PUC

Series: Monographs in Computer Science
and Computer Applications
Nº 5/75

Information Systems Group
Report nº IS-1-75

ON THE IMPLEMENTATION OF DATA GENERALITY

by

Arndt von Staa
and
Carlos J. Lucena

Computer Science Department - Rio Datacenter

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

Series: Monographs in Computer Science
and Computer Applications
Nº 5/75

Information Systems Group
Report nº IS-1-75

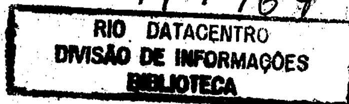
ON THE IMPLEMENTATION OF DATA GENERALITY *

by
Arndt von Staa
and
Carlos J. Lucena

Associate Professors
Computer Science Department
Deptº Informática - PUC/RJ

Series Editor: Larry Kerschberg September, 1975

* This research was partially supported by the Brazilian Government Agency FINEP under contract Nº 244/CT.



M 1467
RIO DATACENTRO
DIVISÃO DE INFORMAÇÕES
BIBLIOTECA

DIVISÃO DE INFORMAÇÕES
BIBLIOTECA
código/registro 1662
data 11/12/75
RIO DATACENTRO

ABSTRACT : Data generality is the property through which program modules are able to communicate via arbritary data structures. Since the design of module interfaces is a very difficult and error prone activity in systems design, the implementation of data generality is a very desirable goal in programming. The present work describes one approach to the implementation of data generality and sketches the algorithm for its implementation.

KEYWORDS : Data Generality, Data Types, Classes, Clusters, Modularity.

RESUMO : Generalidade de tipos é a propriedade necessária para que módulos de programas possam se intercomunicar utilizando estruturas de informação arbitrárias. Como o projeto de interfaces de módulos é bastante complexo e sujeito a erros, a implementação da generalidade de tipos é desejável em programação modular. O presente trabalho descreve uma maneira de se obter generalidade de tipos e esboça um algoritmo para a sua implementação.

PALAVRAS CHAVE: Generalidade de Tipos, Tipos de Dados, "Classes", "Clusters", Modularidade.

1. INTRODUCTION:

A very comprehensive definition of Program modularity has been proposed by Dennis in [1]. According to his definition, a program segment written in a given programming language can be called a module if it follows the properties of syntactic non-interference, semantic context independence and data generality. Syntactic non-interference accounts for the possibility of combining program segments without having to make syntactic changes in any of the segments. Semantic context independence assures that a given segment cannot cause side-effects and cannot be affected by side-effects. In other words, its output assertion (specification) remains invariant no matter where the segment is used within a programming system. The property of data generality requires that modules be able to communicate via arbitrary data structures. Data generality allows for the full application of Parnas' "hiding principle" [2] through which each module's programmer needs to know only about another module's specification and not about its internal implementation.

Difficulties arise in practice with respect to the implementation of the concept of data generality. A software module specification language as proposed by Parnas [3] still requires a reference to the module's internal data structures in the assertions that describe the module. The classical programming solution of carefully planning the module interface data structures is but a distant managerial approximation of the data generality concept (see [4] for a well-designed example of the principle).

Two approaches, by v. Staa [5] and Lucena [6], have been proposed for the full application of the concept of data generality. Both approaches are based on the concept of abstract data types (or simply a data type). For the purpose of this introduction an abstract data type can be thought of as a heterogeneous algebra (defined over more than one set of objects) that can be specified by giving the family of sets or domains and the operations defined on them. A data type can be specified formally through, for instance, the algebraic theory

developed by Zilles [7]. They can be implemented through programming features such as classes [8] or clusters[9].

In the approach by Lucena [6], the issue of data generality arises when clusters are used to model data types used in the context of a very high level language. The same data types, say SET, can be associated to different variables and yet be implemented differently for each variable. If these variables need to interact within the program, the standard compiler representation is used to effect the conversions during execution time.

In the approach by v. Staa [5], the module writers know the specification of a type T whose meaning is global to a program. Each module writer may implement T in a different way and yet he may transfer variables of abstract type to another module without knowing the other modules internal representations. In this paper we discuss the underlying ideas of this proposal and present a sketchy algorithm that describes how data transmition can be achieved with generality.

2. BASIC CONCEPTS

An abstract data type, or simply a type, is an algebra T=(V,0) where V is a set* of values and 0 is a set of operators defined over these values. Types are used to model data abstractions of the kind used in connection with very high level languages. Algorithms expressed in terms of variables with associated abstract types are often called a program specification. A specification can be transformed into a program either by the automatic compilation of a type into a standard data representation or by the possibly automatic substitution of a type by a valid representation selected from a set of possible representations.

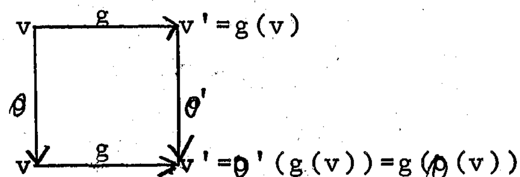
A data representation τ of an abstract data type is a triple $\tau=(V',0',D)$ where $V' \subset V$, $0'$ is a set of constructive

---

* For simplicity of notation and without loss of generality we will overlook the fact that types are, in fact, heterogeneous algebras.

models of the operators in the set 0 (programs that implement the definitions of the operations in 0, the set of axioms that define 0 and 0' are equivalent) and D is the description of a particular representation for the elements of V' for a given base machine (a set data declarations on a given programming language).

Let Φ be a special element of V and V' called the undefined value, and let g be a function* from V to V'. We say that a data representation represents (rep) a data type T, if the following diagram commutes forevery $o \in 0$ and $o' \in 0'$



if τ rep T and additionally for every $v \in V$, such that $v \neq \Phi$, $g(v)=v'$ and $v' \neq \Phi$, we say that data representation represents exactly the abstract data type.

Let $A \subseteq V$ be the set of the values in the domain of g such that $g(a)=\Phi$ and $a \in A$, for a given data representation τ and relation rep. The cardinality of A, $|A|$, is called the degree of approximation of the abstract data type by a data representation. The smaller the cardinality, the better the approximation.

We will call $B \subseteq V$ a set such that $\forall b \in B$, g(b) and g(o(b)) are defined and the "values" o'(g(b)) that correspond to the g(o(b)) are error messages. The cardinality of B, $|B|$, reflects the restrictions that occur in practice in the implementation of the operation o'.

To illustrate the above notions we can use as an example the type INTEGER. INTEGER can be defined by the pair:

---

* In some works about the correctness of data representation g is defined from V' to V. The way we formulated the definition will help presenting our forthcomming ideas.

(set of integer numbers, {+,-,*,/})

Let us call INT a possible representation of INTEGER. INT will be defined as follows:
$(\{ i + -2^{-16} \leq i < 2^{16} \}, \{+,-,*,/\}$, stored as a 16-bit word in two's complement notation)

INT is an approximation of INTEGER because while $v_1 * v_2$ is always defined for INTEGER it may be undefined for INT (provoking an overflow). INT is clearly a *simulation* of INTEGER.

A type can be represented in a variety of ways. We will call

$$IS_T = \{\tau_1, \tau_2, \ldots, \tau_n\}$$

the *implementation set* of a type T. The representations $\tau_1, \ldots, \tau_n$ simulate the type T with different degrees of approximation.

For example, let us assume that the abstract type T models the concept of *stack*. A stack can be characterized, for instance, by the well-known operations: *empty*, *pop*, *push*, etc. In this case τ_1 could be a list implementation of T, τ_2 an array implementation etc.

Let F be a function with $n \geq 0$ arguments of type T to some domain D.

$$F: T x T x \ldots x T \rightarrow D$$

To achieve data generality we want to be able to use the function F within a program module, using τ_i as an approximation for T ($\tau_i \in IS_T$) in the case where the arguments of F were passed as parameters from a module that used τ_j as an approximation of T.

In the appendix we illustrate a programming mechanism which incorporates this concept. We left this example to the appendix in order not to interfere with the central ideas of this work.

Before discussing alternative solutions to the data generality problem we need to introduce another definition.

A <u>conversion</u> is a function from a data representation to another data representation. We will write $x:\tau_i$ to mean that x is a variable of representation (type) τ_i, that is, x takes its values from the domain D_{τ_i}. A conversion C_{ij} is then specified as

$$C_{ij} : D_{\tau_i} \rightarrow D_{\tau_j}$$

Where $\tau_i, \tau_j \in IS_T$. A conversion is said to be meaning preserving if $\forall v,\ v \in V$ and for the function g_{τ_i} from T to the ith representation τ_i:

$$a)\, g_{\tau_i}(v) = \Phi <=> C_{ij}(g_{\tau_i}(v)) = \Phi$$

$$b)\, g_{\tau_i}(v) \neq \Phi, C_{ij}(g_{\tau_i}(v)) \neq \Phi \text{ and}$$

$$g_{\tau_j}(v) = C_{ij}(g_{\tau_i}(v))$$

3. SELECTION OF DATA REPRESENTATIONS

We are interested in the alternative approaches to choose the representation into which parameters should be converted when passing the approximation of a type from a module to another. One possible choice would be to choose a standard representation and use it as a default option. The approach we will favor later on will be motivated by efficiency considerations. We will sketch an algorithm through which a systematic selection of representations can be made.

The standard representation approach can be carried out along the following lines. Let us define for each type a canonical implementation (representation) $\tau_c \in IS_T$, such that parameters of type T, being shipped from another module, would always be converted into τ_c. This operation implies that

$$C_{ic}: D_{\tau_i} \rightarrow D_{\tau_c} \text{ and}$$

$$C_{ci}: D_{\tau_c} \rightarrow D_{\tau_i}$$

are defined (if we want the modules to be able to communicate). Several objections can be raised with respect to this approach. For one thing, τ_c can lead to an inefficient execution of a given critical (oftenly executed) operation of T. Besides, C_{i_c} may not be definable, in the sense that τ_c is not a meaning preserving representation of τ_i, or may be a very costly operation.

To overcome these difficulties, we propose a method which is presently being considered as a basis for a language being designed at PUC*.

We start by defining a set of meaning preserving conversions

$$C_{ij} : D_{\tau_i} \to D_{\tau_j}$$

for significant pairs of implementations $\tau_i, \tau_j \in IS_T$. As a next step we associate a cost function k_{ij} with each c_{ij}. We foresee the mechanism being described, used in connection with a language with a strong typing capability. That is, the language is capable of knowing the actual type τ_i and formal type τ_j at parameter association time.

When converting from τ_i to τ_j we may not find a conversion function C_{ij} that performs the operation. Instead, we may need to look for a sequence of conversions ~~which~~ whose composition produces the same effect of C_{ij}. That is, we need to be able to detect a sequence of conversions such that:

$$C_{ij} = C_{12} \circ C_{23} \circ \ldots \circ C_{n-1\ n}$$

Where

$\tau_i = \tau_1$, $\tau_j = \tau_n$ and $1<n$, and such that

$C_{r\ell}$ exists for all the pairs ($r=1$, $\ell=2$), ($r=2$, $\ell=3$, ..., ($r=n-1$, $\ell=n$)

---

* Pontifícia Universidade Católica do Rio de Janeiro

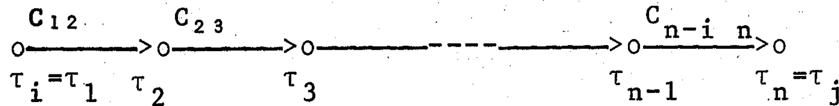
Additionally, the $C_{r\ell}$ are required to be meaning preserving and the sum

$$\sum_{(r=1,\ \ell=2)}^{(r=n-1,\ \ell=n)} K_{r\ell}$$

$k_{r\ell}$ cost function

needs to be minimal over the set of defined conversions.

Graphically this sequence can be visualized as follows



If such a path choice exists within the library of conversions, we can achieve data generality and portability as a byproduct.

It needs to be stressed that if we want to achieve intercommunication between modules (e.g.co-routines) it is required that another sequence of conversions C_{ji} also exists, possibly the inverse of C_{ij}, that allows the conversion from τ_j to τ_i.

In a network environment a programming system with the described capability should be able to go back to the user and interact with him in the case of impossibility of accomplishing a particular conversion. In this case the system should describe the environmental contraints (size of words in each machine etc), that were responsible for the problem. The user in such a situation should be able to reply and propose some "roundings" that would weaken the requirement of meaning preservation, so that the processing could successfully terminate

A number of interesting features can be stressed in the method outlined. It follows directly from the above that the method is very general and oriented to the achievement of maximum efficiency.

The method could be implemented by means of THUNKS [11] and occurs in a ~~degenerate~~ incomplete form in ALTRAN [10]. The

method also suggests a new direction to the open problem of generality efficient code for very high level languages.

It can be proved without too much effort that if we have a library of conversions, all that is required of its structure in order to be able to produce every possible conversion is that it form a strongly connected directed graph of conversions.

In a very high level language all implementation details (in particular data structures that are oriented towards the base machine) are hidden from the user. As a result of that, a general purpose compiler that compiles directly from the abstract data types used by the language (e.g. general sets and sequences) have to be general enough to allow for the implementation of the typically associative operations used at the language level (usually called specification level). This fact has led in all reported experiences to the generation of prohibitively expensive object code. The approach suggested in this paper induces a technique through which a compiler can select the best representation (from an existing repertoire) for a particular application of a type and also allows for the selection of paths of transformations that enable adjustments in the form of implementations for very different uses of the same data type.

Following this suggestion when defining a very high level language, an abstract type needs to be specified through the following tuple

$$T=(O,\ IS_T,\ CS)$$

The meaning of each element can be illustrated through an example:

O is a set of operations which define the type; e.g.{+,-,*,/,:=} for fixed binary

IS_T is the implementation set of this type; e.g.{precisions (15,0),(12,3),(31,0)} for fixed binary)

$$C_{ij}:\ x:\tau_i \rightarrow x:\tau_j$$

3. CONCLUSIONS

In the present paper we attempt to contribute to the better understanding of the problems of communication between program modules. We outlined one approach to data transmission that can help bridge the gap between the goals of ease of programming and program efficiency. We are aware that a number of issues remain to be investigated in this realm and that we will be raising a number of important problems as we proceed with the implementation of the method described here.

APPENDIX

In figure 1 we show the function concat. This function receives parameters and type descriptors associated with these parameters. We also show the minimum set of operators that have to be provided in order for the function to be used.

```
ref type-1 function concat (type type-1 contains(integer size;
          ref type-1 function obtain (integer length);
          type type-2; type 2 elem[*]; ref type-1 a,b);
   begin concat;
      integer length = a->size + b->size;
      concat:=obtain(length);
      for i:=1 until a->size do
        concat->elem[i]:=a->element[i]; od;
      for i:=a->size+1 until length do
        concat->elem[i]:=b->elem[i-a->size]; od;
    end concat;
      The following parameters were made explicit
      ref type-1: obtain(length);
      integer /*fetch function*/size;
      type-2[*]  /* access function*/ elem;
```

Fig.1 Example of a function which accepts types as parameters

In figure 2 we illustrate the definition of a type which will be transmited to concat.

```
type string (integer string-size)=
   begin string;
      outside-scope scope;
         integer fetch function string-length=string-size;
         bit(2) string-elem[string-size]
         ref string function get (integer length)=
           get:=new string(length),
         ref string function constructor(bit(2)vector[*]=
          begin constructor;
             integer length=upper bound(vector),
             constructor:=get(length);
             for i:=1 until length do
                constructor->string-elem[i]:=vector[i];
             od;
          end constructor;
      end scope;
   end string;
```

Fig. 2 Definition of the type "string of bit (2)"

Figure 3 shows how the transmission is accomplished

```
ref string a,b,c;
a:=constructor(array('01'B,(10'B));
b:=constructor(array('11'B));
c:=concat(type-1::string(elem::string-elem,size::string-
                         length,obtain::get,type-2::bit
                         (2)), a::a, b::b);
```

Fig. 3 Use of concat with the type "string"

Note that in figure 1 the operation contains(...) makes explicit the parameters that are transfered when "string" is used. In figure 3 the type "string" is transmited to concat. The names defined by "string" and the names expected by concat. are different. This difficulty is overcome through the association of the parameter names.

The sintax used is:

<name of the formal parameter>::<actual parameter>

The symbol <name of the formal parameter> stands for the textual name of the formal parameter that will be associated with the value of <actual parameter>. The construct array(...) in figure 3 defines a function which creates an array of as many elements as the actual parameters in the list. The elements of array will be initialized to the actual values of the parameters.

REFERENCES

[1] Dennis, J.B., 'Modularity'; in Bauer, F.L. ed; Springer Verlag; 1973

[2] Parnas, D.F., 'Information Distribution Aspects of Design Methodology'; IFIP Congress Proceedings; 1971

[3] Parnas, D.F.; 'A Technique for Software Module Specification with Examples'; Communications of the ACM Vol. 15, nº 5, May 1972

[4] Mc Keeman, W.; 'Compiler Structure'; Proceedings of the First USA - Japan Computer Conference; 1972

[5] Staa, A.v.; Data Transmission and Modularity Aspects of Programming Languages; Research Report CS-74-17, Department of Computer Science, Univ. of Waterloo; 1974

[6] Lucena, C.J.; On the Synthesis of Reliable Programs, Technical Report, Computer Science Department, Univ. of California, Los Angeles; 1975.

[7] Liskov, B.H.; Zilles, S.N.; 'Specification Techniques for Data Abstractions'; IEEE Transactions on Software Engineering, Vol. 1, nº 1; 1975.

[8] Hoare, C.A.R.; 'Proof of Correction of Data Representations' Acta Informatica Vol. 1, fasc 4; 1972

[9] Liskov, B.H.; Zilles, S.N.; 'Programming with Abstract Data Types'; SIGPLAN Notices Vol. 9; 1972

[10] Brown, W.S.; ALTRAN User's Manual. Bell Telephone Lab., Murray Hill, N.J., 1973

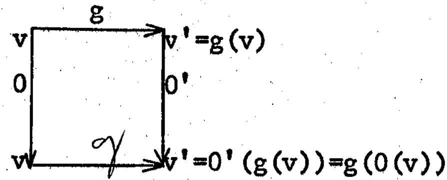
[11] Ingerman, P.Z.; 'THUNKS, a Way of Compiling Procedure Statements with Some Comments on Procedure Declarations'; Comunications of the ACM, Vol. 4, nº 1; April 1961.

[12] Bauer, F.L. ed; Advanced course on Software Engineering; Series: Lecture Notes in Economics and Mathematical Systems nº 81, Springer Verlag; 1973.

ERRATA

VON STAA, Arndt & LUCENA, Carlos J. - On the implementation of data generality...

Pages	Lines	Where it reads	Please, read
3.	2	in the 0 (the sit of...	in the 0; the sit of...
3.	10	for very 0 ε 0...	for every 0 ε 0...
3.	Replace	the figure in the text by the figure bellow	



Pages	Lines	Where it reads	Please, read
4.	8	simultation...	simulation...
4.	9	in a veriety...	in a variety...
4.	24	argument of...	arguments of...
6.	14	for significat...	for significant...
6.	22	conversions which...	conversions whose...
7.	17	sequence of conversion C...	sequence of conversions C...
7.	last	degenerate...	incomplete

DIVISÃO DE INFORMAÇÕES
BIBLIOTECA
código/registro
data
...../...../.....
RIO DATACENTRO

RIO DATACENTRO
DIVISÃO DE INFORMAÇÕES
BIBLIOTECA