Journal of Universal Computer Science, vol. 14, no. 20 (2008), 3358-3369
submitted: 29/4/08, accepted: 16/11/08, appeared: 28/11/08 © J.UCS

An Implementation of CLIM Presentation Types

Timothy Moore
(Red Hat SARL, Paris la Défense, France
timoore@redhat.com)

Abstract: Presentation types are used in the CLIM interface library to tag graphical output with a type and establish an input type context in which the user may use the keyboard to type input, accepted by a parser associated with that presentation type, or click on the graphical representation of an object that has an appropriate presentation type. Presentation types are defined using a syntax reminiscent of the `deftype` syntax of Common Lisp; the input and output actions of the types, as well as aspects of their inheritance, are implemented using a system of generic functions and methods directly based on CLOS. The presentation type system is different enough from the Common Lisp type system that its types, generic functions and methods do not map directly to those of Common Lisp. We describe the presentation type implemention in McCLIM which uses the CLOS Metaobject Protocol to implement presentation type inheritance, method dispatch and method combination without implementing an entire parallel object system next to CLOS. Our implementation supports all types of method combination in the presentation methods, including user-defined method combination.
Key Words: Common Lisp, CLIM, presentation types, metaobject protocol
Category: D.3.3, D.1.5, D.2.2

1 Introduction

The specification of the Common Lisp Interface Manager (CLIM) [McKay, n.d.; Rao *et al.*, 1991] describes a graphical interface toolkit for Common Lisp [ANSI, 1996] in which program objects are explicitly associated with graphical representations of those objects, called *presentations.* Based on the user's interactive input, and according to a context of desired input established by the program, the previously presented objects are made available as input to commands. Commands are launched either graphically, in the traditional style of GUI interaction, or by typing on a command line, in which case a click on a presentation causes a textual representation of it to be inserted in the command line.

Presentations store a *presentation type* as well as an object, and it is this type that is used to test whether a presentation can satisfy the current input context. In most programs that use CLIM a highlightling rectangle is drawn around objects that match the current input context as the user moves the mouse over them, and a message summarizing the input action that will occur if the mouse buttons are pressed is displayed at the bottom of the screen. Presentation types share similarities both with Common Lisp built-in types and with Common Lisp Object System (CLOS) classes. A system of generic functions and methods, also similar to that in Common Lisp, supports dispatch on types as if they were objects. Several of these *presentation generic functions* are defined by CLIM to control input parsing and output of the objects associated with presentation types, type membership tests, and subtype relations between the presentation types.

The semantics of presentation types are different enough from those of Common Lisp types defined via the `deftype` macro and standard classes such that a naïve implementation would duplicate a lot of the complex method combination and dispatch code that must exist in a Common Lisp implementation to support CLOS. We describe here the the implementation of the presentation type system in McCLIM [Strandh & Moore, 2002], an open source implementation of CLIM. We used the Metaobject Protocol [Kiczales & des Rivieres, 1991] present in most Common Lisp implementations to implement presentation types and generic functions. McCLIM was written from scratch with reference to the CLIM specification, a terse and, at times, incomplete and contradictory document, and a few available example CLIM programs. For an introduction to Common Lisp and CLOS refer to texts such as [Seibel, 2005; Graham, 1999; Norvig, 1991; Keene & Gerson, 1989]. An introduction to CLIM can be found in [Rao *et al.*, 1991; Möller, n.d.].

2 Presentation Types

Before describing the definition of presentation types, it is useful to review the ways that new types, called *type specifiers*, are defined in Common Lisp because CLIM presentation types use concepts from these approaches. A type specifier is a name or a list of a name and parameters that can be passed to `typep` to test whether an object is of a certain type or to `subtypep` to determine subtype relationships. Type specifiers are defined using either the `deftype` macro or the `defclass` macro. Instances of the classes defined using `defclass` can be created using `make-instance` and the arguments specifed in the defining `defclass` form. The type specifier of a user-defined class is either the name of the class (as a symbol) or a *metaclass* object created by the system to represent the type.

We ignore types created with the Common Lisp macros `defstruct` and `define-condition` as they are very similar to classes defined with `defclass`.

2.1 Common Lisp type specifiers and `deftype`

The `deftype` macro defines a function that expands a type specifier into another type specifier through a process very similar to macro expansion; indeed, the function created by `deftype` behaves exactly like a macro expander function created by `defmacro`, except that the default argument for optional and keyword arguments in the type specifier form is `*`, the wildcard type specifier. The body of the `deftype` form returns a new type specifier using the arguments, existing type

```
(deftype even-positive-integer (&optional high)
  `(and (integer 0 ,high) (satisfies evenp)))
```

Figure 1: *Example of `deftype` usage*

```
(defclass person ()
  ((name :accessor name :initarg :name)
   (age :accessor age :initarg :age)))
```

Figure 2: *Class definition example*

specifiers, compound type specifiers like `and` and `or` that create intersections and unions of existing types, or the `satisfies` type specifier that uses a function predicate to define a type. [Figure 1] shows a simple `deftype` definition that creates a subset of the `integer` type with parameters to `integer` and a functional predicate. It is important to note that the types created with `deftype` cannot specify objects with new characteristics in Common Lisp; they can only restrict existing types by giving them explicit parameters or perform set operations on the membership of the types. They cannot be specified in `defmethod` argument specializers.

2.2 Classes defined with `defclass`

Classes are user-defined types that have superclasses and that can store data in *slots*. A class is defined using the `defclass` macro. A *slot definition* specifies the name of the slot and optional parameters such as the type of the value of the slot and the names of generic functions that get and set its value.

[Figure 2] shows the definition of a simple class. This class is named `person` and has two slots, `name` and `age`. Classes can inherit from one or more user-defined classes to create a subtype relationship. The new subclass is a subtype of its superclasses. [Figure 3] shows the definition of an `engineer` class that inherits from a `specialty-mixin` class as well as from the `person` class.

2.3 The form `define-presentation-type`

Presentation types combine aspects of type specifiers and classes considered as types. The type is descriptive and parameterized, like a type specifier, but is not instantiable. The concrete representation of a presentation type is either a symbol or a list with arguments. Presentation types support multiple inheritance,

```
(defclass specialty-mixin ()
  ((specialty :accessor specialty :initarg :specialty)))

(defclass engineer (person specialty-mixin)
  ())

(defclass cook (person specialty-mixin)
  ())
```

Figure 3: *Multiple inheritance with a mixin class*

```
(define-presentation-type integer (&optional low high)
  :options ((base 10) radix)
  :inherit-from `((rational ,low ,high)
                  :base ,base :radix ,radix))

(define-presentation-method presentation-typep
    (object (type integer))
  (and (integerp object)
       (or (eq low '*)
           (<= low object))
       (or (eq high '*)
           (<= object high))))

(defmethod presentation-type-of ((object integer))
  'integer)

(presentation-typep 42 '(integer 6 43))
T
```

Figure 4: *Example of presentation type, its definition, and a presentation method*

and can participate in a kind of method dispatch and combination in which parameters of the type are available inside the methods.

[Figure 4] shows the definition of a presentation type, `integer`, that is a part of CLIM. The parameters `low` and `high` specify the members of the type. This type also specifies options that do not affect type tests and membership but do affect how presentations with this type will be displayed and how input will be parsed in this input context. The `:inherit-from` argument is a form that specifies the supertypes of the presentation type and can use the parameters and options as arguments in a limited way: the form must be able to create its result without referring to the actual value of the parameters and options. This allows the `:inherit-from` form to be analysed using dummy arguments at the time of the type definition.

To support dispatching on a single presentation type argument, CLIM provides *presentation generic functions* and *presentation methods* that are similar to their CLOS equivalents – for example, method combination and effective method computation work as expected – but that also make parameters and options available as implicitly defined variables in the methods, properly transformed for the presentation type. This style of magic slot access in methods is not found elsewhere in Common Lisp today but is retained for compatibility with an earlier presentation-based system found in Dynamic Windows in the Symbolics Genera environment.[1] In [Figure 4], `presentation-typep` is a presentation generic function defined by CLIM. The `type` argument is a presentation type. This method is properly ordered with respect to other applicable presentation methods such as, for example, a method for the presentation type `rational`. The parameters and options of the presentation type are available as bound variables inside the method.

The call to `presentation-typep` in [Figure 4] shows a typical use of presentation types. `presentation-typep` is a function that invokes the presentation generic function `presentation-typep`. This computes and invokes the effective method for this call which then calls the presentation method for the type `integer`. [Figure 5] shows two more basic presentation methods, `present` for output and `accept` for input. Presentation methods make use of the options available in presentation types.

CLOS class names and metaclass objects are valid as presentation types. Many builtin Lisp types have a presentation type equivalent with the same name.

In order to make presentation types less abstract, [Figure 6] shows some experiments with presentation types in the Listener application supplied with McCLIM. The `present` function writes output annotated with a presentation type, called a *presentation,* to an output stream. The function `accept` reads input, either typed by the user or entered by clicking on a presentation with a compatible presentation type. In this case “42” is acceptable because the presentation type `(integer 0 50)` is a subtype of `(real 0 100)`. The pointer documentation pane at the bottom of the listener window shows the action if the user clicks on the left mouse button: “42” will be accepted. The number 42 satisfies the `presentation-typep` predicate of the presentation type `integer` with the parameters 0 and 50; the behavior is undefined if an object is presented that is

[1] The designer of this feature now says “I can now say that this was a mistake, and that we should have simply implemented a `with-slots`-like macro that did the right thing.” [McKay, 2008]

```
(define-presentation-method present
    (object (type integer) stream (view textual-view)
     &key acceptably for-context-type)
  (declare (ignore acceptably for-context-type))
  (let ((*print-base* base)
        (*print-radix* radix))
    (princ object stream)))

(define-presentation-method accept ((type integer)
                                    stream (view textual-view)
                                    &key (default nil defaultp)
                                    default-type)
  (let ((*read-base* base))
    (let* ((token (read-token stream)))
      (when (and (zerop (length token))
                 defaultp)
        (return-from accept (values default default-type)))
      (parse-integer token))))
```

Figure 5: *Example `present` and `accept` methods. The presentation options `base` and `radix` are used in these methods.*

not actually a member of the presented presentation type.

3 Implementation of Presentation Types

3.1 Presentation Types and Presentation Methods

Although they are represented as lists, presentation types have many characteristics of CLOS objects. Their parameters and options are similar to class slots, and they have an inheritance relation with their supertypes. However, parameters and options are not inherited from supertypes – they parameterize the supertypes and they may be arbitrarily transformed within the limits imposed on the `:inherit-from` specification. A parameter may have a different value in a presentation method written on a supertype than it does in a subtype method; this is the opposite of the behavior of slots, which have a single value in an object. Nevertheless, if a presentation type could be represented as a CLOS object, then presentation method dispatch could be implemented easily using normal CLOS method dispatch. The CLIM specification seems to point in this direction, saying "Every presentation type is associated

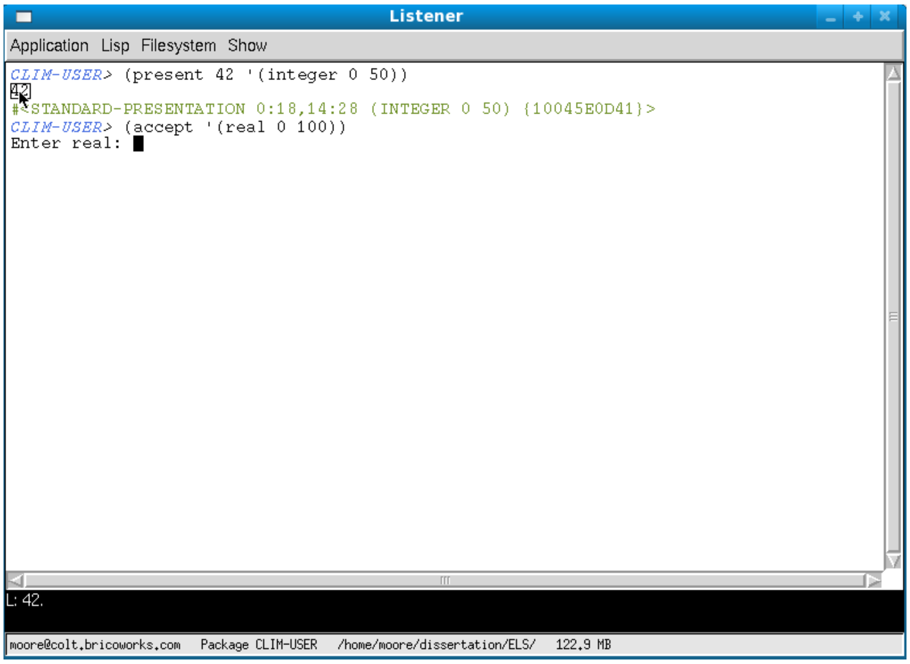

Figure 6: *Presentation types in action. "42" has been presented to the screen with a presentation type that is a subtype of* `integer`*; that value can be accepted if a subtype of* `real` *is requested.*

with a CLOS class... `define-presentation-type` defines a class with metaclass `presentation-type-class` and superclasses determined by the presentation type definition." Also, the lambda list of a presentation generic function must contain a mysterious "`type-key` or `type-class` [argument]; this argument is used by CLIM to implement method dispatching." There only needs to be a single type key object for a presentation type because presentation method dispatch is not influenced by a presentation type's parameters.

There are some awkward complications with this approach. It is easy to construct a type key for presentation types defined via `define-presentation-type`; it can be created as part of the evaluation of the defining form. But CLOS classes are implicitly presentation types too, and it is not obvious how to create an instance of an arbitrary class without any knowledge of its required initialization arguments. It is reasonable to define presentation methods on `standard-object`, the superclass of all CLOS classes, but many presentation types do not have `standard-object` as a supertype and so those methods should not be applicable when a presentation generic function is called on such a type.

3.2 The Metaobject Protocol

Fortunately most implementations of Common Lisp implement the Metaobject Protocol, or MOP, as described in [Kiczales & des Rivieres, 1991]. This exposes many of the internal details of class definition, generic function definition and method dispatch and allows them to be customized. The implementation of presentation types makes use of two major features of the MOP. The MOP specifies that a *class prototype* object, which is an instance of a class with undefined slot values, exists for all classes. This is obviously ideal to use as the type key object. Also, the MOP supports broad customization of the selection of applicable methods in a generic function call via the generic functions `compute-applicable-methods` and `compute-applicable-methods-using-classes`. Even Common Lisp implementations that do not support the full MOP usually have some internal functionality that is equivalent to these features and that can be used in the presentation types implementation.[2]

3.3 Implementation using the MOP

A class metaobject of type `presentation-type-class`, a subclass of `standard-class`, is created for each defined presentation type. The class is given a fake name so that there is no conflict between presentation types and built-in types of the same name. This class stores details about the presentation type including a function that produces the `:inherit-from` form from parameter and option arguments. The supertypes of the presentation type, retrieved by running the `:inherit-from` function with dummy arguments, become the direct superclasses of the metaobject. A hash table maps presentation type names to these metaobjects. CLOS classes that are mentioned in `define-presentation-type` forms are represented by a presentation type class that is not a metaclass but that does contain a reference to the metaclass of that class.

According to the CLIM specification [McKay, n.d.], presentation generic functions are called using the macros `funcall-presentation-generic-function` and `apply-presentation-generic-function`. This extra syntax is rather awkward but, in actual CLIM programming, presentation generic functions are not called directly by the programmer; they are invoked indirectly by calling functions defined in the CLIM specification. For example, a program calls the `present` function, and that function calls the presentation generic function of the same name, perhaps after establishing dynamic state and defaulting arguments. One of the arguments in the presentation generic function call will be a presentation type specifier which is examined to find the presentation type metaobject and thence the associated class prototype. This is passed as an argument to the

[2] This work was originally done using OpenMCL, which at the time did not have a full MOP implementation.

```
(defmethod compute-applicable-methods :around
    ((gf presentation-generic-function) arguments)
  (let ((methods (call-next-method)))
    (if (typep (class-of (car arguments))
               'presentation-type-class)
        (remove-if
         #'(lambda (method)
             (eq (car (clim-mop:method-specializers method))
                 *standard-object-class*))
         methods)
        methods)))
```

Figure 7: *`compute-applicable-methods` implementation which removes any presentation methods defined for CLOS types from methods for a non-CLOS presentation type. The code for `compute-applicable-methods-using-classes` is similar.*

presentation generic function as the type key object. If the presentation type argument is a CLOS class, the prototype of that class is passed.

The type key object, and all the other arguments of the presentation generic function, are used to compute the applicable methods for the function invocation.

The Metaobject Protocol specifies that one of two generic functions, `compute-applicable-methods` or `compute-applicable-methods-using-classes`, can be called when computing the applicable methods for a particular function invocation. The latter is called in situations where only the classes of arguments are relevant for object dispatch i.e., there are no `eql` specializers in the argument lists of the generic function's methods, and allows a CLOS implementation to optimize this step and memoize the results. Any programmer customization of this process must define methods for both of these functions. The `presentation-generic-function` metaclass, a subclass of `standard-generic-function`, has specialized versions of these two methods which eliminate any potentially applicable method that is specialized on `standard-object` if the presentation type argument is not a CLOS class, as shown in [Figure 7]. The presentation methods themselves are just regular methods with an additional argument for the type key. [Newton & Rhodes, 2008; Verna, 2008] have also found these MOP generic functions useful starting points for their own object system experiments and customization.

The body of a presentation method is wrapped by code that expands the presentation type argument from its actual type to the supertype expected by the method. Each presentation type's `:inherit-from` function can translate a type

```
(defmethod %presentation-typep
    ((type-key |(presentation-type common-lisp::integer)|)
     object type)
  (block presentation-typep
    (let ((#:massaged-type2397
           (translate-specifier-for-type
            (type-name-from-type-key type-key) 'integer type)))
      (let ((parameters (decode-parameters #:massaged-type2397)))
        (declare (ignorable parameters))
        (with-presentation-type-parameters
            (integer #:massaged-type2397)
          (and (integerp object) (or (eq low '*) (<= low object))
               (or (eq high '*) (<= object high)))))))))
```

Figure 8: *The expansion of the `define-presentation-method` form for the `presentation-typep` method of the `integer` presentation type.*

specifier to that of its supertypes i.e., produce a new list with type parameters appropriate for the supertype. This can be done repeatedly on a subtype and its supers until the type specifier of an arbitrary supertype is produced. Once this is in hand, the parameter and options are decoded and bound to variables in the method body. [Figure 8] shows the expansion of the method definition in [Figure 4].

In effective methods that contain many constituent methods, this strategy could lead to poor performance because the expansion functions for the most specific classes need to be be run repeatedly as less specific methods are called. It was thought that it would be useful to introduce a caching mechanism to mitigate this effect, but profiling has not shown this process to be a bottleneck in real applications that use McCLIM. An alternate strategy would be to perform the expansion outside of the method body, in the method combination code. This approach avoids unnecessary expansion, but it breaks all non-standard method combinations. The simpler approach used in McCLIM, which keeps the type argument expansion inside the method body, allows all standard and user-defined method combination to "just work."

4 Conclusion

We have described the implementation of a complex part of the CLIM specification, presentation types, using features of the Common Lisp Metaobject Protocol. We were able to use a small slice of the MOP to our advantage.

The implementation of presentation method dispatch, using class prototypes, is straightforward and would require no new metaclasses if we did not have a requirement to eliminate some presentation methods from the list of applicable methods in a presentation generic function call. The major remaining complexity is in the translation of presentation subtype parameters to parameters for the supertypes, and that is handled by runtime support added by somewhat complicated macro programming.

At the time that CLIM was first specified and implemented (1992), the MOP was quite new and not well supported in CLOS implementations. CLOS itself was new, and CLIM was a major driver of the evolution of CLOS implementations. If the MOP had been well supported, some syntactic choices in CLIM (such as the `funcall-presentation-generic-function` macro) would have undoubtedly been different, and the specification could have referenced MOP concepts such as the class prototype directly [McKay, 2008]. Nevertheless the concept of presentation types and the implementation details presented here are useful even considered separately from the specific syntax of CLIM.

Without MOP support the implementation of presentation types and presentation method dispatch requires an enormous amount of coding; this daunting task had blocked progress in the McCLIM project for some time. The realization that use of a small part of the Metaobject Protocol coupled with layers of macro support could be used to implement this part of CLIM resulted in a robust presentation type system for McCLIM in a fairly short time. This in turn supports a large amount of functionality, including presentation methods for standard types and the full machinery for `accept` and `present`, that make McCLIM a real implementation of CLIM. The successful applications built using McCLIM help to confirm that the MOP is a powerful tool for implementing object systems and programmers should not be intimidated by it.

Acknowledgements

I would like to thank Robert Strandh, who invited me to be a "professeur associé" at the Université de Bordeaux. As a result of this life-changing event much work was done on McCLIM. Also, I would like to thank the entire McCLIM team, in particular Christophe Rhodes and Troels Henriksen, for their bug fixes to the presentation type code in McCLIM. McCLIM can be found at `http://common-lisp.net/project/mcclim/`.

References

[ANSI, 1996] ANSI. 1996. *American National Standard for Information Technology: programming language — Common LISP.* American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. Approved December 8, 1994.

[Graham, 1999] Graham, Paul. 1999. *ANSI Common LISP*. Second edn. Englewood Cliffs, NJ 07632, USA: Prentice-Hall.

[Keene & Gerson, 1989] Keene, Sonya E., & Gerson, Dan. 1989. *Object-oriented programming in Common LISP: a programmer's guide to CLOS*. Reading, MA, USA: Addison-Wesley.

[Kiczales & des Rivieres, 1991] Kiczales, Gregor, & des Rivieres, Jim. 1991. *The art of the metaobject protocol*. Cambridge, MA, USA: MIT Press.

[McKay, n.d.] McKay, Scott. *Common Lisp Interface Manager CLIM II Specification*. Available at `http://www.stud.uni-karlsruhe.de/~unk6/clim-spec/`.

[McKay, 2008] McKay, Scott. 2008 (April). personal communication.

[Möller, n.d.] Möller, Ralf. *User Interface Management Systems: the CLIM perspective*. Available at `http://www.sts.tu-harburg.de/~r.f.moeller/uims-clim/clim-intro.html`.

[Newton & Rhodes, 2008] Newton, Jim, & Rhodes, Christophe. 2008. Custom specializers in object-oriented Lisp. *Pages 75–87 of: European Lisp Symposium*.

[Norvig, 1991] Norvig, Peter. 1991. *Paradigms of artificial intelligence programming: case studies in Common LISP*. Los Altos, CA 94022, USA: Morgan Kaufmann Publishers.

[Rao *et al.*, 1991] Rao, Ramana, York, William M., & Doughty, Dennis. 1991. A guided tour of the Common Lisp interface manager. *SIGPLAN Lisp Pointers*, **IV**(1). Updated 2006 by Clemens Frühwirth.

[Seibel, 2005] Seibel, Peter. 2005. *Practical Common Lisp*. Apress.

[Strandh & Moore, 2002] Strandh, Robert, & Moore, Timothy. 2002. A Free Implementation of CLIM. *In: Proceedings of the 2002 International Lisp Conference*.

[Verna, 2008] Verna, Didier. 2008. Binary methods programming: the CLOS perspective. *Pages 91–105 of: European Lisp Symposium*.