Object-Oriented Programming
Versus
Abstract Data Types

William R. Cook
Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA, 94303-0969, USA

Abstract: This tutorial collects and elaborates arguments for distinguishing between object-oriented programming and abstract data types. The basic distinction is that object-oriented programming achieves data abstraction by the use of *procedural abstraction*, while abstract data types depend upon *type abstraction.* Object-oriented programming and abstract data types can also be viewed as complementary implementation techniques: objects are centered around the constructors of a data abstraction, while abstract data types are organized around the operations. These differences have consequences relating to extensibility, efficiency, typing, and verification; in many cases the strengths of one paradigm are the weaknesses of the other. Most object-oriented programming languages support aspects of both techniques, not a unification of them, so an understanding of their relative merits is useful in designing programs.

Published in the Proceedings of the REX Workshop/School on the Foundations of Object-Oriented Languages, LNCS 173, Springer-Verlag, 1990, pp. 151-178.

1 Introduction

The development of abstract data types and object-oriented programming, from their roots in Simula 67 to their current diverse forms, has been prominent in programming language research for the last two decades. This tutorial is aimed at organizing and collecting arguments that distinguish between the two paradigms. The focus of the arguments is on the basic mechanisms for data abstraction, illustrating the differences with examples. Although more advanced topics, like inheritance, overloading, and mutable state, are important features of one or the other paradigm, they are not considered in this presentation. The interpretations of “abstract data type” and “object-oriented programming” compared in this paper are based upon major lines of development recorded in the literature and in general use.

Abstract data types are often called *user-defined data types*, because they allow programmers to define new types that resemble primitive data types. Just like a primitive type *INTEGER* with operations $+$, $-$, $*$, etc., an abstract data type has a type domain, whose representation is unknown to clients, and a set of operations defined on the domain. Abstract data types were first formulated in their pure form in CLU [31, 30]. The theory of abstract data types is given by *existential types* [33, 13]. They are also closely related to algebraic specification [20, 23]. In this context the phrase “abstract type” can be taken to mean that there is a *type* that is “conceived apart from concrete realities” [41].

Object-oriented programming involves the construction of *objects* which have a collection of *methods*, or procedures, that share access to private local *state*. Objects resemble machines or other things in the real world more than any well-known mathematical concept. In this tutorial, Smalltalk is taken as the paradigmatic object-oriented language. A useful theory of objects associates them with some form of *closure* [2, 37, 8], although other models are possible [26]. The term “object” is not very descriptive of the use of collections of procedures to implement a data abstraction. Thus we adopt the term *procedural data abstraction* as a more precise name for a technique that uses procedures as abstract data. In the remainder of this paper, procedural data abstraction (PDA) will be used instead of "object-oriented programming". By extension, the term "object" is synonymous with *procedural data value*.

It is argued that abstract data types and procedural data abstraction are two distinct techniques for implementing abstract data. The basic difference is in the mechanism used to achieve the abstraction barrier between a client and the data. In abstract data types, the primary mechanism is *type* abstraction, while in procedural data abstraction it is *procedural* abstraction. This means, roughly, that in an ADT the data is abstract by virtue of an *opaque* type: one that can be used by a client to declare variables but whose representation cannot be inspected directly. In PDA, the data is abstract because it is accessed through a procedural interface – although all of the types involved may be known to the user. This characterization is not completely strict, in that the type of a procedural data value can be viewed as being partially abstract, because not all of the interface may be known; in addition, abstract data types rely upon procedural abstraction for the definition of their operations.

Despite the very different approaches taken by ADT and PDA, they can be understood as orthogonal ways to implement a specification of a data abstraction. A data abstraction can be characterized in a very general way by defining *abstract constructors* together with *abstract observations* of the constructed values. Using these notions, a data abstraction may be defined by listing the value of each observation on each constructor. The difference between PDA and ADT concerns how they organize and protect the implementation of a data abstraction. The choice of organization has a tremendous effect on the flexibility and extensibility of the implementation.

ADTs are organized around the *observations*. Each observation is implemented as an operation upon a concrete representation derived from the constructors. The constructors are also implemented as operations that create values in the representation type. The representation is shared among the operations, but hidden from clients of the ADT.

PDA is organized around the *constructors* of the data abstraction. The observations become the attributes, or methods, of the procedural data values. Thus a procedural data value is simply defined by the combination of all possible observations upon it.

Viewing ADTs and PDA as orthogonal ways to organize a data abstraction implementation provides a useful starting point for comparing the advantages and disadvantages of the two paradigms. The decomposition used, either by constructor or observer, determines how easy it is to add a new constructor or observer. Adding a new feature that clashes with the organization is difficult. In addition, the tight coupling and security in an ADTs makes it less extensible and flexible, but supports verification and optimization. The independence of PDA implementations has the opposite effect. However, in several cases where PDA might be expected to have difficulty, the problems are lessened by the use of inheritance and subtyping.

Most existing programming languages that support PDA also make use of ADTs. Smalltalk, for example, is based upon efficient but inflexible ADTs for small integers and arrays. The PDA numbers and collections are constructed upon this base. Simula and C++ support both ADTs and PDA in the same framework; the effect is more of interweaving than unification, since the trade-offs between the two styles are still operative. The explicit opaque types in Modula-3 are based on objects, so good programming style seems to call for implementing ADTs using objects [5, 10]. CLOS also supports a mixture of the two forms in an even more flexible way with multimethods and multiple inheritance.

The next section traces the history of ADTs and procedural data abstraction. Section 3 outlines a matrix form of specification for abstract data, and identifies ADTs and PDA as orthogonal decompositions of this matrix. Section 4 compares the two techniques with respect to incremental programming, optimization, typing, and verification. Section 5 discusses the use of ADT and PDA techniques in existing programming languages. Finally, Section 6 summarizes the important points of the paper.

2 Historical Overview

In 1972, David Parnas published his seminal work on modularization [35]. He showed the value of decomposition of a system into a collection of modules supporting a procedural interface to hidden local state. He pointed out the usefulness of modules for facilitating modification or evolution of a system. His specification technique [36] for describing modules as abstract machines has not been generally adopted, but the module concept has had a great impact, especially on the development of languages like Modula-2 [44]. Although Parnas recognized that modules with compatible interfaces can be used interchangeably, he did not develop this possibility. As a result, modules are not first-class values, so they cannot be passed as arguments or returned as values.

In 1973, Stephen Zilles published a paper on "Procedural abstraction: a linguistic protection technique, " [45] which showed "how *procedures can be used to represent* another class of system components, *data objects*, which are not normally expressed as programs" (emphasis added). His notion of procedural abstraction is very similar to Parnas's modules; however, he views them as data and discusses passing them as arguments to other procedures, and returning them as values. He also noted the similarity to objects in Simula. He illustrated them by discussing *streams* represented as a vector of procedures with local state. Calling an operation was defined as an indirect procedure call through the vector. He shows that different classes of stream objects can be defined by building an appropriate vector of procedures. He also presents two of the main methodological advantages of objects: encapsulation and independence of implementations.

The following year, in 1974, Zilles published an influential paper with Barbara Liskov on ADTs and CLU [31]. Gone was any mention of PDA; type abstraction had taken its place. The formalism of ADTs was still presented as closely related to Simula; the main difference was claimed to be that Simula allowed full inspection of object representations. The publication of this paper initiated a decade of intense research on ADTs, which soon branched into work on languages [30, 34], algebraic specification [20, 23, 24, 18, 19], and existential types [33].

In 1975, John Reynolds published a paper called "User-defined data types and procedural data structures as complementary approaches to data abstraction" [39] in which he compares procedural data abstraction to user-defined data types. He argued that they are complementary, in that they each has strengths and weaknesses, and the strengths of one are generally the weaknesses of the other. In particular, he found that PDAs offer extensibility and interoperability but obstruct some optimizations. ADTs, on the other hand, facilitate certain kinds of optimizations, but are difficult to extend or get to interoperate. He also discussed the typing of the two approaches, and identified recursion in values and types as characteristic of PDA. One limitation of his presentation is that the objects in his examples only have a single method. The introduction of a second method was described as an intellectual "tour de force", implying that multiple methods are too complicated for use in practical designs.

After 1975 little was written that related to the theory of object-oriented programming, while investigation of ADTs continued. Yet development of object-oriented languages, like Smalltalk [21] and Flavors [40], continued, especially in the context of extensible, interactive, open systems which encouraged user programming.

Theoretical interest in object-oriented programming was sparked in 1984 by Cardelli's paper on "The semantics of multiple inheritance" [9]. This paper identified the notion of *subtyping* as central to an understanding of object-oriented programming. Subtyping and parametric polymorphism were combined to form *bounded quantification*, which could describe aspects of update operations on records [13]. The study of various calculi for records operations and subtyping has continued along a number of lines [42, 43, 38, 12].

A good explanation for the complementarity noted by Reynolds was presented by Abelson and Sussman [1] in 1985, although they do not cite his work. They discuss "data-oriented programming" as a technique for writing flexible and extensible programs in Lisp. They note that abstractions are characterized by observations and representations, where the operation needed to perform an observation depends upon the representation. Data-oriented programming works by grouping all the observations on a particular representation together as components, or methods, of a value containing that representation. This is in contrast to operation-oriented programming, or ADT programming, where a function is written for each observation with cases for each representation. By organizing the observations and constructors into a two-dimensional matrix, it becomes clear that ADTs and object-oriented programming arise from a fundamental dichotomy: there are two ways to organize this table: either by observers for ADTs or by constructors for PDAs.

There are number of other papers that contribute to the comparison of ADTs and PDA. A distinction between ADTs and procedural data abstraction is present in the work on PS-algol, which can implement both techniques using persistent procedures [4]. Danforth and Tomlinson [17] survey the work on type

	constructor of *s*	
observations	*nil*	*adjoin(s′, n)*
null?(s)	true	false
head(s)	**error**	*n*
tail(s)	**error**	*s′*

Figure 1: An observer/constructor specification for lists.

	Constructor of l	
Constructor of m	*nil*	*adjoin(l′, x)*
nil	*true*	*false*
adjoin(m′, x)	*false*	*(x=y)* **and** *equal(l′, m′)*

Figure 2: Specification of the binary *equal(l, m)* observation.

and subtype theories for ADTs and early models of data objects without procedural components. They conclude that neither is an adequate explanation of PDA or inheritance. The proposal of *exemplars* for Smalltalk is a clear presentation of the PDA model [28]; it is also the inspiration for the basic *list* example used in this tutorial.

3 Distinguishing ADTs and PDA

3.1 Data Abstraction

Data abstraction refers to a range of techniques for defining and manipulating data in an abstract fashion. Abstract data is useful because the conceptual view of the properties of data are often very different from the properties of the data's detailed representation in a computational system. In this paper *immutable* data abstractions are discussed, however, many of the same considerations apply to mutable data abstractions, or state abstractions.

A general view of abstract data is based on the notion of abstract *constructors* and abstract *observations*. The constructors create or instantiate abstract data, while the observations retrieve information about abstract data. The behavior of a data abstraction is specified by giving the value of each observation applied to each constructor. This information is naturally organized as a matrix with the constructors on one axis and the observers on the other. Each cell in the matrix defines the behavior of the abstraction for a given observer/constructor pair.

As an example, consider a data abstraction for integer lists. The constructors are *nil*, which constructs an empty list, and *adjoin*, which takes a list and an integer, and forms a new list with the integer added to the front of the list argument. The observers are *null?*, *head*, and *tail*. *Null?* is a predicate that returns true if its argument is the empty list; i.e. if it is equal to *nil*. *Head* returns the first integer in a non-empty list. *Tail* returns the rest of a non-empty list.

The behavior of the observers on each constructor is given in Figure 1. The cells of the matrix contain values defined in a Pascal-like syntax. The variables *x*, *y*, and *z* are used to represent integers, while *l*, *m*, and *n* denote lists. The value of the *null?* observation on the *nil* constructor is *true*, and on the *adjoin* constructor it is *false*. The *head* and *tail* observations on *nil* both result an error condition; how such errors are treated is not specified.

	Constructor of l	
Observations	*nil*	*adjoin(l′, x)*
equal(l, m)	*null?(m)*	**not** *null?(m)* **and** $x =$ *head(m)* **and** *equal(l′, tail(m))*

Figure 3: Behavior of *equal(l, m)* with cases only for *l*.

	constructor of s	
observations	*nil*	*adjoin(s′, n)*
null?(s)	true	false
head(s)	**error**	n
tail(s)	**error**	s'

Figure 4: The list specification decomposed into observations.

In Figure 1, all of the observations are *unary*, in that they observe only a single value of abstract data. It is often necessary to have more complex observations. To compare two values, it is necessary to observe pairs of abstract data. This complicates the specification, because the value of the observation must be defined for all combinations of possible constructors for the abstract values being compared. This technique is illustrated for the binary *equal* observation in Figure 2. This table can be viewed as adding a third dimension to the basic table for the unary observations.

This third dimension can be removed by encoding its cases directly into the behavioral specification of the two-dimensional case. Such a flattened definition for *equal* is given in Figure 3. In this version, an explicit classification is made only on the first argument of the observation, while the second is referenced abstractly with *null?*, *head*, and *tail*.

Using this framework, a mutable data abstraction could be specified by the addition of an abstract state. The specification would then include not just the value of the observations, but their effect upon the abstract state of their arguments.

While such tabular presentations of a data abstraction have certain advantages, it is also useful to decompose the table into more modular units. There are two immediately obvious ways to do this, by partitioning the table into horizontal or vertical slices. Decomposition has the advantage of reducing the table to a collection of smaller, uniform pieces. One disadvantage of decompositions is that the symmetry of the table is lost.

3.2 Abstract Data Types

3.2.1 Decomposition by Observations

ADTs can be viewed as a decomposition of specification matrix into observations, as illustrated in Figure 4. This slices the table horizontally into a stack of rows, where each row collects the information about a single observer together in a unit. Information about a given constructor is spread across components.

Each observation is formed into an independent operation that returns the appropriate result when applied to any of the constructors' values. The constructors are also included as operations. The connection between the constructors and the observations is via a shared *representation*. In order to keep the representation abstract, its structure is hidden from clients of the ADT. The clients can only create val-

```
adt IntList
representation
    list = NIL | CELL of integer * list
operations
    nil = NIL

    adjoin(x : integer, l : list) =
        CELL(x, l)

    null?(l : list) = case l of
        NIL  ⇒  true
        CELL(x, l)  ⇒  false

    head(l : list) = case l of
        NIL  ⇒  error
        CELL(x, l′)  ⇒  x

    tail(l : list) = case l of
        NIL  ⇒  error
        CELL(x, l′)  ⇒  l′

    equal(l : list, m : list) = case l of
        NIL  ⇒  null?(m)
        CELL(x, l′)  ⇒  not null?(m)
                         and x = head(m)
                          and equal(l′, tail(m))
```

Figure 5: Implementation of an ADT for lists.

ues of the type by using the constructors, and inspect them only with observer operations. The concrete representation is usually derived from the form of the constructors, but alternative representations are also possible. Since they all share access to the real representation of the abstract type, the operations are tightly coupled.

3.2.2 Implementing ADTs

A wide variety of languages support the implementation of abstract data types. These languages include use of private types in Ada packages, Clu clusters, ML abstype definitions, and opaque types in Modula-2. The overall structures of these facilities are very similar. The key element is of course that the representation of abstract values is hidden from users of the operations. Exactly how the representation type is defined and how the operations are implemented depends upon the data types and control structures of the language.

Figure 5 defines an ADT implementing integer lists. The syntax is based loosely on ML. The ADT has two distinct parts: a representation and a set of operations. The representation is defined as a labeled union type, or variant record, with cases named *NIL* and *CELL*. The *NIL* variant is simply a constant, while the *CELL* variant contains a pair of an integer and a list.

The constructors *nil* and *adjoin* are defined as operations that build appropriate representation values. The observations are defined by a case statement over the representational variants which returns the appropriate value from the specification. *null?* is a query operation that determines if a list is nil. *head* and *tail* are accessors which return the first integer in the list, and the abstract list representing the tail, respectively. The equality operation has a case statement over its first argument but uses operations to query its second argument. Improvements in the implementation of *equal* are discussed in Section 4.3.

	constructor of s	
observations	*nil*	*adjoin*(s', n)
null?(s)	true	false
head(s)	**error**	n
tail(s)	**error**	s'

Figure 6: The list specification decomposed into constructors.

A client of the ADT is able to declare variables of type *list* and use the operations to create and manipulate list values.

var l : list = adjoin(adjoin(nil, 4), 7);
if equal(nil, l) **then** ...

However, the client cannot inspect the representation of list values. Attempting to determine if *l* is equal to *NIL* would be rejected by the compiler as a violation of the encapsulation of the ADT.

3.3 Procedural Data Abstraction

3.3.1 Decomposition by Constructors

PDA can be viewed as a decomposition of a data abstraction specification into constructors, as illustrated in Figure 6. This slices the table into a series of columns, each of which collects all the information about a given constructor into a unit. The values of the different observations are spread across the constructors.

Each constructor is converted into a template, or *class*, for constructing procedural data values, or objects. The arguments to the constructor become the local state, or *instance variables* of the procedural data value. In the list example the *nil* object has no local state, while the *cons* object has two pieces of state: one to hold the integer value and the other to hold the rest of the list.

The observations become components or fields of the objects made by a constructor. The observations are often called *attributes* or *methods*. Each object is represented as the combination of the observations applicable to it. Since the result of an observation on a constructor is what a client is interested in, only the mechanism for producing the observed value needs to be hidden from the client.

In organizing the matrix in this way, a case discrimination is done on the operation to be performed, not on the constructor representations as in an ADT. Since operations are visible to the user, there is no need for a hidden case statement: the user can simply select the appropriate observation directly.

3.3.2 Implementing PDA

As combinations of procedural observations with shared local state, PDAs are naturally implemented as *closures* containing records of procedures [37]. The procedures are derived from the specification of the data abstraction. The record is formed by using the observations as field names and the procedures as values. The closure is used to encapsulate the constructor's arguments, which act as local state for the procedures.

The class constructs in most PDA languages can be viewed as special mechanism for creating closures of records. This form of closure is different from the kind commonly found in functional languages like Lisp for two reasons. First, in functional languages it is often only possible to form closures over functions, not records. Thus records must be simulated as an explicit case over a set of field names [2]. Second, the closures are not a general construct of the language, but are provided only within the class construct.

```
Nil = recursive self = record
      null? = true
      head = error;
      tail = error;
      cons = fun(y) Cell(y, self);
      equal = fun(m) m.null?
      end

Cell(x, l) = recursive self = record
      null? = false
      head = x;
      tail = l;
      cons = fun(y) Cell(y, self);
      equal = fun(m) (not m.null?)
                         and (x = m.head)
                          and l.equal(m.tail) end
```

Figure 7: Implementation of lists as PDAs.

The two constructors for list objects are defined in Figure 7. The constructor functions, *Nil* and *Cell*, return record values. The constructor for cells takes two arguments, *x* and *l*, which play the role of instance variables of the object. In this example they are not changed by assignment, though there is no essential reason why they could not be modified (if, for example, a *set-head* method were introduced).

Each constructor creates a record with fields named *null?*, *head*, *tail*, *cons*, and *equal*. The implementation uses recursive records, where the identifier *self* refers to the record being constructed. An observation *m* on an object *o*, which is written *m(o)* in the specification, is implemented by selecting the *m* field of the object: *o.m*. Explicit functional abstraction is introduced to represent the methods: the notation **fun***(x)b* represents a functions of one argument named *x*. Unlike the ADT implementation, there are no explicit case statements in the PDAs. An implicit case statement is used to select the appropriate observation from the objects.

There are several levels of recursion in the implementation [7]. The *Nil* and *Cell* objects pass themselves as an argument to the *Cell* constructor, in response to a *adjoin* message. In addition, the *Cell* constructor function is itself recursive, because it is called from within the cell *adjoin* method. It is also possible for objects to return themselves as values of a method, as is common in Smalltalk. An additional layer of recursion, in the types of objects, will be discussed in Section 4.4.

A client of the PDA creates objects and sends requests, or messages, to them.

```
var l = Nil.adjoin(4).adjoin(7);
if Nil.adjoin(8).equal(l) then ...
```

As in the case of ADTs, the client cannot inspect the internal representation of list values, although the external format of all the messages is known.

4 Comparing ADTs and PDA

The difference between ADTs and procedural abstraction involve both the client's use of the abstraction and the implementors definition of the abstraction. The differences are illustrated in the areas of incremental programming, optimization, typing, and verification.

The client has an abstract view of data in both ADTs and PDA. The major difference between them is the technique used to enforce the encapsulation and abstraction. In an ADT the mechanism is *type* abstraction, while in PDA it is *procedural* abstraction. Another major difference is that in PDA the objects act as clients among themselves, and so are encapsulated from each other. In an ADT, the abstract values are all enclosed within a single abstraction, and so they are not encapsulated from each other.

4.1 Adding New Constructors

Adding new or different constructor to a data abstraction can extend the semantics of the abstraction or provide a more efficient description of an existing construction. For example, lists of random numbers, intervals of sequential integers, or a list that is computed from a function might be added to the basic lists. These examples provide efficient representations for lists that would be cumbersome to represent explicitly. A constructor for explicit circular lists would also constitute an extension, because they cannot be represented in the original specification. The random and functional lists may also extend the specification to allow infinite lists. It is also useful to allow a constructor to be part of more than one specification. This can be viewed as allowing a constructor from one specification to be added into another specification.

The question of whether a client can add a new constructor, or whether the addition must be made by the implementor of the abstraction. If the client can add a constructor, the new object should be allowed to intermix with existing object.

In general, it is much easier to a new construction to a data abstraction implemented as PDA than as an ADT. The issues are illustrated by adding a constructor for intervals of integers to the previously defined list abstractions. The representation for an interval of integers is as a pair of bounding integers.

4.1.1 Extending an ADT Representation

Adding a new constructor to an ADT involves extending its concrete representation. This, in turn, requires that all of the case statements in the operations be extended to cover the new representational variant. This effect is illustrated in Figure 8, where the *INTERVAL* representation is added throughout the implementation.

A new operation, *interval*, is introduced to construct interval lists; it tests whether the bounds define an empty interval, returning *nil* if they do. Since the *tail* operation is extended to call the abstract *interval* constructor rather than simply create an *INTERVAL* representation directly, it will return *NIL* when the interval becomes empty.

Adding a new constructor to an ADT requires intrusive changes to the existing implementation. This is because the ADT paradigm organizes the observer/constructor matrix according to operations, so that adding a constructor clashes with the natural structure of the implementation.

4.1.2 Adding a New PDA Constructor

Adding a new constructor in an PDA is easy, and is common practice in such programs. The representation is decentralized, with each object specifying its own local state. For example, intervals can be created by defining a new object constructor, as in Figure 9.

The constructor simply returns the *Nil* object if the interval is empty. Otherwise it constructs an object with methods that return the first number in the interval or construct a new interval object. By interpreting records, the construction of new interval objects can be made lazy: they are only created as needed. A practical realization of this would involve inserting an empty function abstraction in front of each field value.

The PDA can even be extended beyond the original specification of the data abstraction, to allow infinite lists. One implementation of infinite lists is as a *stream*, whose elements are successive applications of a transition function. A *Stream* constructor is defined in Figure 10. The state of the stream is given by an initial value and the transition function. A program designed to use only the original list implementations *Nil* and *Cell* will operate just as well on intervals or streams. Of course, if the equality observation is included, then it may diverge if applied to a stream.

```
adt IntList
representation
    list = NIL | CELL of integer * list
            | INTERVAL of integer * integer

operations
    nil = NIL

    interval(x : integer, y : integer) =
        if x ≤ y then INTERVAL(x, y) else nil

    null?(l : list) = case l of
        NIL  ⇒  true
        CELL(x, l)  ⇒  false
        INTERVAL(x, y)  ⇒  false

    head(l : list) = case l of
        NIL  ⇒  error
        CELL(x, l′)  ⇒  x
        INTERVAL(x, y)  ⇒  x

    tail(l : list) = case l of
        NIL  ⇒  error
        CELL(x, l′)  ⇒  l′
        INTERVAL(x, y)  ⇒  interval(x+1, y)

    adjoin(x : integer, l : list) =
        CELL(x, l)

    equal(l : list, m : list) = case l of
        NIL  ⇒  null?(m)
        CELL(x, l′)  ⇒
            not null?(m)
             and x = head(m)
              and equal(l′, tail(m))
        INTERVAL(x, y)  ⇒
            not null?(m)
             and x = head(m)
              and equal(tail(l), tail(m))
```

Figure 8: Implementation of lists with intervals as an ADT.

4.2 Adding New Observations

A new observation can extend the functionality of a specification or provide more efficient access to specific information. The example given here is the addition of a *length* observation to the list abstraction.

Since ADTs are organized around observations, it appears that it should be easy to add new observations. Conversely, it should be difficult to one to a PDA. However, this is not always the case.

4.2.1 Adding Observations to an ADT

Adding new observations to an ADT is complicated by the fact that the hidden representation must be shared by all operations. In some way the new observation operation must be inserted within the scope of the representation type. Existing languages do not support this capability, but there is little to prevent a simple language extension to allow it. Without this extension, the definition of the ADT must be changed to insert the new code into the scope of the representation.

```
Interval(x : integer, y: integer) =
    if x > y then
        Nil
    else
        recursive self = record
            null? = false
            head = x;
            tail = Interval(x+1, y)
            cons = fun(z) Cell(z, self);
            end
```

Figure 9: A PDA for interval lists.

```
Stream(f : integer → integer, x : integer) =
    recursive self = record
        null? = false
        head = x;
        tail = Stream(f, f(x))
        cons = fun(y) Cell(y, self);
        end
```

Figure 10: A PDA for infinite streams.

Adding a length operation involves inserting the following operation defined in Figure 11 into theADT defined in Figure 5.

4.2.2 Adding PDA Observations

To add a length message to the PDA lists, it is necessary to add it to each constructor. Even though the addition of an operation clashes with the natural organization of PDA programs according to constructors, the mechanism of inheritance is available to make this easier. An example will illustrate this simple use of inheritance to add an operation. The precise meaning of this inheritance mechanism is omitted (see [15, 16]).

A new family of object constructors is defined in Figure 12 by reference to the original implementations in Figure 7 that adds the length field.

4.3 Optimizing Operations

Optimizing operations is an important consideration in programming. One of the benefits of abstraction is that some optimizations can be performed in isolation within an abstraction. However, abstraction can also prevent optimization because it prevents access to the information on which the optimization would be based.

When the interval representation is added to the lists, the equality operation becomes very inefficient because it must create the complete sequence of numbers in the interval. It is easier to optimize the ADT implementation because the list values are not encapsulated from each other as they are in the procedural data abstraction.

```
length(l : list) = **case** l **of**
    NIL  ⇒  0
    CELL(x, l′)  ⇒  1+length(l′)
```

Figure 11: A length operation for the list ADT.

```
NilWithLength =
    inherit Nil
    with [
        length = 0
        ]

CellWithLength(x, l) =
    inherit Cell(x, l)
    with [
        length = l.length+1
        ]

IntervalWithLength(x : integer, y: integer) =
    inherit Interval(x, y)
    with [
        length = (y - x + 1)
        ]
```

Figure 12: Adding a length operation to list constructors using inheritance.

4.3.1 Optimizing ADTs

In the ADT it is possible to improve the efficiency, because the representations of both arguments to the equality function may be inspected. The equality operation in the list ADT can be improved by adding cases for the constructors of both arguments to the operation. Previously, all operations performed a case statement on only their first argument. By using case statement on both arguments of the *equal* operation, as shown in Figure 13, a much more efficient comparison of intervals is possible. This is still not the most efficient implementation possible, but it does illustrate examination of more than one representation.

4.3.2 Optimizing PDAs

In PDA it is much more difficult to optimize operations, because the representation of argument to the equality observation cannot be determined. For example, the equality method on an interval object cannot be optimized because there is no way to determine if its argument is also an interval. This is the cost of the flexibility of objects.

The optimization of methods is only one form of optimization; addition of specialized representations can also be viewed as a form of optimization, which is supported by objects. Another promising approach involves compilation techniques that create special code for common combinations of arguments to methods [25, 22, 14].

Direct optimization of methods is possible in some cases. By adding additional messages to the object, it is possible for other objects to query these messages and perform more efficiently. These messages can easily degenerate into simply specifying representational details. The trick is to define sufficiently abstract queries that provide quick answers for some implementations, while not prohibiting other implementations. To give a simple example illustrating this, consider the addition of an append

```
equal(l : list, m : list) =
    **case** l **of**
        NIL  ⇒  null?(m)
        CELL(x, l′)  ⇒
            (**not** null?(m))
              **and** (x = head(m))
               **and** equal(l′, tail(m))
        INTERVAL(x, y)  ⇒
            **case** m **of**
                NIL  ⇒  *false*
                CELL(y, m′)  ⇒
                    (x = y) **and** equal(tail(l), m′)
                INTERVAL(x′, y′)  ⇒
                    (x = x′) **and** (y = y′)
```

Figure 13: Efficient comparison of ADT intervals.

operation to the list ADT and PDA. A simple ADT operation for append is easily defined.

```
append(l : list, m : list) = case l of
    NIL  ⇒  m
    CELL(x, l′)  ⇒  new adjoin(append(l′, m), x)
```

This operation is inefficient because it will copy its first argument in cases where the second argument is null. Thus a better implementation will check its second argument as well.

```
append(l : list, m : list) = case l of
    NIL  ⇒  m
    CELL(x, l′)  ⇒
        case m of
            NIL  ⇒  l
            CELL(y, m′)  ⇒
                new adjoin(append(l′, m), x)
```

This optimization can be implemented in the PDA version because the *nil* operation is already present for determining when a list is null.

```
NilWithAppend =
    inherit Nil
    with [
        append = fun(m) m
        ]

CellWithAppend(x, l) =
    inherit Cell(x, l)
    with [
        append = fun(m)
            if m.null? then
                self
            else
                l.append(m).adjoin(x)
        ]
```

Another example involves optimization of the equality method. If the list interface included a quick *length* message, then equality could be defined to first check equality of the length of the lists.

4.4 Typing

Types are partial specifications of a program. Type systems are usually defined to be as expressive as possible while still permitting efficient checking by a compiler.

Several different levels of specificity are possible for describing interfaces, from simply a simple method list to full behavioral specification. The most that a language can be expected to check automatically is syntactic compatibility, based on interfaces with method names and argument types. This will be taken as the definition of interface from this point on.

4.4.1 Signatures of ADTs

The type theory of ADTs is derived by analogy with abstract algebra. An ADT implementation is an algebra; that is, it is a set together with operations on that set. As in algebra, the exact nature of the set is considered to be less important than the relative effects of the different operations.

The operations in an algebra is characterized by its *signature*. A signature provides a name for the carrier set, or *sort*, of the algebra, and specifies the operations on the carrier. The signature names each operation and defines the type of its arguments and return value. A signature for the integer list algebra is given below.

```
signature List
    sort t
    operations
        nil : t
        null? : t  →  boolean
        head : t  →  integer
        tail : t  →  t
        adjoin : t × integer  →  t
```

The *List* signature does not ensure that the algebra defines operations that behave correctly for lists, only that it has certain format of operations. One way a signature can be expanded into a type that specifies behavior is by adding axioms. This is discussed in Section 4.5.

A signature is a type that classifies algebras. It must not to be confused with the sort *t* of the algebra, which is the type of integer list values. Thus there are at least two types involved in the analysis of ADTs: the type of the implementation and the type of abstract values manipulated by the implementation.

A third type is present, but invisible, within an ADT. This is the type of the concrete representation of the abstract type values. This type is known only within a particular implementation, but is shared among all the operations.

Signature types are especially useful if ADT implementations are first-class values. Using signature types, it is possible to pass ADT implementations or return them as values from a procedure. In most languages, including Ada, ML (at least in its abstype construct), and Modula, abstract data types are not first-class values and each of their signature types is statically bound to a single implementation; of course, this implementation can be replaced or refined as the program is modified. The ML module system allows parameterization of ADT values, although they are still not completely first-class.

ADTs have been formalized as *existential types* [33, 13]. Using existential types, it is possible to parameterize fully over the implementations of abstract data types.

```
List = **Exists** t . [
    nil : t,
    null? : t  →  boolean,
    head : t  →  integer,
    tail : t  →  t,
    adjoin : t × integer  →  t
    equal : t × t  →  boolean ]
```

Every ADT supports an abstract type of values. In languages where the ADT implementation is statically bound to its signature, the name of the ADT often serves as the name for the abstract sort of values.

In languages where ADTs are first-class values, the ADT must be *opened* around a given scope in which the abstract type will be defined. The requirement that an ADT must be 'opened' before it can be used can lead to difficulties. One problem is that each time an ADT is opened it creates a new distinct type, so that two openings of the same ADT cannot use each others values or operations [32]. The practical effect of this is to force all ADTs to be opened outside the scope of their use, often at the outermost block level of a program. Techniques for avoiding this problem have been developed, by allowing a compiler to determine in some cases that two openings are equal [11].

4.4.2 Interfaces to Procedural Data Values

The *interface* to a procedural data value is a type that specifies the names, arguments, and return values of each procedure in the value. Since objects are passed as arguments and returned as values, the types of objects are *higher-order.* If the object returns itself as a value, or requires other parameters like itself, its interface will be *recursive.* Interfaces are not sufficient to guarantee that an object behaves properly; though behavior descriptions can be easily attached to them. The interface for the integer lists are recursive and higher-order.

```
list = [
    null? : boolean;
    head : integer;
    tail : list;
    adjoin : integer  →  list;
    equal : list  →  boolean
    ]
```

This type is very similar to the body of the ADT signature, except that the first occurrence of the existentially bound type *t* has been removed from each operation. This argument is missing because it is the object on which the interface is defined. The other uses of *t* are converted to a recursive reference to *list*.

4.4.3 Partially Abstract Interfaces

A hybrid typing model with partially abstract interfaces seems to be more appropriate for understanding object-oriented languages like Simula, C++, and Eiffel. In these languages the types have both explicit procedural interface information and also hidden components. Thus they are only *partially abstract.* A model for partially abstract types is given by *bounded existential quantification* [13]. A partially abstract list type includes message information as a bound on the abstract type.

```
List = Exists t ⊆ [
            null? : boolean,
            head : integer,
            tail : t,
            adjoin : integer  →  t
            equal : t  →  boolean
            ].
       [ nil : t ]
```

This type gives the overall structure of an ADT implementation. The abstract type *t* is constrained to be a *subtype* of the list object interface. An extended notion of bounded polymorphism is used to allow the bound variable *t* to appear in the bound [6].

The *nil* constructor is the only operation supplied directly in the ADT. The other operations are defined as messages to values in the partially abstract type.

4.5 Verification

While ADTs were developed with correctness proofs in mind, PDA has evolved without the benefit (or hindrance) of concern about formal verification. Specification of immutable procedural data abstractions, as illustrated in this paper, is equivalent to specification of higher-order functions. Correctness proofs for mutable PDA are much more difficult.

The practical consequences of the correctness problem in a procedural data abstraction is easily illustrated. Since there is no centralized control over abstractions, there is little to stop an incorrect object from finding its way into places that it is not appropriate. For example, there are many objects which resemble lists but do not behave properly. For example, this object is particularly uncooperative:

```
Bad = recursive self = record
    null? = true,
    head = 10,
    tail = Nil,
    adjoin = fun(y) self
    equal = fun(l) true
    end
```

It has a *head* value of 10 and its tail is empty, so it seems to represent the list (10). But its *adjoin* method always returns the same list (10); it is expected that *Bad.adjoin(4)* should return a list that represents (4, 10). In addition, *Bad* always claims that it is empty. It also claims that it is equal to every other list.

However, *Bad* has type *List*, because it satisfies the superficial description of methods and arguments. This is not to say that interfaces are worthless, only that while they do not guarantee correct behavior, they can prevent certain kinds of incorrect behavior.

Several proof systems for PDA have recently been proposed. The first is Pierre America's axiomatic specification system for a simple language [3]. It supports the behavioral specification of objects, verification that an implementation meets a specification, and a subtyping relation between specifications. The object language includes message passing and mutation of instance variables; however, it does not allow objects to be passed as parameters of a message – all the arguments to messages must be primitive values (non-objects). The problem of aliasing must be solved before this technique can be extended to allow objects as first-class values.

The second is an algebraic specification system [29]. This system handles class specifications and subtyping, but does not allow mutation of objects. It also allows any object to be passed as first-class values. Thus this work should be directly applicable to the immutable data abstractions considered in this paper. However, the problem of aliasing must be addressed before the system can be extended to cover state abstraction.

5 Languages Supporting PDA

5.1 Simula and Related Languages

Simula 67 was the first object-oriented language. It was defined as an extension of Algol 60 by allowing blocks to be detached from the normal nested activation scheme and have an independent lifetime. The declarations in a detached block were made accessible to other parts of the program through a reference. The definition of such blocks were called *classes*, which also acted as types or *qualifications* on references. Classes could also be defined by extension of previous classes, resulting in an inheritance hierarchy. Early versions of the language did not provide sufficient encapsulation of the attributes of classes, but later versions corrected this problem.

Simula was the inspiration for both the pure ADT languages, like CLU, and the pure PDA language Smalltalk. This is not surprising, because Simula embodies aspects of both techniques. This composite approach has been preserved in most of its statically-typed descendants, including C++, Beta, and Eiffel.

A class definition is both constructor of objects and a type. This dual nature can be seen in the two contexts in which class names are used: in a *new* expression, or in declaration of a variable. The type defined by a class is partially abstract, as outlined in Section 4.4.3, because they may have both public and hidden components. If the hidden part is empty then the class resembles an object-oriented interface. Such classes are sometimes called *abstract classes.* If a class with private components is used as a type, then it is acting more like an ADT.

Simula and C++ also support a distinction between *virtual* and *non-virtual* operations. When a virtual operations is invoked, the method to be called is determined from the object on which the operation is being performed. This is the behavior that has been assumed as normal in the general discussion of PDA. All operations are virtual in Smalltalk, Eiffel, and Trellis. The method for a non-virtual operation is determined from the class of the variable used to refer to the object, not from the object itself. Non-virtual operations model the operations in an ADT, because they are taken from the implementation of the type, not from the abstract values themselves. Classes in which all of the operations are virtual are called virtual classes.

Thus PDA is expressed in these languages by using virtual abstract classes as types. The use of non-abstract or non-virtual classes are used as types indicates the use of ADT techniques. Thus classes are able to express both ADTs and PDA in the same syntactic form, but the distinction between the two techniques still exists, and the practical trade-offs are still operative.

5.2 Smalltalk

Smalltalk was developed from ideas in Simula 67 and Lisp. It has gone farthest in developing a pure object-oriented language. The language has no static typing, but it is still a strongly typed language, because erroneous message sends and variable accesses are caught at runtime.

Since it lacks a static type system, it is impossible for it to have user-defined type abstraction. Instead, Smalltalk relies consistently on procedural abstraction, even at the level of basic control structures like conditionals and iterations.

The Smalltalk library contains a mixture of mutable and immutable object classes. The immutable ones include the booleans (True and False), the numbers (SmallInteger, LargeNegativeInteger, LargePositiveInteger, Integer, Fraction, and Float), the ParseNodes (one for each kind of expression in the language). The mutable objects include the collections (Set, Dictionary, Array, LinkedList), except Interval, which is immutable. These classes are all implemented as PDAs.

However, at the lowest levels of the system, in the handling of primitive numbers and arrays, Smalltalk relies upon built-in ADTs. One type represents small integers in the range 0-256. Another is used for basic floating-point numbers. In addition, a primitive array construct is built into the basic class mechanism. The PDA versions of integers, factions, complex numbers, and collections are built upon these primitive ADTs.

The way in which this is done is very interesting. For integers, the system is implemented so that arbitrary new integer representations may be defined and intermixed with other implementations. Since binary operations cannot know the exact representation of their arguments, a unary protocol is used to extract digits one at a time from the argument object. The digit protocol includes a message *digitLength* to determine the number of digits in an integer, and *digitAt:* n to access n^{th} digit. The number n must be a primitive small integer, which restricts the number of digits, however, the digits are represented in base 256, and the number of digits is typically on the order of 65,000, so extremely large integers can be represented. The interesting thing about this arrangement is that special representations may be invented to represent large numbers. For example, $2^m - 1$ can be represented by a special object that returns digits with the appropriate number of 1 bits set depending on the argument n of *digitAt.* In order to fully integrate a new integer representation, it may be necessary to change other methods so that they create the new integer when appropriate.

For other kinds of numbers, including fractions, floats, and complex numbers, there is a *generality* hierarchy. This is a total ordering of the various Number classes. A general number class must define how to convert less general numbers into the extended representation without losing information, and also how to truncate the extended representation so that it can be used where a less general number is expected.

5.3 The Lambda-Calculus

According to the definitions given here, the lambda-calculus is the oldest and most pure language supporting PDA. Since its only construct is functional abstraction, all data must be represented as functions. Its purity comes from the fact that it lacks not only user-defined ADTs but also the familiar builtin data types like integer and boolean found in Pascal and Fortran. On the other hand, the lambda-calculus also lacks imperative constructs for assignment of variables, which are often considered essential for PDA in practice.

Programming languages based on the lambda-calculus, like Scheme, use these same techniques to implement PDA [2]. It is possible to define PDAs in statically-typed derivatives of the lambda-calculus, like ML, but much of the flexibility of the technique is lost due to the lack of subtyping. Adding types and parametric polymorphism to the lambda-calculus enables it to accurately model ADTs [33].

5.4 CLOS

The Common Lisp Object System (CLOS [27]) seems to defy characterization using the distinction between PDA and ADTs.

The model was originally motivated as a generalization of the classical object model in Smalltalk. In this model a message is sent to a distinguished object, and the method to invoke is determined by examining the type, or class, of that object. The arguments of the message do not play a role in determining the method to invoke. In the *generalized object model*, the message is viewed as an operation on a collection of arguments, and any or all of the arguments may play a part in determining which method to run. Whereas the selection process is localized to a single object and its class in the classical model, in the generalized model the system must select a method based on interactions among classes.

It may be possible to view CLOS as being organized around the original multi-dimensional specification of a data abstraction. This matrix is not decomposed as it is in the ADT and PDA approaches, but is managed as a whole by the system. In this scheme, representational structures and operations are relatively independent entities. The resulting system is less modular than the ADT or object organizations, but has the advantage of symmetry and flexibility. The degree of encapsulation provided by the system is also not clear. An appropriate type system has also not been identified.

6 Conclusion

The essence of object-oriented programming is *procedural data abstraction*, in which procedures are used to represent data and procedural interfaces provide information hiding and abstraction. This technique is complementary to ADTs, in which concrete algebras are used to represent data, and type abstraction provides information hiding.

The two paradigms can be derived from a fundamental dichotomy in decomposing a matrix of observers and constructors that specify abstract data. PDA decomposes this matrix into constructors: a class is associated with each constructor and the observations become attributes, or methods, of the class's instances. In effect, the values in the abstraction are nothing more or less than the collection of legal observations on them. ADTs, on the other hand, decompose the matrix into observations: each observation is an operation defined on an abstract type that includes the constructors as representational variants.

As would be expected, given the organization biases of the two paradigms, they are complementary in the sense that each has advantages and disadvantages. Using PDA it is easy to add new constructors, and the absence of a shared abstracted type reduces code interdependence. With inheritance and subtyping it is also possible to add new observations (methods). However, the use of strong functional abstraction prevents optimizations that might be performed if more than one representation could be inspected at a time. Binary observations in particular cause difficulties for PDA.

Using ADTs, it is difficult to add fundamentally new constructors: the shared representation type and all of the operations must be changed. It is easier to add a new observation, although support for this is not provided in most languages. Subtyping allows different implementations to be used, but implementations cannot be intermixed. On the other hand, the unifying properties of type abstraction allow optimizations which require access to more than one value's representation at a time.

Simula was the inspiration for the development of both abstract data types (exemplified by CLU) and PDA (exemplified by Smalltalk). This is not surprising, because Simula embodies a combination of both techniques, a characteristic preserved by its descendants C++, Eiffel, and Beta. The combination is more of an interweaving than a unification, because the trade-offs outlined above are still operative.

In general, the advantages of one paradigm are the disadvantages of the other. A detailed analysis of the trade-offs involved must be made in order to choose whether an ADT or PDA is better suited for a problem. Some relevant questions in this choice are: How likely is it that the system will be extended? Must the security of the system be preserved, even at the expense of extensibility? Is it possible the unforeseen interactions may be desired? Is the environment dynamically extensible, and must the additions interact in complex ways with existing code? How much efficiency is required? Is it likely that there will be a large number of binary operations with complex behavior? Unfortunately, the future changes in requirements may invalidate this choice. In this case it is possible to mechanically translate from one paradigm to the other. In this context, questions like "Which paradigm is better for modeling the real world" are virtually meaningless. The choice depends upon how the world is conceptualized, and what additional properties are expected of the model.

Acknowledgment

I would like to thank the sponsors and participants of the school/workshop on the foundations of object-oriented programming for providing a forum for the initial presentation of these ideas, and the other speakers at the school for motivating the effort. The point of view presented here was developed in collaboration with Peter Canning, Walt Hill, and Walter Olthoff. Valuable contributions to the presentation were made by Warren Harris, Dinesh Katiyar, and Alan Snyder.

References

[1] Abelson and Sussman. *The Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] N. Adams and J. Rees. Object-oriented programming in Scheme. In *Proc. of the ACM Conf. on Lisp and Functional Programming*, pages 277–288, 1988.

[3] P. America. A behavioral approach to subtyping object-oriented programming languages. In *Proc. of the REX School/Workshop on the Foundations of Object-Oriented Languages*, 1990.

[4] M. P. Atkinson and R. Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, 7(4):539–559, 1985.

[5] M. R. Brown and G. Nelson. IO streams: Abstract types, real programs. Technical Report 53, Digital Equipment Corporation Systems Research Center, 1989.

[6] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proc. of Conf. on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[7] P. Canning, W. Cook, W. Hill, and W. Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 457–467, 1989.

[8] P. Canning, W. Hill, and W. Olthoff. A kernel language for object-oriented programming. Technical Report STL-88-21, Hewlett-Packard Labs, 1988.

[9] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–68. Springer-Verlag, 1984.

[10] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report (revised). Technical Report 52, Digital Equipment Corporation Systems Research Center, Dec. 1989.

[11] L. Cardelli and X. Leroy. Abstract types and the dot notation. Digital Equipment Corporation Systems Research Center, 1990.

[12] L. Cardelli and J. C. Mitchell. Operations on records. Technical Report 48, Digital Equipment Corporation Systems Research Center, 1989.

[13] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1986.

[14] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 49–70, 1989.

[15] W. Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[16] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 433–444, 1989.

[17] S. Danforth and C. Tomlinson. Inheritance and type theory. Technical report, MCC, 1987.

[18] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* Springer-Verlag, 1985.

[19] J. Goguen and J. Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In *Functional and Logic Programming*, pages 295–363. Prentice-Hall, 1986. An earlier version appears in Journal of Logic Programming, Volume 1, Number 2, pages 179-210, September 84.

[20] J. Goguen, J. Thatcher, E. Wagner, and J. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24(1):68–95, January 1977.

[21] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation.* Addison-Wesley, 1983.

[22] J. Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages.* PhD thesis, University of Illinois, 1989.

[23] J. Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):297–404, June 1977.

[24] J. V. Guttag and J. J. Horning. An introduction to the larch shared language. In *IFIP*, 1983.

[25] R. E. Johnson. Type-checking smalltalk. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 315–321, 1986.

[26] K. Kahn, E. Tribble, M. Miller, and D. Bobrow. Vulcan: Logical concurrent objects. In *Research Directions in Object-Oriented Programming*, pages 75–112. MIT Press, 1987.

[27] S. E. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Wesley, 1989.

[28] W. R. LaLonde, D. A. Thomas, and J. R. Pugh. An exemplar based smalltalk. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 322–330, 1986.

[29] G. T. Leavens and W. E. Weihl. Reasoning about object-oriented programs that use subtypes. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, 1990. to appear.

[30] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20:564–576, 1977.

[31] B. Liskov and S. Zilles. Programming with abstract data types. *SIGPlan Notices*, 9(4):50–59, 1974.

[32] D. B. MacQueen. Using dependent types to express modular structure. In *Proc. of the ACM Symp. on Principles of Programming Languages*, 1986.

[33] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. In *Proc. of the ACM Symp. on Principles of Programming Languages*, 1985.

[34] U. S. D. of Defense. Reference manual for the Ada programming language. ANSI/MIL-STD-1815 A, 1983.

[35] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[36] D. Parnas. A technique for software module specification. *Communications of the ACM*, 15:330–336, 1972.

[37] U. S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. of the ACM Conf. on Lisp and Functional Programming*, pages 289–297, 1988.

[38] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 77–88, 1989.

[39] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New Advances in Algorithmic Languages*, pages 157–168. Inst. de Recherche d'Informatique et d'Automatique, 1975.

[40] R. Stallman, D. Weinreb, and D. Moon. *Lisp Machine Manual.* MIT AI Lab, 1983.

[41] J. Stein, editor. *Random House Dictionary of the English Language.* Random House, 1967.

[42] M. Wand. Complete type inference for simple objects. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987.

[43] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 92–97, 1989.

[44] N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1983.

[45] S. N. Zilles. Procedural encapsulation: A linguistic protection mechanism. *SIGPlan Notices*, 8(9):142–146, 1973.