LISP AND SYMBOLIC COMPUTATION, 10, 39–60 (1997)

Adding Support for Persistence to CLOS via Its Metaobject Protocol

ARTHUR H. LEE alee@psl.korea.ac.kr
Department of Computer Science, Korea University, Seoul 136-701 Korea

JOSEPH L. ZACHARY zachary@cs.utah.edu
Department of Computer Science, University of Utah, Salt Lake City, UT 84112 USA

Abstract. Language-level support for object persistence frees programmers from having to confront a broad class of database issues from within their applications. By virtue of its metaobject protocol, CLOS is a language whose semantics can be tailored by individual programmers. We used the metaobject protocol to extend CLOS with support for object persistence. Our goal was to obtain a version of CLOS with persistence to which we could easily port a commercial geometric CAD modeling system. We describe the design and implementation of our persistence extension and highlight the strengths and weaknesses exhibited by the CLOS metaobject protocol during our experiment. For many aspects of the implementation we found that the metaobject protocol was ideal. In other cases we had to choose among paying a large performance penalty, extending the protocol, and bypassing the protocol by modifying the language implementation directly.

Keywords: Open Implementation, Metaobject Protocol, Object Persistence, CLOS

1. Introduction

Language-level support for object persistence frees application programmers from having to deal with a broad class of software engineering concerns. Without adequate support for object persistence at the language level, programmers must confront database issues from within their applications. We faced this issue within the context of the Conceptual Design and Rendering System (CDRS) [14, 15], a geometric CAD modeling system, written in the Common Lisp Object System (CLOS), that is being used in dozens of major automotive and product design companies worldwide.

CLOS [3] has an open implementation by virtue of its metaobject protocol. By encapsulating a fundamental portion of its semantics within a set of default classes and empowering the programmer to derive new versions of these classes, the designers of CLOS have provided a language whose semantics can be tailored by individual programmers.

We used the metaobject protocol to extend CLOS with support for object persistence. Our goal was to obtain a version of CLOS with persistence to which we could easily port CDRS. We originally wanted to modify CLOS strictly via the metaobject protocol, so that no changes to the compiler, runtime system, or the existing application program would be required. Although we ultimately compromised slightly on this point and devoted considerable engineering effort to the implementation, the final product, although fully expressive, was judged too inefficient for commercial use.

In this paper we will focus on describing the design and implementation of our persistent CLOS and highlight the strengths and weaknesses exhibited by the CLOS metaobject

protocol during our experiment. (A detailed presentation of the design and an analysis of performance measurements appears in [17].) We will also describe some of the main issues that we had to resolve while adding persistence to CLOS. Adding persistence to CLOS is no small undertaking, and the metaobject protocol is quite general, so we are convinced that our experience is relevant to programming at the metalevel in general. For many aspects of the implementation we found that the metaobject protocol was ideal. In other cases we had to choose among paying a large performance penalty, extending the protocol, and bypassing the protocol entirely by modifying the language implementation directly. Some of these difficulties were language-specific (due to CLOS) while others were problem-specific (due to the nature of implementing persistence).

The remainder of this paper is organized as follows. In Section 2 we discuss object persistence, open implementation, the CLOS metaobject protocol, and our approach to adding persistence via metalevel programming. In Section 3 we describe in detail how we used the metaobject protocol to add support for object persistence. In Section 4 we describe a minor extension that we had to make to the protocol to deal with one level of indirection on slot accesses. In Section 5 we detail how incremental saves of objects to the database are handled, and in Section 6 we describe how we cope with shared, structured, non-object data. After we survey other uses of programming at the metalevel in Section 7, we conclude in Section 8.

2. Background

2.1. Motivating Application

The Conceptual Design and Rendering System (CDRS) [14, 15] is a geometric CAD modeler that is being used by designers in dozens of major automotive and product design companies worldwide. The work presented in this paper was initially motivated by the problems of object persistence encountered in CDRS. CDRS is an object-intensive application written mostly in Common Lisp [29] as extended by CLOS. A typical model manipulated by CDRS contains tens of thousands of objects that may not all fit into memory, exhibits a wide variation in the sizes of objects, requires complex data structures within objects, and has rich semantic and structural relationships among objects. Supporting object persistence is particularly difficult in the presence of these characteristics.

CDRS uses a naive file-based, batch-oriented approach to object persistence that has proven ill-suited to the mix of objects that it manipulates [16]. All objects in a model are saved to a file at the end of a design session and are reloaded at the beginning of the next session. This approach requires a huge amount of physical memory, frequent large garbage collections, and a long time to load and save models. CDRS uses 500 megabytes of swap space, requires up to 128 megabytes of main memory, and spends almost 30 minutes loading or saving a typical model. To make matters worse, users tend to save models frequently for fear of losing them due to reliability problems.

The solution that we sought was to introduce a tighter interface with a database system by providing incremental saves and loads. We attempted to do this not by modifying CDRS but by modifying CLOS via its metaobject protocol.

2.2. Open Implementation

Traditionally, black box abstraction has been used to control the complexity of a software module by exposing its functionality while hiding its implementation. Recently, however, researchers have explored a different kind of modularity called *open implementation* [10, 11, 12, 13, 19].

Under black box abstraction, the implementation decisions encapsulated within a module are generally made in the light of assumptions about the way in which the client will ultimately use the module. Kiczales [11] terms such decisions, necessarily made in the face of incomplete information, *mapping decisions*. A *mapping conflict* occurs when the assumptions made by the implementor work at cross purposes to an eventual client's actual needs. Clients typically resolve mapping conflicts by adding extra code to their applications to compensate for the mapping decisions made in the module. Kiczales calls this "coding between the lines."

The open implementation approach separates implementation decisions into *base* and *meta* parts. The base part of the implementation is closed as in a black box abstraction. The meta part is open to modification by clients via the meta interface specified by the module designer, who carefully decides what will be open and what will be closed. An open implementation thus has both a conventional interface (exposing functionality) and a meta interface (exposing aspects of the implementation).

2.3. The CLOS Metaobject Protocol

CLOS has an open implementation, and the CLOS metaobject protocol is the language's meta interface. It is implemented in an object-oriented fashion by exploiting reflective techniques [26, 27]. Via the metaobject protocol, users can alter the semantics of CLOS by using the standard object-oriented techniques of subclassing, specialization, and method combination.

Programmers build applications in CLOS by making use of the five basic elements of the language: classes, slots, methods, generic functions, and method combinations. Each such element in a program is represented as a CLOS object. This object is called a *metaobject*, and its class is called a *metaclass*. For example, a user-defined class `student` will be represented as a metaobject that contains the structure and gives the semantics of the `student` class.

Because of this organization, the default semantics of CLOS can be given by the implementations of five metaclasses, one for each of classes, slots, methods, generic functions, and method combinations. The larger part of these implementations comprise the *base* part of CLOS.

The five metaclasses from which metaobjects are made behave much like other classes in CLOS. Thus, one can change the semantics of a metaobject by modifying its metaclass. Although a user class definition can be freely changed, a metaclass definition can only be changed *incrementally* via subclassing, specialization, and method combination. Those aspects of the language that can be changed in this fashion comprise the *meta* part of CLOS. This meta interface—the metaobject protocol—is part of the CLOS definition.

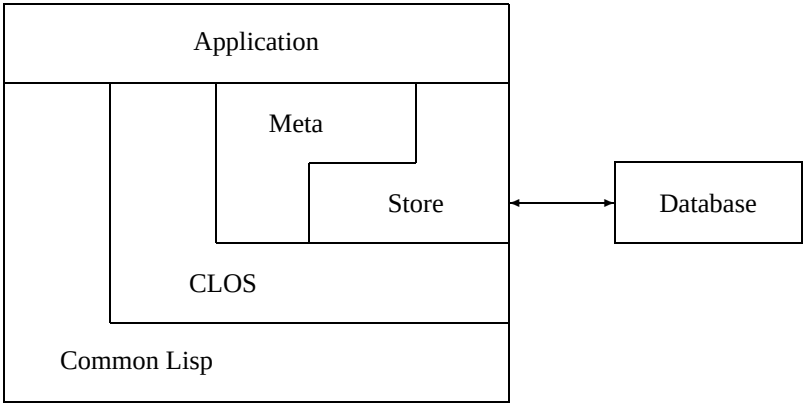

Figure 1. MetaStore architecture

2.4. MetaStore

We used the metaobject protocol to add persistent objects to CLOS. The resulting system, MetaStore, has two major components: a language extension portion (`Meta`) that was implemented via the CLOS metaobject protocol, and a persistent object store (`Store`) that provides database management. The organization of MetaStore is diagrammed in Figure 1. Our goal in extending CLOS and creating MetaStore was to incorporate persistence into the existing CDRS implementation in as transparent a manner as possible. We will focus in this paper on describing the language extension portion and detailing the lengths to which we eventually went to provide transparent persistence via the metaobject protocol.

MetaStore supports the creation of persistent objects at the user level via inheritance. An application programmer specifies that the objects of some class, say `C`, are to be persistent by specifying the metaclass of `C` to be `persistable-metaclass` rather than the default `standard-class`. The `standard-class` metaclass encapsulates the properties that all user-defined classes have in common, and `persistablemetaclass` makes persistence one of those properties.

We distinguish *transient objects* from *persistable objects*. A transient object is an object in the conventional sense, whereas a persistable object is one whose value can persist between sessions. Based on our experience with CDRS, there are many classes whose objects need not be saved because they can be easily reconstructed. In the case of CDRS, treating all the data in a program as persistable, as is done in PS-Algol [6] and by Jacobs [9], would have been both unnecessary and impractical.

A persistable object becomes a *persistent object* when it is eventually saved to the object base. This distinction is important, since our experience with CDRS shows that about 90% of the data that is created is never written to the database; instead, it is collected as garbage. We also distinguish between persistable and transient slots within a persistable object. Only the persistable slots of a persistable object are ever saved to the object store; the contents of transient slots are discarded.

Finally, we distinguish atomic and composite slots. An *atomic* slot is one whose value is of an atomic data type, whereas a *composite slot* has as its value composite data such as arrays and lists. In CDRS, we found that over 85% of the space occupied by a typical object was consumed by a small number of large slot values. Saving and restoring an object as a single entity, including all of these large slot values, would have been impractical. Instead, we decided to save and restore composite slot values as separate entities.

The application programmer's interface to MetaStore is illustrated by the class definition below.

```
(defclass student ()
  ((name  :initform "")
   (id    :initform -1)
   (major :initform 'undecided)
   (hobby :initform 'guitar :TRANSIENT T))              (1)
  (:METACLASS PERSISTABLE-METACLASS))                   (2)
```

Line (2) of the `student` class causes MetaStore to treat `student` objects as persistable, while line (1) specifies that `hobby` is a transient slot. Our original goal was that the inclusion of `:METACLASS` and `:TRANSIENT` options be the *only* hooks from an application into MetaStore. As we detail in the next section, we ultimately retreated somewhat from this goal in the face of compelling efficiency concerns.

A persistable object, like any other object, is created by a call to the CLOS method `make-instance`. In MetaStore, when a persistable object is created, it is assigned a unique persistent identifier (PID). The persistable object in Figure 2 contains a PID slot in addition to the user-defined atomic slot `a` and composite slot `b`. A unique PID is necessary to map physical addresses to logical addresses as objects are saved to the object base, and to map from logical addresses to physical addresses as objects are loaded from the object base. Address translations in MetaStore are done by pointer swizzling [16].

When an atomic persistable slot of a persistable object is modified, the *dirty bit* of the object is set (see Figure 2). This slot, just like the PID, is added automatically when a persistable object is created.

Composite slots are handled differently. Upon creation of a persistable object, an intermediary data structure called a *phole* (*p*ersistent *hole*, much like a placeholder in MultiScheme

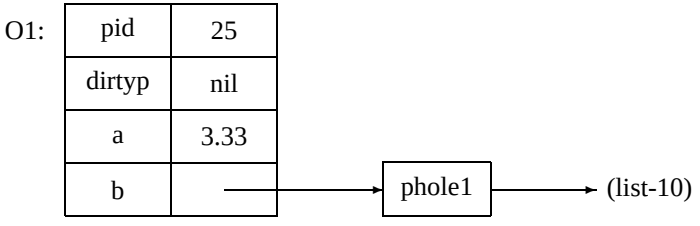

Figure 2. A persistable object in MetaStore

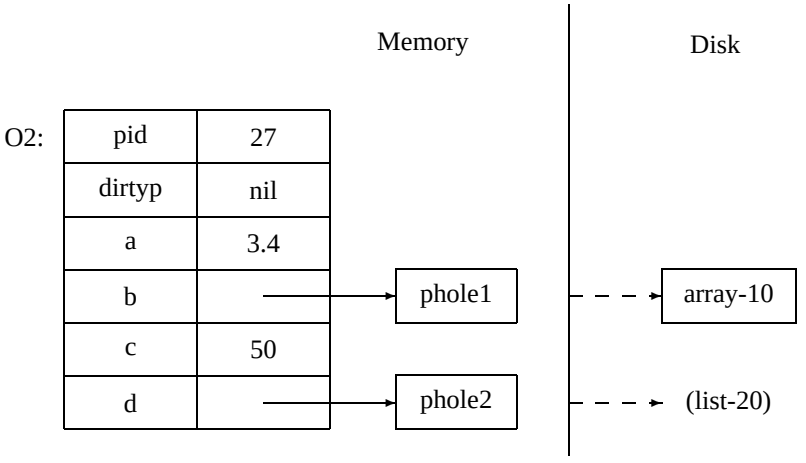

Figure 3. A husk object

[20]) is added between the object and each of its persistable composite slot values. A dirty bit for the composite slot is maintained in its phole. When the slot value is modified or mutated it is the phole, instead of the object, that is marked as dirty. Thus, MetaStore uses a two-level dirty bit scheme to support its slot-level persistence granularity.

The dirty bits in the pholes within an object are used to determine which persistable composite slots must be written to the database when an object is saved. The persistable atomic slots of a dirty object are saved as an object, and the dirty composite slots are saved separately. All newly created-objects and composite values are treated as dirty, of course, until they are first saved.

MetaStore implements persistence by maintaining a virtual object space within virtual memory. As the object space fills, MetaStore flushes some of the objects to disk and makes their virtual images available for garbage collection. When an object is *flush*ed, only the values of its composite slots are saved to disk and removed from virtual memory. The object that remains, with its composite slots referring to uninstantiated pholes (Figure 3), is called a *husk*[1].

When the process that created an object terminates, the husk itself is finally saved (if necessary) and removed from virtual memory. When another process loads the saved object, only the husk is initially loaded (unless the object is requested to be fully instantiated). The values of the composite slots are loaded lazily as demanded by the application. From an application's point of view, this approach to loading, along with the incremental saves, amortizes the costs of saving and loading models over an entire design session, thus reducing the user's waiting time.

3. Structure and Behavior of Persistable Objects

The metaobject protocol is ideal for making language extensions that involve modifications to the structure of objects or simple changes to their behavior. We made a number of such extensions in MetaStore, including such things as maintaining PIDs and dirty bits, differentiating transient and persistable objects, and specializing the behavior of read and write accesses. We describe some of the key modifications in this section.

3.1. Persistable Class Metaobject Class

The first step in modifying the behavior of objects is to define `persistablemetaclass` as a subclass of `standard-class`. Whereas `standard-class` encapsulates the standard behavior of classes as defined by CLOS, `persistable-metaclass` is used to modify that behavior in support of persistence. The new class is an example of a specialized class metaobject class [12].

```
(defclass persistable-metaclass
          (standard-class)
  ())
```

This new class, of course, specifies the same structure and behavior as its superclass at this point. Aspects of the behavior of each instance (i.e., object) of a user-defined persistable class are governed by an instance (i.e., metaobject) of this metalevel class. By adding to `persistable-metaclass` using the mechanism established by the metaobject protocol, we were able to make the extensions described in the remainder of this section.

3.2. Persistence via Inheritance

MetaStore treats transient and persistable objects differently by using inheritance to differentiate the two kinds of objects. MetaStore defines `persistable-root-class` and makes it a superclass of each persistable user class. A user class does not have to specify `persistable-root-class` explicitly as a superclass, and thus it remains invisible to user programs. Instead, the `initialize-instance` phase of class definition is modified via the metaobject protocol to incorporate the superclass automatically.

The purpose of `persistable-root-class` is twofold. Structurally, it adds extra information such as a persistent ID (`pid`) and a dirty bit (`dirtyp`) to each object:

```
(defclass persistable-root-class ()
  ((oid    :initform (make-oid ...))
   (dirtyp :initform t ...)
   )
  (:metaclass persistable-metaclass))
```

Behaviorally, `persistable-root-class` provides the following functionality for all persistable classes:

- It provides the default method for checking the consistency of objects, which MetaStore runs before an object is saved. This default method is a dummy routine; it exists to provide a method that application programs can specialize via inheritance. It is common for an object in CDRS to have "wrong" data; this method affords a way for an application to detect and repair problems before an object is saved.
- It handles flushing out objects as required by the virtual object memory manager. For a detailed description of the virtual object memory management algorithm see [16].
- It handles encoding objects for saving and decoding them for loading. Address translations are done as a part of this process.

3.3. Persistent Identity and Creating Persistable Objects and Husks

MetaStore customizes the behavior of the `make-instance` method. When a persistable object is created by a call to `make-instance`:

- Pholes are added to each persistable composite slot.
- The necessary information for the virtual object memory is recorded.

When an object is loaded, it is loaded as a husk. Husk creation is handled somewhat differently from object creation. When an object is created via a call to `make-instance`:

- An instance is allocated by the protocol routine `allocate-instance`.
- The allocated instance is initialized with values specified for `initforms` in slot definitions.

When a husk `O` is to be created from an object stored on disk, we could do so by calling `make-instance` and then replacing the atomic slot values of `O` with the saved values and the composite slot values of `O` with empty pholes. This would waste time and space because the values specified by the `initforms` would be computed and immediately discarded. Instead, a husk is created as follows:

- An instance is allocated by the routine `allocate-instance`.
- Only the transient slots are initialized with values specified for `initforms` in slot definitions.

3.4. Accessing Objects

Each read or write access is intercepted by the metaobject protocol so that appropriate persistence-related actions can be performed.

On a read access, if the accessed slot is a persistable composite slot that is not yet loaded, then the value of the slot is read in from disk. If the slot is a transient or atomic slot, then

the value should already be in memory and is returned. All this is handled by modifying the behavior of the `slot-value-using-class` method, which is the workhorse of the user accessible routine `slot-value`. This is done by defining an `:around` method on `slot-value-using-class`.

A write access is more complicated than a read access. On a write access to a non-transient slot, the following are taken care of by MetaStore:

- If both the current and new values are atomic, set the object's dirty bit.
- If the current value is atomic (or uninitialized) and the new value is composite, add a phole to the slot with its dirty bit set. Set the dirty bit of the object as well.
- If the current value is composite and the new value is atomic, remove the phole and set the object's dirty bit.
- If both the current and new values are composite, set the phole's dirty bit.

All of this is handled by modifying the behavior of `(setf slot-value-usingclass)`, which is the workhorse of the user accessible routine `(setf slot-value)`. This is done by defining an `:around` method on `(setf slot-value-using-class)`.

3.5. **Persistable** *Slot-Definition* **Metaobject Class**

When a class definition is processed, a *slot-definition* metaobject class is created for each slot. MetaStore defines an extra slot option, `:transient`, so that each slot can be declared as transient or persistable. This is supported in two places:

- `standard-direct-slot-definition`: The instances of this class hold intermediate, not yet fully processed slot-related information from the class definition form. We define `persistable-standard-direct-slot-definition` as a subclass of `standard-direct-slot-definition` with an extra slot, `transientp`, and its `:initarg, :transient`.
- `standard-effective-slot-definition`: The instances of this class hold slot-related information that has been fully processed (*finalized*) using inheritance rules. We define `persistable-standard-effective-slot-definition` as a subclass of `standard-effective-slot-definition` with extra slot `transientp` and its `:initarg, :transient`.

Thus far, we have taken care of the static parts. We also have to tell the system which *slot-definition* metaobject class should be instantiated to implement each persistable slot. We do this by giving implementations of `persistable-standarddirect-slot-definition` as well as `persistable-standard-effective-slotdefinition`. These are used by two generic functions: the former is used by `direct-slot-definition-class` and the latter by `effective-slot-definitionclass`. Both initialization and reinitialization of

instances are funneled to the generic function `shared-initialize`. Here, we first make the value of slot option `:transient` available for use.

There is one more thing to take care of. A rule for inheritance regarding transience of slots must be enforced. A slot is treated as transient only if all classes in the inheritance chain that define a slot with that name have the same declaration. (This was also done in [23].) This is done at the time effective slot definitions are computed by the generic function `compute-effective-slot-definition`.

4. Indirection on Slot Access

MetaStore supports persistence at the slot level, thus requiring us to maintain one level of indirection for each persistable composite slot. The contents of a persistable composite slot is a pointer to a phole, which (among other things) contains a pointer to the actual composite slot value.

When a user program issues a `slot-value` call to a persistable slot, MetaStore must follow pointers and return the value stored in the phole. The implementation of MetaStore, however, must sometimes directly obtain the phole via the same call. Supporting this behavior was not entirely straightforward.

The solution requires providing two different semantics for `slot-value` depending upon where and for what purpose it is called. The metaobject protocol provides no support for this differentiation. Solving this problem involved making minor modifications to the protocol. Specifically, we had to add an extra method for accessing slot values at the protocol implementation level. This kind of language-specific problem in CLOS could be avoided by a minor change to the design of the protocol.

5. Maintaining Dirty Bits

Only dirty (modified) persistable objects are ever saved to the database. Because the smallest grain size of persistence is the composite slot, each persistable object and persistable composite slot value has its own dirty bit. Maintaining the dirty bits of objects was straightforward. We will concern ourselves here with the much more challenging problem of maintaining the dirty bits of composite slots.

The dirty bit of a composite slot is kept in its phole, and must be set whenever the slot is written or its contents are modified. Doing this via the metaobject protocol proved difficult. Performing a write on a slot value via the public interface of the containing object, i.e., via `(setf slot-value)`, poses no problem. The problem occurs when programmers obtain a slot value via a *read* access (via `slot-value` or an accessor) and then mutate that value. The following code fragment demonstrates the problem.

```
(let ((arr1 (slot-value object1 'slot1)))
  (setf (aref arr1 3) 4.5))
```

Here, the value (an array) of the slot `slot1` is read and locally bound to `arr1`. The array is then modified. However, since this modification is not made through the phole associated

with `slot1`, the dirty bit in the phole cannot be set. To make sure that the dirty bit is set, the user program could do the following.

```
(let ((arr1 (slot-value object1 'slot1)))
  (setf (aref arr1 3) 4.5)
  (setf (slot-value object1 'slot1) arr1))          (1)
```

The extra call, labeled `(1)`, would solve the problem since `(setf slot-value)` can be easily modified via the metaobject protocol to maintain dirty bits.

Although we use an array as an example here, all composite values except objects pose this problem. The fundamental problem is that the Common Lisp functions that mutate structured values such as arrays cannot be customized via the metaobject protocol.

In the remainder of this section, an expensive but complete solution that maintains dirty bits without any help from either the application program or the compiler is described first, followed by a more practical solution that requires the cooperation of user programs. We ultimately implemented the simpler solution in MetaStore.

5.1. A Complete Solution

The goal is to support incremental saving without requiring application programs to deal with dirty bits at all. That is, we want the maintenance of dirty bits to be transparent to user programs. We describe an algorithm that satisfies this goal.

Although all non-object composite values pose this problem, we will concentrate on arrays, which are entirely typical. The basic idea behind this solution is to maintain dirty bits indirectly via a hash table. The solution requires that we

- maintain an *"eq"*-test hash table, `ht1`, containing dirty composite values and
- overload the write access routine for arrays, `(setf aref)`.

When an array `a1` is write accessed, as in:

```
(setf (aref a1 3) a-new-value)
```

`a1` is added to `ht1`. Since only dirty values are added to `ht1`, the number of entries will stay small if frequent incremental saves are made.

We overload `(setf aref)` as follows:

```
(defsetf aref (arr &rest subscripts) (new-val)
  `(progn (setf (gethash ,arr ht1) t)
          (setf (sysaref ,arr ,@subscripts) ,new-val)))
```

This new version of `(setf aref)` has the additional task of adding the array being modified to `ht1`. This modification assumes that the workhorse system routine for `aref` is `sysaref`. If we don't have access to this routine due to restrictions posed by the substrate vendor, as is the case with Lucid Common Lisp [18], then we must require our application programs to use a different name in place of `aref`, `paref`, and include the following definition instead:

```
(defmacro paref (arr &rest subscripts)
  `(aref ,arr ,@subscripts))

(defsetf paref (arr &rest subscripts) (new-val)
  `(progn (setf (gethash ,arr ht1) t)
          (setf (aref ,arr ,@subscripts) ,new-val)))
```

Thus, a user program would have to use `paref` and `(setf paref)` to access any potentially persistent array. Failure to do so would result in loss of data consistency.

When a persistable object is ready to be saved, we treat each persistable composite slot value as follows. We traverse the entire structure reachable from the slot, stopping when we reach objects. If any structured value that we encounter is in `ht1`, we treat the entire slot value as dirty. If the slot value is dirty, we save it as a unit into the object base and then remove all dirty substructures contained in the slot value from `ht1`.

This solution does not work properly if non-object structured data is shared among slots. We discuss this problem in detail in Section 6.

5.2. Feasibility of the Complete Solution

There are both performance and engineering problems with the solution that we have just described.

- Each write access to a composite value must bear the cost of adding an entry to the hash table.
- The entire structure of each persistable composite slot of a persistable object must be traversed to see if any substructure is dirty. Each composite value encountered during the traversal must be looked up in the hash table.
- Each time a model is saved, every persistable object in the entire system must be traversed.
- Under some implementations of CLOS, existing applications must be modified to use `paref` in place of `aref`.

5.3. A Practical Solution

The extra costs associated with the solution described in the previous section led us to a more pragmatic compromise solution that puts more of a burden on the application programmer. If a slot value is modified via the public interface, i.e., via `(setf slot-value)`, then there is no extra responsibility on the part of an application program. However, if a user program mutates a slot value, then it must inform MetaStore that there is a modification being made. There are two ways to do this:

- `(mark-dirty object1 &rest slot-names)`: For each slot in `slot-names`, `object1` is marked dirty if the slot is a persistable atomic slot, and the slot itself is marked dirty if the slot is a persistable composite slot. If no slot name is given, i.e., `slot-names` is `nil`, then `object1` and all the persistable composite slots are marked dirty.
- Instead of the usual `with-slots` macro of CLOS, user programs can use an alternative, `with-pslots`, with the following syntax and semantics.

```
(with-pslots (slot1 (slot2 :dirty) (slot3 :dirty)) object1
  (body-of-usual-with-slots))
```

  This specifies that sometime during the evaluation of the body of this construct, the values of `slot2` and `slot3` will be modified. `with-pslots` will signal to MetaStore that these slots are dirty and then call the usual `with-slots`. This way, if a persistable value is accessed many times in a tight loop, the dirty bit will be set only once.

6. Shared Structures

Structured data in Common Lisp can be shared freely. This greatly complicates the problem of supporting persistence of shared structures. We could find no acceptably efficient solution within the metaobject protocol. The central problem is that structures such as arrays and lists, unlike objects, cannot be given unique identifiers via the protocol.

To illustrate the problem, suppose a composite slot value, the array `a1` of the object `O1` in Figure 4, is ready to be saved. Also suppose that `a1` has another array, say `a2`, as one of its elements. Finally, suppose that a slot of another object `O2` also has `a2` as its value through a third array `a3`. Thus, `a2` is shared indirectly by `O1` and `O2`.

Assuming that only objects have dirty bits, and also assuming both `O1` and `O2` are dirty, if both `O1` and `O2` are saved, two copies of `a2` will be saved: once by `O1` and again by `O2`. When `O1` and `O2` are both loaded at some later time, `b` of `O1` and `c` of `O2` will have their own copies of the original array `a2`.

A complete solution that, while inefficient, allows persistable structures to be shared is described first, followed by the practical solution that was implemented in MetaStore in which non-object structured data are not allowed to be shared unless they are encapsulated within objects. This second approach was found acceptable for CDRS.

6.1. A Complete Solution

Supporting slot-level persistence for composite values required adding pholes as a level of indirection between each composite slot and its value. This was easy to do within the metaobject protocol, since slots are parts of objects. Supporting persistence for shared structures requires adding pholes as a level of indirection in front of each shared structure. This, unfortunately, cannot be done within the metaobject protocol.

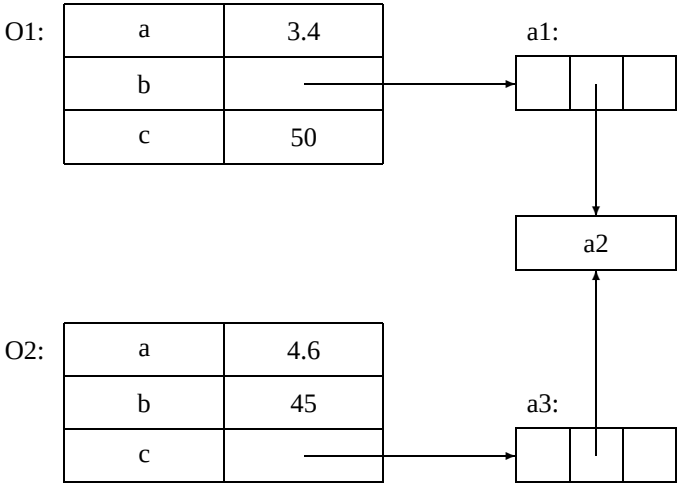

Figure 4. An array shared by two objects

Just as we maintain virtual dirty bits for substructures of composite slot values via a hash table, we maintain virtual pholes for shared structures via another hash table. Our solution requires that we do the following:

- Create an *"eq"*-test hash table, `ht2`, that maps a key (an array) to a value (an array or a phole). At any given time, `ht2` will contain all arrays for which pholes have been created, as well as all arrays that are contained by other arrays.
- Overload the write access routine for arrays, i.e., `(setf aref)`.

Figure 5 shows a snapshot in the execution of MetaStore. The arrays `a1` and `a3` are indirectly contained in their respective slots via pholes, and `ht2` contains a mapping from the shared `a2` to the virtual phole `phole2`. (Ideally, `a1` and `a3` would not contain `a2` directly, but would do so indirectly via a non-virtual `phole2`, and `ht2` would be unnecessary.) Although not pictured, `ht2` also maps `a1` and `a3` to `phole1` and `phole3` respectively.

When an array `a4` becomes a substructure of another array `a5` via the call

```
(setf (aref a5 1) a4)
```

the following procedure is followed (in addition to the procedure for dealing with dirty bits):

1. If `a4` is in `ht2`, then:

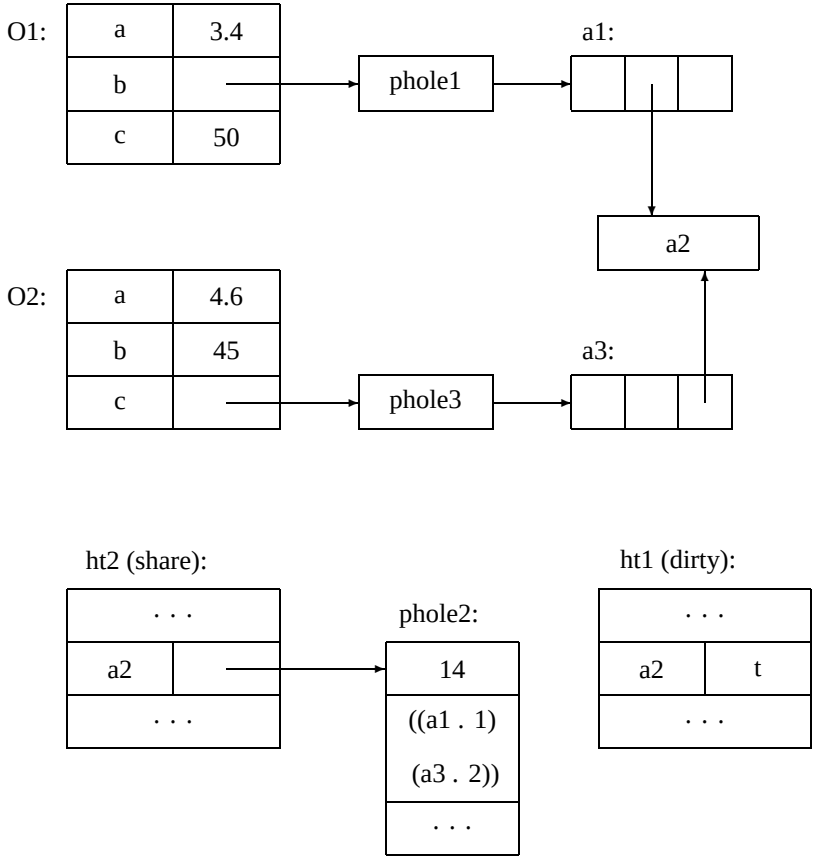

Figure 5. A persistable array shared by two persistable objects

(A) If `ht2` maps `a4` to a phole (say `phole4`), then add `a5` to the reference list, `referrers`, of `phole4`.

(B) If `ht2` maps `a4` to another array (say `a6`) then *create* a phole (say `phole4`) and add both `a5` and `a6` to the `referrers` of `phole4`. Modify `ht2` so that `a4` is mapped to `phole4`.

2. If `a4` is not in `ht2`, then add a new entry to `ht2` that maps `a4` to `a5`. Notice that a virtual phole is not created for an array until that array is shared. This is critical because our experience with CDRS shows that structure sharing is very rare although many small arrays (e.g., 3D point data arrays) are often referenced by a parent array.

This algorithm associates a phole with each array as soon as it is shared by at least two other arrays. If the array is a direct member of an object slot, of course, a phole is

```
(defsetf aref (arr &rest subscripts) (new-val)
  `(progn
     (setf (gethash ,arr ht1) t)            ;for dirty bit
     (when (composite-p ,new-val)           ;for sharing
       (let ((in-ht2 (gethash ,new-val ht2)))
         (if in-ht2
             (if (phole-p in-ht2)
                 (push ,arr (phole-referrers in-ht2))
                 (let ((phole2 (make-phole)))
                   (push in-ht2 (phole-referrers phole2))
                   (push ,new-val (phole-referrers phole2))))
             (setf (gethash ,new-val ht2) ,arr))))
     (setf (sysaref ,arr ,@subscripts) ,new-val)))
```

Figure 6. Modification to array updates for sharing

associated with it immediately. The virtual pholes contained in `ht2` come into play only for write accesses.

This algorithm can be implemented by modifying the write access routine as shown in Figure 6 in CLOS code. Arrays are used in the example code, thus modifying the array mutating operator `(setf aref)`.

The routine in Figure 6 assumes that the workhorse system routine for `aref` is `sysaref`. If we do not have access to this routine, then we will have to require our application programs to use `paref` instead of `aref` which would require similar definitions instead as we saw with the dirty bits (Section 5.1).

Suppose now that we want to save `O1` and `O2`, both of which are dirty. Suppose also that `a1`, `a2`, and `a3` are dirty. First, by saving `O1`, we will save `a1`, `a2`, and `O1`. At that point, they will also be marked clean. Second, by saving `O2`, we will save `a3` and `O2`, and mark them clean. Note that by the time `O2` is saved, `a2` is already clean, thus not saved multiple times. Also note that this algorithm saves each composite value as a separate entity in the object base.

Suppose now `O1` and `O2` are loaded in that order by a different process at some later time. When the value of `b` of `O1` is loaded, the value of `phole1` will be set with `a1`. The second element of `a1` will then get `a2` loaded by using the slot ID, 14, in the object table. Note, however, that `a1` references `a2` directly rather than the phole, `phole2`. When the value of `c` of `O2` is loaded, `a2` would have already been loaded and `a3` will reference `a2` directly again by going through the object table.

When an object is declared deleted by a user program, MetaStore releases all the handles that MetaStore is keeping track of, including the hash tables for dirty bits and structure sharing.

6.2. Feasibility of the Complete Solution

As with the complete solution for tracking dirty bits, there are both performance and engineering problems with the solution that we have just described. Because this solution depends on the complete solution for tracking dirty bits, supporting this solution incurs the following extra cost on top of the cost of dealing with dirty bits (Section 5.1):

- Each write access to a composite value must bear the cost of the case analysis done in the presented algorithm and the cost of adding an entry to the hash table.
- With this solution, each composite value is saved as a separate entity in the object base, thus the granularity of persistence is a composite value rather than a composite slot value. This algorithm would suffer much more dealing with the smaller grain size.
- The hash table, `ht2`, will in general be quite large because of many small arrays (e.g., arrays of length three for 3D point data in a geometric application such as CDRS), thus potentially severely affecting each write access.

Although many small arrays are referenced by a parent array, sharing is not very common in CDRS. Even though this complete solution would preserve the semantics of Common Lisp for sharing, supporting it in the context of persistence would be impractical in a production quality application like CDRS.

6.3. A Practical Solution

The extra costs associated with the solution described in the previous section would outweigh the benefits. Thus, we adopted a more pragmatic solution that limits the sharing of non-object structured data.

Because a persistable object already has a unique ID, it can be shared freely. Therefore, any composite value that is not an object can be made sharable by encapsulating it within an object. Thus, the array `a2` in Figure 5 can be made sharable by making it a slot value of an instance of the sharable composite class, say `sharable-composite-class`:

```
(defclass sharable-composite-class ()
  ((sharable-composite :initform nil
                       :initarg :sharable-composite
                       :accessor sharable-composite))
  (:metaclass persistable-metaclass))
```

With this class defined, `a2` now can be replaced by an object created by calling,

```
(make-instance 'sharable-composite-class :sharable-composite a2)
```

Using the accessor `sharable-composite` and its dual `(setf sharable-composite)`, one can conveniently perform read and write accesses respectively. The cost of using this class is one extra slot access on both read and write accesses. However, this would

give much better system performance than would the "complete solution" presented in the previous section. Because the amount of sharing done in an application like CDRS is so small, this was considered an acceptable alternative.

7. Related Work

Metaprogramming has been used in a variety of different applications by a number of researchers. Interestingly, none of these researchers reported the kinds of problems with metaprogramming that we have observed. We believe that this is because our application was much more ambitious than any of the others.

Rodriguez, with Anibus [24, 25], investigated whether it was possible to use the metaobject protocol approach to develop an open parallelizing compiler in which new "marks" for parallelization could be defined in a simple and incremental way. Anibus has its own metaobject protocol. Unlike the metaobject protocol of CLOS, which is intended to be used in executing CLOS programs at run time, that of Anibus was intended to be used to map a Scheme [28] program to an SPMD Scheme [24] program at compile time.

The authors in [1] present three examples of how the CLOS metaobject protocol can be used. The first example shows how atomic objects can be implemented for concurrency control. Their second example outlines how persistence can be implemented through metalevel manipulations. This supports persistence at the object level. Their final example illustrates how graphic objects can be implemented via the protocol.

PCLOS [22] is CLOS extended with persistence via the metaobject protocol of CLOS. PCLOS also supports persistence at the object level. It uses data base management systems for secondary storage management, which suffers from the phenomenon known as impedance mismatch [2, 7].

Unlike PCLOS [22] and the work described in [1], MetaStore supports persistence at the slot level, which we believe is critical for the performance of a CAD application. Therefore, neither of these efforts experienced the kinds of problems that we described in sections 3, 4, and 6. Two other important differences are that MetaStore, unlike [22] and [1], supports incremental saves and addresses the problem of supporting persistence of shared structures.

There are other systems of interest, though they are more oriented toward persistence than towards metaprogramming.

The authors in [9] implemented persistence of Common Lisp values featuring orthogonal persistence, concurrent transactions, and compiled code support.

PS-Algol [6] is S-Algol [21] extended with persistence by applying the principle of orthogonal persistence. The runtime system of PS-Algol maintains two separate heaps, one in RAM and the other on disk, thus using two types of addresses: persistent and local. A dereference instruction traps all persistent addresses and triggers the loading of objects from disk. When a program finishes, the runtime system copies objects from the local heap in RAM to the persistent heap in disk.

STATICE [30, 31], a database system for Symbolics LISP workstations with a multiple clients/single server architecture, provides the usual features of an object-oriented database system such as object identity, multiple inheritance, user-defined literal types, and a query language.

Hosking describes a Smalltalk system extended with persistence in [8]. He used the object faulting approach by modifying the language runtime system to give the illusion of a large heap of objects, only some of which are actually resident in memory. When the runtime system detects a reference to the contents of a non-resident object, an object fault occurs, causing the object to be made resident.

Most of these systems support orthogonal persistence, which is theoretically attractive. Some of them were found to be efficient enough for some applications, but the non-selective persistence supported in most of these systems would not be practical enough for applications such as CDRS. Based on our experience, the majority of data/objects created in CDRS never persist.

These other systems were also implemented with the benefit of having almost unlimited access into the low-level implementation. For example, STATICE had access to both the underlying object system implementation and the Common Lisp implementation. The work by [9] had access to the underlying Common Lisp implementation. PS-Algol had access to the runtime system, as did Hosking's system.

In the case of Hosking's work, the underlying system itself is a purely object-based language, Smalltalk, that makes the persistence extension much easier. In comparison, the implementation of MetaStore was constrained by both the metaobject protocol and by the efficiency restrictions imposed by the commercial product CDRS.

8. Summary

Adding persistence to CLOS is no small undertaking. The protocol is sufficient to support language extensions as long as these extensions involve modifying or augmenting the structure or behavior of objects. Since most of what was required to extend CLOS with object persistence was related to objects, it was done easily via the protocol. We are convinced that the idea of programming at the metalevel is the right approach for applications such as ours. A few extensions to the protocol, coupled with better implementation techniques, would yield a uniquely useful tool.

Our implementation revealed that the currently available design and implementation of the CLOS metaobject protocol was lacking in several ways:

- The current implementations of the protocol are fairly efficient for the most part. However, we encountered some features that performed rather poorly. For example, we observed up to 300 time slowdowns in PCL [4], and up to 50 time slowdowns in Lucid CLOS, with `:around` methods. Specializing default behavior by the use of `:around` methods is one of the most commonly used features in the metaobject protocol.

- To support persistence at the slot level requires one level of indirection on slot accesses and the current protocol does not provide this feature. We were, however, able to deal with this by extending the protocol by adding two more interface routines.

- Maintaining dirty bits for composite values and handling persistence of shared non-object structured data were difficult because they are not object related and do not belong to the domain of the metaobject protocol. Instead they belong to the base

language implementation level, thus requiring help from the language compiler and the runtime support system. Since we could not get the necessary help from these either, we handled them with some help from application programs. Most of the performance penalty was caused by having to deal with dirty bits and shared non-object structured data.

Based on our implementation we propose the following for the CLOS metaobject protocol:

- Add two new routines to the protocol so that one level of indirection on slot accesses can be done. An even better solution would be to extend the semantics of method combinations in CLOS in such a way that specialized methods, such as `:around` methods, can be optionally skipped during execution.

- Extend the design in such a way that all composite data types can be implemented as objects so that they can be included in the metaobject protocol. As it is, there is an abstraction mismatch: maintaining dirty bits and dealing with non-object structured data belong to the base language, yet we had to handle them at the metaobject level. This extension would require a significant effort.

If the overhead currently inherent to `:around` methods in Lucid CLOS were eliminated, we estimate that write accesses to persistent objects would be about seven times slower than to non-persistent objects, that object creation would be about four times slower, and that read accesses would be about the same. We also believe that these overheads would be tolerable in CDRS, which is governed by the speed of user interaction. Nevertheless, we are exploring some changes to the design to MetaStore to reduce these performance penalties.

The poor performance results and the difficulties we experienced with dirty bits and shared structures should be interpreted with caution. It is important to differentiate the costs incurred by the metaobject protocol itself from the costs stemming from our persistence requirements. Much of our performance hit was due to the severe requirements of supporting almost-transparent persistence without any help from the compiler or the runtime system. Our experiment was perhaps too ambitious for the pioneering, experimental implementations of the metaobject protocol that were available. We remain excited about the potential impact of open implementations and metaprogramming on software engineering and software science in general. The latest work on open implementations can be found in [13] and [19].

Acknowledgments

Thanks go to Gregor Kiczales for helping us with the metaobject protocol and open implementation ideas and its implementation in PCL; to Andreas Paepcke for helping us with the metaobject protocol of PCL; to Bob Kessler, Gary Lindstrom, and Mark Swanson for stimulating discussions on technical issues; and to the members of the CDRS group at Evans & Sutherland Computer Corporation for the opportunity to be persistent on object persistence and for many technical discussions.

Notes

1. PCLOS [22] also uses the notion of a husk object, but with a different meaning. A husk in PCLOS is a placeholder for an object and is used to make memory-resident references to the instance, that is not yet loaded, work properly. Thus, a husk there is much like a phole in MetaStore, except that a phole is used for a composite slot, whereas a husk in PCLOS is used only for objects. The notion of a husk object as used in MetaStore does not exist in PCLOS.

References

1. G. Attardi, C. Bonini, M.R. Boscotrecase, T. Flagella, and M. Gaspari. Metalevel programming in CLOS. In *Proceedings of the European Conference on Object-Oriented Programming* (1989).
2. F. Bancilhon and D. Maier. Multilanguage object-oriented systems: new answer to old database problems? In *Programming of Future Generation Computers II,* Fuchi, K. and Kott, L. editors. Elsevier Science Publishers B.V. (North-Holland) (1988).
3. D.G. Bobrow, L. DeMichiel, R.P. Gabriel, G. Kiczales, D. Moon, and S.E. Keene. *The Common Lisp Object System Specification:* Chapters 1 and 2. Technical report 88-002R, X3J13 Standards Committee Document (1988).
4. D.G. Bobrow and M. Stefik. *The Loops Manual.* Intelligent Systems Laboratory, Xerox Palo Alto Research Center (1983).
5. G. Bracha. The programming language Jigsaw: mixins, modularity, and multiple inheritance. Ph.D. thesis, Department of Computer Science, University of Utah (1992).
6. W.P. Cockshott. *PS-Algol Implementations: Applications in Persistent Object-Oriented Programming.* Ellis Horwood Limited (1990).
7. G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (June 1984). *ACM SIGMOD Record 14, 2* (1984).
8. A.L. Hosking. Lightweight support for fine-grained persistence on stock hardware. Ph.D. thesis, University of Massachusetts at Amherst (1995).
9. J.H. Jacobs and M.R. Swanson. Syntax and semantics of a persistent Common Lisp. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (1993).
10. G. Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures* (1992).
11. G. Kiczales. Why are black boxes so hard to reuse? (Towards a new model of abstraction in the engineering of software.) Invited Talk at the OOPSLA'94 Conference on Object-oriented Programming Systems, Languages, and Applications, Portland, Oregon (October 1994).
12. G. Kiczales, J. des Rivières, and D.G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press (1991).
13. G. Kiczales, J. Lamping, C.V. Lopes, A. Mendhekar, and G. Murphy. Open implementation design guidelines. (submitted to ICSE-97: 19th International Conference on Software Engineering, Boston, Mass.)
14. A.H. Lee. An object-oriented programming approach to geometric modeling. In *Proceedings of Evans & Sutherland Technical Retreat,* Ocho Rio, Jamaica (1989).
15. A.H. Lee. Managing complex objects. Internal document, Evans & Sutherland Computer Co. (1990).
16. A.H. Lee. The persistent object system MetaStore: persistence via metaprogramming. Ph.D. thesis, Department of Computer Science, University of Utah (1992).
17. A.H. Lee and J.L. Zachary. Reflections on metaprogramming. *IEEE Transactions on Software Engineering, 21, 11* (November 1995) 883-893.
18. *Lucid Common Lisp/MIPS Version 4.0, Advanced User's Guide.* Lucid, Inc. (1990).
19. C. Maeda, A.H. Lee, G. Murphy, and G. Kiczales. Open implementation analysis and design$^{[tm]}$. (submitted to ICSE-97: 19th International Conference on Software Engineering, Boston, Mass.)
20. J.S. Miller. MultiScheme: a parallel processing system based on MIT Scheme. Ph.D. thesis, Massachusetts Institute of Technology (1987).
21. R. Morrison. *S-Algol Language Reference Manual.* CS-79-1, University of St. Andrews (1979).
22. A. Paepcke. PCLOS: stress testing CLOS: experiencing the metaobject protocol. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (1990).

23. A. Paepcke. User-level language crafting: Introducing the CLOS metaobject protocol. A. Paepcke, ed., *Object-Oriented Programming: The CLOS Perspective* (1992).
24. L.H. Rodriguez, Jr. Coarse-grained parallelism using metaobject protocols. M.S. Thesis, Massachusetts Institute of Technology (1991). (Also available as Technical report SSL-91-06, Xerox Palo Alto Research Center, 1991.)
25. L.H. Rodriguez, Jr. Towards a better understanding of compile-time metaobject protocols for parallelizing compilers. In *Proceedings of IMSA'92: International Workshop on Reflection and Meta-level Architecture*, Tokyo, Japan (1992).
26. B.C. Smith. Reflection and semantics in a procedural language (Ph.D. thesis). Technical Report TR-272, Laboratory for Computer Science, MIT (1982).
27. B.C. Smith. Reflection and semantics in Lisp. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, Salt Lake City, Utah (1984).
28. G. L. Steele, Jr and G.J. Sussman. *Scheme: An interpreter for the extended lambda calculus.* Memo 349, MIT Artificial Intelligence Laboratory (1975).
29. G.L. Steele, Jr. *Common Lisp: The Language*, Second edition. Digital Press (1990).
30. D. Weinreb. An object-oriented database system to support an integrated programming environment. *IEEE Data Engineering, 11, 2* (June 1988).
31. D. Weinreb, N. Feinberg, D. Gerson, and C. Lamb. An object-oriented database system to support an integrated programming environment. R. Gupta and E. Horowitz, eds, *Object-Oriented Databases with Applications to CASE, Networks, and VLSI Design*, Prentice Hall, Englewood Cliffs, NJ (1991) 117-129.