# Reflections on Metaprogramming

Arthur H. Lee
Joseph L. Zachary

UUCS-93-004

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

March 1, 1993

## Abstract

The spread of object-oriented technology has led to object-oriented programming languages with object-oriented implementations. By encapsulating part of the semantics of a language within a set of default classes and empowering the programmer to derive new versions of these base classes, a designer can provide a language whose semantics can be tailored by individual programmers. The degree to which such languages are simultaneously flexible and efficient is an open question. We address this question by reporting our experience with using this technique to incorporate support for persistence into the Common Lisp Object System via its metaobject protocol. For many aspects of our implementation we found that the metaobject protocol was perfectly suitable. In other cases we had to variously extend the protocol, pay an unacceptable performance penalty, or modify the language implementation directly. Based on our experience we propose some improvements to the protocol. We also present some performance measurements that reveal the need for improved language implementation techniques.

# Contents

## 1 Introduction

The spread of object-oriented technology has led to object-oriented programming languages with object-oriented implementations. By encapsulating part of the semantics of a language within a set of default classes and empowering the programmer to derive new versions of these base classes, a designer can provide a language whose semantics can be tailored by individual programmers. The process of modifying language semantics in this way is called *metaprogramming.*

The degree to which such languages are simultaneously flexible and efficient is an open question. In this paper we address this question by reporting our experience with using metaprogramming to incorporate support for persistent objects into the Common Lisp Object System (CLOS) [BDG+88]. This experiment was possible because CLOS provides for metaprogramming via its *metaobject protocol* [KdRB91].

The goal of our experiment was to see if we could obtain a version of CLOS with persistence to which we could easily port a commercial CAD system already written in CLOS. We originally wanted to modify CLOS strictly via the metaobject protocol, so that no changes to the compiler or run-time system 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.

Our intent in this paper is to highlight the strengths and weaknesses exhibited by the CLOS metaobject protocol during our experiment. 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 metaprogramming in general. For many aspects of the implementation we found that the metaobject protocol was perfectly suitable. In other cases we had to choose among paying a large performance penalty, extending the protocol, and bypassing the protocol entirely and modify the language implementation directly. Based on our experience we propose some improvements to the protocol. We also present some performance measurements that reveal the need for improved language implementation techniques.

The remainder of this paper is organized as follows. In Section 2 we briefly describe the CLOS metaobject protocol, the problem of object persistence, and our approach to adding persistence via metaprogramming. In Section 3 we describe in detail how we made four particular extensions and discuss the problems that we encountered; in section 4 we propose some improvements to the protocol; and in section 5 we present some performance measurements. After we survey other uses of metaprogramming in section 6, we conclude in section 7.

# 2 Background

## 2.1 The CLOS Metaobject Protocol

CLOS has an object-oriented implementation. This allows users to alter the semantics of the language by using the standard object-oriented techniques of subclassing and specialization.

In the design of CLOS, the basic elements of the programming language—classes, slots, methods, generic functions, and method combinations—are made accessible as objects. Because these objects represent fragments of a program, they are given the special name *metaobject*. Individual decisions about the behavior of the language are encoded in a protocol operating on these metaobjects—thus the term metaobject protocol. For each kind of metaobject a default class is created, which delineates the default behavior of the language in the form of methods.

In the metaobject protocol, for example, the meaning of object instantiation is implemented by a small number of generic functions. These semantics can be changed by defining a subclass in which these generic functions are specialized. In doing this, the user is making an incremental adjustment to the meaning of the language. Most aspects of the language's behavior and implementation remain unchanged, with just the semantics of instances being altered.

## 2.2 An Application

Our work was initially motivated by the problems of object persistence encountered with the Conceptual Design and Rendering System (CDRS) [Lee89,Lee90]. CDRS is a geometric CAD modeler that is used by designers in a dozen major automotive and product design companies worldwide. It is written mostly in Common Lisp [Ste90] as extended by CLOS. A typical model manipulated by CDRS contains tens of thousands of objects that may not all fit in virtual memory, a wide variation in the sizes of objects, complex data structures within objects, and rich relationships (both semantic and structural) among objects.

CDRS uses a naive batch approach to object persistence that has proven to be ill-suited [Lee92]. All objects in a design session are saved to a file at the end of a modeling session and are reloaded at the beginning of the next session. This approach requires a huge amount of virtual memory, frequent large garbage collections, and a long time to load and save models. For example, CDRS usually 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.

## 2.3 MetaStore

We tried to address these problems by adding the notion of persistent objects to CLOS. The resulting system, which we call MetaStore, has two major components: the language extension portion implemented via the metaobject protocol, and the database management portion that provides a persistent object store. We are concerned in this paper with the language extension component and the degree to which the metaobject protocol facilitated and frustrated our efforts. For a complete discussion of the resulting system see [Lee92].

MetaStore maintains a virtual object space within virtual memory. As the object space fills, it writes the least recently used persistent objects to disk and makes their virtual images available for garbage collection. Saves are done incrementally (only modified objects are written to disk) and loads are done on demand (objects are loaded as they are needed). This approach amortizes the cost of saving and loading models over the entire design session. To support this, we had to make substantial modifications to the way objects are represented and manipulated by CLOS. The question that concerned us throughout design and implementation was whether the overhead imposed by the metaprogramming would be too costly. It was.

## 3 Implementation Experience

We used the metaobject protocol to make dozens of substantial modifications to CLOS. In this section we discuss four of these modifications, which illustrate the kinds of situations in which the metaobject protocol is and is not applicable. For each problem and difficulty we encountered, we indicate whether it was a language-specific problem (due to CLOS) or a problem-specific difficulty (due to adding persistence to a language).

### 3.1 Structure and Behavior of Objects

The metaobject protocol is ideal for language extensions that involve modifications to the structure of objects or simple changes to their behavior. Changes to other kinds of data structures (such as arrays) are much more difficult. We were able to make a wide variety of such modifications, a few of which we describe here. All of these modifications were supported by changing the class meta-class via inheritance.

In addition to the user-defined slots, we maintain several other slots in each object. A unique object identifier and a "dirty bit" which flags unsaved objects are included. Since the majority of the objects in a typical model are not intended to be saved, we must be careful to distinguish between transient and persistent objects. This burden is borne by the programmer, who must choose between defining classes relative to the standard class meta-class (in the case of transience) or the derived class meta-class (in the case of persistence).

Each read or write access to a persistent object is intercepted so that appropriate actions can be carried out. For example, a read access may result in a composite slot being loaded from disk, and a write access results in a dirty bit being set. These kinds of actions are implemented by modifying the appropriate methods via inheritance.

## 3.2 Indirection on Slot Access

MetaStore supports persistence at the slot level, and Common Lisp allows structure sharing. These two facts required us to maintain one level of indirection for each persistent composite slot. The contents of a persistent composite slot is a pointer to a data structure called a *phole*, which (among other things) contains a pointer to the composite value.

When a user program issues a `slot-value` call to a persistent 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 the method (`slot-value`) depending upon where and for what purpose it is called. The metaobject protocol provides no support for this. 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.

## 3.3 Maintaining Dirty Bits

Only dirty (modified) persistent objects are ever saved to disk. Because the smallest grain size of persistence is the composite slot, each persistent object and persistent composite slot value has its own dirty bit. We will concern ourselves here with composite slots. The dirty bit of a composite slot is kept in its phole.

The dirty bit of an object or composite slot must be set whenever a write access is made. Doing this via the metaobject protocol proved difficult. Performing a write upon 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 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 via the phole of `slot1`, the phole's dirty bit 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. However, requiring this

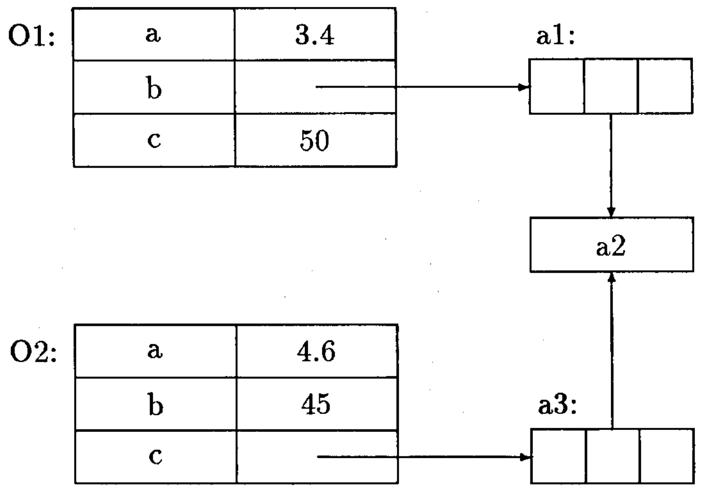

Figure 1: An array shared by two objects

extra call changes the semantics of CLOS. This is a problem-specific difficulty due to adding persistence to a programming language.

An expensive solution that maintains dirty bits without any help from either the application program or the compiler is described in [Lee92] although we chose to implement a simpler solution along the lines suggested above that requires help from user programs for efficiency reasons.

## 3.4 Shared Structures

Structured data in Common Lisp can be shared freely. This freedom adds much difficulty in supporting persistence of shared structures. We could find no acceptably efficient solution within the metaobject protocol since it does not deal with structured data that are not objects. 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 1, 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`.

This sort of sharing is perfectly legal in Common Lisp. 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`, say `a2-1` and `a2-2`. This again is a problem-specific difficulty due to adding persistence to a programming language.

A solution that, while inefficient, handles persistent shared structures entirely within the metaobject protocol is described in [Lee92]. We chose, however, a different approach in MetaStore in which we require that composite structures be shared only at the slot level. This approach works because a phole can contain a unique identifier for a composite slot value.

## 4 Observations

### 4.1 Abstraction Mismatch

The metaobject protocol is designed to support language extensions that have to do with the structure or behavior of objects. As soon as we try to augment the language with a feature that is not a property of objects, the protocol is no longer sufficient.

As we have seen, supporting object persistence required some changes that were object related as well as others that were base language related. Dealing with the kinds of modifications described in sections 3.3 and 3.4 was difficult because there are no metaobjects corresponding to the nonobject structures; i.e., there is an abstraction mismatch.

CLOS can be viewed as having five levels of implementation ranging from high-level to low-level: CLOS objects, Common Lisp, garbage collection, data types, and memory. Dealing with dirty bits and structure sharing can best be done at levels such as "Common Lisp" and/or "garbage collection" in the list above. In MetaStore we tried to solve these issues at the "CLOS objects" level, so it is not surprising that it was not natural. We had to leave the metaobject protocol at times to deal with these issues by devising extra mechanisms that required some help from user programs and/or the Common Lisp compiler.

### 4.2 Short-Term Improvements

Based on the experience of adding object persistence to CLOS in MetaStore, a few minor improvements are proposed here to the existing protocol. They are related to slot accessing as described in section 3.2. We propose that the protocol support a mechanism for one level of indirection on slot accesses. One possibility would be to provide two more routines as follows:

- `slot-value-using-class-direct`:
  This routine is identical in all respects to `slot-value-using-class`, which performs read accesses to slots. We sometimes want to use the default behavior of `slot-value-using-class` and at other times the changed behavior, and this new routine would always give the default behavior. This changed behavior is typically obtained by specializing the default method. With the current protocol, once we modify the behavior of a method this way, we cannot use the default behavior anymore.

- `(setf slot-value-using-class-direct)`:
  This is the dual of `slot-value-using-class-direct` for write accesses to slots.

When a method in CLOS is changed via specialization, there is no easy way to get the default behavior any more. We may want to extend the semantics of method combinations as follows. Even after a method is specialized, we are given the option of executing the original version alone. This is not an easy extension to support in general since it requires elaborate control. The `copy-as` operation of Jigsaw [Bra92] would solve this problem.

### 4.3 Long-Term Improvements

As discussed in sections 3.3 and 3.4, a seamless extension to CLOS of object persistence requires support from the base language implementation level. Judging from our experience with MetaStore, the metaobject protocol of CLOS seems well designed to support extensions to CLOS as long as the extension is inherently object-oriented.

To stay with the spirit of the metaobject protocol of CLOS to "open" up the language, it would be useful to push the metaobject protocol idea further down to the level of the base language implementation. If we could support the metaobject protocol at the Common Lisp data type level or at the garbage collection level, the problems that we experienced in MetaStore (dirty bits and shared structures) could be easily solved. With this change, the protocol would allow more flexibility for extensions of the sort done in MetaStore. (Perhaps we would then call it the *metadata protocol* or *metatype protocol.*)

## 5 Performance Measurements

In this section we present some performance comparisons of two implementations of the CLOS metaobject protocol. We also give performance measurements for our implementation of MetaStore. The machine and the configuration we used are not included since we are primarily interested in relative comparisons.

### 5.1 PCL and Lucid Performance Measurements

We had originally intended to use the Lucid CLOS version of the metaobject protocol, but it did not have a complete implementation of slot-level metaobjects. As a result we were forced to use the PCL version, even though it is not an industrial-strength implementation.

Before we present the measurements of MetaStore, we describe one significant inefficiency we found with PCL [BS83] and Lucid CLOS [Luc90]. In implementing MetaStore, a number of `:around` methods are used to specialize the protocol routines. (An `:around` method is a commonly used CLOS construct for specializing other methods.) The three most notable are `make-instance` for creating objects, `slot-value-using-class` for read accessing an object, and `(setf slot-value-using-class)` for write accessing an object. To measure the cost of `:around` methods, an `:around` method was defined for each of these methods whose body did nothing but call `call-next-method`. The measurements in this section are based on PCL and we also present what we learned about Lucid CLOS where appropriate.

- *Creation:* Creating 1,000 objects showed:

| | *Time* | *Bytes Consed* |
|---|---|---|
| Transient | 3.40 sec | 256,008 |
| After :around methods | 4.32 | 368,008 |
| Ratio | 1.27 | 1.43 |

  Other measurements showed that creating objects with `:around` methods was about 50 times slower than without in Lucid CLOS [LZ92]. In PCL we do not see as big a difference as with Lucid CLOS since creating objects in PCL without `:around` methods is already much slower than it is in Lucid CLOS.

- *Read Access:* Read accessing a slot of the "self" object 100,000 times showed:

| | *Time* |
|---|---|
| Before MetaStore | 0.13 sec |
| After :around Methods | 34.86 |
| Ratio | 268.15 |

  Dummy `:around` methods made read accesses almost 300 times slower than the normal transient read accesses.

- *Write Access:* Write accessing a slot of the "self" object 100,000 times showed:

| | *Time* |
|---|---|
| Before MetaStore | 0.11 sec |
| After :around Methods | 35.90 |
| Ratio | 326.36 |

  Dummy `:around` methods made write accesses over 300 times slower than the normal transient write accesses.

The extent to which dummy `:around` methods compromise the performance of both the PCL and Lucid implementations of CLOS belies the claim of [KdRB91] that the metaobject protocol is both elegant and efficient. Specializing default behavior by the use of `:around` methods is the most commonly used tool in the metaobject protocol.

Notice that in PCL we observed a 300 times slowdown when reading and writing slots, whereas in Lucid we observed a 50 times slowdown when creating objects. Neither of these figures can be tolerated in a commercial application. In fairness, we must emphasize that Lucid is in general far more efficient than PCL.

## 5.2 Implementation of Persistence in MetaStore

In this section we present measurements based on our implementation of the MetaStore kernel. This is the cost of the basic mechanism of MetaStore that allows the minimum functionality of MetaStore: being able to define persistent classes, being able to selectively declare slots to be persistent, being able to perform incremental saves, being able to load on demand, etc. We maintain such things as object identities, pholes, the object table, dirty bits, and model identities for interfacing the object base at this level. These measurements also include the cost of metaobject classes, `:around` methods, and slot level persistence. Shared structures and virtual object memory are not included.

- *Creation:* The measurements were made while creating 1,000 objects:

| | *Time* | *Bytes Consed* |
|---|---|---|
| Transient | 3.40 sec | 256,008 |
| MetaStore Kernel | 56.35 | 6,152,008 |
| Ratio | 16.57 | 24.04 |

  The MetaStore kernel made creating objects about 16 times slower than creating transient objects. Creating objects in the MetaStore kernel used about 24 times more space than creating transient objects.

- *Read Access:* The measurements were made while read accessing an object 100,000 times. A slot of the "self" object within a method was accessed that many times:

| | *Time* |
|---|---|
| Transient | 0.13 sec |
| MetaStore Kernel | 35.62 |
| Ratio | 274.00 |

  Read accesses in the MetaStore kernel was about 270 times slower than the normal transient read accesses.

- *Write Access:* The measurements were made while write accessing an object 100,000 times. A slot of the "self" object within a method was accessed that many times:

| | *Time* |
|---|---|
| Transient | 0.11 sec |
| MetaStore Kernel | 254.70 |
| Ratio | 2,315.45 |

  Write accesses in the MetaStore kernel was over 2,000 times slower than the normal transient write accesses.

Read accesses in the MetaStore kernel did not add any additional cost because there is no extra work added to the read mechanism at the kernel level. The main cost added on object creation is due to the addition of pholes to persistent composite slots and the case analysis of slot values that are being used as the initial values. Write accesses added substantial extra cost. Most of it is caused by (i) the use of `:around` methods and (ii) the case analysis on the slot values in order to add or remove a phole if necessary.

If we were to factor out the constant overhead imposed by the use of `:around` methods, object creation and write accesses would be about 13 and 7 times slower respectively than the transient case; read accesses would be about the same as the transient case.

If MetaStore were to run on Lucid CLOS with the `:around` overhead removed, our measurements predict that object creation and write accesses would be about 4 and 7 times slower respectively than the transient case; read accesses would be about the same as the transient case. Although obviously not ideal, we believe that these overheads would be tolerable in CDRS, which is governed by the speed of user interaction.

## 6 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 [Rod91,Rod92], 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 [SS75] program to an SPMD Scheme [Rod91] program at compile-time.

The authors in [ABB89] present three examples of how the CLOS metaobject protocol could be used. The first example shows how atomic objects could be implemented for concurrency control. Their second example outlines how persistence could be implemented through metalevel manipulations. This supports persistence at the object level. Their final example illustrates how graphic objects could be implemented via the protocol.

PCLOS [Pae90] 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 [BM88,CM84].

Unlike PCLOS [Pae90] and the work described in [ABB89], 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 section 3. Two other important differences are that MetaStore, unlike [Pae90] and [ABB89], supports incremental saves and persistence of shared structures.

## 7 Summary

The authors in [KdRB91] state that they have simultaneously achieved elegance and efficiency by basing language design on metaobject protocols. Our experience of extending CLOS with persistence via the metaobject protocol shows that current implementations do not live up to this claim. The extent to which even dummy `:around` methods compromise the performance of both the PCL and Lucid implementations of CLOS makes them unacceptable for production programming. We observed up to 300 time slowdowns in PCL, and up to 50 time slowdowns in Lucid. Specializing default behavior by the use of `:around` methods is the most commonly used tool in the metaobject protocol.

Nevertheless, most of the extensions required to support object persistence were easily carried out in the metaobject protocol. We are convinced that the idea of metaprogramming 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. 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 metaprogramming in general.

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.

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. We propose that two new routines be added 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 an `:around` method can optionally be skipped during execution.

There were a few difficulties that we faced that could not be resolved with the protocol alone. They were maintaining dirty bits for composite values and handling persistence of shared nonobject structured data. 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 run-time support system. Since we could not get help from these either, we handled them with some help from application programs. Here, we propose that all persistent data types be implemented as objects so that they can be included in the metaobject protocol. This would be a significant effort and we consider this a long-term goal.

## References

[ABB89] 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.

[BM88] F. Bancilhon and D. Maier. Multilanguage object-oriented systems: new answer to old database problems? In *Programming of Future Generation Computers II,* K. Fuchi and L. Kott, editors. Elsevier Science Publishers B.V. (North-Holland), 1988.

[BDG+88] 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.

[BS83] D. G. Bobrow and M. Stefik. *The Loops Manual.* Intelligent Systems Laboratory, Xerox Palo Alto Research Center, 1983.

[Bra92] G. Bracha. The programming language Jigsaw: mixins, modularity, and multiple inheritance. Ph.D. dissertation, Dept. of Computer Science, Univ. of Utah, 1992.

[CM84] 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).

[KdRB91] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[Lee89] A. H. Lee. An object-oriented programming approach to geometric modeling. In *Proceedings of Evans & Sutherland Technical Retreat*, Ocho Rio, Jamaica, 1989.

[Lee90] A. H. Lee. Managing hierarchical complex objects. Internal report, Evans & Sutherland Computer Co., 1990.

[Lee92] A. H. Lee. The persistent object system MetaStore: persistence via metaprogramming. Ph.D. dissertation, Dept. of Computer Science, Univ. of Utah, 1992.

[LZ92] A. H. Lee and J. L. Zachary. Using metaprogramming to add persistence to CLOS. Technical report UUCS-93-001, Dept. of Computer Science, Univ. of Utah, 1993.

[Luc90] *Lucid Common Lisp/MIPS Version 4.0, Advanced User's Guide.* Lucid, Inc., 1990.

[Pae90] 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.

[Rod91] 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.)

[Rod92] 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.

[SS75] G. L. Steele, Jr and G. J. Sussman. *Scheme: An interpreter for the extended lambda calculus.* Memo 349, MIT Artificial Intelligence Laboratory, 1975.

[Ste90] G. L. Steele, Jr. *Common Lisp: The Language,* Second edition. Digital Press, 1990.