

Abstract Types Defined as Classes of Variables

D. L. PARNAS

*Research Group on Operating Systems (1)
Computer Science Department
Technische Hochschule Darmstadt
61 Darmstadt, West Germany*

and

*Information Systems Staff
Communications Sciences Division*

and

JOHN E. SHORE AND DAVID M. WEISS

*Information Systems Staff
Communications Sciences Division*

April 22, 1976



NAVAL RESEARCH LABORATORY
Washington, D.C.

Approved for public release; distribution unlimited.

20. Abstract (Continued)

variables is then achieved by allowing the substitution of one variable in an equivalence class for another within specified contexts such as parameter substitution, assignment, and data sharing.

The user of a language with an abstraction facility based on the preceding idea is free to group variables into types in ways meaningful to him but must state the conditions under which members of a group are equivalent. Several kinds of types are proposed for inclusion in such a language. Some examples are types defined by identical externally visible behavior (equivalent specification), types whose members differ according to the value of one or more parameters, and types whose members have identical representations.

The preceding approach to type definition has a strong effect on the possibilities for code sharing, data sharing, and parameter passing. New or unusual compiling techniques will probably be required to exploit these possibilities.

ABSTRACT TYPES DEFINED AS CLASSES OF VARIABLES

INTRODUCTION

The concept of *type* has been used without a precise definition in discussions about programming languages for 20 years. Before the concept of user-defined data types was introduced, a definition was not necessary for discussions of specific programming languages. The meaning of the term was implicit in the small list of possible types supported by the language. There was even enough similarity between different languages so that this form of definition allowed discussions of languages in general. The need for a widely accepted definition of type became clear in discussions of languages that allow users to add to the set of possible types without altering the compiler. In such languages the concept of type is no longer implicitly defined by the set of built-in types. A consistent language must be based on a clearer definition of the notion of type than we now have.

PREVIOUS APPROACHES

We have found the following five approaches to a definition of type in the literature (sometimes implicitly):

- *Syntactic*. Type is the information that one gives about a variable in a declaration. If in old languages one could write "VARIABLE X IS INTEGER" and one can now write "VARIABLE X IS ***,", then *** is a type. Such an approach only avoids the problem. The basic need of a definition appears when one tries to decide what should go under ***.
- *Value Space*. A type is defined by a set of possible values. One may therefore discuss unions, cartesian products, and other mathematically acceptable topics [1,2].
- *Behavior*. A type is defined by a value space and a set of operations on elements of that space [3].
- *Representation*. A type is determined by the way that it has been represented in terms of more primitive types [4,5]. This determination is repeated until one reaches primitive data types that are usually hardware (or compiler) implemented.
- *Representation Plus Behavior*. A type is determined by a representation plus the set of operators that define its behavior; these operators are defined in terms of a set of procedures operating on the representation [6-8].

We have been unable to use any of these approaches to produce a definition of type in an "extensible" language that allowed us to achieve both certain practical goals (such as strong compile-time type checking of arrays with dynamic bounds) as well as the aesthetic

Manuscript submitted March 17, 1976.

goal of having a simple language with a clear and simple set of semantic rules. Each simple set of rules led to the exclusion of cases of practical importance; the inclusion of those cases invariably resulted in a set of exceptions that made the basic semantics of the language hard to understand.

As a result of these experiences we have taken a new approach. We consider the notion of a variable and its permitted contexts within a program as primitive, and we define types as equivalence classes of variables. We do not include a precise definition of a variable, since variables have essentially the same meaning in all commonly used programming languages, and since there is no evidence of any practical difficulty resulting from the lack of a definition. As a result we feel justified in taking the concept of variable to be primitive and using that concept as a basis of our definition of mode and type. For this purpose we consider constants and temporary variables for the storage of intermediate results to be variables as well.

MOTIVATIONS FOR TYPE EXTENSIONS

We begin with a brief discussion of the reasons for including user-defined types, sometimes called type extensions, in a programming language. Including a type definition facility in a language will not increase the class of functions that can be computed by programs in the language, nor will it make possible the generation of better machine code than was possible before. We believe however that type extension can support the following four goals:

- *Abstraction.* An abstraction is a concept that can have more than one possible realization. The power of abstraction, in mathematics as well as in programming, comes from the fact that by solving a problem in terms of the abstraction one can solve many problems at once. The user-defined data type is generally an abstraction from many possible structures of more primitive data elements and many possible procedure implementations. Languages that allow the definition of abstract data types support the use of such programming methodologies as "structured programming," "stepwise refinement," and "information hiding" [9-12].
- *Redundancy and Compile Time Checking.* In defining new types and declaring variables to be of those types, one is providing additional information about the intended use of the data, thereby restricting the set of meaningful operations on the data. In a correct program this information is redundant, but for programs being developed it allows more checking and error detection by the compiler. It is widely believed that such redundancy will lead to more reliable programs and lower program development costs.
- *Abbreviation.* If a program is written in terms of operations on data types and structures defined by the user, it can be shorter than an equivalent program written in terms of data types defined for general purposes. The shorter program is easier to write, understand, prove correct, modify, etc. The source text requires less storage space. Code sharing can occur if the user-defined data elements and their associated operations are suitable for use in more than one program or in more than

one "module" of a large program. Extensive abbreviation and code sharing in programs was made possible by the invention of the subroutine. Data abstractions are the next step.

- *Data Portability.* Often a programmer is faced with using or producing data defined according to a data organization not under his control. Sometimes the programmer must process data independently produced in several systems using different formats. The ability to define data types and write programs in terms of those data types should help reduce the amount of program rewriting made necessary by the introduction of new data or new data organizations. In many important applications the data may be self-describing, so that its characteristics can be completely determined only at the time of actual processing. Extensible languages should be helpful in this situation as well.

In our opinion languages currently being discussed have not achieved the foregoing goals. Current trends seem to favor strongly typed languages that use a representation approach or representation plus behavior approach to type definition [5,7,8]. The definition of types in terms of representation plus behavior interferes with the goal of abstraction, because a technique for handling more than one implementation of an abstraction at a time in one program has not been developed. The desire for compile-time checking interferes with abbreviation and code sharing, because strong type checking tends to prevent code developed to work with data of one type from use on data of another type, even when its application is meaningful. One reason APL programs can be so short is the "everything goes" attitude taken toward the types of variables expected by operators. A definition of type in terms of value spaces interferes with the goal of compile-time checking, since variables that count pears and variables that count light bulbs have the same value space. (Because so many distinct properties of objects can be described with the same value space, it has proven necessary to use the concept of units. We see support for the definition of the units in which a quantity is expressed as being an obligation of the concept of user-defined types. We also note that units as such are often inadequate for our purposes. Very different properties, such as length and height of an object, may be measured in the same units.)

A NEW APPROACH

The extent to which a programming language supports the goals discussed in the previous section depends almost entirely on the situations in which one variable may be substituted for another. Abstraction from the differences between two variables is achieved when one variable may be substituted for the other without making the program illegal or meaningless. Compile-time type checking prohibits the substitution of one variable for another in certain contexts.

All of the practical difficulties that we have encountered in our attempts to use the five approaches to a definition of type previously listed appeared because each definition placed certain restrictions on the context in which variable substitutions were allowed. Those restrictions then prevent the achievement of one or more of the four goals outlined. As a result we have chosen to consider a less restricted definition in which the concept of variable is considered primitive and types are defined as various equivalence classes of

variables that may be legally and meaningfully substituted for one another. To explore such a definition, we need to introduce the concept of a variable's *mode*. (The use of the terms *type* and *mode* here is not consistent with that found in the literature, which is itself inconsistent in the use of these terms. In particular the use of these terms here is different from the use adopted in Ref. 13.)

THE MODE OF A VARIABLE

When a variable is declared in conventional programming languages, such as FORTRAN, ALGOL 60, or PASCAL, the compiler is given enough information to determine how the data referred to by means of the variable is to be represented and which procedures are allowed to operate on the representation. We refer to all variables that are identical with respect to data representation and access as being of the same mode. Once the mode of a variable is determined, the compiler has enough information to produce machine code that will operate on the machine representation of the variable.

Mode defines an equivalence class on variables. Any value that can be stored in one variable of a given mode can be stored in another of the same mode. Any program that operates correctly on a variable of a given mode will operate on any other variable of the same mode under exactly the same conditions (initial values etc.).

TYPES AS CLASSES OF MODES

Each mode defines a simple class of variables, namely, variables whose substitution for each other in any context will not result in a compile-time error. We introduce the concept of type in order to define classes of variables whose substitution for each other is permitted by the compiler only in some restricted contexts. Since we need never distinguish between two variables of the same mode, types can be thought of as classes of modes.

We are unwilling to restrict ourselves to types that consist of a single mode because of our goals of abstraction and abbreviation. In a practical language it should be possible to write programs that can be applied to variables of more than one mode. (Generic procedures are currently often used to solve this problem.) On the other hand the goal of increased redundancy and type checking forbids allowing the compiler to compile code whenever a meaningful interpretation can be imagined. Such an approach is often euphemistically called automatic type conversion. Because we have never seen a system of automatic type conversions that performed all conversions that agreed with our intended use of the data and refused to perform any others, we favor languages that have no automatic conversions. The alternative to types consisting of a single mode is to define types as classes of modes and to specify in terms of types the set of permissible operands for newly defined operators. This allows the programmer who defines a new type to determine the set of permissible operator/operand combinations and requires that he define the meaning of expressions involving his type. Additional burdens are thereby placed on the definer of a data type, but the user of that type is relieved from the onerous task of writing explicit calls on conversion routines in many situations.

In our view the term *abstract data type* is properly applied to the preceding concept of type; that is, a data type deserves the name *abstract* only if it includes more than one mode and if one can deal with all members of the mode class without distinguishing among them.

One can combine modes into types for a variety of reasons, such as to support the goals of abstraction, abbreviation, and code sharing without sacrificing type checking. In the following sections we will discuss examples of situations in which modes should be grouped into types. These examples should not be interpreted as a refinement of the basic definition. The language user should be able to group modes into types almost arbitrarily. The situations we will describe are merely those which we expect to occur most often. Any language that will not allow us to define types in the situations we describe is not satisfactory.

TYPES CONSISTING OF MODES WITH IDENTICAL EXTERNALLY VISIBLE BEHAVIOR (SPEC-TYPES)

For any mode defined by a representation and a set of permissible operators, one can describe those characteristics that can be observed by operating on the representation using only the operators provided. This black-box picture of the mode can be termed its specification [14]. If this specification of the mode contains less information than is contained in a description of its implementation, other modes also satisfy the specification. As an example the specification for modes used to implement complex numbers need not define whether the internal representation is in terms of real and imaginary parts or in terms of argument and magnitude.

The set of modes that satisfy a given specification constitute an important class, which we call a *spec-type*. Any program that is written to operate on variables of a *spec-type* and that can be proven correct without assuming more information about the type than is given in the specification will be correct for another variable of that type even if the mode is different. When the mode is changed, recompilation may be needed, but the program text need not be changed. Given a procedure with a parameter specified to be of a given *spec-type*, it should be possible for the compiler to verify that the parameter is operated on only as permitted by the type specification. The compiler can then permit the procedure to be shared by anyone who wishes to call it with a variable whose mode is a member of the given *spec-type*.

TYPES CONSISTING OF MODES WITH IDENTICAL REPRESENTATIONS (REP-TYPES)

It is common to find data with quite different meanings having the same representation. For example integers and real numbers are often both represented by a single machine word. A user may choose to represent both a two-dimensional position and a complex number by a structure with two real elements. A frequent complaint about languages with strong type checking is that one cannot use the common properties of two modes. These restrictions have been introduced in part to prevent the writing of programs

whose correctness depends on implementation details that are not part of the language definition. (Such programs could become incorrect if a compiler change is made.) Unfortunately the restrictions extend beyond the protection of implementation details. Many operations (such as storage management) can be usefully applied (with the same meaning) to all modes having the same representation. A program may also be useful when applied to variables of different modes that have common representations even though interpretation of the effect of that program may be quite different in each case.

For example the same program could calculate distance from origin for a point in two-dimensional cartesian space and the magnitude of a complex number whose representation is in terms of real and imaginary parts. However nice the aesthetic properties of a language may be, if it forces users to write duplicate programs or forces the code generated to be larger than otherwise necessary, the language will have difficulty gaining acceptance by organizations with strong cost, time, and memory constraints. Under pressure the users of such a language will resort to the dirtiest of dirty tricks to meet their time and space constraints.

From these considerations we conclude that a user should be able to declare as a type a group of modes that have the same representation and to define a set of operations on variables of that type in terms of that representation. We call such a type a *rep-type*. We do *not* want a compiler to recognize common representations. Membership in a rep-type should be declared explicitly in such a way that the compiler can detect undeclared representation dependence. The decision to have a representation-dependent program should be explicit, and the points at which representation dependence is introduced should be easily recognized.

TYPES CONSISTING OF MODES THAT ARE INVOCATIONS OF PARAMETERIZED MODE DESCRIPTIONS (PARAM-TYPES)

One of our goals is the achievement of code sharing, that is, the ability to use the same code to operate on variables of different types. With current computers, compilers, and macro generators it is easy to write code that can be applied to variables that are alike except for the value of one or more descriptive parameters. For example the same code can invert a matrix whether it be 5 by 5 or 60 by 60. As the CLU language shows [7,8], it is also possible to write code that will implement a stack of integers or a stack of variables of type complex. These are examples of the use of parameterized mode descriptions. Both are descriptions in which certain symbols are designated to be parameters. Defining values of those parameters completes the mode description. Thus *integer array* [M:N], where M and N are parameters, defines the mode *integer array* [2:3] if M has the value 2 and N the value 3. Similarly TYP *array* [M;N], where TYP is considered to be a parameter, can generate integer arrays, real arrays, etc. The class of all modes that can be obtained by assigning values to the parameters of a parameterized mode description can be considered as a type. Code sharable by all members of this type can then be written, because it can refer to the parameters.

An example of a language that does not allow code sharing in such situations is PASCAL, which excludes even dynamic arrays as they were known in ALGOL 60. There

have been several proposals to make a special case of such arrays. Rather than recognize one or two special cases, we choose to allow the user to declare modes to be members of the same type if they can be generated by assigning values to the parameters of a mode description. We call this type a *param-type*.

TYPES CONSISTING OF MODES WITH SOME COMMON PROPERTIES (VARIANT-TYPES)

A weaker form of spec-types is needed for situations in which a variety of modes do not have identical specifications but have some common properties that one wants to exploit. (This example of a type can be regarded as a catchall, since it allows handling of anything not falling into any of the previous situations.) Consider a personnel records system. There may be many different modes for representing employees, because the data kept for each class of employee may be quite different. However all of these will have a birth date. An organization may want to invite its employees to a free dinner on their birthdays and will want a program that goes through the personnel records and produces a file sorted according to birth date containing only the name, address, and (in Germany) titles and degrees, so that a proper invitation can be issued a few days before the birthday. This program should be written so that it ignores (abstracts from) those aspects of the personnel record that are irrelevant to the program. It should be possible to declare a type that includes all personnel records and makes them all appear to have only those attributes needed by this program. The program need not be changed when it is applied to new modes. It is only necessary to include the new mode in the abstract type for which the program was written. This type is also defined by a specification, and the operators specified to be common to all variables of the mode must be implemented for the new type in accordance with those specifications. We call such a type a *variant-type*.

MODES BELONGING TO MORE THAN ONE TYPE

An advantage of our basic definition of type is that there are no conceptual difficulties involved in considering one variable to be of more than one type. That arises whenever one mode is a member of more than one class of modes. Types may also be declared to be subsets of other types, or a set of types may be combined because of some common property and considered (for some programs) as a single type.

For example one may have two forms of strings that are members of a spec-type defining the meaningful set of string operations. One of these modes keeps the strings tightly packed for storage efficiency; the other keeps them in a form more suitable for changes. For convenience exactly the same set of string operations are defined for both of them, so that certain programs can be written to operate on variables of either mode (on any variable whose mode is a member of the spec-type) without converting one into the other. The representations of these two modes will not be the same, and if a rep-type is declared for one of the modes, there will be programs that can operate on one but not on the other. These programs will be shared with other members of the rep-type but not with other members of the spec-type. Efficiency requires that we allow such differences to exist; the dictates of structured programming and modularity require that we confine knowledge of those differences to small parts of the system [9-11].

A language that allows a variable to be of more than one type might be regarded as unstructured (unrestricted), because it allows one to write a program that will be correct for one variable of a given type but not for another variable of the same type. It is clear that abuse of this facility could lead to programs that are hard to understand and maintain. Our position is that representation or machine-dependent programs will be written whenever cost considerations demand it; it is better to provide a mechanism that allows the control of such dependency than to force the programmer to use dirty tricks.

THE TIMES AT WHICH CODE SHARING MAY OCCUR

If two variables are members of the same spec-type, they may share the source code of a program but not necessarily the compiled code. This allows source code to be shared among versions of the programs [15]. If several members of the spec-type are expected to be operated on by the same piece of compiled code, then either the code compiled must contain a branch on the mode of the variable or the procedures required to satisfy the spec-type's specifications must be passed with the variable at run time. If the program is written so that the mode of the variable can be determined at compile time, more efficient code can be compiled than if the program is left more general (if only the spec-type is known).

NEED FOR A GENERAL EQUIVALENCE FACILITY

If the concepts in this report are to be used, it will be necessary to have programs in which one data item appears to be of two different types in two different programs. One program will operate on a variable as a variable of type "personnel record," and another will operate on it as a variable of type "officer record." Some part of the system must be responsible for making sure that the same record is operated on in both cases and that the changes are kept consistent. This can be achieved by representing both variables with the same data item. This is an equivalencing facility — but one without all of the dangerous properties of the FORTRAN EQUIVALENCE statement. In FORTRAN, the users of a variable declare the EQUIVALENCE. In our proposal, only the program responsible for implementing the abstract types can make two variables equivalent. Users of the modes cannot. The responsibility for consistency rests with one implementor rather than with all possible users of the data.

DESCRIPTION OF FORMAL PARAMETERS FOR PROCEDURES, MACROS, ETC.

In a language based on the concepts discussed it is vital that the description of a formal parameter given with the declaration of a procedure be permitted in terms of types as well as modes. It is worth remembering that in the ALGOL 60 reference language [3] formal parameter descriptions, although syntactically similar to variable declarations, were referred to as specifications and did not always provide as much information as needed in a declaration. (Implementations of ALGOL 60 often require that the parameter specification include information allowed but not required by the reference language.) The

clear distinction between parameter specifications and variable declarations in ALGOL 60 is often lost because of the syntactic similarity in the two. Allowing user-defined types makes the distinction vital. A parameter specification may be any kind of type, but a variable declaration must always determine a mode.

Allowing the full range of possibilities suggested by this report would appear to require some new or unusual compiling techniques. We view the issues involved in parameter bindings to be among the most important and most difficult remaining problems for developers of languages that allow user-defined data types.

APPLYING THESE CONCEPTS TO DESIGNING A LANGUAGE

Although we feel that the preceding view of types provides a clearer conceptual basis for a language design than the others that we have considered, we have not yet developed a language syntax that embodies our concepts. Syntaxes to accommodate some of these ideas within the context of data-base-management systems have been proposed (as in Ref. 16). We know that the declaration of a mode will resemble languages such as PASCAL and CLU, but we expect the declaration of a type to look quite different. It must be possible to define a type by enumerating the member modes (or types) or by making a declaration of the required properties of member modes or by some combination of the two techniques. For some types a set of operations will exist that must be defined for all modes that are of that type; the compiler must then check that the necessary operators are available for each member mode. Since we do not expect to be able to check for correctness, the compiler is required to check only that an operator of the proper name and form (parameters etc.) exists. We believe that the syntax for parameterized mode descriptions should resemble the syntax for procedures, which we view as parameterized statements.

ACKNOWLEDGMENTS

The authors are grateful to Mr. Warren Loper of the Naval Electronics Laboratory Center and Drs. James Miller and John Nestor of Intermetrics, Inc., for many useful discussions. We are especially grateful to Prof. Dr. Hoffmann of the Technische Hochschule Darmstadt for constructive suggestions. We also thank the referees for pointing out the lack of clarity in earlier versions of this report.

The ideas expressed in this report have been stimulated by the authors' involvement in the Navy's design of a new programming language (CS-4) [13].

REFERENCES

1. C.A.R. Hoare, "Notes on Data Structuring," in *Structured Programming*, O-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Academic Press, 1972.
2. B. Wegbreit, "The Treatment of Data Types in EL 1," *Communications of the ACM* 17 (No. 5), 251-264 (May 1974).

3. P. Naur, editor, "Revised Report on the Algorithmic Language ALGOL 60," *Communications of the ACM*, 6 (No. 1), 1-17 (Jan. 1963).
4. A. Van Wijngaarden et al., "Report on the Algorithmic Language ALGOL 68," *Numerische Mathematik* 14, Feb. 1969.
5. N. Wirth, "The Programming Language PASCAL (revised report)," *Berichte der Fachgruppe Computer - Wissenschaften, Eidgenossische Technische Hochschule, Zurich*, Dec. 1973.
6. O-J. Dahl, B. Myrhaug, and K. Nygaard, "Simula 67, Common Base Language," Norwegian Computing Center, Oslo, Norway, 1968.
7. B. Liskov and S. Zilles, "Programming with Abstract Data Types," *SIGPLAN Notices* 9, 50-59 (Apr. 1974).
8. B. Liskov, "A Note on CLU," *Computation Structures Group Memo 112, MIT, Project MAC*, Nov. 1974.
9. E.W. Dijkstra, "Notes on Structured Programming," in *Structured Programming*, O-J Dahl, E.W. Dijkstra, and C.A.R. Hoare, Academic Press, 1972.
10. D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM* 15 (No. 12), 1053-1058 (Dec. 1972).
11. N. Wirth, "Program Development by Stepwise Refinement," *Communications of the ACM* 14 (No. 4), 221-227 (Apr. 1971).
12. J.M. Aiello, "An Investigation of Current Language Support for the Data Requirements of Structured Programming," *MAC Technical Memorandum 51, MIT, Project MAC*, Sept. 1974.
13. "CS-4 Language Reference Manual," Oct. 1975; available from the Naval Electronics Laboratory Center, Code 5200, San Diego, Calif. 92152.
14. D.L. Parnas, "A Technique for Software Module Specification with Examples," *Communications of the ACM* 15 (No. 5), 330-336 (May 1972).
15. D.L. Parnas, "On Methods for Developing Families of Programs," *Technical Report, Forschungsgruppe Betriebssysteme (I), Technische Hochschule Darmstadt, Darmstadt, West Germany*.
16. R. Boyce and D. Chamberlin, "Using a Structured English Query Language as a Data Definition Facility," *IBM Technical Report RJ 1318, IBM Research Laboratory, San Jose, California*, Dec. 1973.