Lisp and Symbolic Computation, 1, 39–51 (1988)

Oaklisp: An Object-Oriented Dialect of Scheme

KEVIN J. LANG AND BARAK A. PEARLMUTTER
Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213

Abstract

This paper contains a description of Oaklisp, a dialect of Lisp incorporating lexical scoping, multiple inheritance, and first-class types. This description is followed by a revisionist history of the Oaklisp design, in which a crude map of the space of object-oriented Lisps is drawn and some advantages of first-class types are explored. Scoping issues are discussed, with a particular emphasis on instance variables and top-level namespaces. The question of which should come first, the lambda or the object, is addressed, with Oaklisp providing support for the latter approach.

1. Introduction

Scheme has always been object-oriented in several senses. Like any dialect of Lisp, Scheme has the sort of interactive computational model that the Xerox PARC people have so eloquently and tirelessly promoted. Unlike earlier dialects of Lisp, Scheme has lexical closures that can encapsulate code with little pieces of private state. As *Lambda: The Ultimate Declaritive* [12] pointed out, viewing lambdas as objects makes it possible to write Scheme programs that have classical message-sending semantics. Unfortunately, programs written in this style are rather contorted, since the method selection process must be made manifest in every lambda.

The designers of T [10] decided that they could better support an object-oriented programming style by adding a special mechanism that implicitly looks up methods according to a mapping that is associated with each lambda. This facility was great for writing print methods and for defining generic operations on abstract types such as tables. However, T programmers who adopted an object-oriented point of view soon wanted an inheritance mechanism, so features were added that allowed a limited form of inheritance. The T object-oriented programming facility could not be improved further, since the original idea of viewing lambdas as objects was more of an intellectual exercise than a serious suggestion for a language design. The biggest shortcoming of T is that method tables cannot be incremen-

This work was supported by grants from DARPA and the System Development Foundation. Barak Pearlmutter is a Hertz Fellow.

tally modified; it is impossible to add a method to an existing type when there is no way to refer to a type. In fact, types don't even exist at the language level.[1]

To arrive at a full-blown object-oriented language that is still Scheme, a different approach is necessary. Instead of building objects out of lambdas, Oaklisp builds lambdas and other Lisp data types on top of a simple object-oriented kernel. Because types are represented by first-class, anonymous objects, real object-oriented programming can be supported without a large number of special linguistic mechanisms.

2. Overview of the language

As in T and later languages such as CommonLoops, Oaklisp uses function call syntax for message sending. When `(car x)` is evaluated, a message containing the `car` operation is sent to the object `x`, and a method is then selected to run based on the type of `x`. Oaklisp types are arranged in the usual sort of multiple inheritance hierarchy, so each type needs to supply only those methods that are needed to distinguish the type from its supertypes.

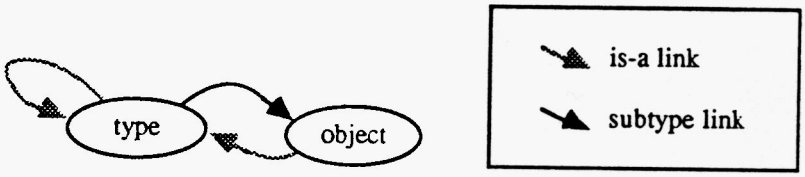

Fig. 1. The root of the Oaklisp type hierarchy.

2.1 Fundamental types

The Oaklisp type system is structured by the relations *is-a* and *subtype*. An object is related to its type by *is-a*, and a type is related to its supertypes by *subtype*. Each of these relations defines a directed graph that includes all of the objects in the system.

The fundamental types in the system are `type` and `object` (Fig. 1). They are distinguished by their position at the top of the *is-a* and *subtype* hierarchies, and by their circular definitions.

[1] In T, types do exist at an invisible implementation level; those who know the right magic functions to call *can* define methods incrementally.

`type` *Type*

This type is the root of the *is-a* hierarchy. It is the type of types, so new types are created by instantiating it.

`object` *Type*

This type is the root of the *subtype* hierarchy. Every type is a subtype of `object`, so default methods for operations such as `print` are defined for `object`.

2.2 Operations on objects

The following predicates are defined on all objects:

(`eq?` *object1 object2*) *Operation*

Defines object identity.

(`is-a?` *object type*) *Operation*

Determines whether *object* is an instance of *type* or one of its supertypes. Ordinary Lisp type predicates such as `pair?` and `number?` can easily be implemented in terms of this.

2.3 Operations on types

A predicate for testing the *subtype* relation on types is also available. The circularity at the top of the type system may be expressed by the assertions (`is-a? object type`) and (`subtype? type object`).

(`subtype?` *type1 type2*) *Operation*

Determines whether *type1* is a subtype of *type2*.

Types are distinguished from other objects by the fact that they handle the `make` operation, which is the mechanism for generating new objects.

(`make` *type . args*) *Operation*

Returns a new instance of *type* . `make` creates objects by allocating them and then sending them an `initialize` message. When `make` receives extra arguments, it includes them in the call to `initialize`.

(`initialize` *object . args*) *Operation*

Returns *object*. When a type requires special initialization, it shadows the default `initialize` method, which is a no-op.

2.4 Making new types

Since types are objects, new ones are created by sending a `make` message to the appropriate type object, which in this case is `type`.

(`make type` *ivars supertypes*) *Operation*

Returns a new type-object with the supertypes and instance variables specified by the argument lists. The order of the types in *supertypes* is significant because it influences run-time method selection. The method which is invoked for an operation on an object of a given type is the first one that would be encountered on a left-to-right depth-first traversal of the supertype tree starting at that type. Instances of the type contain all of the instance variables named in *ivars*, plus a block of instance variables for each ancestor type, with duplicate types removed.

2.5 Methods

The mechanism for manipulating the mapping from operations and types to methods is the following special form:

(`add-method` (*operation* (*type* . *ivar-list*) . *arg-list*) . *body*) *SpecialForm*

Returns *operation* after adding a method for *operation* to the method table of *type*.[2] *Operation* and *type* are evaluated positions. The body of the form is surrounded by an implicit `block`. The arguments to the method are specified by *arg-list*, and instance variables that are to be referenced are declared in *ivar-list*. Methods are closed over free variable references when the form is evaluated.

2.6 Functional syntax

Sometimes it is convenient to adopt a more conventional Lisp viewpoint when writing programs. This viewpoint considers functions to be the primary abstraction, with objects downgraded to the status of data which are passed around between functions. The key to this programming style is the ability to write functions that can accept arguments of any type.

Oaklisp readily accommodates a functional programming style, since methods can be defined for `object`, which lies at root of the *subtype* hierarchy. To give the language a familiar appearance when this programming style is used, the following macros are provided:

[2]Conceptually but not necessarily implementationally.

(lambda *arg-list* . *body*) *Macro*
  => (add-method ((make operation) (object) . *arg-list*) . *body*)
(define (*variable* . *arg-list*) . *body*) *Macro*
  => (set *variable* (lambda *arg-list* . *body*))

2.7 An example

A cursory reading of the preceding language description could leave the false impression that Oaklisp is similar to Smalltalk or Flavors. The following code fragment is intended to emphasize the Scheme-like nature of even the object-oriented portion of Oaklisp. This function is used in the kernel of our implementation to give objects a crude but informative way of printing themselves, namely by printing a name for the type of the object followed by a weak pointer that could be used to generate a reference to the object as long as it exists.

```
(define (add-simple-print-methods types names)
  (map (lambda (the-type the-name)
         (add-method (print (the-type) instance stream)
           (format stream "#<~A ~A>"
                   the-name (get-weak-pointer instance))))
       types names))
```

A call to this function would look like `(add-simple-print-methods (list type operation locale) '("Type" "Op" "Loc"))`. It is worth pointing out that types are given names only in the user interface; types themselves are just anonymous objects, which is why it is easy to pass a list of them around. This example also shows than an `add-method` form can be nested inside of arbitrary code, even inside a loop. Because the type and operation slots of `add-method` are evaluated, we are able to provide the same method for a number of different types. Each instantiation of this method does the right thing because it is closed in its own environment with an appropriate binding for `the-name`.

2.8 Other features of Oaklisp

All of the usual Scheme functions and special forms are available. When a functional programming style is adopted, Oaklisp is essentially Revised↑3 Scheme with generic versions of predefined functions such as `length`.

3. Variables

Like every dialect of Scheme, Oaklisp is lexically scoped, with all variable references resolved at compile time. Lexical scoping allows upward funargs to work

correctly, and makes it possible to compile the language efficiently. Because variables live in a well-defined textual scope, programs are both more readable and more resistant to remotely induced bugs.

The lexical scoping rules for Oaklisp instance variables are designed to preserve these properties. Instance variables are introduced into the naming environment in exactly the same way that ordinary parameters are: by declaring them in the header of an `add-method` form. Although it sounds restrictive to insist that instance variables be declared before they can be used, the alternative would be to have names invisibly inserted into the lexical environment when an `add-method` boundary is crossed. This would not only make programs harder to read, it would allow the addition of a new variable to a far away type definition to suddenly shadow out a variable from a `let` around an `add-method`, or even a global variable.[3]

Modern dialects of Lisp are supposedly distinguished by their rationalized scoping systems. Unfortunately, the scoping class of variables may not be visually apparent. In Common Lisp [14], the expression `(let ((foo 3)) (bar foo))` could have counterintuitive semantics if the name `foo` had been declared special somewhere in the compilation environment.[4] To avoid the debugging difficulties inherent in this kind of system, programmers have adopted naming conventions such as putting stars around variables that are dynamically scoped. Oaklisp eliminates the problem by making it impossible to have fluid variables that look like lexical variables. As suggested in *The art of the interpreter* [13], fluid variables are referenced through a special form, so references to the fluid variable `foo` look like `(fluid foo)`. Since parameters and instance variables are strictly lexically scoped and fluid variables have a visually distinguishable form, every mystery variable in a piece of Oaklisp code is not really mysterious at all; it has to be an ordinary global variable that lives in the top-level namespace, which is structured using the mechanism discussed in the next section.

4. Locales

Multiple namespaces were one of the first modularity tools to be invented, and are a feature of almost every modern programming language. In C, for example, every file lives in a private namespace that inherits from the global namespace. Identifiers are exported to the global namespace with the `extern` form.

Many dialects of Scheme incorporate a more flexible facility based on first-class namespace objects, or locales [9]. This approach decouples files from namespaces and permits more elaborate patterns of identifier sharing. An Oaklisp locale specifies partial mappings from symbols to storage locations and from symbols to

[3]Object-oriented Lisps are susceptable to modularity problems of this sort if they are not carefully designed. For a detailed exploration of these issues, see [11].

[4]If `foo` has been declared special, the `let` introduces a binding with pervasive scope, and any reference to a global variable `foo` in the function `bar` is shadowed out.

macro expanders. The gross structure of Oaklisp's top-level namespace (Fig. 2) is provided by a mechanism that allows one locale to inherit the bindings of others. Finer control may be achieved by sending messages to locales, instructing them to alter their mappings. Because many compiler optimizations are legal only if it can be determined that the value of a given global variable will never be modified, locales record which variables the user has promised never to change. An autoloading facility is provided by a subtype of locale that lazily loads its bindings from a file.

It is worth pointing out that locales are not analogous to Common Lisp packages. In Common Lisp, there is only one top-level namespace; naming conflicts are prevented[5] by providing each symbol with an implicit prefix. This approach was necessary in the dynamically scoped ancestors of Common Lisp, since in a dynamically scoped Lisp there is a one-to-one correspondence between symbols and variables.

5. Uniform temporal semantics

Aside from syntactic and environmental issues, the difference between Pascal and Scheme lies in the uniformity of their temporal semantics. In Pascal, some things (e.g., creating a procedure) can be done only at compile time, while in Scheme, everything that can be done at compile time can be done at run time as well.

Historically, object-oriented features have destroyed the uniform temporal semantics of Lisp dialects into which they were incorporated. For example, in

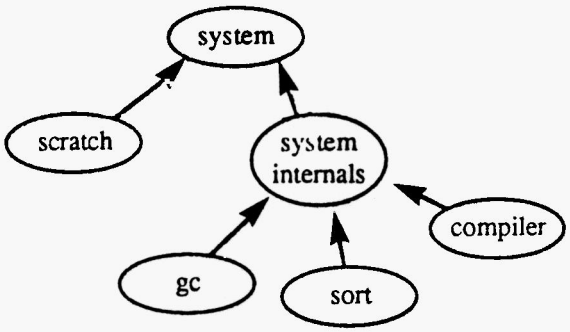

Fig. 2. The Oaklisp namespace configuration.

[5]Although packages solve the problem of conflicting global variable names, they interact poorly with the overloading of symbols in Lisp. For example, when a program imports a macro that uses noise words as part of its syntax, as does the MIT `loop` macro, the noise symbols must be imported along with the macro. These symbols might have bindings as variables or functions completely divorced from their role in the macro, and if the importing program is using these symbols for its own functions or variables, a conflict will result.

Flavors [6, 15] it is possible to define a new flavor or add a method to an existing flavor only at compile time; it would be unthinkable to put a `defmethod` form inside a loop. To retain Lisp's uniform temporal semantics while fully supporting the object-oriented paradigm, first-class types are necessary.[6]

```
(map plus (list 5 7)
          (list 14 19))

(map mix (list rear-scrolling-mixin neon-borders-mixin)
         (list geometry-window coke-monitor-window))
```

In Oaklisp, just as the `plus` operation can be mapped across two lists of numbers yielding a list of their sums, the `mix` operation can be mapped across two lists of types yielding a list of new types. Because types are first-class objects, the full power of the language is available for manipulating them. Special purpose, temporally inconsistent mechanisms (such as Flavors' `defflavor`) are unnecessary.

In addition to making the language simpler and more consistent, first-class types make it easy to extend the typing system in useful ways. For example, the Oaklisp coercion facility is based on coercible types, which are instances of a user defined subtype of `type`. A coercible type differs from other types in that it possesses an operation for coercing things to itself which it returns when sent a `coercer` message. Assuming that an appropriate coercion method has been defined, an object `x` can be coerced to type `vector` by writing `((coercer vector) x)`. For an example of a coercion method, consider the following code for coercing any sequence to a string:

```
(add-method ((coercer string) (sequence) self)
  (let* ((l (length self))
         (s (make string l)))
    (dotimes (i l)
      (set (nth s i) (nth self i)))
    s))
```

This coercion mechanism is both extensible and defined at the user level; there is no magic involved in either the definition of `coercible-type` or the definition of particular coercion methods. It works because types are first class.

Another advantage of making types first class is that they are anonymous, which makes it possible to have an automatic facility that manages the creation and tracking of a pool of composite types. For example, to find a type consisting of generic-window, flashing-borders-mixin, and active-margins-mixin, one could

[6]The designers of T were able to avoid breaking the clean semantics of Scheme by adding only one of the features normally associated with object-oriented programming: generic procedures. Because types were completely implicit, inheritance and textually dispersed method definitions could not be supported; T is not object-oriented in the same sense that Smalltalk is.

simply type (mix-types window-type-manager (list generic-window flashing-borders-mixin active-margins-mixin)), which eliminates the burden of remembering whether this type has already been defined and what it might be called.

6. Operations

An operation can be thought of as a generic function. When an operation is applied to an object, the type of the object is used to select a method that can perform the abstract computation specified by the operation. When exactly one universally applicable method has been defined for an operation, the operation is functionally equivalent to a lambda. Thus we see that lambdas are a special case of operations. Because Oaklisp operations are just objects that fit into the regular type system, it is possible to create new types of operations that respond to messages and that contain useful structure.

T was the first language to exploit this possibility by defining settable operations, which can be used to access and side-effect generalized locations. The operation for changing the value of a location is obtained by sending a setter message to the corresponding accessor operation. The advantage of settable operations is that only half as many operations need to be kept around and given names; it is easier to remember (setter car) than rplaca. This facility is used in conjunction with a macro that expands a set form whose location slot is a list into the appropriate call using setter. For instance, (set (car x) y) macro expands to ((setter car) x y), which sends the car operation a setter message and applies the returned operation to x and y.[7]

This theme can be developed further. In Oaklisp, the make-locative form is used to create locatives to variables, but (make-locative x) where x is a list expands to a call to a locater operation. For instance, (make-locative (car x)) expands to ((locater car) x). Locatable operations are settable as a matter of course, since the accessor operation on locatives is contents, which is settable. This fact is reflected in the Oaklisp operation type hierarchy, where locatable operations are a subtype of settable operations. Moreover, the preceding argument is put to use in the initialization code for locatable operations, which automatically provides accessor and setter methods in terms of the (as yet nonexistent) locater method. These methods simply invoke the locater operation and then use the resulting locative to

[7]Readers familiar with Common Lisp or ZetaLisp will note the similarity of this facility to setf. However, because setter deals with anonymous operations while setf deals with symbols, setf fails in situations such as

```
(defun set-positions (func l z)
   "Set the FUNC of successive members of L to Z+1, Z+2,
   (dolist (x l)
      (setf (funcall func x)
            (setq z (+ z l))))).
```

perform the appropriate side effect or access. Thus, when a locatable operation is created, it requires a method to be defined only for the operation's locater operation; the rest of the functionality defaults properly.

The Oaklisp compiler open codes some operations, such as `plus` and `(setter car)`, and constant folds others. To support these optimizations, a number of mix-ins are available for combining with the other operation types. The `open-coded-mixin` type adds information on how an operation should be open coded, while the `foldable-mixin` type allows its instances to be constant folded.

7. Lisp types and inheritance

Until recently, most object-oriented versions of Lisp were created by adding a separate message facility to an existing language. Languages of this sort had two problems: object-oriented code had to be written in a sort of pidgin Lisp that lacked expressive power, and all of the traditional Lisp data types were implemented in an ad hoc manner outside of the object-oriented type system. The resulting dichotomy between the two sides of the language compromised the traditions of openness and extensibility long enjoyed by both Lisp and Smalltalk.

CommonLoops [3] demonstrates that with enough effort and machinery it is possible to do a good job of making an existing Lisp uniformly object-oriented. An easier way of ensuring that a Lisp dialect is completely object-oriented is to define an object-oriented kernel with the right semantics[8] and then use its extension facilities to build all of the necessary Lisp data types. Because inheritance provides an underlying structure for the types, it it possible to regularize the system and introduce abstract types at appropriate points in the hierarchy. The usual Lisp predicates and functions can then be defined generically at just the right level of abstraction, leaving the system open to the creation of generalized versions of traditional types. For example, the types and functions necessary for list processing are defined in the portion of the Oaklisp type hierarchy diagrammed in Figure 3.

`Pair` is an abstract type that is never directly instantianted. Methods for printing and mapping are defined for `pair` and are shared by all of its subtypes, while the subtypes themselves are responsible for handling `car` and `cdr` messages. Ordinary cons cells are instances of the type `cons-pair`, but other useful subtypes of `pair` can be defined as well. For example, the following program fragment sets up a type of lazy pair that only computes its car and cdr when they are actually needed.

```
(set lazy-pair (make type '(car-thunk car-flag
                            cdr-thunk cdr-flag)
                          (list pair object)))
```

[8]Smalltalk would not be appropriate because it does not support upward funargs.

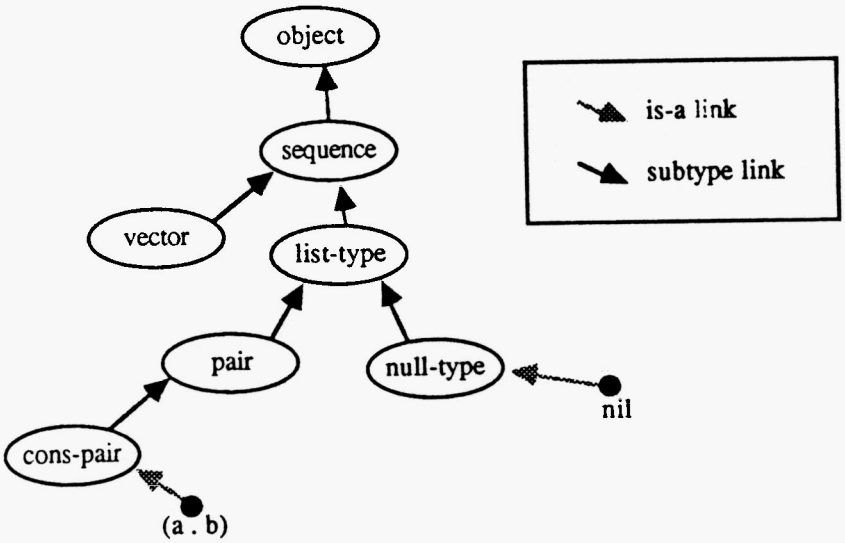

Fig. 3. The Oaklisp cons hierarchy.

```
(add-method (initialize (lazy-pair car-thunk car-flag
                                   cdr-thunk cdr-flag)
                        self thunk1 thunk2)
  (set car-flag nil)
  (set cdr-flag nil)
  (set car-thunk thunk1)
  (set cdr-thunk thunk2)
  self)

(add-method (car (lazy-pair car-thunk car-flag) self)
  (cond (car-flag car-thunk)
        (else (set car-thunk (car-thunk))
              (set car-flag t)
              car-thunk)))
```

`Lazy-pair` uses a call by need strategy, in which thunks are used to represent the potential values of the car and cdr until they are needed. For example, to create an infinite list of squares we can write

```
(define (make-omega-squares-from n)
  (make lazy-pair (lambda () (* n n))
                  (lambda () (make-omega-squares-from
                     (+ n 1)))))
(set infinite-squarelist (make-omega-squares-from 0))
```

If we were to now print `infinite-squarelist`, "(0 1 4 9 16 25 36 49 . . .)" would appear on the screen, since `lazy-pair` has inherited the print method defined for `pair`, which already knows how to abbreviate long lists using the "..." syntax.

8. Implementation and future work

The current implementation of Oaklisp is based on a bytecode machine for portability. References are 32 bits long, with the 2 low bits devoted to tags. Our compiler differs from those written for purely instructional dialects of Scheme by emphasizing runtime efficiency rather than hooks for the debugger. For instance, appropriate `labels` forms are compiled as tight loops that do not generate lambdas. This orientation also led us to provide for low-level access to native machine resources. Our implementation runs under Unix™ and on the Apple Macintosh.™

Although we currently dispatch only on the type of the first argument, we are modifying the language to allow dispatching on the types of multiple arguments. The syntax of `add-method` has been extended to (`add-method` (*operation* [[(*type*. *ivar-list*)] *arg*]*) . *body*), and the appropriate modifications to the implementation should be in place before this paper appears.

In addition to this minor linguistic enhancement, our colleague Bruce Horn is planning a number of technological upgrades, including a fancy user interface. Our vision is closer to the Smalltalk user interface than to that of current Lisp machines. By using a model-view-controller [4] paradigm we hope to integrate the functionality normally associated with debuggers and inspectors by giving continuations appropriate viewers and controllers.

We are also considering the addition of futures [2, 5] to the language, one of the motivations being the possibility of using them as tokens for swapped out objects in an object based virtual memory system. This facility would allow the Macintosh implementation to swap to disk, and give users fine control over paging policies.

9. Conclusion

Scheme demonstrated that Lisp could be made both simpler and more powerful by rationalizing its scoping rules and making procedures truly first class. However, Lisp still suffers from its ad hoc type system, and attempts to enhance the language with object-oriented features usually compound the problem. Oaklisp shows that Lisp can benefit from having a clean type system designed with Scheme aesthetics in mind.

References

1. Abelson, H. et al. The revised revised report on Scheme. Tech. Rept. AI Memo 848, MIT AI Lab, Cambridge, MA, 1985.

2. Baker, H.G. Jr. Actor systems for real-time computation. Tech. Rept. TR-197, MIT Laboratory for Computer Science, Cambridge, MA, March, 1978.
3. Bobrow, D. et al. CommonLoops: Merging Common Lisp and object-oriented programming. ACM Conference on Object-Oriented Systems, Programming, Languages and Applications, September 1986, pp. 17-29.
4. Goldberg, A.J., and Robson, D. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.
5. Lang, K.J., and Pearlmutter, B.A. Oaklisp: An object-oriented scheme with first-class types. ACM Conference on Object-Oriented Systems, Programming, Languages and Applications, September 1986, pp. 30-37.
6. Moon, D.A. Object-oriented programming with Flavors. ACM Conference on Object-Oriented Systems, Programming, Languages and Applications, September 1986, pp. 1-8.
7. Halstead, R.H. "Multilisp: A language for concurrent symbolic computation". *Transactions of Programming Languages and Systems 7,* 4 (Oct. 1985), 501-538.
8. Lang, K.J., and Pearlmutter, B.A. The Oaklisp language and implementation manuals. Tech. Rept. CMU-CS-87-103, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, 1987.
9. Rees, J.A., and Adams, N.I., IV. T: A dialect of Lisp or, Lambda: The ultimate software tool. ACM Symposium on Lisp and Functional Programming, August 1982.
10. Rees, J.A. et al. *The T Manual.* Yale University Computer Science Department, New Haven, CT, 1984.
11. Snyder, A. Encapsulation and inheritance in object-oriented programming languages. ACM Conference on Object-Oriented Systems, Programming, Languages and Applications, September 1986, pp. 38-45.
12. Steele, G.L., Jr. Lambda: The ultimate declarative. Tech. Rept. AI Memo 379, MIT AI Lab, Cambridge, MA, 1976.
13. Steele, G.L., Jr., and Sussman, G.J. The art of the interpreter. Tech. Rept. AI Memo 453, MIT AI Lab, Cambridge, MA, 1978.
14. Steele, G.L., Jr. *Common Lisp—The Language.* Digital Press, Burlington, MA, 1984.
15. *Symbolics Release 7 Documentation,* Vol. 2A. Symbolics, Inc., Cambridge, MA, 1986.