Common Lisp Object System Specification

3. Metaobject Protocol

This document was written by Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon.

Contributors to this document include Patrick Dussud, Kenneth Kahn, Jim Kempf, Larry Masinter, Mark Stefik, Daniel L. Weinreb, and Jon L White.

CONTENTS

Status of Document

WARNING: This document is uneven in its coverage. Some sections are only sketched out, and others go into significantly more detail. All suggestions for improvements in writing and/or content gratefully accepted.

Terminology

For almost all the generic functions described below, there is only one method defined in CLOS for each generic function. The specializers for these methods are usually one or more of standard-class, standard-method, standard-generic-function, and standard-object. In the case where there is only one method specified by the CLOS kernel, we use the term ”standard method” to refer to this method. In the section listing all the generic functions, we actually provide the full signature for the methods.

Introduction

This document describes the Common Lisp Object System kernel (CLOS kernel). The Common Lisp Object System kernel comprises those classes, methods and generic functions which implement the Common Lisp Object System system described in chapters 1 and 2. The function of the CLOS kernel is to make accessible the "real" CLOS interpeter as an object oriented program written in CLOS. By "real" we mean two things. First, the interpereter we describe is the one that is running; that is, changes made to the generic functions described will effect the operations of the system. Secondly, a "real" interpreter includes not only the direct operations of the language, but provides access to mechanisms to control and support compile and run time optimizations usually hidden from users. We return to this below.

The CLOS kernel provides three levels of accessibility for users. First, it defines a programmatic interface that supports alternative interfaces for the construction of programs. Programmers need not construct and evaluate forms using the interface macros *defmethod*, *defgeneric* and *defclass*. Instead they can invoke directly the generic functions that implement some or all of the behavior associated with these forms.

Secondl, the CLOS kernel provides users a detailed model of the implementation. The kernel classes define the data structures used, and the generic functions and methods define the behaviors of the kernel. In this way it is like the metacircular interpreter for Lisp or Scheme. By understanding those interpreters, users can predict the behavior of the system. But the CLOS kernel is not just a model of the implementation. The generic functions specified are the ones that actually implement the system. The data stuctures are instances of classes in the CLOS kernel, and the operations of the system depend on the types and contents of these instances. Because these generic functions, classes and instances comprise the metacircular interpreter, we refer to them collectively as the metaobjects of CLOS. The *metaobject protocol* refers to the operations (generic function with methods) defined on these metaobjects.

Finally, the CLOS kernel allows for detailed control and extensions to the object system. In Lisp, this type of control is usually achieved by building an interpreter that tests many conditions explicitly, and providing the users with ways of manipulating the conditions. In this distributed, object oriented interpreter, the standard behavior is supported through methods specialized to the standard objects in the system. Specialization of the behavior of the interpreter is achieved by defining classes that are subclasses of those provided by Common Lisp Object System , and specializing some of the generic functions described. By defining methods on the generic function specialized on these new subclasses, programmers can build extensions or alternatives to the basic CLOS framework. This requires programming only the few methods that implement the difference in behavior – a standard way of extending an object-oriented program. The remaining functionality is shared. The defined classes and generic functions thus provided a distributed extensible metacircular interpreter for the Common Lisp Object System.

This chapter is organized as follows. The section *Standard Classes in CLOS* provides a detailed description of the classes that are provided with the CLOS system. Many of these classes were mentioned in chapters 1 and 2, but here we give a full specification of their structure, and related accessors.

After describing the classes, we provide a functionally organized view of the kernel. We call each piece of this kernel a *Protocol*, a set of closely related generic functions that implement a particular feature of the Common Lisp Object SystemẆe group these descriptions of generic functions, providing details in each section of the operation of each generic function. In describing each generic function, we distinguish two different contracts. The first is the one that a caller of the generic function is concerned with. This includes specification of the arguments for the function, the value returned, and expected side effects. A second part of the description focusses on particular methods, usually the standard method. For these methods, we specify other generic functions that are called and more explicit side effects. Finally, there are some invariants to be maintained to ensure proper operation of the kernel. This may require that some generic functions be called whenever certain conditions hold.

The section *Named Class Definition Protocol* specifies the generic function for the expansion of the *defclass* user interface macro, and the generic functions that support the behavior of named standard-classes. The closely related *Slot Parsing Protocol* creates and initializes slot description objects. The underlying anonymous classes are manipulated through the *Class Update Protocol* and the *Slot Inheritance Protocol*.

Allocation and access to instance structure is supported at three levels in the Common Lisp Object System kernel. The *Slot Level* connects the user functions *slot-value*, *slot-boundp* and *slot-makunbound*, and *allocate-instance* with the metaobjects of the CLOS kernel. At the other extreme, the *Index Level* provides a mechanism for the manipulation of indexed blocks of storage. Conecting these two is the *Symbolic Level* that maps slot names into indices. It also provides the appropriate level to provide a hook for optimized storage access.

Generic functions and methods are supported by three protocols. First is the *Named Method and Generic Function Protocol*, which supports the expansion of the user interface macros *defmethod* and *defgeneric*. Secondly, there is the *Generic Function Update Protocol*, part of which has been described earlier (add-method, remove-method and get-method). The other part of this protocol supports maintenance of the method database, and the link from classes to generic functions specialized on them. The *Method Lookup Protocol* supports generation of the code for generic functions, and the computation of effective methods for appropriate combinations of arguments.

Class Organization in the CLOS Kernel

The earlier version of this section has been removed since it was too concerned with design rationale and not concerned enough with the user ramifications of the design decision.

This section should layout the class organization we decide on and then describe how it is supposed to be used. It will need to talk about the how users are expected to define subclasses of standard-class which specialize or override behavior.

For now, the class organization described below is the one (in PCL) where all the classes are subclasses of standard-class and standard-class is a subclass of object. Once we get that locked in, we will have to write this section.

The Classes in the CLOS Kernel

In defining the classes that make up the kernel of CLOS, we find it useful to distinguish three categories of information associated with a class. The distinction between structural, intrinsic and interface is based on the how the information is stored, and consequently how it is retrieved and modified.

Structural Access.

Structural information is explicitly stored in a slot of an instance, it is accessed using the slot-value function. The kernel itself uses slot-value and (setf slot-value) to read and write the information. This means that changes to the the value in a slot will affect the behavior of the kernel. Even so, the kernel often expects certain updating protocols to be followed when the value of a slot is changed. In these cases, the updating protocol insures that parts of the kernel which depend on the value of the slot will behave properly after the change. An example of this is the class-direct-superclasses slot of classes. In order for the kernel methods to behave properly the value of this slot must be changed with update-class generic function rather than (setf slot-value).

Intrinsic Access:

Information in this category is accessed by calling a generic function. The kernel itself also calls this generic function to access the information. This means that a user who has defined her own class as a subclass of a kernel class can specialize these generic functions to affect the behavior of kernel code which is inherited in her own class. Instrinsic accessors fall into one of two categories:

Reading and writing done with a generic function and setf of the generic function respectively. Examples of this kind are:

class-name (setf class-name)

Reading and writing done with different generic functions. Examples of this kind are:

class-direct-subclasses add-direct-subclass remove-direct-subclass

Interface Access

There are certain kinds of information which are maintained by the CLOS kernel but which cannot be modified directly. They include various kinds of derivative information such as back-pointers. Examples include:

class-direct-subclasses class-direct-methods class-direct-generic-functions

The kernel itself does not call these generic functions. Other programs, including parts of a given implementation's programming environment may call these generic functions. A user who has defined her own class as a subclass of a kernel class may specialize these generic functions to affect all callers of the generic function.

The following table shows the slots, instrinsic accessors and interface accessors of the kernel classes. Note that implementations are free to add other slots, instrinsic or interface accessors to any of the classes but these must not conflict with the kernel specification as described here.

Each of the slots, intrinsic accessors, and interface accessors needs more documentation. For now, infer the 'obvious' behavior.

standard-class

Supers: (object)

Slots: direct-superclasses direct-slots direct-class-options class-precedence-list slots class-options inheritance-finalized-p

Intrinsic Accessors: class-name (setf class-name) class-prototype (setf class-prototype) class-direct-subclasses add-direct-subclass remove-direct-subclass

The class prototype is an unitialized instance of the class. It is used when an instance of a class needs to be passed to make method lookup work right. A legal definition of class-prototype would be:

```
(defun class-prototype (class)
  (allocate-instance class))
```

Interface Accessors: class-direct-methods class-direct-generic-functions class-all-initargs class-all-initarg-defaults class-all-slot-initargs class-direct-initargs class-direct-initargs-defaults class-direct-slot-initargs

Initialization Protocol (initargs): Accepts the same keyword arguments as update-class does. The :after initialize-instance method for standard-class simply calls update-class with its arguments. Specifically, :direct-superclasses, :direct-slots and :direct-options.

```
(defmethod initialize-instance :after
           ((class standard-class) &rest keys &allow-others-keys)
  (apply #'update-class class keys))
```

forward-referenced-class

Supers: (standard-class)

built-in-class

Supers: (standard-class)

structure-class

Supers: (standard-class)

funcallable-standard-class

Supers: (standard-class)

standard-slot-description

Supers: (object)

Slots: name initform initfunction initarg-names accessors readers type

Intrinsic Accessors:

Interface Accessors: slotd-allocation

Initialization Protocol (initargs): :name :initform :initfunction :initarg :accessor :reader

The first three of these are slot filling initargs. The last three are processed by the :after initialize-instance method. They can be specified more than once, all the values will be collected to fill the appropriate slot. See the Slot Parsing Protocol Section.

standard-class-slot-description

Supers: (standard-slot-description)

Slots: value

Intrinsic Accessors:

Interface Accessors: slotd-allocation

Initialization Protocol: The :after initialize-instance-method calls the initfunction to set the value of the value slot. The value slot is where the value of the slot for the entire class is stored.

standard-method

Supers: (object)

Slots: specializers qualifiers function

Intrinsic Accessors: method-generic-function method-lambda-list

Interface Accessors: documentation

Initialization Protocol (initargs): :specializers :qualifiers :function :lambda-list :documentation

standard-accessor-method

Supers: (standard-method)

Slots: slot-name

Initialization Protocol (initargs): :slot-name

standard-reader-method

Supers: (standard-accessor-method)

standard-writer-method

Supers: (standard-accessor-method)

standard-generic-function

Metaclass: funcallable-standard-class

Supers: (function object)

Slots: methods method-class lambda-list argument-precedence-order method-combination-type method-combination-arguments

Intrinsic Accessors: generic-function-name generic-function-declarations add-method remove-method get-method

Initialization Protocol (initargs): :declare :lambda-list :argument-precedence-order :method-combination :method-class :documentation

The :after initialize-instance method calls update-generic-function to store these values. See the Updating a Generic Function Protocol.

The following is a DAG for the classes in CLOS, as described above. Implementations are free to interpolate additional classes, provided that the order of inheritance of the classes specified is the same.

This really should be a picture.

```
T
    standard-object
        standard-generic-function (also has function as superclass)
        standard-method
            standard-accessor-method
                standard-reader-method
                standard-writer-method
        standard-slot-description
                standard-class-slot-description
                standard-structure-slot-description
        standard-class
            built-in-class
            structure-class
            forward-referenced-class
            funcallable-standard-class
```

The following is the DAG of classes that correspond to the Common Lisp data types. The classes that have multiple supers are indicated with a *; the most specific super is the one that comes first (highest) in the figure below.

This really should be a picture too.

```
T
    function
    number
        rational
            ratio
            integer
        complex
    character
    array
        vector*
            string
            bit-vector
    symbol
        null*
    sequence
        vector*
            string
            bit-vector
```

```
list
    cons
    null*
```

The Named Class Definition Protocol

Here there should be a little summary which describes the basic function of the named class definition protocol. It should cover all the relevant generic functions:

```
expand-defclass
add-named-class
class-for-redefinition
```

expand-defclass

In order to allow metaclasses to effect the processing of defclass forms form, the expansion of defclass forms is controlled by a the expand-defclass generic function. The expand-defclass generic function is called by the defclass macro to compute the expansion of the macro. The syntactic processing and checking done by the defclass macro is minimal. It only parses the arguments to defclass into name, superclasses, slot specifications and options. Furthermore, it scans the options to see if the :metaclass option was specified. All other syntactic checking of the arguments to defclass is done later. If the :metaclass option was not specified it defaults to standard-class. The metaclass option is used to compute the first argument to expand-defclass. The first argument to expand-defclass is the prototype instance of the metaclass. The remaining arguments are as described above. The defclass macro behaves as if it was defined by:

```
(defmacro defclass (name superclasses slots &rest options &environment env)
  (let ((metaclass (or (dolist (option options)
                         (when (and (listp option)
                                    (eq (car option) ':metaclass))
                           (return (cadr option))))
                       'standard-class)))
    (expand-defclass (class-prototype (symbol-class metclass))
                     name
                     superclasses
                     slots
                     options
     env)))
```

expand-defclass *prototype-instance name superclasses slots options environment* [*Generic*

Function]

The purpose of the expand-defclass generic function is to compute the expansion of the defclass form. The expand-defclass generic function is supplied with the information about the metaclass option specified in the defclass form, and receives the remaining arguments to the defclass form. expand-defclass also receives the environment the defclass form appeared in as an argument.

Typically, expand-defclass returns a form which includes a call to add-named-class, but this is not required.

expand-defclass `(prototype-instance standard-class) name direct-superclasses direct-slots options environment` [*Primary Method*]

The standard method for expand-defclass expands the defclass form into a form which includes a call to the generic function add-named-class. This method also normalizes the slot specifications which appeared in the defclass form.

Normalization of the slot specifications converts the three kinds of slot specifications which can appear in defclass to a single 'pure plist' form. This is done as follows:

If the slot-specification is a symbol: <slot-name> the normalized slot specification is: (:name <slot-name>)

If the slot-specification is a list like: (<slot-name> . <slot-options-and-values>) the normalized-slot-specification looks like: (:name <slot-name> . <slot-options-and-values>)

During normalization, the standard method on expand-defclass adds the :initform-function slot option to the slot options that were specified in the defclass form. The :initform-function slot option is used to pass in the function of no arguments which can be called to evaluate the :initform in the proper lexical environment.

The expanded form may include other implementation-dependent code, but it will always include a call to add-named-class. The form which calls add-named-class will behave as if it was:

Note that the issue of compile time environments is being finessed a bit here and throughout the document. We had decided that the environment argument would contain all the magic and that would be all we had to specify, but its not clear that will really be enough.

```
`(progn
   (eval-when (compile load eval)
     (add-named-class
       (class-prototype
         (class-named
           ',(class-name
               (class-of prototype-instance))))
       ',name
```

```
                  ',direct-superclasses
                  ',(mapcar #'normalize-slot-specification direct-slots)
                  ',options
                  ',environment)))
```

Example of specializing expand-defclass

Suppose one wanted to have a metaclass which treated classes defined with defclass differently than classes defined with add-named-class. This metaclass might want to supplement the class options with a special marker which said that the call to add-named-class was the result of a defclass expansion. The following code would have that effect:

```
(defmethod expand-defclass ((prototype-class my-class)
                            name superclasses slots options environment)
  (call-next-method prototype-class
                    name
                    superclasses
                    slots
                    (cons '(defclass t) options)
                    environment))
```

add-named-class

add-named-class *prototype-instance name direct-superclasses direct-slots options environment* [*Generic Function*]

I think this would be more useful if it was converted to take keyword arguments like update-class does.

This generic function is the programmatic interface for defining named classes. The prototype-instance argument should be the class-prototype of the class of the class being defined. The direct-superclasses argument can be a list of either symbols (class names) or class objects. The slots argument should be a list of slot specifications as they would appear in a defclass form. The options should be a list of the class options as they would appear in a defclass form. The environment is the environment in which the definition should take place, this is used to distinguish between compiler and non compiler environments. If there is no class with the given name, a new class is created. If there is already a class with the given name the results depends on the metaclass.

add-named-class `(prototype-instance standard-class) name direct-superclasses direct-slots options environment` [*Primary Method*]

The standard method on add-named-class implements the behavior described for defclass in chapter 1.

The first step performed by this method is to determine the class object which will be used for the new definition. If there is no existing class with the given name, this method creates a class of the specified class. If there is already a class with the given name, this method calls class-for-redefinition to get the class object to use.

Once the class object has been determined, the slot-specifications are parsed to turn them into slot description objects. For details on how this is done see the slot parsing protocol.

Then update-class is called to store the specified superclasses, slot-descriptions and options in the class. For details on the operation of update-class see the class updating protocol.

Once the class has been updated to reflect the specified superclasses slots and options, it is stored in the symbol-class association table, and has its own class-name attribute updated.

class-for-redefinition

...

class-for-redefinition *prototype-instance old-class* [*Generic Function*]

class-for-redefinition is called by the standard method on add-named-class when there is already a class with the given name. The class-for-redefinition generic function is expected to return the class object which should be used for the new definition. For standard-class, the class object used is the old class object since standard class supports the notion of updating old instances to reflect new definitions of the class. Other metaclasses might not support this notion, they might want new class definitions to use a new class object or even signal an error if an attempt is made to redefine a class.

class-for-redefinition `(prototype-instance standard-class) (old-class standard-class)` [*Primary Method*]

This method on standard class first calls make-instances-obsolete on the old-class argument and then returns the old-class argument.

class-for-redefinition `(prototype-instance t) (old-class structure-class)` [*Primary Method*]

what to say about this?

class-for-redefinition `(prototype-instance t) (old-class built-in-class)` [*Primary Method*]

what to say about this?

Example Specialization of class-for-redefinition

Sometimes, a user wants to declare that certain classes, when they are defined, should have a particular metaclass. This can be the case when someone takes a program which is already written and wants to compile and load it using an optimizing metaclass. The user explicitly does not want to have to edit the original defclass forms to specify the metaclass option; the user would like to use a simple macro to make this declaration. Something like:

```
(defclass-optimized A)
```

Given that the optimizing metaclass already exists and is called optimized-class, this can be done using class-for-redefinition. The following code will work.

```
(defclass forward-referenced-optimized-class (forward-referenced-class)
    ())

(defmethod class-for-redefinition
           ((existing-class forward-referenced-optimized-class)
            (proposed-new-class standard-class)
            name
            supers
            slots
            options)
  (change-class existing-class
                (class-prototype (class-named 'optimized-class)))
  existing-class)

(defmacro defclass-optimized (class-name)
  `(add-named-class
     (class-prototype (class-named 'forward-referenced-optimized-class))
     ',class-name
     ()
     ()
     ()
     ()))
```

The Slot Parsing Protocol

Standard classes store two distinct lists of slots. The first is the list of slots defined in the class proper. The second is the total list of slots the class has, this includes inherited and locally defined slots. Both of these are stored as lists of slot description objects.

As part of defining a class, the normalized slot specifications passed to add-named-class must be converted to a list of slot description objects. This conversion process is done using the slot parsing protocol.

The slot parsing protocol is quite simple. It only contains only two steps: a call to the generic function slot-description-class and a call to make-instance.

Normalized slot-specifications are always parsed with respect to the class they specify a slot for. This allows the class the slot description is being produced for to control the class of the slot description object itself. This slot-description-class generic function is called with the class and the normalized slot specification to determine the class of slot description which should be produced for the class.

Once the appropriate class for the slot description has been determined, the actual parsing is achieved by applying make-instance to the class and the normalized slot specification.

This means that the legal set of slot option names for a given class of slot-description is the same as the legal set of initarg names for that class. See lambda-list-congruence rules.

slot-description-class

The slot-description-class generic function..

SLOT-DESCRIPTION-CLASS ((class standard-class) normalized-slot-specification)

slot-description-class

make-instance of slot-descriptions

The standard method ...

MAKE-INSTANCE ((class standard-slot-description))

This fills in the slots of the class standard-slot-description with the appropriate values in the initargs

MAKE-INSTANCE ((class standard-class-slot-description))

This fills in the slots and also fills in the class-value slot.

The Class Update Protocol

This protocol supports the invariants that must be maintained between local information in a class, such as direct-slots and direct-superclassess and the derived information of the class from its position on the class lattice. It consists of four parts: entry, propagation, local updating and finalization.

The entry part of the class update protocol is implemented by the generic functions update-class, default-class-supers, legal-class-option-p, and compatible-super-metaclass-p. A call to update-class is the only guaranteed consistent way to update the slots of superclasses of a standard class. The keyword arguments of update-class allow specification of new direct superclasses, new direct slots and new options. The generic function default-class-supers is used to compute the minimum default superclasses for a standard class. Some error checking is done using the legal-class-option-p and compatible-super-metaclass-p.

The change propagation part of this protocol is implemented by the generic functions propagate-class-update. This generic function walks as a depth first tree the changed class and all its direct subclasses recursively, notifying each class it reaches that a change has occurred.

The local updating of classes in the lattice at the time of a change is the contract of update-class-locally. The minimum contract of this generic function is that it will store information in the class that has been explicitly changed, and will mark as needing updating classes that have had some change made in the lattice at or above them.

The inheritance finalization part of the class update protocol is implemented by the generic function finalize-inheritance. It allows implementations to update any precomputed caches used for instance allocation and access. It must be called sometime after update-class has returned. It must be called before an instance is made of any updated class, or before an obsolete instance is updated to the newly defined structure.

Update Entry

update-class

The generic function update-class is used to update existing classes. It also is used to initialize a class that has just been created. It deals with the classes as anonymous objects. update-class is the only interface to change the direct-slots, direct-superclasses, or class-options of a class. It is undefined what happens if these slots of a class are changed in any other way.

The standard method of add-named-class calls update-class. An :after method of initialize-instance on standard-class calls update-class. Specialized methods of add-method and remove-method call update-class when a new method is added on the generic function initialize-instance.

The value returned by the generic function is the updated class.

The following defines the argument list of the generic function:

```
update-class (class
              &rest key-arguments
              &key (direct-superclasses () new-supers-p)
                   (direct-slots () new-slots-p)
                   (options () new-options-p)
                   (init-method-keys () new-init-method-keys-p)
              &allow-other-keys)
```

update-class `(class standard-class) &rest key-arguments &key (direct-superclasses () new-supers-p) (direct-slots () new-slots-p) (options () new-options-p) (init-method-keys () new-init-method-keys-p)` [*Primary Method*]

In this method on standard-class, *class* is the class to be updated; *direct-superclasses* is a list of class objects (no symbols); *direct-slots* is a list of slot-description objects; *options* is a list of class options; *init-method-keys* is list of the keyword arguments accepted by all the initialize-instance methods on this class.

If direct-superclasses is given, the value actually used to update the class is the value of: (default-class-supers class supplied-supers). This call to default-class-supers implements the feature that standard-classes have the class named object as their default superclass if () is provided as the superclasses list (say by the defclass form).

For each direct superclass, the generic function check-super-metaclass-compatibility is called to check if the given superclass has a metaclass compatible with the class being defined. It is expected that check-super-metaclass-compatibility will signal an error if there is any problem.

For each of the options provided, the generic function legal-class-option-p is called to check the legality of each option given. If legal-class-option-p returns NIL, then this method on update-class signals an error.

Since direct-slots are slot objects, no further error checking is required for them.

After legality checking, if direct-supers have been provided, the pointers from old and new direct-superclasses to the updated class are changed using the generic functions add-direct-subclass and remove-direct-subclass. The newly provided direct superclasses are stored in the slot *direct-superclasses*.

If new-slots have been provided, this method on update-class maps through the old and newly provided slot-description objects to determine reader and writer methods, removing no longer required reader and writer methods, and adds newly required methods using the generic functions. It does this by calling the generic functions remove-reader-method, remove-writer-method, add-reader-method, add-writer-method. It then stores the new direct-slots in the slot in the class called *direct-slots*.

If options have been provided, the standard method on update-class stores the new options in the slot *options* in the class.

To inform all subclasses of the updated class of changes that might affect them, a call is made to the generic function propagate-class-update, passing it all the arguments to update-class.

default-class-supers

This generic function is called to determine the direct-superclasses for a class. It is called by the standard method for update-class when direct-supers have been supplied. It receives as arguments the class for whom the superclassess are intended, and the supplied superclasses. It returns a list of classes.

default-class-supers *class supplied-superclasses* [*Generic Function*]

default-class-supers `(class standard-class) supplied-supers` [*Primary Method*]

If *supplied-supers* is NIL, or the list just containing the class named T, then this method returns a list containing the class named object. Otherwise it returns its argument *supplied-supers*.

default-class-supers `(class structure-class) supplied-supers` [*Primary Method*]

If *supplied-supers* is NIL then this method returns a list containing the class named T. Otherwise it returns its argument *supplied-supers*.

default-class-supers `(class funcallable-standard-class) supplied-supers)` [*Primary Method*]

If *supplied-supers* is NIL then this method returns a list containing the class named function and the class named object. Otherwise it returns its argument *supplied-supers*.

Example of specializing default-class-supers

Suppose we have loops-class as a subclass of standard-class, and we want all instances of loops-class to have the class named loops-object as their default super.

```
(defclass loops-class (standard-class) ())

;;;
;;; Implement the rule that where standard-class would have made
```

```
;;; the superclasses be a list of the class object, we return a
;;; list of the class loops-object.
;;;
(defmethod default-class-supers ((class loops-class) supplied-supers)
  (let ((default (call-next-method)))
    (if (and (null (cdr default))
             (eq (car default) (class-named 'object)))
        (list (class-named 'loops-object))
        default)))
```

check-super-metaclass-compatibility

The generic function check-super-metaclass-compatibility tests whether the proposed superclass has a metaclass compatible with being the a direct-superclass of the class being defined. It should signal an error if there is a compatibility problem.

check-super-metaclass-compatibility *class proposed-superclass* [*Generic Function*]

check-super-metaclass-compatibility `(class t) (proposed-superclass t)` [*Primary Method*]

The default method signals an error unless the metaclasses are EQ.

check-super-metaclass-compatibility `(class standard-class) (proposed-superclass forward-referenced-class)` [*Primary Method*]

Standard classes support having superclasses that are not yet defined. These superclasses are represented by instances of forward-referenced-class. Hence, this method returns T.

Question: can check-super-metaclass-compatibility have a side effect on any class – that is make things compatible by changing the metaclass of one or more classes. Should this be a predicate, like legal-class-option-p, and have the error signalled in update-class standard method. YES and NO respectively

legal-class-option-p

This generic function is used to check the legality of class options provided to update-class. It uses or method combination type, and returns true if one of the applicable methods believes that the option is legal. This generic function is called by the standard method on update-class, which signals an error if legal-class-option-p returns false for an option.

legal-class-option-p *class option-option* [*Generic Function*]

legal-class-option-p `(class standard-class) option` [*Primary Method*]

This method checks for the allowed options described in chapter 1.

Update Propagation

propagate-class-update

The generic function propagate-class-update guarantess to visit all the subclasses (direct or indirect) of the changed-class at least once. It receives as arguments all the information passed to update-class. It also receives *class*, the class that is to notice the change, and *changed-class*, the class that was the orginal argument to update-class. It is called from the standard method of update-class.

The value of propagate-class-update is not defined.

propagate-class-update *class changed-class* `&restkey-arguments` [*Generic Function*]

propagate-class-update `(class standard-class) changed-class &rest key-arguments` [*Primary Method*]

The standard method on propagate-class-update calls the generic function update-class-locally on the given class. It passes it all the arguments it received.

The standard method on propagate-class-update then calls propagate-class-update recursively on each of its direct-subclasses in order. This has the effect of making a depth first walk of the subclasses of a class, possibly visiting some subclasses more than once.

```
(defmethod propagate-class-update ((class standard-class) changed-class &rest
key-args)
  (let ((new-key-args
          (append (apply #'update-class-locally class changed-class key-args)
                  key-args)))
   (dolist (sub (slot-value class 'direct-subclasses))
     (apply #'propagate-class-update sub changed-class new-key-args))))
```

Fix update-class-locally to say it returns nil, fix other places to talk about the value it can return.

Local Class Updating

update-class-locally

This generic function is responsible for ensuring that appropriate changes will be made if a class has been changed either directly (by update-class) or indirectly, by being a subclass of a directly changed class.

The arguments passed to update-class-locally are the same as those that were passed to propagate-class-update. The named arguments are the same as those passed originally to update-class. This generic function is called from the standard method for propagate-class-update.

The value of update-class-locally is used by propagate-class-update in its recursive call, this allows update class locally to pass information down to the subclasses that will also be updated.

```
update-class-locally
  (class changed-class
  &rest key-arguments
  &key (direct-superclasses () new-supers-p)
        (direct-slots () new-slots-p)
        (options () new-options-p)
        initialize-instance-changed-p)
```

update-class-locally `(class standard-class) changed-class &rest key-arguments &key :direct-superclasses :direct-slots :options :initialize-instance-changed-p` [*Primary Method*]

This method on update-class-locally sets to NIL the slot *inheritance-finalized-p*. This slot is used as a flag to determine if certain methods should call finalize-inheritance. The standard methods on make-instance and update-instance-structure check this flag to determine if they should call finalize-inheritance.

What happens next in this method in update-class-locally is dependent on whether the class *class* has instances. This is determined in this method by a call to the generic function class-has-instances-p. If the class does not have instances, not further updating is done in this method. This is postponed until finalize-inheritance is called.

If *class* has no instances, this method on update-class-locally returns immediately. What follows is what happens if *class* does have instances.

If direct-superclasses are provided, this method sets the value of the slot *class-precedence-list* to the result obtained by calling the generic function compute-class-precedence-list.

If direct-superclasses or direct-slots are provided, this method on update-class-locally sets the value of the slot *slots* in the class to the result obtained by calling the generic function collect-slotds.

If the result returned by collect-slotds specifies a different list of instance slots, then the generic function make-instances-obsolete is called on this class. It is because this must be done immediately that class-precedence-list and slots must be updated if the class has instances.

class-has-instances-p

This generic function is used to tell if there are any existing instances of a given class. Implementations are allowed to be conservative and return T if this class has ever had an instance created.

This generic function is called by the standard method on update-class-locally.

class-has-instances-p *class* [*Generic Function*]

class-has-instances-p `(class standard-class)` [*Primary Method*]

This is the only method that must exist in the standard. It must return T if there are current instances of the class and/or there are instances of an obsolete version this class that may be updated to the current instance structure. It may be conservative. It may even return T all the time. The only penalty will be possible additional work in updating classes.

compute-class-precedence-list

This generic function computes the class precedence list of a class as described in Chapter 1. The value is a list of class objects in order.

compute-class-precedence-list *class* [*Generic Function*]

compute-class-precedence-list `(class standard-class)` [*Primary Method*]

The standard method on class-precedence-list treats instances of forward-referenced-class as classes with no superclasses but the class named T.

Finalizing Class Inheritance

finalize-inheritance

The generic function finalize-inheritance is used to optimize the creation of instances by precomputing information based on inherited. It is called by the methods on standard-class for make-instance and update-instance-structure if a flag stored in the class slot *inheritance-finalized-p* is NIL. It may also be called by the user.

Users with special optimization requirements can write methods on finalize-inheritance to precompute their own information based on inherited information, and be assured they will be called when ever changes occur.

The value of finalize-inheritance is undefined.

finalize-inheritance *class* [*Generic Function*]

finalize-inheritance `(class standard-class))` [*Primary Method*]

This method sets the value of the slot *class-precedence-list* to the result obtained by calling the generic function class-precedence-list.

This method warns if any of the superclasses are instances of forward-referenced-class.

It sets the value of the slot *slots* in the class to the result obtained by calling the generic function collect-slotds.

This method sets the flag in the slot *inheritance-finalized-p* to T.

Adding and Removing Accessor Methods

As part of the processing of the class option, readers and accessors for particular slots may have to be added. If there was a previous definition of the class being changed, some readers and writers may need to be removed. The following generic functions are used to implement this facility. They are called from the standard method for update-class.

For each of these generic function, the *class* argument is the class on which the slot is to be found. The slotd is the slot-desciption object. The caller of these generic functions, the standard method on update-class, has these slot-description objects in hand at the time of the call. The *generic-function-name* is a symbol. All the methods on these generic functions call ensure-generic-function with the name and constructed lambda-list to get the generic fucntion to add the method too.

The value of each of these generic functions is the newly added (removed) method, or NIL if it was unsuccessful..

add-reader-method

This generic function adds a reader method for the slot in *class* described by *slotd* to the generic function named by the symbol *generic-function-name*. It returns the method object added.

add-reader-method *class slotd generic-function-name* [*Generic Function*]

add-reader-method `(class standard-class) slotd generic-function-name` [*Primary Method*]

This method ensures that generic-function-name is the name of an appropriate generic function by calling ensure-generic-function. It then creates a method object that is an instance of the class standard-reader-method. The effect of this standard method on add-reader-method is as though it evaluated:

‘(defmethod ,generic-function-name ((c ,class)) (slot-value c ’,(slot-value slotd ’name)))

Implementations are free to provide special mechanisms for these readers.

add-writer-method

This generic function adds a writer method for the slot in *class* described by *slotd* to the generic function named by the symbol *generic-function-name*. It returns the method object added.

add-writer-method *class slotd generic-function-name* [*Generic Function*]

add-writer-method `(class standard-class) slotd generic-function-name` [*Primary Method*]

This method ensures that generic-function-name is the name of an appropriate generic function by calling ensure-generic-function. It then creates a method object that is an instance of the class standard-writer-method. The effect of this method is as though it evaluated:

‘(defmethod (setf ,generic-function-name) ((c ,class)) (new-value) (setf (slot-value c ’,(slot-value slotd ’name)) new-value))

Implementations are free to provide special mechanisms for these writers.

remove-reader-method

This generic function removes a reader method for the slot in *class* described by *slotd* from the generic function named by the symbol *generic-function-name*. It returns the method object removed, or NIL if none was found.

remove-reader-method *class slotd generic-function-name* [*Generic Function*]

remove-reader-method `(class standard-class) slotd generic-function-name` [*Primary Method*]

This method uses get-method to locate the reader method on the named generic function. It then removes the method located. If there is no such generic function or there is no such method on the generic function, this method on remove-reader-method returns NIL. Otherwise it returns the removed method.

remove-writer-method

This generic function adds a writer method for the slot in *class* described by *slotd* from the generic function named by the symbol *generic-function-name*. It returns the method object removed, or NIL if none was removed.

remove-writer-method *class slotd generic-function-name* [*Generic Function*]

remove-writer-method `(class standard-class) slotd generic-function-name` [*Primary Method*]

This method uses get-method to locate the writer method on the named generic function. It then removes that method. If there is no such generic function or there is no such method on the generic function, it returns NIL. Otherwise it returns the removed method.

The Slot Inheritance Protocol

The total set of slots for any given class is computed by combining the locally defined slots for the class and all of its superclasses. For standard classes, this combination proceeds according to the rules described in chapter 1. This combination is implemented by the slot inheritance protocol.

The slot-inheritance protocol is a two level protocol.

collect slotds collects up all the slotds and then calls compute-effective-slotd to condense them into one slotd. Need to make some statement about the ordering constraints on what collect slotds will do. Perhaps there aren't any.

The computation of the set of slots and their descriptions are controlled at two levels. For each slot, the set of slots with that name, ordered by class precedence list (most specific first), is used to compute an effective slot description for the slot locally, using

compute-effective-slotd (class slotds)

The standard method for this generic function supports the inheritance of slot options that is described in Chapter 1. It returns a slot-description object that can be used locally.

The generic function collect-slotds (class local-slots cpl) collects an ordered list of effective slot descriptions for this class. It takes the local-slots as an argument, and recursively builds up the list of all slots that need to be in this class. It calls compute-effective-slotd to combine multiple definitions of a single slot found in classes on the class precedence list.

Example of Specializing compute-effective-slot-description

Suppose a user wanted to define a new metaclass which implemented a different rule for the inheritance of the :type slot option. This new rule might want to say that a subclass must specify a type which is at least as specific as the type specified by any of the superclasses. If none of the superclasses specified a type, the local class may either not specify a type at all or may specify any type it likes.

```
(defmethod compute-effective-slot-description ((class my-class)
                                               slot-descriptions)
  (when (car slot-descriptions)
    (when (slot-boundp (car slot-descriptions) 'type)
      ;; The class has a local specification for this slot and
      ;; the :type option is specified in that specification.
      ;; Make sure the specified type is at least as specific
      ;; as all the other types specified.
      (let ((local-type (slot-value (car slot-descriptions) 'type)))
        (dolist (super-slot (cdr slot-descriptions))
          (when (slot-boundp super-slot 'type)
```

```
        (unless (subtypep slotd-type (slot-value super-slot 'type))
          (error "~S is not a subtype of ~S"
                 local-type
                 (slot-value super-slot 'type))))))))
  (call-next-method))
```

The Instance Structure Protocol

This section does not yet include descriptions of any setf functions. While reading it, you should assume that the obvious functions have setf functions with the obvious meanings.

It sure would be real nice if there were some abbreviated way to discuss setf functions. Putting them in line all the time is real painful and interrupts the flow of the text. I don't think its possible though.

Metaclasses determine the structure of their meta-instances. This includes allocating the memory for and managing the layout of the instance. This is handled by the instance structure protocol.

The instance structure protocol has several levels. At the lowest level, it permits the allocation and access to two kinds of instances: standard-class and structure-class. At this level, instances appear to be be vector-like blocks of memory with the additional property that the type system (including class-of) can determine their class. At this level, positive integers called indexes can be used to access the elements of the instance. For this reason this is called the Indexed Level of instance structure. At this level there is also support for implementation specific mechanisms for controlling the packing and garbage collection parameters of this access.

At the next level, there is a mapping from symbolic descriptions of the elements of an instance to the information about where the slot is stored. This level is called the Symbolic Level of index structure. For standard class this mapping is from the symbol which names a slot to the actual either index in the instance where the slot is stored or a specification that the slot is a :class slot.

At the highest level, the instance appears to contain a set of slots as described in chapter 1. This is called the Slot Level of index structure. Since only metaclass programmers make use of the levels below the slot level, it is often useful to think of this as the user level.

User defined metaclasses can define new elements of instances at any of the three levels.

The rest of this section describes these three levels and describes how the standard-class and structure-class metaclasses use them.

There is a bit of design rationale I would like to put in here, but I can't figure out how. Specifically, the reason we specify just the two meta-instance types and don't specify a general way to allocate new kinds of instances is that we don't want to have to specify a more powerful portable way to extend an implementation's type system. Also, it turns out that this provides as much power as a seemingly more general portable mechanism would provide. This is because all the more general schemes I have been able to come up with turn out to be essentially equivalent to this.

Instance Allocation

At the indexed level of the protocol, an instance of a certain type and size is allocated by calling the appropriate allocation function. At the symbolic level, information about what will be stored in the instance – the slots – is used to determine the appropriate size for the instance. At the slot or user level, the metaclass determines the kind of instance which is allocated.

Index Level Instance Allocation

At the lowest level, there are two functions used for allocating instances. These are allocate-standard-instance and allocate-structure-instance. Each of these takes as arguments an instance size. The size indicates the size of the elements in index numbers (see the low level instance access section). The optional argument storage-information is an implementation-specific value which can be used to specify packing and garbage collection information for the instance.

allocate-standard-instance *class size* `&optional` *storage-information* [*Function*]

allocate-structure-instance *class size* `&optional` *storage-information* [*Function*]

The value returned is the newly allocated instance.

standard-instance-p *thing* [*Function*]

Returns true if thing is a standard instance (was created by a call to allocate-standard-instance).

structure-instance-p *thing* [*Function*]

Returns true if thing is a structure instance (was created by a call to allocate-structure-instance).

Implementations are free to make standard instances and structure instances be the same but they must do so consistently. In other words if any value returned by allocate-structure-instance is standard-instance-p they must all be and vice versa.

Symbolic Level Instance Allocation

At the symbolic level, information about what is to be stored in the instances of the class is used to determine the appropriate size for the instance. This level acts as a translation between the information stored at the slot level (about what slots the instance has) and the indexed level.

compute-instance-size

Methods on compute-instance-size take care of this conversion. The kernel methods on compute-instance-size return a size greater than or equal to the number of slots that must be stored in the instance.

compute-instance-size `(class standard-class)` [*Primary Method*]

This method returns a number greater than or equal to the number of :instance slots of the class.

compute-instance-size `(class structure-class)` [*Primary Method*]

This method returns a number greater than or equal to the number of slots of the class.

Slot Level Instance Allocation

...

allocate-instance

At the slot level, instances are allocated using the generic function allocate-instance. Methods on allocate-instance take care of calling the appropriate index level instance-allocation function. These methods determine the appropriate size for the instance by calling the compute-instance-size generic function.

allocate-instance `(class standard-class) &key &allow-other-keys` [*Primary Method*]

This method allocates instances using the allocate-standard-instance function. The size argument to the allocate-standard-instance function is determined by calling the compute-instance-size generic-function with the class as its only argument. Whether the packing-information argument to allocate-standard-instance is supplied is implementation dependent.

allocate-instance `(class structure-class) &key &allow-other-keys` [*Primary Method*]

This method allocates instances using the allocate-standard-instance function. The size argument to the allocate-standard-instance function is determined by calling the compute-instance-size generic-function with the class as its only argument. Whether the packing-information argument to allocate-standard-instance is supplied is implementation dependent.

Instance Access

The instance access part of the instance structure protocol operates at the same three levels as the instance allocation part does.

Index Level Instance Access

At the index level, instances are accessed as if they were vector-like blocks of memory. They are accessed with functions specific to the kind of instance being accessed.

standard-instance-ref *instance index &optional storage-info* [*Function*]

Takes a standard-instance and returns the element stored at index number index. If the instance argument is not a standard-instance the results are undefined. If the instance argument is smaller than the index specified the results are undefined.

Specific implementations may extend the meaning of the storage-info argument to provide mechanisms for data packing and garbage collection control.

structure-instance-ref *instance index &optional storage-info* [*Function*]

Takes a structure-instance and returns the element stored at index number index. If the instance argument is not a structure-instance the results are undefined. If the instance argument is smaller than the index specified the results are undefined.

Specific implementations may extend the meaning of the storage-info argument to provide mechanisms for data packing and garbage collection control.

standard-instance-boundp *instance index &optional storage-info* [*Function*]

If there is a value stored in index number index of the standard-instance instance this returns true. If there is no value stored there returns false. If the instance argument is not a standard-instance the results are undefined. If the instance argument is smaller than the index specified the results are undefined.

Specific implementations may extend the meaning of the storage-info argument to provide mechanisms for data packing and garbage collection control.

standard-instance-makunbound *instance index &optional storage-info* [*Function*]

Causes there to be no value stored in element number index of the standard instance standard-instance. If the instance argument is not a standard-instance the results are undefined. If the instance argument is smaller than the index specified the results are undefined.

Specific implementations may extend the meaning of the storage-info argument to provide mechanisms for data packing and garbage collection control.

Symbolic Level Instance Access

The symbolic storage layer provides indirection from symbolic descriptions of an element of an instance to the index number at which that element is stored. Standard-class and structure-class use this layer to map from slot names to the index number at which the slot is stored.

The symbolic storage layer is designed to provide an interface to this symbolic mapping which can be used by metaclass programmers to take advantage of implementation specific optimization mechanisms.

index-in-instance

The generic function index-in-instance is used to convert a symbolic description of an element of an instance to a specification of where that slot is stored. The kernel methods for index in instance support symbolic descriptions which are symbols, specifically slot-names. If the slot is a :instance allocated slot these methods return the index number at which that element is stored. If the slot is a :class allocated slot these methods return the slot description representing the slot. User defined methods can extend this mechanism to use other kinds of symbolic descriptions.

index-in-instance `(class standard-class) instance description` [*Primary Method*]

If description is a symbol which is the name of a :instance slot in the class, returns the index number at which that slot is stored. If description is a symbol which is the name of a :class slot in the class returns the slot description which represents that slot. Otherwise this returns nil. For more information about exactly how the index number is computed see the section on computing slot inheritance.

index-in-instance `(class structure-class) instance description` [*Primary Method*]

If description is a symbol which is the name of a slot in the class, returns the index number at which that slot is stored. Otherwise this returns nil. For more information about exactly how the index number is computed see the section computing slot inheritance.

Optimized Symbolic Level Instance Access

The standard-instance-access function provides the basic interface to the implementation-specific standard instance access optimization. This function is just a simple combination of more primitive instance access mechanisms, but it is designed to be the place where the implementation provides its optimization. In most implementations calls to this function will be replaced by just a few instructions. Specific implementations of CLOS are expected to implement their instance access optimization by optimizing these functions and then using the Instance Access Optimization Protocol to convert instance accesses to calls to this function.

The effective definition of standard-instance-access is:

```
(defun standard-instance-access
       (instance description trap not-bound-function missing-function)
  (let* ((class (class-of instance))
         (index (index-in-instance class description)))
    (cond ((null index)
           (funcall missing-function instance description))
  ((not (numberp index))
   (slot-value index 'value))
          ((null (standard-instance-boundp instance index))
           (funcall not-bound-function instance description))
          (t
```

```
            (standard-instance-ref instance index)))))
```

In order to support the optimization there is a contract between standard-instance-access and the kernel methods which provide the class updating protocol. Specifically, standard-instance-access is allowed to call index-in-instance at finalize-inheritance time. This means that any user defined methods on index-in-instance which might affect uses of standard-instance-access must guarantee that their value only changes when a class update happens.

In order to allow flexible use of the optimization standard-instance-access provides, there is a mechanism for deoptimizing calls to standard-instance-access for a particular class. This mechanism causes all the calls to standard-instance-access for a particular class to call the trap function instead. The trap function received the instance and the description as its arguments.

deoptimize-standard-instance-access *class* [*Function*]

...

There is a similar set of functions for structure instance. But, structure-instance-access doesn't support deoptimization, and structure-instance-access is free to call index-in-instance at load time. This reflects the different performance optimization structure-class provides.

Slot Level Instance Access

At the highest level, instance access is in terms of slots. The basic functions for accessing the slots of an instance are described in chapters 1 and 2. In this section we describe the generic functions underlying those functions. The functions rely entirely on these generic functions to implement their behavior. Each of the corresponding functions calls the generic functions directly, the only difference is that the class of the object is included as the first argument to the generic function. In the case of setf functions, the class is the second argument. For example the slot-value and (setf slot-value) functions are implemented in terms of slot-value-using-class and (setf slot-value-using-class) as follows:

```
(defun slot-value (instance slot-name)
  (slot-value-using-class (class-of instance) instance slot-name))

(defun (setf slot-value) (new-value instance slot-name)
  (setf (slot-value-using-class (class-of instance) instance slot-name)
        new-value))
```

slot-value-using-class

The generic function **slot-value-using-class** is called by the function **slot-value**.

slot-value-using-class returns the value of the slot with the given name. All methods on slot-value-using-class call slot-missing if the slot with the given name does not exist. Some methods on slot-value-using-class may do additional checks, for example to see if the slot is bound.

slot-value-using-class `(class standard-class) instance slot-name` [*Primary Method*]

Returns the value of the slot named slot-name if such a slot exists and is bound. If the slot does not exist calls slot-missing. If the slot exists but is not bound calls slot-unbound.

slot-value-using-class `(class structure-class) instance slot-name` [*Primary Method*]

Returns the value of the slot named slot-name if such a slot exists. If the slot does not exists calls slot-missing.

slot-boundp-using-class

The generic function **slot-boundp-using-class** is called by the function **slot-boundp**.

The generic function **slot-boundp-using-class** tests whether a specific slot in an instance of a given class is bound. Not all metaclasses support this operation.

slot-boundp-using-class `(class standard-class) instance slot-name` [*Primary Method*]

If a slot with the given name exists, and that slot is bound, returns true. If a slot with the given name exists and that slot is not bound returns false. If no slot with the given name exists the function slot-missing is called.

slot-boundp-using-class `(class structure-class) instance slot-name` [*Primary Method*]

If a slot with the given name exists, returns true. If no slot with the given name exists the function slot-missing is called.

slot-makunbound-using-class

The generic function **slot-makunbound-using-class** is called by the function **slot-makunbound**.

For metaclass which support unbound slots, the generic function **slot-makunbound-using-class** restores a slot to its unbound state. Attempting to read a slot after it has been made unbound will result in a call to **slot-unbound**.

slot-makunbound-using-class `(class standard-class) instance slot-name` [*Primary Method*]

If a slot with the given name exists in the class the slot is restored to its original unbound state. If there is no slot with the given name in the class calls slot-missing.

slot-makunbound-using-class `(class structure-class) instance slot-name` [*Primary Method*]

Since this operation is not supported by structure-class, this method signals an error.

slot-exists-p-using-class

The generic function **slot-exists-p-using-class** is called by the function **slot-exists-p**.

The generic function **slot-exists-p-using-class** tests whether a slot by the given exists in the instance.

slot-exists-p-using-class `(class standard-class) instance slot-name` [*Primary Method*]

If either a :instance or :class slot slot with the given name exists in the class returns true. Otherwise returns false.

slot-exists-p-using-class `(class structure-class) instance slot-name` [*Primary Method*]

If a slot with the given name exists in the class returns true. Otherwise returns false.

Example of Using the Instance Structure Protocol

This example also makes use of the Optimizing Instance Access Protocol, to fully understand it see that section also.

```
;;;
;;; Define a new metaclass faceted-slot-class which provides one facet
;;; for each :instance slot in the class.  In the instances, the facets
;;; are stored between the slots, each facet comes immediately after its
;;; corresponding slot.
;;;

(defclass faceted-slot-class (standard-class) ())

(defmethod compute-instance-size ((class faceted-slot-class))
  (* 2 (call-next-method)))

(defmethod index-in-instance ((class faceted-slot-class) description)
  (cond ((symbolp description)
         (* 2 (call-next-method)))
        ((and (listp description)
              (eq (car description) 'facet))
         (1+ (index-in-instance (cadr description))))
        (t
         (error "Don't understand the description ~S." description))))
```

```
(defun slot-facet (instance slot-name)
  (standard-instance-access instance
                            (list 'facet slot-name)
                            nil
                            #'facet-unbound
                            #'facet-missing))

(defun (setf slot-facet) (new-value instance slot-name)
  (setf (standard-instance-access instance
                                  (list 'facet slot-name)
                                  nil
                                  #'facet-unbound
                                  #'facet-missing)
        new-value))

(defun facet-unbound (instance facet)
  (error "The facet ~S is unbound in the object ~S" (cadr facet) instance))

(defun facet-missing (instance facet)
  (error "The facet ~S is missing from the object ~S" (cadr facet) instance))

(defmethod optimize-instance-access
           ((class faceted-class) function args instance-arg context)
  (cond ((or (equal function 'slot-facet)
             (eq function #'slot-facet))
         `(standard-instance-access ,(car args)
                                    '(facet ,(cadr args))
                                    #'slot-facet
                                    #'facet-unbound
                                    #'facet-missing))
        ((or (equal function '(setf slot-facet))
             (eq function #'(setf slot-facet)))
         `(setf (standard-instance-access ,(cadr args)
                                          '(facet ,(cadr args))
                                          #'slot-facet
                                          #'facet-unbound
                                          #'facet-missing)
                ,(car args)))
        (t
         (call-next-method))))
```

The Instance Access Optimization Protocol

As described in chapters 1 and 2, most code access instances at the slot level. But, as described in the Instance Structure Protocol section, a call to slot-value results in a call to the slot-value-using-class generic function which then calls standard-instance-access. If every call to slot-value had to do this generic function call, slot access would be too slow.

To solve this problem, CLOS provides a mechanism for optimizing calls to slot-value. At compile-time, this mechanism optimizes calls to slot-value where it is possible to convert the call to a use of standard-instance-access. This requires that the compiler be able to ascertain the class of an instance (it can be a subclass of that class at run-time).

This mechanism is general enough that it can be used to optimize any access to instances whose class is known at compile time.

There is a notion which needs to be defined here. It is the concept of a context in which a call to standard-instance-access can be optimized. I don't understand quite how to define this. It needs to be phrased in a portable way.

The fundamental hook for this mechanism is the optimize-instance-access generic function. This generic function is called on any instance accessing form which, if it were converted to a call to standard-instance-access could be furthur optimized. This gives the metaclass programmer an opportunity to optimize any instance accessing form into a call to standard-instance-access whenever it would do any good.

optimize-instance-access receives as arguments the class of the instance being accessed (or a superclass) the function being called on the instance, all the arguments to the function, the particular one of those arguments which will be the instance at run-time and information about the context the access is in. The context will be the symbol :effect if the compiler is guaranteeing that this access is for effect only.

optimize-instance-access `(class standard-class) function arguments instance-argument context` [*Primary Method*]

```
(defmethod optimize-instance-access
           ((class standard-class) function args instance-arg context)
  (cond ((or (equal function 'slot-value)
             (eq function #'slot-value))
         `(standard-instance-access ,(car args)
                                    '(facet ,(cadr args))
                                    #'slot-value
                                    #'slot-unbound
                                    #'slot-missing))
```

```
((or (equal function '(setf slot-value))
     (eq function #'(setf slot-value)))
 `(setf (standard-instance-access ,(cadr args)
                                  '(facet ,(cadr args))
                                  #'slot-value
                                  #'slot-unbound
                                  #'slot-missing)
        ,(car args)))
(t nil)))
```

When a metaclass optimizes slot accesses, it may do so in a way that makes them deoptimizable. A deoptimized slot access is one that goes through the full access protocol rather than the optimized access. If a metaclass can deoptimize its slot accesses, it should return true from can-deoptimize-slot-accesses-p, if not it should return false.

can-deoptimize-slot-accesses-p `(class standard-class)` [*Primary Method*]

Returns true.

can-deoptimize-slot-accesses-p `(class structure-class)` [*Primary Method*]

Returns false.

deoptimize-slot-accesses `(class standard-class)` [*Primary Method*]

Deoptimizes its optimized slot accesses by calling deoptimize-standard-instance-access.

can-deoptimize-slot-accesses-p `(class structure-class)` [*Primary Method*]

Signals an error.

[[[[Expanding the defgeneric form]]]]

The defgeneric form is expanded into a call to ensure-generic-function, followed by a call to defmethod for each method-description clause in the defgeneric form. The behavior of ensure-generic-function is described in Chapter 2.

The Named Method Definition Protocol

The generic function expand-defmethod is used to compute the expansion of defmethod forms.

expand-defmethod (proto-method name qualifiers lambda-list body environment)

Whatever value expand-defmethod returns will be used as the expansion of the defmethod. Before expand-defmethod is called, the defmethod form is parsed according to the syntax defined in the CLOS spec, so methods on expand-defmethod can't be used to change the syntax of defmethod, but can be used to change the expansion for methods of a particular class. Note that for many uses, it is more appropriate to define a special method on expand-method-body.

The arguments of the standard method for expand-defmethod are as follows:

proto-method:

An instance of the class of method this defmethod form is supposed to define. This class is the one specified by the generic function's :method-class option. It can be the prototype instance of the method class.

name:

The name argument to the defmethod form. It is the name of the generic function that this method should be added to.

qualifiers:

A list of the method qualifiers as specified in the defmethod form

lambda-list:

The specialized lambda-list as specified in the defmethod form

body:

The body as specified in the defmethod form.

environment:

The lexical environment the defmethod form appeared in. This is what the defmethod macro got as its &environment argument.

For a typical defmethod like:

(defmethod move :before ((p position) x y) "Move the position to x,y and update the display" (setf (pos-x p) x) (setf (pos-y p) y) (update-display))

The arguments would be:

name: MOVE qualifiers: (:BEFORE) lambda-list: ((p position) x y) body: ((setf (pos-x p) x) (setf (pos-y p) x) (update-display)) environment: <some structure or NIL>

—

Example of specializing expand-defmethod

Suppose the user wants some methods to broadcast to other machines, but not have calls to those same generic functions that are broadcast to rebroadcast.

(defmethod expand-defmethod ((proto-method broadcast-method) name qualifiers lambda-list body environment) (call-next-method name qualifiers (add-key-argument lambda-list '(broadcaster nil broadcast-p)) ‘(multiple-value-prog1 (progn ,@body) (or broadcast-p ,(broadcast-call name lambda-list))) environment))

The standard method on expand-defmethod calls the generic function expand-method-body. expand-method-body is also called by the other method defining forms. This means that it can affect lexically defined methods as well. This generic function gets the opportunity to do extra processing of the body of the method. This processing can include things like inserting declarations, wrapping a special lexical environment around the body etc.

expand-method-body (mex-method generic-function-name body env)

The mex-method argument is an instance of the same method class that the defmethod form will define (the same class as the method-instance argument to expand-defmethod). Unlike the method-instance argument, the mex-method argument has the qualifiers, lambda-list, and specializers slots filled in. This provides a general mechanism for expand-defmethod to communicate information about the method that will be defined to expand-method-body. If defmethod is being evaluated at load time (as opposed to compile time), the mex object is in fact the method that will be returned by the evaluation of the defmethod form.

add-named-method and friends need to go in here. add-named-method, ensure-generic-function, get-method

The Generic Function Update Protocol

The generic functions get-method, add-method and remove-method previously described provide and interface for directly accessing and manipulating the methods of a generic function.

In order to update links between classes and generic functions that have used the classes as specializers, the standard method on generic-function-changed calls

add-method-on-specializer(method standard-method specializer)

remove-method-on-specializer(method standard-method specializer)

For other changes to a generic function, the update-generic-function generic function is used. It is called with the generic function as a first argument and keywords describing the change that should be made to the generic function.

update-generic-function *generic-function* `&key class method-combination-type method-combination-arguments argument-precedence-order` [*Generic Function*]

The standard method makes the change as specified and then calls compute-discriminator-code to compute new discriminator code for the generic function.

The Method Lookup Protocol

When a generic function is called with particular arguments, it must determine the code to execute. This code is called the effective method for those arguments. The effective method is a combination of the applicable methods in the generic function. A combination of methods is a Lisp expression that contains calls to some or all of the methods. If a generic function is called and no methods apply, the generic function **no-applicable-method** is invoked.

When the effective method has been determined, it is converted to an actual function and the actual function is applied to the same arguments as were passed to the generic function. Whatever values it returns are returned as the values of the generic function.

The specification for the precise way the kernel computes the effective method appears in chapter 1. This section describes the protocol used to compute and invoke the effective method.

The key component in this protocol is the discriminator code for the generic function. The discriminator code for a generic function is called whenever the generic function is called; it computes the effective method and invokes it. The discriminator code for a generic function is computed each time the generic function changes. When the generic function itself is called, the pre-computed discriminator code is called. The protocol described here is used to compute the discriminator code, thus the protocol described here is not invoked when the generic function is called, it is invoked whenever the generic function is updated. This allows method lookup to be a fast operation.

The standard-method on update-generic-function calls compute-discriminator-code whenever the generic function changes. compute-discriminator-code is expected to return the discriminator code for the generic function. This discriminator code must be valid until the next time update-generic-function is called.

The standard-method on compute-discriminator-code provides the documented behavior of computing the effective method and calling it. It does this by providing a second layer of protocol, specifically the compute-effective-method generic function. compute-effective-method is called by the standard-method on compute-discriminator-code to compute the effective method for a set of applicable methods.

There are also several support functions supplied by the kernel to assist users in extending the method lookup protocol. These support functions implement certain key parts of the kernel method lookup behavior.

compute-discriminator-code *generic-function* [*Generic Function*]

compute-discriminator-code `(generic-function standard-generic-function)` [*Primary Method*]

```
(defmethod compute-discriminator-code
           ((generic-function standard-generic-function))
  #'(lambda (&rest args)
      (let* ((lambda-list (slot-value generic-function 'lambda-list))
             (methods (compute-applicable-methods generic-function args))
             (function
               (make-effective-method-function
                  generic-function
                  (compute-effective-method
                    generic-function
                    methods
                    (slot-value generic-function
                                'method-combination-type)
                    (slot-value generic-function
                                'method-combination-arguments)))))
        (check-keyword-arguments lambda-list methods args)
        (apply function args))))
```

compute-effective-method *generic-function applicable-methods method-combination-type method-combination-arguments* [*Generic Function*]

compute-effective-method `(generic-function standard-generic-function) applicable-methods method-combination-type method-combination-arguments` [*Primary Method*]

Actually, there is one method here for each pre-defined method-combination-type. This needs to be explained in terms how define-method-combination expands into a defmethod for compute-effective-method.

Support Functions for Method Lookup

In order to help the user use the method lookup protocol the CLOS kernel provides some helpful support functions.

compute-applicable-methods *generic-function arguments* [*Function*]

Given a generic function and a set of arguments, this uses the standard rules to determine the ordered set of applicable methods.

compute-combination-points *generic-function* [*Function*]

Computes all the combination points for this generic functions. That is all the points at which (if you are using combined-methods) methods must be combined. For each point it also provides the ordered set of methods applicable at the point.

This may sound like it is too implementation specific to be useful in the metaobject protocol, but I think that is because of the way I am describing it. I believe a lot of method lookup hackers are going to want to compute this, and given that it is hard to compute accurately and quickly I think we should provide it.

check-keyword-arguments *generic-function lambda-list methods args* [*Function*]

This implements the keyword congruence rules specified in chapter 1. If the keyword arguments in args are OK, this returns t. Otherwise it signals an error.

make-method-call *method-list* `&key operator identity-with-one-argument` [*Function*]

This is documented in chapter 2.

make-function-call *function* [*Function*]

This has behavior similar to make-method-call. It produces a call to the function with all the arguments of the generic function.

we should invent a better name for this.

make-effective-method-function *generic-function effective-method-body* [*Function*]

This takes the effective method body as computed by compute-effective-method-body and converts it to a function which implements the effective method. This function accepts the same arguments the generic function accepts. This function does the standard keyword congruence checking. This function arranges to call all the methods "as if with :allow-other-keys t".

Basically, make-effective-method-function, make-method-call and make-function-call are the ones that have the contract that makes calling methods work. They communicate the information about what parameters the arguments will be bound to, how to hack :allow-other-keys t etc.

I am concerned that actually we have to get rid of make-method-call and have a call-method special form. Otherwise, I don't know how someone is going to build a portable stepper for standard generic functions.

Example of using the Method Lookup Protocol

This example defines a special class of tracing generic function. This class of generic function provides two kinds of tracing facilities. The first kind allows the user to specify that calls to particular generic functions should cause breakpoints. The second allows the user to specify that calls to the effective method for particular sets of methods should cause breakpoints.

```
(defvar *trace-generic-functions* ())
```

```
(defvar *trace-effective-methods* ())

(defclass tracing-generic-function (standard-generic-function) ())

(defmethod compute-discriminator-code ((gf tracing-generic-function))
  (let ((real-discriminator-code (call-next-method)))
    #'(lambda (&rest args)
        (when (member gf *trace-generic-functions*)
          (break "The generic function ~S is one of the generic~%~
                  functions on *trace-generic-functions*"
                 gf))
        (apply real-discriminator-code args))))

(defmethod compute-effective-method ((gf tracing-generic-function)
                                     methods
                                     method-combination-type
                                     method-combination-arguments)
  `(progn
     (when (member ',methods *trace-effective-methods* :test #'equal)
       (break "The set of methods ~S is one of the sets of~%~
               methods on *trace-effective-methods*."
              ',methods))
     ,(call-next-method)))
```