CommonLoops
Merging Lisp and Object-Oriented Programming

Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales,
Larry Masinter, Mark Stefik, and Frank Zdybel
Xerox Palo Alto Research Center
Palo Alto, California 94304

CommonLoops blends object-oriented programming smoothly and tightly with the procedure-oriented design of Lisp. Functions and methods are combined in a more general abstraction. Message passing is invoked via normal Lisp function call. Methods are viewed as partial descriptions of procedures. Lisp data types are integrated with object classes. With these integrations, it is easy to incrementally move a program between the procedure and object-oriented styles.

One of the most important properties of CommonLoops is its extensive use of meta-objects. We discuss three kinds of meta-objects: objects for classes, objects for methods, and objects for discriminators. We argue that these meta-objects make practical both efficient implementation and experimentation with new ideas for object-oriented programming.

CommonLoops' small kernel is powerful enough to implement the major object-oriented systems in use today.

Introduction

Over the last decade many systems have been written that add objects to Lisp (e.g., Flavors, Loops, ObjectLisp.) Each of these has attracted a group of users that recognize the benefits of message sending and specialization and have endorsed an object-oriented style. The object languages in these systems have been embedded in Lisp with different degrees of integration.

Lisp continues to be an important and powerful language for symbol manipulation and is widely used for programming in artificial intelligence applications. One of Lisp's interesting strengths is its ability to absorb other languages, that is, its use as a base for implementing experimental languages. Within the procedure-oriented paradigm, Lisp provides an important approach for factoring programs that is different from common practice in object-oriented programming. In this paper we present the linguistic mechanisms that we have developed for integrating these styles. We argue that the unification results in something greater than the sum of the parts, that is, that the mechanisms needed for integrating object-oriented and procedure-oriented approaches give CommonLoops surprising strength.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/ or specific permission.

© 1986 ACM 0-89791-204-7/86/0900-0017 75¢

We describe a smooth integration of these ideas that can work efficiently in Lisp systems implemented on a wide variety of machines. We chose the Common Lisp dialect as a base on which to build CommonLoops (a Common Lisp Object-Oriented Programming System). because Common Lisp is supported on almost all commercial Lisp workstations. A portable implementation of CommonLoops is available and is being used in many Common Lisp implementations.

With respect to Lisp, CommonLoops has tried to satisfy a number of different, sometimes conflicting goals:

Compatibility: CommonLoops is compatible with Lisp's functional programming style. Message sending uses the same syntax as function call. Method definition is an extension of Lisp function definition. This is described in section I. Object space is defined as a natural extension of the Common Lisp type space. This is described in section II. Integrated syntax and type spaces allow incremental conversion of programs from a functional to an object-oriented style.

Powerful base: CommonLoops is rich enough for building interesting applications without the need for higher level object languages. It also provides several desirable extensions to object-oriented programming. Method lookup can be based on the class of more than one argument (a "multi-method"). Behavior for an individual object can be specified.

Portability: CommonLoops provides a small kernel that is easy to integrate into Common Lisp

implementations. CommonLoops is currently running in five different implementations of Common Lisp.

Flexibility: CommonLoops can be used to implement the major object languages in use today (e.g., Flavors, Smalltalk and Loops) as well as new languages like ObjectLisp. CommonLoops supports intercallability among objects from these different languages. The use of meta-objects in CommonLoops supports variations in object representation, method syntax, combination and optimization. This makes CommonLoops open-ended enough to support research and experimentation with future object and knowledge representation languages, while providing a base for standardization.

Efficiency: Using proven software techniques, described in section III, CommonLoops can run efficiently without special hardware support. This is important because Common Lisp runs on a wide variety of hardware bases.

I. Methods and Functions

In Lisp, functions are applied to arguments. The code that is run is determined only by the name of the function. The lisp form

```
(foo a b)
```

can be interpreted in terms of a function calling primitive, `funcall` as:

```
(funcall (function-specified-by 'foo).
         a b).
```

In object-oriented systems one "sends messages" to objects. The code that is run is determined by both the name of the message and the type (class) of the object. Methods defined for a particular selector are associated with a class. In the next section we will indicate how we merge the ideas of Lisp datatypes and object classes. The following message using selector `sel`:

```
(send a 'sel b)
```

can be interpreted as the function call:

```
(funcall
 (method-specified-by 'sel (type-of a))
 a b).
```

The collection of all methods defined for `sel` define the "generic" function for that selector. Which method is run when a generic function is invoked is determined by the type of the first argument. Thus a method is a partial description of a generic function restricted to objects of a particular type. With this understanding of method invocation, we can reinterpret all standard Lisp calls:

```
(foo a b)
```

as meaning

```
(funcall
 (method-specified-by 'foo (type-of a))
 a b).
```

if there is a method defined for `foo` and `(type-of a)`. Some of the ideas described here were independently invented and implemented in New Flavors [MoonKeene86]. Because of their similarity, we will contrast CommonLoops with New Flavors where appropriate. We adapted their term, generic function, to describe the collection of methods for a selector.

A method for `move` applicable only when the first argument is of type `block` is defined in CommonLoops as follows:

```
(defmeth move ((obj block) x y)
  <code for moving a block>)
```

The code for this method is added to the generic function for `move`, and is invoked for objects of type `block`, or any subtype. If there was an existing method for the same selector and type, `defmeth` replaces that method. To invoke this method, one simply writes:

```
(move block1 x-pos y-pos)
```

Given that `block1` is of type `block`, the code above will be invoked. Other methods for move could be defined for first argument being a window, a sketch, etc. If more than one method is applicable (because of subclassing), the most specific method is used.

Default Methods

If one uses the `defmeth` form without specifying any type as in:

```
(defmeth move (thing x y)    ...)
```

this code is run when no more specific method of the generic function for `move` is applicable. When only such a default method is supplied, it is like defining an ordinary Lisp function. There is no speed penalty for using such default methods instead of functions.

The difference between defining a default method and defining an ordinary Lisp function is that the latter is not allowed to be augmented by specialized method definitions. This protects users from inadvertently overriding or specializing predefined functions where

perhaps special compilation optimizations have been used. For example, in most Lisp implementations calls to the primitives `car`, `cdr`, and `cons` are compiled specially for efficiency. Specializing these functions could either have disastrous effect on system efficiency, or no effect on previously compiled code.

However, it is often useful to be able to define methods which specialize existing Lisp functions. To make the Lisp function `print` specializable, one uses:

```
(make-specializable 'print
                    '(thing stream))
```

This declares that the pre-existing lisp function `print` is to become the default method for the generic function, and that additional methods can be added.

Multi-Methods

CommonLoops extends Lisp's function call even further. It allows the method to be specified in terms of the types of any number of arguments to the form. It interprets the form:

```
(foo a b)
```

as

```
(funcall
 (method-specified-by 'foo
                      (type-of a)
                      (type-of b))
  a b).
```

Thus, unlike most other object-oriented schemes, CommonLoops allows method-lookup to be based on more that the class of the first argument. For example,

```
(defmeth insidep
   ((w window) (x integer) (y integer))
   ...)
```

defines the method for `insidep` when the first argument is a window and the second and third arguments are integers.

For any set of arguments, there may be several methods whose type specifications match. The most specific applicable method is called. Method specificity is determined by the specificity of the leftmost type specifiers which differ. However, as discussed below, other regimes can be implemented using the meta-objects facility.

Method and Discriminator Objects

In CommonLoops all the data-structures used to implement the system are objects. In particular, defining a method creates three objects, the *method*, the *discriminator* and the *discriminating-function*.

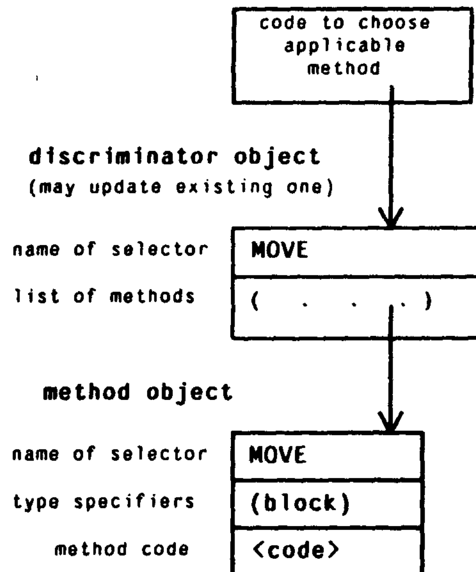

Figure 1: These three objects are used for interpretation of a call. The discriminating function is the code object that selects the method to be called. The discriminator object describes the generic function and is used to construct the discriminating function. It uses the information in the method object; the method object is also used in the compilation of the code for the specific method.

The method object represents the method being defined. The method object contains the type-specifiers and the code for the method. The discriminator object contains a list of all the methods defined on a particular selector. Hence, it describes the generic function. Together, the discriminator and all of its methods produce the discriminating function, a piece of Lisp code which is called when the selector is invoked to determine which method to call. Because the method-lookup and calling mechanisms are under control of the discriminator and method objects, specialized method-lookup and method-combination mechanisms can be implemented by defining new classes of discriminators and methods which specialize parts of the method-lookup protocol.

One such special class are methods which are specialized to individuals. By this we mean that some methods are applicable only if called with a specific object as argument. For example, this would allow a special-case for a connection to a particular host on a network for some period of time when special rerouting needs to be done. Standard protocol makes a

method applicable to an individual more specific than any method just specified on types.

Method Combination

Frequently, when one specializes behavior for a given class of object, the desire is to add only a little behavior to the methods of the super-classes.

The primary mechanism for method combination in CommonLoops is run-super which is defined to run the most specific method matching the arguments of the current method that is more general than the method in which the run-super occurs. If there is no such method an error is signaled.

For example,

```
(defmeth move
  ((w bounded-window)
   (x integer)
   (y integer))
  (cond ((in-bounds-p w x y)
(run-super))
        (t ...   ;; move to closest point inside
          (run-super))))
```

defines a method which specializes the move method on window so that it always moves in-bounds.

The run-super is essentially the mechanism of method combination found in Smalltalk, Loops, Director and Object-Lisp. It is both powerful and simple. It allows arbitrary combination of inherited code with current code using Lisp as the combination language.

Sometimes it is more useful to have a declarative means of specifying method combination. In Flavors, for example, before and after parts can be specified for any method, and these will be run before and after any directly specialized method without requiring any statement in the specialized method. Before and after parts can be attached any place in the inheritance chain, and are combined in a single method at definition time. In CommonLoops we implement this feature using a special discriminator object that indexes these parts and does the method combination. We have implemented in CommonLoops the interface for user defined method combination specified for New Flavors.

Method and discriminator objects are used to implement both run-super and the user defined method combination mechanism. This provides the flexibility of choosing either of the standard kinds of combination in use today. In addition, the existence of these meta-objects allows experimentation with other kinds of combination and invocation. We are currently looking at integrating logic programming into the CommonLoops framework. Logic programming requires specialized method and discriminator objects to combine method clauses using backtracking search.

Processing of method code

The code that implements a method is interpreted and compiled in a context in which the method object is available. The method can use information from the type-specifiers to optimize parts of the method body, or to provide special syntax within the body of the method to access the slots of arguments to the method. Because this processing is done using a defined protocol of messages to the method object, it can be extended by users.

II. Defining Classes

CommonLoops uses defstruct to define its classes, extending the syntax of the construct found in Common Lisp for defining composite structures.

```
(defstruct position
  (x-coord 0)
  (y-coord 0))
```

defines a class named position, and specifies that instances of that class should have two slots, x-coord and y-coord, each initialized to 0. As a side effect of defining this structure, defstruct also defines a function to make instances of type position, and functions position-x-coord and position-y-coord to access the slots of an instance. An updating form using setf and these access functions is used to change the values in the slots, e.g.:

```
(setf (position-x-coord i-1) 13)
```

In addition, defstruct can define an extension of apreviously defined class.

```
(defstruct (3d-position
            (:include position))
  (z-coord 0))
```

The new structure is a subclass of the old, and includes all of its slots and may add slots of its own. Thus 3d-position has slots x-coord, y-coord, and z-coord, and inherits all methods defined on position.

Meta-classes

In CommonLoops, as in Smalltalk, classes are themselves instances of other classes. These special

classes are known as meta-classes. The figure below indicates the relationships of the classes defined above, and their meta-class `structure-class`.

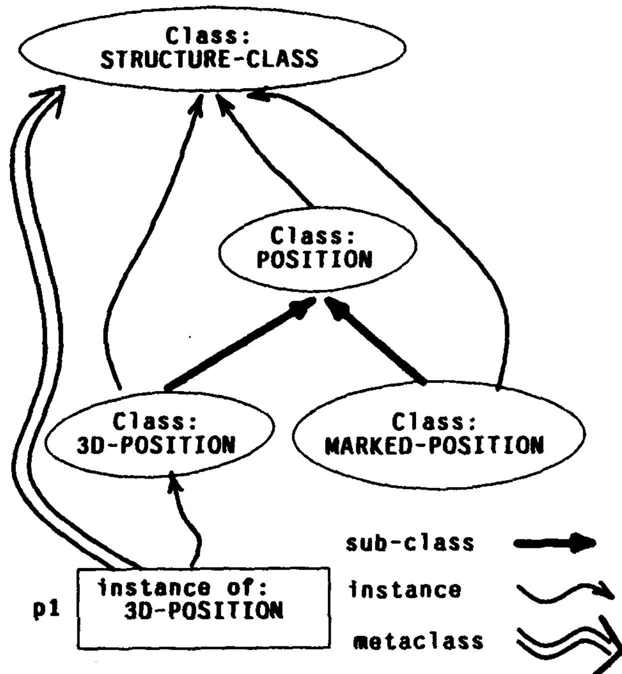

Figure 2: Three different relations are illustrated in this diagram. 3d-position and marked-position are both subclasses of position, and inherit its structure and behavior. p1 is an instance of 3d-position, the three position classes are instances of structure-class. We call structure-class the "meta-class" of p1, since it is the class of its class.

Meta-classes control the behavior of the class as a whole, and the class-related behavior of the instances such as initialization, as do Smalltalk meta-classes. In Flavors, the Flavors themselves are not instances of any Flavor, and hence their behavior is uniform.

In CommonLoops, meta-classes have important additional roles. A meta-class controls the interpretation of the defstruct form; it also controls the representation of instances of the class; it specifies the order of inheritance for classes; finally, it controls allocation and access to instance slots.

Interpreting the Defstruct Form

Because some meta-classes need to provide defstruct options not provided by other meta-classes, CommonLoops separates the interpretation of the defstruct form into two parts. In the first part, the defstruct is checked to make sure it conforms to basic `defstruct` syntax and the meta-class is determined by looking for a :class option. Then `expand-defstruct` is called on the meta-class and the `defstruct` form. This allows the meta-class to process the `defstruct` form and interpret the options as it chooses.

Representation of Objects

Meta-classes control the representation of instances. Consider the following definitions of the class position:

```
(defstruct (position
             (:class structure-class))
  (x-coord 0)
  (y-coord 0))
```

```
(defstruct (position (:class class))
  (x-coord 0)
  (y-coord 0))
```

In the first definition, the `structure-class` meta-class is specified. An instance of position created using this definition will be represented as a linear block of storage with two data items. This very efficient in space. The second definition specifies the meta-class `class` which causes the instances to be represented in a more flexible way, with a level of indirection between a header and the storage for the data. This allows such an instance to track any changes in its class (adding or deleting intance variables) without users of the instance needing to do anything to update the instance. Automatic updating occurs when access to slots is requested. The instance can even change its class, and invisibly update its structure. Because the meta-class is responsible for the implementation of the instance, it is also responsible for access to slots of the instance. We return to this below.

Multiple inheritance

Many meta-classes allow multiple inheritance. These meta-classes extend the syntax of the :include defstruct option to allow a list of included classes. For example,

```
(defstruct
  (titled-window
    (:include (window titled-thing))))
```

defines a new-class, `titled-window`, which includes both `window` and `titled-thing` as super-classes. Under control of the meta-class, the new class will inherit slots from the super-classes. Although the usual inheritance for slots is to take the union of those specified in the included-classes, some meta-classes could signal an error if there were an overlap in

names.

The class being defined is the root of a sub-lattice from which descriptions are inherited. The specified order of the included classes determines a local precedence among the classes. This is used as the basis of the precedence relation for specificity. The specificity of classes with respect to this new one is cached in the class as an ordered list that we call the *class precedence list*.

The meta-class determines the algorithm for computing the class precedence list from the local precedences. The algorithm used by the meta-class `class` is left to right, depth first up to joins, with the constraint that the local ordering of any local precedence list must be maintained. Except for the last constraint, this is the same as the algorithm used in Loops. The constraint is violated when a local precedence list contains C1 before C2, and C1 is somehow a super of C2. In this case, CommonLoops signals an error. This algorithm produces the same ordering as the one used in New Flavors.

for structures of CommonLisp, the built-in types, and the meta-class `class` designed to facilitate exploratory programming [Sheil]. The user can define a new meta-class to provide other functionality for a different object system. For example, with Gary Drescher, we have looked at defining a metaclass that supports ObjectLisp [Drescher] inheritance and behavior.

Slot-Options in Class

The representation of instances used by `class` allows three additional allocation strategies for slots in addition to the usual direct allocation of storage in the instance. These provide facilities that have been found useful in a number of object systems. In each case, the user of the class does not need to change the form of access to change the form of allocation.

A `:class` allocation specifies that the slot is stored only in the class; no storage is allocated for it in the instances. Thus, the slot is shared by all instances of the structure. Updating the value in one instance is seen by all. This option provides functionality similar to class variables in Smalltalk and Loops, except that CommonLoops class variables share the same name space with instance variables.

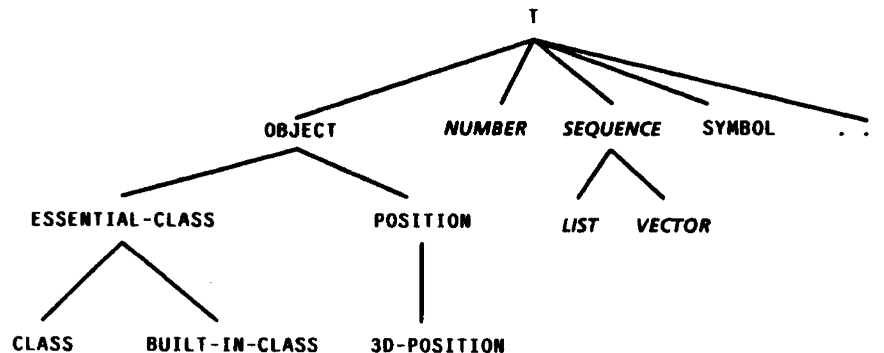

Figure 3. Classes in italics are instances of `built-in-class`, *all others are instances of* `class`. `t` *is the super-class of everything in CommonLoops. It corresponds to the Common Lisp type specifier of the same name.* `essential-class` *is a primitive class used to implement meta-classes. All meta-classes have* `essential-class` *as a super-class. It defines default behavior which all metaclasses must have.*

Initial classes in CommonLoops.

CommonLoops uses the flexibility provided by meta-classes to define classes which correspond to the primitive Lisp types. These classes are part of the same class lattice as all other CommonLoops classes. Thus the Lisp data-type space is included in the CommonLoops class lattice. This means that methods can be defined on the Lisp built-in classes as well as on types defined by defstruct. This is a significant difference from New Flavors.

As shown in Figure 3, CommonLoops provides several pre-defined meta-classes which provide functionality

A `:dynamic` allocation specifies that storage for this slot should be allocated in the instance, but only when the slot is first used. If the first access is a fetch, then storage is allocated, the `default-init` form is evaluated, the value is stored in the slot and returned. If the first access is a `setf`, then storage is allocated, the value is stored in the slot, and returned. This allows infrequently used slots to have initialization declarations, but take storage only if needed.

An allocation option of `:none` means that the slot should not exist in this instance. This is used to override inheritance of slots defined by a super class.

The meta-class `class` also allows objects to have slots that do not appear in the defstruct declaration. This gives objects their own property lists: this is analogous to Flavor's plist-mixin flavor. It differs from a plist-mixin in that there is uniform access to slots independent of whether they were declared.

III. CommonLoops Implementation

CommonLoops can be implemented efficiently, even on conventional machines. The most important cases for time-critical applications are well understood and have been implemented in several object-oriented systems.

Method Lookup.

Implementation of method lookup can be specialized with respect to four cases: where there is only one method defined for a particular selector, where the only method has no type specification, where all the methods have specifiers only on their first argument, and the general case.

Single Method: In this case there is only one non-default method defined on the selector. A static analysis of Loops and Flavors code shows that approximately 50% of the selectors fall in this category. In this case the method can be compiled with a type check to make sure it is applicable. The method-lookup time is only the time required to check the types of the arguments.

Default Method Only: This case is similar to the single method case except that the method has no type specifiers at all, so it is always applicable. In this case there is no type check required. It is implemented as a if it were defined as a function.

Classical Methods Only: When there are multiple methods, all of which only have type specifiers on their first argument the situation is the same as in Smalltalk and Flavors. We call this "classical" to stress its equivalence to classical object programming systems. On stock hardware this can be implemented using any of the proven method-lookup caching schemes. The cache can either be a global cache, a selector-specific cache, a callee cache, or a caller cache. Variations have been used in Smalltalk-80 systems [Krasner], Loops, and Flavors. On specialized hardware this can be implemented using the same mechanisms as in Flavors. A default method can easily be combined with a set of classical methods, calling it instead of a standard error.

Multiple Multi-Methods: In the remaining case, a selector has more than one method, and at least one of them has a type specifier on other than the first argument. A standard case might have type specifiers for the first two arguments, e.g. where the types for `show` could be

```
(square, display-stream)
(square, print-stream)
(circle, display-stream)
...
```

In our current implementation of multi-method invocation, we have implemented a straightforward extension of the caching techniques used for classical method lookup. We do not have enough experience with multi-methods to know what other common patterns should be optimized.

In classical object-oriented programming, this example could be handled by introducing a second level of message sending. Instead of having separate multi-methods for each case, one could (by convention) write two methods for each case. Thus, the `show` message for square would send a second message to the stream (`show-square-on`) that would embed the type information about square implicitly in the selector.

Multi-method lookup in CommonLoops is faster than multiple sequential method lookups. The overhead for doing lookup is the time of an extra function call (a call to the discriminator function which then calls the chosen method) plus the time of a type check for each specialized argument.

Slot access and Meta-classes

Slot access can be implemented in a variety of ways. The meta-class `class` uses a caching technique similar to that used in Smalltalk. The meta-class `structure-class`, because it does not allow multiple inheritance, can compile out the slot lookups in the standard way. Another meta-class could use the self-mapping-table technique used in Flavors.

We have also experimented with ways to compile out the cost of method lookup and slot lookup entirely. Having meta-classes and discriminator objects allows the specification of special ways of accomplishing a call to a selector when the types of some the arguments are known at compile-time. In certain cases the appropriate method can be determined at compile-time so that no method lookup need occur at

run-time. The body of the method can even be compiled in-line.

Compilation of calls to accessor functions is a common case where in-line expansion works well. The resulting code can access the slot directly. Meta-classes which do this kind of optimization are useful in production versions of applications where the time to change a program vs. program execution speed tradeoffs can be pushed completely towards execution speed.

Flexibility to use different slot-access or method-lookup schemes based on the meta-class is an important feature of CommonLoops. Efficiency is a matter of tradeoffs. Object systems without meta-classes must choose one set of tradeoffs and implement it as well as possible. Then, users have to live with it. In CommonLoops, several different sets of tradeoffs can be implemented as well as allowing users to choose which set of tradeoffs is appropriate for a given situation.

IV. CommonLoops and other Systems

In this section we consider several important object-oriented languages. All of these languages have been influential in the design of CommonLoops, and we try to note similarities and differences. A general overview of features of object languages and multiparadigm systems can be found in [StefikBobrow86]

Loops

Loops [BobrowStefik83] is a multi-paradigm system for knowledge programming implemented in Interlisp-D. It is integrated into the interactive environment provided by Interlisp-D. It also provides special environmental capabilities, such as class browsers and object inspectors. The design of CommonLoops draws on our experience with Loops, but is a major departure from it.

CommonLoops provides new functionality but also introduces many minor incompatibilities and lacks some functionality of Loops as discussed below. Features of Loops such as composite-objects that are appropriately implemented in terms of the CommonLoops kernel are not discussed. Modifying Loops to run on top of CommonLoops will require a substantial programming effort.

Class variables. Loops supports the notion of class variables that are accessed via special functions. CommonLoops provides class variables which provide nearly equivalent functionality. There are not, however, different name spaces for instance variables and class variables as there are in Loops. We now believe that the advantages for modifiability of a program outweigh the advantages of multiple name spaces.

Default values. Loops supports the notion of a default value which at slot access time finds the default value in the class or the super classes of the class. CommonLoops provides init-forms in slot-descriptions that specify how to compute the default value at creation time. The essential difference is that in Loops an instance tracks the slot description until given a local value while CommonLoops always gives a local value at creation time. The Loops behavior can be implemented in CommonLoops using annotated values as described in the section on open design questions. In our experience, initial values are satisfactory for most of the applications of default values.

Slot properties. In Loops a slot can have named properties in addition to a value. This provides a convenient way to store more information about a value without interfering with access of the value. This can be supported using annotated values.

Active values. In Loops a value can be active, so that specified functions can be run when a slot containing an active value is accessed. CommonLoops can provide comparable capabilies, as discussed below in the section on open design questions.

Smalltalk-80 System

The Smalltalk-80 system [GoldbergRobson] is both an object-oriented programming language and a vertically integrated programming environment that is uniformly object structured. The strength and importance of the Smalltalk-80 system rests not only with its object-oriented programming style, but also in the careful engineering of the set of kernel classes and their behavior that define the Smalltalk-80 image.

In terms of its provisions for class definition, name lookup, method discrimination, and method combination CommonLoops can be viewed as a superset of Smalltalk-80, with some notable exceptions:

The Smalltalk-80 virtual machine directly supports only single superclass inheritance. Nevertheless, additional inheritance schemes can be implemented (by changing the manner in which new classes are defined) and multiple superclass inheritance is

included as part of the standard Smalltalk-80 environment. It operates substantially the same as in CommonLoops, except that multiply inherited methods for the same selector must be redefined at the common subclass, or else an error will result when the method is invoked. This Smalltalk-80 feature is inconvenient for mixin classes that specialize standard methods as used in Flavors and Loops.

The Smalltalk-80 multiple inheritance scheme provides an explicit scheme for method combination: objects can send messages to themselves in a way that specifies from which superclass method lookup is to proceed. This is done by composing the name of the superclass with the selector, e.g. an instance of ReadWriteStream may send itself the message ReadStream next to indicate that the ReadStream superclass is to supply the method. This explicitness can cause problems because methods build in as constants information about the class hierarchy, which may change.

Classes and meta-classes bear the same relationship to each other and there is some overlap of function in both systems. However, there are some significant differences in functionality. Instances of all Smalltalk classes (except for the compiled method class) are realized in terms of just three basic implementations: pointer objects, word objects, and byte objects. The class definition directly determines which implementation is to be used. By convention in Smalltalk each class has a unique meta-class.

In Smalltalk enumerating the instances of a class is intended to be computationally bearable (just how bearable depends on implementation dependent factors, e.g. whether and how virtual memory is implemented.) As a result, Smalltalk classes can broadcast to their instances. This makes them extensional, as well as intensional, characterizations of sets of objects. Since even integers have a class in CommonLoops, it is not generally useful to enumerate all instances of every class. It is straightforward in CommonLoops to implement a meta-class that allows a class to keep a list of instances it has created.

In the Smalltalk-80 system, one can find all references to an most types of objects. It is even possible to interchange all the references to an object with all the references to some other object, regardless of their respective classes. In effect, the two objects exchange identities. This operation is inexpensive if the object memory indirects references via an object table, which is the standard practice. This capability enables, among other things, cheap re-sizing of instances of variable length classes. In CommonLoops, instances of classes created by the meta-class `class` can easily modify their contents and class pointers to achieve the same functionality.

Smalltalk provides class variables, which are shared by all the instances of a class and its subclasses, and pool variables, which are shared by all instances of some set of classes and their subclasses. The effect of class variables is directly achieved in CommonLoops through the :allocation class slot option. The effect of Smalltalk's pool variables can be achieved through the expedient of defining a common superclass among the classes to be "pooled", which contributes nothing but a shared slot.

Smalltalk differs more fundamentally from CommonLoops in that Smalltalk objects are encapsulated, and control primitives are based upon message passing. In Smalltalk, unlike CommonLoops, only methods of an object can access and update the state directly (this is not strictly true, but the operations provided for breaking encapsulation are viewed as just that, and used primarily for building debuggers, viewers etc.). All other methods must send messages.

Conditionals, iteration, and the like in Smalltalk are done via message passing, and contexts (stack frames) are first class objects. CommonLoops relies upon the Common Lisp control constructs which in general are special forms and cannot be specialized.

New Flavors

CommonLoops is practically a superset of New Flavors. CommonLoops and Flavors share the notion of generic function. In developing CommonLoops we have included the Flavors mechanism for user-defined method combination.

The New Flavors algorithm for computing class-precedence is a refinement of the old Flavors algorithm which solves problems found in old Flavors, Loops and earlier versions of CommonLoops. We have described our equivalent to the New Flavors algorithm.

To be entirely compatible with New Flavors, CommonLoops would need to provide some syntactic support for the mechanisms for defining classes and methods. Machine dependent support is also necessary (and easily added) to provide the performance on microcoded machines.

The important difference between CommonLoops and New Flavors is the existence of meta-objects in

CommonLoops. Meta-objects make CommonLoops much more extensible. Meta-objects allow experimentation with other kinds of object systems. They allow CommonLoops to treat primitive Lisp types as classes. Methods can be defined on those types, and the standard CommonLoops mechanisms for accessing the slots of a structure can be used to access the fields of primitive Lisp objects.

Other Object languages

ObjectLisp [Drescher] also integrates objects and Lisp. Unlike CommonLoops, ObjectLisp distinguishes fundamentally between Lisp types and ObjectLisp objects. This means that one cannot define methods on existing types. Another difference is that ObjectLisp supports only classical methods.

T shares with CommonLoops the common syntax for message sending and function call. Like ObjectLisp, T supports only classical methods and there is no integration of Lisp types with objects. [ReesAdams]

IV. Open design questions

In this section we present some extensions to CommonLoops which seem attractive, and which suggest directions for future research.

Complex type specifiers

Extending CommonLoops to handle more complex type specifiers is attractive. The simplest extension is to allow logical combinations of the simpler type specifiers, for example

```
(or block window)
```

Another extension would be to allow an arbitrary predicate to be used. Yet another extension would allow specification based on the number of arguments.

The problem occurs when there are ambiguities about which method to use. For example, which of the two:

```
(or block window) (or block house)
```

is more specific with respect to `block`.

For this reason, we have chosen to disallow method type-specifiers which cannot be ordered by specificity in the kernel of CommonLoops. A user who wants to add such methods to CommonLoops can do so by defining a special method class and using the method-lookup protocol to specify different method-lookup rules.

We believe that handling incomplete type specifiers and the possible resolutions (backtracing, unification, production system rules) is a fertile area for language design and research, and perhaps a foundation for a graceful merger of Lisp, Prolog and production systems.

Structural versus Procedural Views of Objects

The object-oriented programming community is split on the issue of whether the specification or interface description of a class of objects should be strictly procedural, or whether it should be split into procedural and structural parts. In the procedural view an object is defined by its message protocols. As a matter of principle, programs that interact with an object should make no assumptions about the internal representations of the object. A procedural view of a complex number is defined in terms its response to messages such as `x`, `y`, `rho`, `theta`, `plus`, `print` etc. CommonLoops continues the Common Lisp convention by usually generating access methods for structural components.

A description which includes a structural description could include the fact that `x`, `y`, `rho`, and `theta` are structural components of a complex number, that is, these named pieces are intended to have memory-cell semantics. Notice that the structural description need not be isomorphic with the implementation structure. For example, a complex number may be implemented as a pair `x` and `y`, with `rho` and `theta` computed.

Those that favor including a structural description argue that language forms should support this way of thinking. In CommonLoops, a uniform procedure `ref` is available to access structural parts by name, e.g.

```
(ref some-complex 'theta),
```

instead requiring the use of an access function for each

```
(complex-number-theta some-complex).
```

Use of the `ref` form allows one to write code that can iterate through the slots of a structure, for example, as in a comparison routine. Also, in CommonLoops, the developmental meta-class supports slots in instances that are not declared in the class. The `ref` form provides a uniform way of accessing both declared and undeclared slots. The `ref` form has some advantages with respect to expressivity.

```
(ref x y)
```

is equivalent to the wordier

```
(funcall
  (find-accessor-function
    (class-of x)
     y)
  x y).
```

A `ref` form is also generally an appropriate first argument for `setf`, unlike `funcall`, because `ref` is used with memory-cell semantics.

A shortcoming with a structural description is that sometimes one wants procedures to be invoked upon access to a structural component. A way to achieve this within the structural framework uses annotated values as described below.

Annotated Values --- Views and Implementations

Access-oriented programming is one of the popular features of Loops and several frame languages such as KEE, UNITS, and STROBE. The merits of this feature are often confounded with the merits of its various implementations. In this section, we try to separate these issues, and indicate alternative implementations available in CommonLoops.

In access-oriented programming, fetching from or storing in an object can cause user-defined operations to be invoked. *Procedural annotations (or active values)* associate objects with slots so that methods are invoked when values are fetched and stored. It is also useful to associate other information with a slot in addition to its value. *Structural (or property) annotations* associate arbitrary extendible property lists with a value in an object. Collectively these kinds of annotations are called annotated values. These annotations can be installed on slots and can be nested recursively.

Annotated values reify the notion of storage cell and are a valuable abstraction for organizing programs. Structural annotations can be used for in-core documentation. They are also used for attaching records for different purposes. For example, such annotations can record histories of changes, dependencies on other slots, or degrees of belief. Procedural annotations can be used as interfaces between programs that compute and programs that monitor those computations. For example, they can represent probes that connect slots in a simulation program to viewers and gauges in a display program.

Annotated values are conveniently represented as objects, and must satisfy a number of criteria for efficiency of operation and non-interference [Stefik]. When multiple annotations are installed on the same value, the access operations must compose in the same order as the nesting. Annotated values can be implemented in different ways that optimize performance depending on the expected patterns of common use.

One implementation of annotated values in CommonLoops would require the slot-access primitives of the meta-class check whether the value is an active value object. The active value check can be made fast if the active value objects are wrapped in a unique data type. This technique for implementing active values has been used sucessfully in Loops. Hardware or microcode support of this fast check would allow the use of annotated values in ordinary Lisp structures (e.g. in cons-cells), greatly extending the utility of this abstraction.

Alternatively, a procedural implementation of annotated values could be built upon the ability in CommonLoops to specialize methods with respect to individuals. For those slots for which a special action is desired upon access, one can define methods for those accessors and objects that do the special action.

CommonLoops is capable of supporting either implementation. In addition, we believe that it is appropriate in CommonLoops to provide meta-classes that can support annotated values according to the needs of optimization. If active values are to be attached and detached frequently, checking dynamically for annotated values may be preferable to changing the discriminator frequently. If probes are usually installed only once, then one may prefer the lower overhead of the procedural implementation. If access to properties is relatively rare compared with the access to values, then differentiating property access at compile-time might be preferred.

It is useful to be able to view a program that uses annotations in terms of that abstraction, rather than in implementation terms. The issue of supporting views of programs is discussed more generally in the next section.

Programming Environment Support

Programming environments must provide computational support for particular views (or perspectives) of programs [BobrowStefik86]. A view is said to support a particular programming abstraction when the elements of the view are in the terminology of the abstraction and the operations possible within that viewer are those appropriate for the abstraction.

For example, a viewer that supports the view of a program in terms of annotated values would show annotated values, not methods or wrappers that make up their implementation. The installation and nesting of annotated values are the appropriate actions available in the viewer.

Another important and popular view of object-oriented programs is that classes are defined by their slots and methods. While program listings often show structure and methods separated, it is useful to view such programs as organized in terms of classes with access to slot and method descriptions. CommonLoops

viewers in definition groups [Bobrow] also provide access to any multi-method from all of its associated classes. Thus, CommonLoops supports the classical view with appropriate extensions.

Views of classes can be organized around semantic categories, as in the standard Smalltalk-80 browser, or around a graph of the class inheritance lattice of some portion of the system, as in Loops, and Commonloops. In the latter case, certain operations become natural to perform directly through the lattice browser -- for example, promoting methods or slots to more general classes, or changing the inheritance structure. Changing the name of a slot or selector through a browser can invoke analysis routines that can find and change all occurrences of the name in code.

Viewers on CommonLoops also support a procedural abstraction. For example, they provide static browsers of program calling structure, where each discriminator is considered as a single function. However, through these browsers, one can get access to individual method definitions from the corresponding discriminator.

To provide viable support for programming with an abstraction, the viewers must be integrated with the debugging system. For example, to support a view of program in terms of methods, it should not be necessary to understand how methods are implemented or to refer to methods created automatically by the system. Rather, debugging should use the same terms that the programmer uses in writing the program.

V. Summary and Conclusions

Over the last ten years many systems have been written that add object-oriented programming to Lisp (e.g., Flavors, Loops, Object-LISP.) Each of these has attracted a group of users that recognize the benefits of message sending and specialization and have endorsed the object-oriented style. The object-languages in these systems have been embedded in Lisp with different degrees of integration.

Interest in object-oriented programming has also been spurred by work in expert systems. Several knowledge programming systems (ABE, ART, KEE, Strobe, UNITS, etc.) have emerged. These systems have included variations and extensions on object-oriented programming and tools for creating knowledge bases in terms of objects. As research continues, additional knowledge programming systems will emerge. Each of these will have their advocates and perhaps their niche in the range of applications and computer architectures. All of these systems can benefit from an object-oriented base that is efficient and extensible.

The creation of a good base involves both theoretical language design and engineering concerns. CommonLoops has attempted to respond to several kinds of pressure on the design of such a system.

The applications community wants to use a system for its work. The language must be suitable for state-of-the-art applications and systems that they build on top of it. The language must have an efficient implementation. It is an advantage if the language is a graceful extension of Common Lisp because existing code and existing programming skills can be preserved.

Vendors share these interests. They want their systems to provide a suitable base for a large fraction of the applications. They want the kernel of the language to be lean, easy to maintain, and efficient; they want the kernel to be principled and free of idiosyncratic features with no enduring value beyond their history. Vendors don't want to implement multiple versions of object languages, gratuitously different and incompatible.

The research community has somewhat different interests. Like the application community, it needs to be able to share code, but it is concerned with being able to try out other ways of doing things. New ideas for languages come out of the experience of the research community. To build higher level languages, the base must provide mechanisms for open-ended experimentation.

CommonLoops has responded to these pressures by providing a base for experimentation through the use of meta-objects, while capturing in its kernel the ability to implement the features of current object-oriented systems. By integrating classes with the Lisp type system, and using a syntax for method invocation that is identical to Lisp function call, CommonLoops makes possible a smooth and incremental transition from using only the functional paradigm for user code to using the object paradigm. As a portable system implemented in a widely available base, it allows users the choice of hardware and environments, and a road to the future.

Acknowledgments

Many people read early drafts of this paper, helped us to sharpen the ideas and present them more coherently. Thanks to Eric Benson, John Seely Brown, Margaret Butler, Johan de Kleer, Peter Deutsch, Richard Gabriel, Stanley Lanning, Henry Lieberman, Mark S. Miller, Sanjay Mittal, Randy Trigg, Bill van Melle, Daniel Weld, and Jon L. White.

References

[Bobrow] Bobrow, D. G., Fogelsong, D. J., Miller, M. S., *Definition Groups, Making Sources First Class Objects,* ISL Report, Xerox PARC, 1986

[BobrowStefik83] Bobrow, Daniel G., Stefik, Mark. *The Loops Manual,* Intelligent Systems Laboratory, Xerox Corporation, 1983

[BobrowStefik86] Bobrow, Daniel G. and Stefik, Mark. "Perspectives on Artificial Intelligence Programming", *Science* V231, No 4741, p 951 February 28, 1986

[Drescher] *ObjectLISP User Manual*, LMI, 1000 Massachusetts Avenue, Cambridge, MA 02138.

[GoldbergRobson] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA 1983.

[MoonKeene86] Moon, D., Keene, S, *New Flavors.* ACM 1986 OOPSLA Conference

[Krasner] Glenn Krasner, Ed. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, Reading, MA, 1983.

[ReesAdams] Rees, , J. A. and Adams, N. I. T: a dialect of Lisp or, lamda: the ultimate software tool, ACM Symposium on Lisp and Functional Programming, 1982

[Sheil] Sheil, B. "Power Tools for Programmers", in Barstow, D. et al (editor) *Interactive Programming Environments*, McGraw Hill, 1984

[Steele] Steele, G.L. *Common Lisp: the language.* Digital Press.1984.

[StefikBobrow86] Stefik, M., Bobrow, D.G. Object-oriented Programming: Themes and Variations. *AI Magazine* 6:4, Winter 1986.

[Stefik] Stefik, M., Bobrow, D.G., Kahn, K. Integrating access-oriented programming into a multi-paradigm environment, *IEEE Software*, 1986.