> function
      compute-primitive-reader-using-class cl slotd effective-slotds
        -> function
  ensure-slot-writer cl slotd effective-slotds writer -> function
    compute-primitive-writer-using-slot-description slotd cl
      effective-slotds -> function
      compute-primitive-writer-using-class cl slotd effective-slotds
        -> function
```

Fig. 6. Class initialization protocol.

The EuLisp definition lays out the Telos MOP quite clearly giving the call structure and functionality of the default methods. For example, the `initialize` method for `<class>` (which is the core of the MOP) has the call structure shown in Fig. 6. Thus `initialize` first calls `compatible-superclasses-p` with the new class and its prospective direct superclasses and returns true if the new

```
(defmethod initialize ((cl <class>) (initargs <list>))
  (let ((name
        (find-key :name initargs :anonymous))
       (direct-supers
        (find-key :direct-superclasses initargs (list <object>)))
       (direct-slotds
        (find-key :direct-slot-descriptions initargs ()))
       (direct-inits
        (find-key :direct-initargs initargs ())))
     (call-next-method)
     (setf (class-name cl) name)
     (setf (class-direct-superclasses cl) direct-supers)
     (setf (class-direct-subclasses cl) ())
     (unless (compatible-superclasses-p cl direct-supers)
       (error
       "incompatible superclasses:~%~s can not be a subclass of ~%~s"
       cl direct-supers))
     (let ((cpl (compute-class-precedence-list cl direct-supers)))
       (setf (class-precedence-list cl) cpl)
       (let ((effective-inits (compute-initargs
                              cl direct-inits
                              (compute-inherited-initargs
                              cl direct-supers))))
        (setf (class-initargs cl) effective-inits)
        (let ((inherited-slotds (compute-inherited-slot-descriptions
                                cl direct-supers)))
         (let ((effective-slotds
                (compute-and-ensure-slot-accessors
                 cl (compute-slot-descriptions
                    cl direct-slotds inherited-slotds)
                 inherited-slotds)))
           (setf (class-slot-descriptions cl) effective-slotds)
           (setf (class-instance-length cl)
             (length effective-slotds))
           (mapcar #'(lambda (super)
                        (add-subclass super cl)) direct-supers))))))
  cl)
```

Fig. 7. Code for class initialization.

class is compatible with those superclasses. To determine this, `compatible-superclasses-p` calls `compatible-superclass-p`. Next `initialize` calls `compute-class-precedence-list`, and so on.

Implementation of this description was mostly straightforward and almost a direct transcription from the definition, copying the call structures and the descriptions of the default methods. For example, the code for the `initialize` method for `<class>` is given in Fig. 7.

An overview of this code:

(1) This defines a method on `initialize`, the generic function that takes a newly-created object and initializes it appropriately. It is passed two arguments; the first is the object we are initializing (in this case the class `cl` we are defining), the second is `initargs`, a list of key-value pairs.

(2–9) The `let` form declares some local variables which contain keywords passed to `initialize`. The form `(find-key key keylist default)` looks for the `key` in the `keylist` and returns its associated value, or `default` if it not present.

(10) The first thing we do is call the next most specific method on `initialize`. This will be the general `initialize` method defined on `<object>` and it will perform those initializations common to all objects.

(11–13) These lines set some slot values in the class, namely its name, direct superclasses and direct subclasses (initially empty). The function `compatible-superclasses-p` checks to see if the class we are defining can be a legitimate subclass of its proposed superclasses. If not, an error is signalled – some classes cannot be subclassed, e.g. `<null>`.

(18) The local variable `cpl` contains the precedence list for instances of this class: normally this is the list of superclasses, in order, from this class back up to the topmost class, `<object>`. This list can be used for computation of the order of applicability of methods defined on this class.

(20–23) Next, we compute the effective initargs for this class. This is the list of keywords that the class will recognize. Effective initargs can be direct (line 21) or inherited (lines 22–23). Normally this returns the union of all new and inherited keywords.

(27–31) A similar computation is performed for the slots of the class, though combining inherited and direct slots is a much more complicated procedure, and we need to ensure that accessors defined on slots work correctly (`compute-and-ensure-slot-accessors`), as inherited slots may have changed relative positions in a new subclass (this does not happen in the case of single inheritance but it is required to support multiple inheritance – see Section 9).

(33–36) Finally, there are two bits of bookkeeping: line 33 makes a count of the number of effective slots, and lines 34–35 notify the new class's superclasses that they have a new subclass. The result of the method (line 36) is `cl`, the now fully initialized class.

There are only minor differences between the specification of Fig. 6 and the the implementation of Fig. 7, and they are 'inessentials' such as `add-subclass`, which are not part of the definition, but are useful for development. One more point: the specification does not detail *when* the initialization happens. It may be that an implementation chooses to perform only a small part of the initialization protocol in the `initialize` method, leaving the class to be *finalized* at some later point. This lazy initialization could be valuable in an application that creates large numbers of classes and only uses details of a few.

The rest of the MOP is similar in style. Due to the simplicity of the default methods (no MI, no complicated method combination, etc.) the code for the MOP itself is about 800 lines.

7. Notes on efficiency

Whereas this implementation was not designed with efficiency uppermost in mind, there are several points that are worth mentioning. No explicit reliance on the Lisp compiler was made (we used AKCL primarily, testing additionally on Clisp and CMU Lisp), nor was any Lisp dependent code used (except to avoid bugs in certain compilers).

Slot access costs just one generic function call over a primitive reference: this is due to the design of the MOP, which computes functions to read and write slots at slot-description creation time. An intelligent Lisp compiler that was Telos-aware could feasibly eliminate the generic call, too. There is no recourse to `slot-value` or equivalent as required in CLOS, in fact this function is not even in the EuLisp definition. Of course, `slot-value` is definable by the user if required.

The only explicit attempt at optimization in the code was to implement generic function method cacheing, without which generic function call becomes significantly slower. Even a simple cache for each generic function that maps signatures to methods is enough to reduce the generic function call overhead significantly. The overhead is reduced to some calls of `class-of` to compute the classes of the arguments passed, then a single table look-up in the case of cache hit. In the case of cache miss, the full MOP procedure is followed, going via the method look-up function that was created at generic function definition time. Simple tests (running some applications with and without cacheing enabled) show that even this trivial kind of cacheing can reduce the generic function call overhead by about 50–75%.

On the other hand, there are a few common compiler features of which we could make general use, in particular inlining. Inlining low-level accessors such as `primitive-class-slots` and `primitive-ref` (which returns the value of a slot in a `primitive-class` structure) could improve the speed of the compiled MOP code.

8. Structures

Having implemented the MOP, we next considered extensions of the basic functionality.

As an initial attempt at the use of the MOP we decided to re-implement structures (*à la* CL `defstruct`) using classes. Structures in CL are very simple objects, they have only single inheritance, no slot redefinition and other features which allow them to be compiled very efficiently for fast slot access, quite possibly to a single memory reference.

The principal feature we wished to emulate was this low overhead of slot access. The default way of accessing slots in Telos is by means of generic functions whose methods are created at slot definition time. The default method for creating slot reader methods (Fig. 8) assumes an existing method is correct, i.e. that a method to access the slot inherited from the parent class will also work on the new structure. Otherwise we compute the primitive reader that can actually find the value in a slot (typically this is a vector reference) and add this as a method to the reader function. There is the possibility here of different classes defining slot readers with the same name, but in practice this seems to arise only rarely

```
(defmethod ensure-slot-reader ((cl <class>)
                               (slotd <local-slot-description>)
                               (effective-slotds <list>)
                               (reader <generic-function>))
  (when (null (generic-function-methods reader))  ; if there are no
                                                  ; existing methods
    (let ((primitive-reader                       ; then compute a new
                                                  ; reader
           (compute-primitive-reader-using-slot-description
            slotd cl effective-slotds)))
      (add-method reader                          ; and add it to the
                                                  ; reader
                  (method-lambda                  ; as a method
                   :class (generic-function-method-class reader)
                   ((obj cl))
                   (funcall primitive-reader obj)))))
  reader)                                         ; otherwise return
                                                  ; reader unchanged
```

Fig. 8. Creating slot reader methods.

due to conventions on naming accessors and to the use of modules/packages. A more careful method on `ensure-slot-reader` could check for subclass compatibility and if necessary add a new method that specializes correctly.

When called from `defclass` the result of `ensure-slot-reader` is bound to the name of the reader or accessor as given in the `defclass` slot options. Thus, even in this case, for slot access we only incur a single generic function call overhead more than a standard function call. However, we would like to use the MOP to reduce or avoid this, if possible.

The normal class initialization of new slot accessors proceeds through three functions (Fig. 9) and similar functions for the writer. First, the default method for `compute-slot-reader` creates a generic function, to which a method is added by `ensure-slot-reader`. To avoid creating this generic function we specialize the protocol at `compute-and-ensure-slot-accessors`. See Fig. 10.

This means that the slot readers and writers for metainstances (instances of instances) of this class are simple calls to `primitive-ref`. Thus slot access for these instances are one extra function call over the `primitive-ref`, which is itself one extra indirection to access the `slots` slot within the `primitive-class`. This is not absolutely minimal access, but inlining could help by reducing these overheads, too.

```
compute-and-ensure-slot-accessors
  compute-slot-reader
  ensure-slot-reader
```

Fig. 9. Initialization of slot accessors.

```
;; define the new metaclass of all structures
;; it is both an instance of, and is subclass to the class <class>
(defclass <structure-class> (<class>) () :class <class>)

;; specialize the way we compute slot accessors
(defmethod compute-and-ensure-slot-accessors
  ((c <structure-class>) (effective-slotds <list>)
   (inherited-slotds <list>))
  (declare (ignore c inherited-slotds))
  (structure-c-a-e-s-a effective-slotds 0)
  effective-slotds)

;; compute an accessor that reads slot numbered "index"
(defun structure-c-a-e-s-a (effective-slotds index)
  (unless (null effective-slotds)
    (setf (slot-description-slot-reader (car effective-slotds))
          #'(lambda (obj)                   ; set the reader of this slot
                                            ; to be a
              (primitive-ref obj index)))  ; function that does a single
                                            ; primitive-ref
    (setf (slot-description-slot-writer (car effective-slotds))
          #'(lambda (obj val)               ; ditto for writer
              (setf (primitive-ref obj index) val)))
    (structure-c-a-e-s-a (cdr effective-slotds) (+ index 1))))
                                            ; recursively do next slot
```

Fig. 10. Code to create structure accessors.

9. Multiple inheritance

The EuLisp definition only specifies a single inheritance model in the default MOP. Addition of a metaclass that supports MI is straightforward, though a trifle involved. There are several kinds of MI and we chose to imitate the CLOS variety (many believe CLOS MI is of questionable semantics and recommend simpler models, e.g. mixins in [13] and [14]; see also [15]).

The major specializations are to the functions in Fig. 11. These are the main functions that take care of

```
compute-class-precedence-list
compute-slot-descriptions
  compute-inherited-slot-descriptions
  compute-specialized-slot-description
compute-and-ensure-slot-accessors
  ensure-slot-reader
  ensure-slot-writer
```

Fig. 11. Specializing slot descriptions.

inheritance of slots and ensuring accessor methods work correctly. Other functions require minor changes to take into account possible multiple superclasses (e.g. `compatible-superclasses-p`).

The first, `compute-class-precedence-list`, is straightforward to code, only requiring a topological sorting routine to order the class precedence lists of the superclasses. We must also consider the addition of new methods to `ensure-slot-reader` and `ensure-slot-writer` that take into account the possibility of a slot changing position, and the addition of new methods to `compute-specialized-slot-description` which governs the merging of new and old slot specifications. These rely on the class precedence list (CPL) for their semantics.

It is remarkable how little work is actually needed to add MI to the MOP: once the semantics of slot inheritance are understood it is simply a matter of writing the (admittedly somewhat fiddly) code to implement them, as all the hooks in the MOP (i.e. `compute-slot-descriptions` and friends) are ready and waiting to accommodate the new code.

Complications arise from the peculiarities of the semantics of CLOS inheritance. For example, [15] and [16] highlight the fact that inheritance of behaviour of slots in CLOS depends not only on direct superclasses, but also on classes higher in the inheritance tree (a failure of their monotonicity–incrementality principle). In particular, this manifests itself in the inheritance of slot initialization functions: a slot inherits not the closest function (i.e. as ordered by the CPL), but the function from the closest slot that had one defined directly. To manage this MI classes require slot descriptions to record whether an function was defined directly or was inherited. Such slot descriptions are implemented simply as new subclasses of `<slot-description>`.

10. Method combination

The default MOP only supports primary methods. Another easy addition is a new class of generic functions that allow CLOS-like method combination using *before*, *around*, and *after* methods. These are methods that supplement the existing methods on a generic function, by running before, or after, or instead of the existing methods.

This is a simple extension of the existing code for method discrimination. The call structure for generic function initialization is given in Fig. 12.

We specialize `compute-discriminating-function` and `compute-method-lookup-function` with new methods. The role of the first function is to create a discriminating function for a new generic function; typically the discriminating function implements the cache and resorts to calling the method lookup function in case of a cache miss. The method lookup function, which is computed by `compute-method-lookup-function`, creates a function that returns a sorted list of applicable methods when presented with a set of arguments. Incidentally, `compute-discriminating-function` may also be called by `add-method` to provide extra scope for method optimization whenever a new method is added to a generic function.

```
compute-method-lookup-function
compute-discriminating-function
```

Fig. 12. Generic function initialization.

We create a new subclass `<mc-generic-function>` of `<generic-function>` that has slots for the three new types of method and a new subclass `<mc-method>` of `<method>` that has slots named `before`, `after` and `around` to store flags to indicate the position of the method (this is somewhat wasteful, but admits simple code).

Definition of a method-combination generic is now simple (Fig. 13). The `defgeneric` form indicates that the default class of its methods should be `<mc-method>`.

The macro `defmethod` eventually results in a call of `make` on `<mc-method>` to make the new method, and a call of `add-method` to add it to the generic function (cf. Section 5). The keyword and value `:around t` are passed to the call of `make`, thus initializing the corresponding slot in that method with the value `t` (true). The subsequent call to `add-method` can use this value stored in the method to update the collection of methods attached to the generic function `foo` accordingly.

The new methods on `compute-discriminating-function` and `compute-method-lookup-function` return functions that take into account the position type of the method, and which (respectively) know how to call, and how to choose and sort applicable methods accordingly. This suffices to implement method combination in Telos in a completely straightforward way. Other aspects of method combination (`and`, `progn, ...,` `:most-specific-last`) are merely a matter of writing more complicated methods for method lookup and discrimination.

11. Other behaviours

Other CLOS style behaviours can be similarly implemented. For example, `change-class` (changing the class of an instance) is supported by the MOP functions `primitive-class-of` and `(setf primitive–class-of)`. These functions act on values returned by `primitive-allocate`, the function that gives the lowest level of access to object creation.

EQL specializers (instance-based discrimination) are implemented through adding methods to `compute-discriminating-function` in a very similar manner to the method combination code; class redefinition can be provided by a metaclass that keeps a list of its instances.

Of course, extensions are not limited to copying the default functionality of CLOS. Other extensions are easily available; Telos has been used to implement (among other things) mixin inheritance,

```
;; define a generic function foo that is an instance of class
;;<mc-generic-function>,
;; and takes methods of class <mc-method>
(defgeneric foo (a)
  :class <mc-generic-function>
  :method-class <mc-method>)

(defmethod foo ((a <integer>)) (* a 2))

(defmethod foo :around t ((a <integer>)) (+ (call-next-method) 1))
```

Fig. 13. Method combination.

persistent objects, distributed virtual memory and several types of mathematical hierarchy (e.g. linear algebra, number theory) all as quite modest pieces of code (see [2] for details).

Each new type of behaviour is implemented by the creation of one or more new classes or metaclasses, together with some new methods on the functions as defined in the MOP. Every new metaclass can be defined in isolation of the others, thus allowing the user to pick and choose the required functionality. This allows a great simplification for the programmer, who is able to concentrate on the functionality desired, without the distraction of irrelevant issues.

12. Conclusion

We have a new MOP implemented in CL that is simple enough to require only a relatively small body of code to implement, but flexible enough to provide extensions as powerful as those provided by CLOS. The layered nature allows the user to concentrate on specific aspects of the MOP at any one time – the functionalities of, say, MI and change-class are clearly separated and are therefore much easier to program.

Having the ability to implement, say, MI as user code raises the question of the interaction between multiple implementations. This is not a real problem, as different and incompatible implementations can coexist as separate branches in the class hierarchy. All MI classes of one type will be subclass to a single superclass, so it is perfectly possible for another branch of the hierarchy to have another MI superclass. More problematic would be the likely clashes of names of functions in the two implementations, but the modular namespaces of EuLisp would help with this, just as the package system would do for CL.

The MOP code is held in a single file and there are no complicated bootstrap procedures needed: to use you merely compile and load the code. Some care was taken so that there is never any redefinition of functions in the bootstrap and the MOP: it is possible a compiler may make good use of this fact. Though simply coded, in practice the MOP is inherently efficient. Further, making it more efficient (even in a machine-independent manner) is quite feasible through the structure of the MOP. For example, method lookup could be further enhanced by recomputing the discriminating function whenever a new method is added in the light of the knowledge of the methods available.

There are a few differences with the definition of Telos in EuLisp, mostly derived from the 1-valued nature of EuLisp and the 2-valued nature of CL. Principally, this involves liberal use of `#'` throughout the code, but there are also differences in the manipulation of generic functions, though even this might be removed by the use of Lisp specific code, as in PCL.

There is still scope for improvement in the Telos MOP: one possibility is for class creation and initialization to follow the lead shown by slot accessors by doing more pre-computation at definition time. Rather than have a protocol to be followed as each instance is created, this could be pre-computed and a pre-specialized constructor function could be used. Also, support for class finalization and re-initialization could be made clearer.

Acknowledgement

This research is supported in part by EPSRC grant GR/K27957.

References

1. J. Padget and G. Nuyens. The Programming Language EuLisp (version 0.99). Available by FTP from `ftp.bath.ac.uk,/pub/eulisp/defn-0.99.dvi.z`, 1994.
2. R.R. Kessler. *Lisp and Symbolic Computation* **6** August (1993).
3. G.L. Steele Jr, S.E. Fahlman, R.P. Gabriel, D.A. Moon, D.L. Weinreb, D.G. Bobrow, L.G. DeMichiel, S.E. Keene, G. Kiczales, C. Perdue, K.M. Pitman, R.P. Waters and J.L. White. *Common Lisp: The Language,* (second edition) (Digital Press, Burlington, MA., 1990).
4. G. Kiczales, J. des Rivieres and D. Bobrow. *The Art of the Metaobject Protocol* (MIT Press, Cambridge, MA, 1991).
5. A. Paepcke. *Object Oriented Programming: The CLOS Perspective* (MIT Press, Cambridge, MA, 1993).
6. P. Broadbery *et al.* FEEL – a (mostly) EuLisp implementation. Available by FTP from `ftp.bath.ac.uk,/pub/eulisp/`, 1992.
7. K. Playford. Telosis – Telos in Scheme. Available by FTP from `ftp.bath.ac.uk,/pub/eulisp/Telosis/telosis-1.00f.tar.Z`, 1992.
8. H. Bretthauer, J. Kopp, H.E. Savis and K.J. Playford. Balancing the EuLisp metaobject protocol. *Lisp and Symbolic Computation*, **6** (1993) 119–138.
9. J.A. Padget, G. Nuyens and H. Bretthauer. An overview of EuLisp. *Lisp and Symbolic Computation*, **6** (1993) 9–98.
10. A. Shalit. *Dylan, an Object-oriented Dynamic Language* (Apple Computer Inc., 1992).
11. G.L. Steele Jr, S.E. Fahlman, R.P. Gabriel, D.A. Moon and D.L. Weinreb. *Common Lisp: The Language* (Digital Press, Burlington, MA, 1984).
12. G. Kiczales and L. Rodriguez. Efficient method dispatch in PCL. In *1990 ACM Conference on Lisp and Functional Programming*, ACM Press, June 1990, pp. 99–105.
13. G. Bracha and W. Cook. Mixin-based inheritance. In *ECOOP/OOPSLA ’90 Proceedings*, ACM, October 1990, pp. 303–311.
14. H. Brethhauer, T. Christaller and J. Kopp. Multiple vs. single inheritance in object-oriented programming languages. What do we really want? Technical Report 415, GMD, 1989.
15. R. Ducournau, M. Habib, M. Huchard and M.L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In A. Paepcke (ed.) *Proceedings OOPSLA ’92*, Vol. 27-10 of *ACM SIGPLAN Notices*, ACM, October 1992, pp. 16–24.
16. R. Ducournau, M. Habib, M. Huchard and M.L. Mugnier. Proposal for a monotonic multiple inheritance linearization. In *Proceedings OOPSLA ’94*, Vol. 29-10 of *ACM SIGPLAN Notices*, ACM, October 1994, pp. 164–175.