DATA ABSTRACTION FROM A PROGRAMMING LANGUAGE VIEWPOINT†

Lawrence A. Rowe
Computer Science Division-EECS
University of California
Berkeley, CA 94720

ABSTRACT

This paper traces the development of data abstraction concepts in programming languages. A data abstraction, or abstract data type, describes a collection of abstract entities and operations on the entities. A program which uses a data abstraction can access or modify the entities only through the abstract operations.

Specific research topics discussed in the paper include: the role of type in a programming language, the formal specification of the semantics of a data abstraction, data abstraction language construct design issues, type hierarchies, and type-checking.

1. INTRODUCTION

This paper presents a brief survey of research on data abstraction from a programming language viewpoint. The goal of this research is to discover language notations, software development methodologies, and software management tools which will make it easier to design and implement the data structures used in a program. Moreover, the resulting programs should be easier to understand so that bugs can be fixed faster and so that it is easier to add features to an existing program. While the emphasis here is on language constructs and how they are used, the related methodologies and tools are essential to achieving the expected benefits.

Section 2 discusses the evolution of programming languages from low-level to high-level notations. Section 3 discusses extensible languages which introduced user-defined data types. Section 4 describes two viewpoints on the role of type in a programming language and describes the problems with user-defined data types as a data abstraction mechanism. Section 5 gives an example of the formal specification of the semantics of a data abstraction. Section 6 discusses some of the issues in the design of a data abstraction language construct. Section 7 briefly describes how type systems in programming languages define type hierarchies and discusses the role of generic procedures. Finally, type systems and type-checking and their impact on type protection are discussed.

2. EVOLUTION OF PROGRAMMING LANGUAGES

A programming language is the notation used to specify an implementation of an algorithm which can be executed by a machine. All languages provide an abstraction of a machine. Control abstractions are provided to specify the sequencing between statements in a program (e.g., conditional branches) and data abstractions are provided to specify the entities and their operations used in the algorithm (e.g., arrays with selection and assignment of individual elements).

Low-level languages, such as assembly languages, provide abstractions which are essentially equivalent to the instruction set of a machine. High-level languages, such as PASCAL [Jensen 75], provide abstractions which highlight the algorithm, or computation, and suppress the implementation detail. Figure 1 shows an array assignment in a high- and low-level language. The low-level implementation might be very different depending on the lower bound of the array, the size of the array elements, whether the machine had word or byte addressing, and the availability of registers at this point in the program. The high-level description suppresses this implementation detail and highlights instead the array. Low-level languages give a programmer more control over the run-time efficiency of a program. High-level languages make program development and maintenance easier at a small expense in run-time efficiency.

Very high-level languages (VHL) provide even more abstract notations. For example, an array in PASCAL implies a particular implementation (contiguous storage with indexed addressing). A VHL might emphasize the functional mapping an array represents (from the array's index domain to the element domain) without implying a particular implementation.

```
A[i] := exp
```

(a) High-Level Language

```
load  r0,exp
load  r1,i
store r0,A(r1)
```

(b) Low-Level Language

Figure 1: Array Assignment

† This work was supported by the National Science Foundation under grant MCS-77-28301.

Programming languages have evolved to higher level notations because software development and maintenance costs dominate hardware costs for most applications. Languages evolve when it is discovered that identical patterns of statements are being used many times in different programs. When this happens, a new language construct will be invented to describe the computation. The new construct will suppress the detail which is the same for the different uses and highlight the differences. For example, early scientific programs written in assembly language frequently used the different contiguous storage implementations of an array and described computations by iterating through the array. High-level languages introduced **array**-declarations and **for**-statements. These high-level language constructs specify "what is to be done" while the assembly language code specifies "how to implement it."

Another reason that new programming languages are continually being designed is to solve problems in new application domains. Existing languages often do not have the appropriate built-in data abstractions to write programs in these new domains. Consequently, much of the code in these programs implements these data abstractions.

A good example of this problem was described by Slate and Mittman in a paper describing the development of chess programs:

> "...the largest single obstacle to progress in computer chess is not the lack of effective chess ideas, but rather, the lack of efficient easy-to-use program structures in which to represent these ideas." [Slate 78]

Chess programs have many different data abstractions which are time-consuming and difficult to code in existing languages. For example, structures which represent patterns between pieces rather than the location of pieces on an 8 by 8 board and structures which describe sequences of moves which have been or could be made. While it has not happened yet for chess, special-purpose languages have been developed for other application domains with the relevant data abstractions as built-in data types (e.g., the symbolic manipulation language MACSYMA has rational functions which are stored as ratios of polynomials [Mathlab 77]). Programs in these domains are easier to write and understand because the notation focuses on the computation to be done rather than how to implement it.

The problem with this approach is that designing and implementing a good programming language is very hard and time-consuming. If your interest is chess programs, you probably do not want to spend two years developing tools to write chess programs. Moreover, this approach assumes that a fixed set of data types can be devised to describe all the data abstractions which might be needed. Unfortunately, this approach fails either because some reasonable data abstractions must be left out or because the set gets too large. Another approach is to define a small set of primitive types (e.g., *integer*, *real*, *boolean*, and *char*) and type constructors (e.g., **array**, **record**, and **pointer**) which can be used to define other abstractions. This approach is discussed in the next section.

3. EXTENSIBLE LANGUAGES

Extensible programming languages attempted to solve the problem of proliferation of special-purpose languages by providing constructs which allow a programmer to extend a base language to a new language for a specific problem domain. ALGOL-68 [van Wijngaarden 75] and ECL [Wegbreit 71] are examples of extensible languages.[1]

An extensible language supports syntax, control-structure, data type, and operator extensions. A syntax extension allows a programmer to modify the syntactic structure of the language. A control-structure extension changes the language control structures (typically by defining procedures). A data type extension introduces a new data type into the language. New data types are specified in terms of a small, fixed set of type constructors as mentioned above. In some languages the semantics for built-in primitives (e.g., *print*) can be extended to work on the new data types [Wegbreit 71]. An operator extension defines the semantics of an operator designation when applied to user-defined data types. For example, the plus operator ('+') could be defined to be concatenation when applied to two string values.

Data type and operator extensions can be used to implement the abstract entities and operations used in an algorithm.[2] However, extensible languages do not provide adequate protection to insure that only meaningful abstract operations are performed on a value which represents an abstract entity. Any operation which is legal on the data types used to implement the abstract entity can be performed on it. For example, if an array is used to implement a stack, any element of the stack can be changed because any element of an array can be reassigned. In other words, operations on the representation data type (array) can be performed on the abstract data type (stack) even though those operations are not semantically valid on the abstract type. Extensible languages do not provide a mechanism to restrict the operations on the abstract type. This situation raises the question "what is the role of type in a programming language?"

4. THE ROLE OF TYPE IN A PROGRAMMING LANGUAGE

Two views of the role of type in a programming language have been discussed in the literature. The first view is that a type is a set of values. A data type is defined by specifying the representation for the values. For example, a queue can be described as an array and two indices, *FRONT* and *BACK*, which index the element at the front, respectively back, of the queue. Operations on values of a particular abstract data type are coded as procedures but they are independent of the data type. This view is embedded in most programming languages (e.g., COBOL, FORTRAN, PASCAL, AND PL/1).

The second view, referred to as the "types are not sets" view, is that a type is a language mechanism to enforce authentication and security [Morris 73]. Authentication ensures that any value supplied to an operation is consistent with the type expected by that operation. Security ensures that any operation applied to a value is meaningful for that type. Extensible languages are based on the first view of the meaning of type and, consequently, could not guarantee protection which is intrinsic to the second view.

The relationship between these two views of type can be made clearer by looking at the relationship between the problem a particular program is solving and the implementation of that program. Figure 2 shows the abstract and representation levels inherent in any problem which is solved by a computer. The abstract entities and operations are those a programmer conceptualizes when solving a problem. For example, graphics applications are based on objects such as *points*, *lines*, and *polygons* and operations such as *draw_line* and *shade_polygon*. An implementation of a graphics program may use a pair of real numbers to represent a *point*, three real numbers to represent a *line* (a *point* and a *slope*), and a linked list of pairs of real numbers to represent a *polygon* (a set of *line*'s). The operations

---

[1] LISP was probably the first extensible language since user-defined functions are invoked with the same syntax as built-in functions [McCarthy 62]. This feature, as well as the fact that LISP programs and data use the same representation and that LISP systems are almost always interactive, has lead to an entire subculture of application-specific LISP extensions [Bobrow 74].

[2] The abstract entities and their operations are called an *abstract data type*.

Figure 2: Abstract versus Representation Levels

draw_line and *shade_polygon* can be represented as procedures.

The "set of values" view of type focuses on the representation level. That is, a type which represents a *point* is really just a pair of real numbers. The "types are not sets" view focuses on the abstract level. That is, a type denotes the abstract data type. Programs coded using the abstract data types of the application domain are easier to code and understand because implementation detail is suppressed and type-checking can provide more protection against errors.

From a language viewpoint, an abstract data type is a specification of the implementation of the abstract entities and operations which are encapsulated so that they only way to access or modify the entities is by the abstract operations. Recent languages have included a construct that allows a programmer to define abstract data types. Before discussing the issues involved in the design of these language constructs, research on notations to specify the semantics of an abstract data type is surveyed.

5. SEMANTIC SPECIFICATION OF AN ABSTRACT DATA TYPE

A programmer should think only in terms of the abstract data types, not their implementation. Consequently, he or she only needs to know the semantics of the data abstraction. The problem is how to specify the semantics without giving an implementation.

This problem is the same one faced by programming language designers when trying to describe the semantics of a programming language and, not surprisingly, the same techniques have been tried. There are two kinds of semantic descriptions: operational and definitional. Operational specifications define the semantics of a program or abstract data type by specifying the meaning of each operation in terms of another model, called the *reference model*, in other words, giving an implementation in terms of a simple abstract model of computation (e.g., Vienna Definitional Language computation tree [Wegner 72]). The advantage of this approach is that semantic descriptions are easy to write and understand because it is similar to programming. The disadvantages are:

1. the descriptions can be quite long and complex when the operations are complex or when there are many different operations,[3]
2. it is difficult to show a relationship between two abstract operations (e.g., that they are inverses), and
3. it is difficult to show the equivalence of two operational specifications.

[3] Long descriptions can be avoided by making the reference model more complex. But, this makes the descriptions harder to understand.

Moreover, how is the meaning of the reference model established?

A definitional specification defines the semantics of an abstract data type by specifying properties of the values and operations of the type. Alphard [Wulf 76] includes a definitional semantic description with the declaration of each abstract data type in a program. The description is based on Hoare's axiomatic approach to specifying program semantics [Hoare 69]. Guttag [Guttag 80], Goguen et.al. [Goguen 75], and Liskov and Zilles [Liskov 74] have used a definitional approach based on algebraic relations.

Figure 3 gives an algebraic specification of a queue. The notation used in this example was developed by Guttag [Guttag 80]. The abstract type being specified is *QUEUE* and has a single parameter, named *T*, which specifies the type of the queue elements.[4] The description consists of a syntactic, semantic, and restriction specification. The syntactic specification describes the names, domains, and ranges of the abstract operations. For example, the *create_q* operation takes no arguments and returns a *QUEUE*.

The semantic specification describes the meaning of the abstract operations by a set of equations which relate the operations. For example, semantic equation 3 relates *front_q* to *add_q* by noting that after adding an element to an empty queue, the element is at the front. If the queue was not empty, the element at the front was the one at the front before the new element was added.

The restrictions specification describes limitations on the use of the operations. For example, if a queue is empty, *remove_q* is undefined.

A set of equations should be consistent (i.e., it should be impossible to derive a false statement from the equations when taken together) and complete (e.g., that any well-formed formula or its negation can be proved true). Consistency has been easier to show than completeness. Consequently, research has focused on deriving properties which are analogous to completeness and satisfy practical constraints (e.g., sufficiently complete [Guttag 80]).

The problem with the algebraic approach, as well as the other approaches to semantic specifications, is that the effort required to develop the specification does not

[4] The specification actually defines a family of types, queues with different element types.

type QUEUE [T:TYPE]

syntax

create_q:		--> QUEUE
add_q:	QUEUE × T	--> QUEUE
remove_q:	QUEUE	--> QUEUE
front_q:	QUEUE	--> T
isEmpty_q:	QUEUE	--> BOOLEAN

semantics

declare q:QUEUE, x:T

1. isEmpty_q(create_q()) = **true**
2. isEmpty_q(add_q(x,q)) = **false**
3. front_q(add_q(q,x)) = **if** isEmpty_q(q)
   then x
   else front_q(q)
4. remove_q(add_q(q,x)) = **if** isEmpty_q(q)
   then create_q()
   else add_q(remove_q(q),x)

restrictions

isEmpty_q(q) ==> **failure**(front_q,q)
isEmpty_q(q) ==> **failure**(remove_q,q)

Figure 3: Algebraic Relation Specification

pay-off. Well-written comments are almost always easier to understand.

Some work has been done on automatic selection of implementations for abstract data types [Dewar 79, Gotlieb 74, Low 78, Rowe 78]. However, most of this work used an operational specification of the data type semantics because it is hard to deduce enough information from a definitional specification to produce a practical implementation.

6. LANGUAGE CONSTRUCTS

The SIMULA *class* was the first language construct to associate the abstract operations with the entities on which they were defined [Birtwistle 73]. A class combines the procedure definitions which implement the abstract operations with the type declarations which implement the abstract entities. However, entities declared in a class can be accessed directly without using the class operations. Classes do not provide protection as discussed in Section 4.[5]

CLU [Liskov 77] and Alphard [Wulf 76] were the first languages to provide a data abstraction construct similar to classes which also enforced protection. An example of the declaration of the *QUEUE* abstraction specified in the previous section is given in Figure 4. This implementation is a bounded queue while the semantic specification was for an unbounded queue. The **export** statement lists the identifiers which will be visible outside the abstraction. **Rep** declares the representation for *queue* values. Notice that arguments to the pro-

[5] Interestingly, the designers of SIMULA never thought about limiting access to the class entities [Dahl 79].

```
queue: module[T:type,N:natural]
    export create,isEmpty,add_q,remove_q,front_q

   /*
    * Implement queue by circular buffer with two indexes: front and back. Front indexes
    * the first element on the queue and back indexes the next free location. The queue
    * is empty when front=back.
    */

    rep = record
              q: array 0..N-1 of T
              front,back: 0..N-1
    end

    create: procedure(inout x:rep) =
    begin/* initialize rep when queue created */
        x.front, x.back := 0
    end

    isEmpty: procedure(x:rep):boolean =
    begin
        return(x.front=x.back)
    end

    add_q: procedure(x:rep;y:T) =
    begin
        if (x.back+1) mod N not= x.front then
            x.q[x.back] := y
            x.back := (x.back+1) mod N
        else
            raise queue_overflow
        end if
    end

    remove_q: procedure(x:rep):T =
    begin
        if x.back not= x.front then
            x.front := (x.front+1) mod N
        else
            raise queue_underflow
        end if
    end

    front_q: procedure(x:rep):T =
    begin
        if x.back not= x.front then
            return(x.front)
        else
            raise queue_underflow
        end if
    end

end /* queue */
```

Figure 4: Declaration of a Bounded Queue

cedures in the abstraction are declared to be of type **rep**. When a *queue* value is passed as an argument of type **rep** to a procedure in the abstraction, it is unsealed so that the internal structure of the value can be accessed and modified inside the abstraction. Outside the abstraction the representation of *queue* values cannot be accessed.

Two queues are declared and implicitly initialized (by calls on *create*) as follows:

```
declare q1:queue(integer,25)
        q2:queue(process_type,15)
```

An element is added to the queue of integers by:

```
q1.add_q(i)
```

Q1, the queue on which the operation is performed, is implicitly bound to the first argument to the procedure *add_q*. Notice how the restrictions specified in the semantic specification are implemented in *front_q* and *remove_q*.

Most programming languages designed in the last few years have included a data abstraction construct (e.g., ADA [ADA 79], MESA [Geschke 77], MODULA [Wirth 77], and RIGEL [Rowe 80]). Different languages use different names for the construct (e.g., **capsule**, **cluster**, **envelope**, **form**, or **module**),[6] but each construct serves the same purpose. The remainder of this section discusses some of the language design issues related to these constructs.

The first design issue is whether a data abstraction can declare more than one data type. Up to this point each abstract data type defined only one data type in the program. For example, when the *queue* abstraction is instantiated with a specific type parameter, say *integer*, and number of elements, say 25, a data type *queue*[*integer,25*] is defined. Some data abstractions require the declaration of more than one data type. Moreover, it may be impossible to separate the implementations of these types. An example is a virtual terminal abstraction which allows multiple windows of data to be visible simultaneously on the screen. The abstraction must define a *virtual_terminal* type and a *window* type. The implementations of these abstract data types cannot be separated because operations on *window*'s must have access to the *virtual_terminal* implementation.

A second design issue is whether the interface to a data abstraction (i.e., the types and procedure headers) is separated from the implementation of the abstraction. In the *queue* example, the implementation was not separated from the interface specification. Consequently, notice how you must read through the implementation code to discover the number and type of arguments to an abstract operation, for example *front_q*. Separation of the interface is a good idea because programmers only see the information needed to use the abstraction.

A third design issue is whether data abstractions can be parameterized and, if so, what kinds of parameters are allowed. For example, the *queue* abstraction has two parameters that specify the type of the *queue* elements and the maximum number of elements in the *queue* (must it be a compile-time constant?). The trade-off on whether or not to have parameterized abstractions is between programming ease and implementation efficiency. A parameterized abstraction allows one program fragment to define many distinct abstractions. Consequently, writing and maintaining programs should be easier because there is only one copy of the code. However, to insure run-time efficient implementation, a compiler will be considerably more complex. And, in the case of type parameters, may result in run-time inefficiencies (see the example of a recursive parameterized abstraction in [Atkinson 78]). A related issue is whether parameterized abstractions can be separately compiled.

The last design issue relating to data abstraction constructs is whether constraints can be associated with type parameters passed to data abstractions and with values passed to procedures [Jones 78]. For example, suppose there is an abstraction with a type parameter for the elements of the abstract structure and an abstract operation to sort the structure. Any type value passed to this abstraction must have a *less_than* operation which can be used to define the *sort* operation. If a *less_than* operation is not supplied, *sort* would be meaningless. A constraint is needed on the formal type parameter which can be checked either at compile- or run-time to determine that the *less_than* operation exists.

Another example of a constraint is to limit the operations which can be performed on an entity passed as an argument to a procedure. Specification of these constraints can make programs more reliable since assumptions which must be satisfied for the program to work can be checked by a compiler. Under some circumstances, the constraints may lead to more efficient implementations.

7. TYPE HIERARCHIES AND GENERIC PROCEDURES

Parameterized data abstractions, discussed in the previous section, define a hierarchy between types. The parameterized abstraction is a generalization of an instance of the abstraction which has a particular value bound to the parameter. For example, *queue*[*T:type,N:natural*] is a generalization of *queue*[*integer,25*]. Programming language type systems define other type hierarchies: subtypes and union types.

PASCAL was the first language to introduce subtypes. Subranges of integers and enumerated types could be defined. For example, *1..10* is a subrange of *integer* (called the *base type*). Subranges are used to limit the possible values a variable can have and to define the domain of an array index. Operations on the base type are also defined on the subrange type but may be infeasible (i.e., the result may not be in the subrange).

Union types introduce another kind of type hierarchy.[7] A union type lists a number of alternative types that a particular variable may hold. For example, the declaration

```
declare x:union(integer,real,char)
```

defines a variable which may hold an *integer*, *real* or *char* value. At any point in the computation, *x* will hold a value of one of these types. Later, the type of the variable may be changed to one of the other alternatives.

Union types are useful when a computation is the same except for the types of the values. For example, to sum the elements of an array the program looks very similar when the array elements are integers or reals:

```
SUM: procedure(A: array 1..10 of T):T =
declare x:T
begin
    x := 0
    for i in 1..10 do
        x := x + A[i]
    end for
end /* SUM */
```

The differences in this code when *T* is an *integer* and a *real* are the constant used to initialize *x* and the code produced for the '+' operator. Compilers for languages with union types can solve these problems by generating code for all types in the union and selecting the

[6] **Monitor** constructs in system programming languages [Hoare 74] are similar to data abstraction constructs. They have a notion of process and synchronization as well as encapsulation and protection.

[7] Variant records are another language construct to realize union types. The major difference is that variant records include an explicit declaration of the tag which specifies the current type of the value.

appropriate code at run-time (e.g., ALGOL 68 [van Wijngaarden 75]). Another approach is to force the programmer to perform the test explicitly:

```
case type(x) of
     integer: x := x + A[i]
     real:    x := x + A[i]
end case
```

Notice that the '+' operation on the first (second) line is *integer* (*real*) addition. This approach makes the programmer more aware of the code being generated and allows him/her to leave out cases which are guaranteed not to occur.

In the *SUM* procedure '+' is an example of a *generic* operation because it is defined for many different types. While '+' is a built-in operation, some languages allow user-defined generic operations and procedures (e.g., ADA [ADA 79]). For example, a program can define many procedures named *f* with different numbers and types of parameters. For each call on *f* code for the correct procedure will be generated by examining the number and types of actual parameters. Note that type parameterized data abstractions usually have some generic procedures.

8. TYPE-CHECKING

The goal of type-checking is to insure that the operation performed on a value is meaningful in the sense of Section 4. The usefulness of data abstraction constructs depends on reasonable type-checking rules. If the rules are too complex or restrictive, programmers will not use the data abstraction constructs or they will find a way to circumvent the type system. This section describes the two basic approaches to type-checking, name equivalence and structural equivalence, and some of the problems with them.

Name equivalence defines two values to be type compatible if the "names" of the types are the same. For example, given

```
declare x,y: integer
```

x and *y* are type compatible because the identifier which defines their type, in this case *integer*, is the same. Even if *x* and *y* were declared on separate lines, e.g.,

```
declare x: integer
        y: integer
```

they are still type compatible because the identifier which defines the type is the same.

Suppose, however, that the type was specified by a type-constructor (e.g., **array**, **record**, **file**, or **relation**) rather than by a type identifier. In this case, *x* and *y* do not have the same type identifier so they are not type compatible. For example, given

```
declare x,y: array 1..10 of integer
        z: array 1..10 of integer
```

x and *y* are type compatible, but neither *x* and *z* nor *y* and *z* are type compatible. To make them type compatible a type name must be declared as follows

```
declare T: type = array 1..10 of integer
        x,y:T
        z:T
```

Now, all three variables are type compatible because they have the same type identifier.

Structural equivalence defines two values to be type compatible if the data representation is the same, regardless of how it is declared. For example, given

```
declare T: type = array 1..10 of integer
        x: T
        y: array 1..10 of integer
        z: array 1..10 of integer
```

all three variables are type compatible.

The issue really is whether the declaration of a type identifier (*T* in the examples above) defines a new data type (name equivalence) or whether it is just a shorter notation, or macro, for the type specification (structural equivalence). Because the theme of this paper has been that protection can make programs more reliable, the reader might assume that name equivalence is best. This assumption is false because name equivalence does introduce some problems.

Suppose one wanted to define a library function, named *SUM*, to sum the elements of an array and that type-checking was based on name equivalence. *SUM* should work for any one-dimensional array regardless of how its type is specified.[8] Assuming that the formal argument type is specified by an **array**-constructor, e.g.,

```
SUM: procedure(A: array 1..10 of integer): integer
```

all array variables declared in the program, whether specified by a type identifier or an **array**-constructor, will be type incompatible with *A*.

The same problem arises with user-defined procedures. Whenever a structured variable is to be passed as an argument to or returned as the result of a procedure, a type must be defined. Because the same procedures are needed on different abstract types, either several copies of each procedure can be defined (one for each distinct type) or the distinct abstract types can be defined as one named type. Most times a programmer will do the latter because it saves space and avoids the problem of having to remember to update multiple copies of a procedure when a bug is fixed. Consequently, pure name equivalence may not help type security.

Structural equivalence solves this problem but it has other problems. The major one is the complexity and efficiency of type-checking at compile-time. For example, think about the algorithm required to determine that

```
declare T1: type = record f1: T; f2: ^T2 end
        T2: type = record f1: T; f2: ^T1 end
```

are type compatible. Moreover, depending on how structural equivalence is defined, some bad program bugs may not be detected [Welsh 77]. For example, if

```
var x: record re,im: real end
    y: record im,re: real end
```

are type compatible (both are records with two fields each with the same type name), the assignment of *x* to *y*, or vice versa, will produce very strange results if the assignment is performed by a block copy operation. One solution to this problem is to require that both records have the same number of fields with identical selector names in the same order each with compatible types. Notice how complicated type-checking is becoming and note that the programmer must remember these rules too.

Many variants of these two basic approaches to type-checking are possible. The idea is to define a type-checking rule which will be "right" most of the time, which will be reasonably easy to implement, and which can be used to describe the type protection the programmer desires. Examples of some variants are:

1. structural equivalence can be applied to only some type constructors (e.g., **record** and **array** types but not enumerated types) [Mitchell 79], and

2. type constructors can be type-checked using structural equivalence and named types can be type-checked using name equivalence [Rowe 80].

While these variants, and others, come close to an ideal type-checking rule, no rule is perfect because type-checking is a syntactic implementation of a semantic function.

[8] In this example we ignore the problems of dynamic or variable-sized arrays and of variable element types.

9. SUMMARY

Research in data abstraction is searching for notations, methodologies, and tools to make programs more reliable, easier to read and understand, and therefore, easier to maintain and enhance. Specific directions which have been investigated are:

1. data type extensions to programming languages,
2. formal specification of data abstraction semantics,
3. language mechanisms to encapsulate data abstractions, and
4. type systems which are flexible, yet secure, and can be efficiently type-checked.

Many areas remain to be explored including tools for the development of data abstractions, libraries of data abstractions, specifications of performance information [Bentley 80], and user-controlled compilation and optimization systems.

ACKNOWLEDGEMENT

I want to thank Mike Brodie, Rick Cattell, Joe Cortopassi, Kurt Shoens, and Steve Zilles who read an earlier draft of this paper and offered many suggestions which improved the presentation.

REFERENCES

[ADA 79] Preliminary ADA Reference Manual. *SIGPLAN Notices*, vol. 14, no. 6 (June 1979).

[Atkinson 78] R.R. Atkinson, B.H. Liskov, and R.W. Scheifler. Aspects of Implementing CLU. *Proc. 1978 ACM Nat. Conf.* (1978).

[Bentley 80] J.L. Bentley and M. Shaw. Abstraction and Efficiency: The Interaction of Languages and Analysis. Comp. Sci. Dept., Carnegie-Mellon Univ. (1980).

[Birtwistle 73] G.M. Birtwistle et.al. *SIMULA BEGIN*, New York: Petrocelli, 1973.

[Bobrow 74] D. Bobrow and D. Raphael. New Programming Languages for AI Research. *Computing Surveys*, vol. 6, no. 3 (Sept. 1974).

[Dahl 79] O.-J. Dahl. Personal Communication (Dec. 1979).

[Dewar 79] R.B.K. Dewar, et.al. Programming by Refinement, as Exemplified by the SETL Representation Sublanguage. *ACM Trans. on Prog. Lang. and Sys.*, vol. 1, no. 1 (July 1979).

[Geschke 77] C.M. Geschke, J.H. Morris, Jr., and E.H. Satterwaite. Early Experience with MESA. *Comm. of the ACM*, vol. 20, no. 8 (Aug. 1977).

[Goguen 75] J.A. Goguen et.al. Abstract Data-Types as Initial Algebras and Correctness of Data Representations. *Proc. Conf. on Comp. Graphics, Pattern Rec. and Data Structure* (May 1975).

[Gotlieb 74] C.C. Gotlieb and F.W. Tompa. Choosing a Storage Schema. *Acta Inform.*, vol.3 (1974), pp. 297-319.

[Guttag 80] J. Guttag Notes on Type Abstraction (Version 2). *IEEE Trans. on Soft. Eng.*, vol. SE-6, no. 1 (Jan. 1980).

[Hoare 69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. of the ACM*, vol. 12, no. 10 (Oct. 1969).

[Hoare 74] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *Comm. of the ACM*, vol. 17, no. 10 (Oct. 1974).

[Jensen 75] K. Jensen and N. Wirth. *PASCAL User Manual and Report*. Springer-Verlag Lec. Notes in Comp. Sci. 18, 1975.

[Jones 78] A. Jones and B.H. Liskov. A Language Extension for Expressing Constraints on Data Access. *Comm. of the ACM*, vol. 21, no. 5 (May 1978).

[Liskov 74] B.H. Liskov and S.N. Zilles. Programming with Abstract Data Types. *SIGPLAN Notices*, vol. 9, no. 4 (1974).

[Liskov 77] B.H. Liskov et. al. Abstraction Mechanisms in CLU. *Comm. of the ACM*, vol. 20, no. 8 (Aug. 1977).

[Low 78] J. Low. Automatic Data Structure Selection: An Example and Overview. *Comm. of the ACM*, vol. 21, no. 5 (May 1978).

[McCarthy 62] J. McCarthy et.al. *LISP 1.5 Programmer's Manual*, MIT Press, 1962.

[Mathlab 77] Mathlab Group. *MACSYMA Reference Manual*. Lab. for Comp. Sci., MIT, 1977.

[Mitchell 79] J. Mitchell, et.al. MESA Language Manual (Ver. 5.0). CSL-79-3, XEROX PARC (Apr. 1979).

[Morris 73] J.H. Morris, Jr. Types are Not Sets. *Proc. 1st ACM Symp. Prin. of Prog. Lang.* (Oct. 1973).

[Rowe 78] L.A. Rowe and F.M. Tonge. Automating the Selection of Implementation Structures. *IEEE Trans. on Soft. Eng.*, vol. SE-4, no. 6 (Nov. 1978).

[Rowe 80] L.A. Rowe, et.al. RIGEL Language Specification. Comp. Sci. Div.-EECS, U.C. Berkeley (March 1980).

[Slate 78] D. Slate and B. Mittman. CHESS 4.6 - Where do we go from here? *Inform. Tech.*, J. Moneta (editor). North-Holland, 1978.

[van Wijngaarden 75] A. van Wijngaarden, et.al. Revised Report on the Algorithmic Language ALGOL 68. *Acta Inform.*, vol. 5 (1975).

[Wegbreit 71] B. Wegbreit. The ECL Programming System. *Proc. AFIPS 1971 FJCC*, vol. 39 (Nov. 1971).

[Wegner 72] P. Wegner The Vienna Definition Language. *Computing Surveys*, vol. 4, no. 1 (Mar. 1972).

[Welsh 77] J. Welsh, M.J. Sneeringer, and C.A.R. Hoare. Ambiguities and Insecurities in PASCAL. *Soft. - Prac. and Exp.*, vol. 7 (Nov. 1977).

[Wirth 77] N. Wirth. MODULA: A Language for Modular Multiprogramming. *Soft. - Prac. and Exp.*, vol. 7 (1977).

[Wulf 76] W.A. Wulf, R.L. London, and M. Shaw. An Introduction to the Construction and Verification of Alphard Programs. *IEEE Trans. on Soft. Eng.*, vol. SE-2, no. 4 (1976).