PUC

Series: Monographs in Computer Science
and Computer Applications
Nº 12/75

ON THE DESIGN OF LANGUAGES FOR A DISCIPLINED USE OF REFERENCES

by

Arndt von Staa

Departamento de Informática

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 12/75

ON THE DESIGN OF LANGUAGES FOR A DISCIPLINED USE OF REFERENCES *

by

Arndt von Staa

Series Editor: Sergio E.R. Carvalho December, 1975

* This work was partially supported by the Brazilian Government Agency FINEP under contract Nº 244/CT, and developed by the Computers and Teleprocessing Communication Group, Departamento de Informática, PUC/RJ.

ABSTRACT

In this paper we study programming language design aspects with the purpose of providing means for a disciplined use of references. This objective is achieved through the introduction of the concept of values of type type .

KEYWORDS AND KEYPHRASES

Modular Programming, Programming Language Design, Program Module, References, Data Space Identifications, Values of Type type.

RESUMO

Neste trabalho estudamos aspectos de projetos de linguagens de programação com o propósito de conduzir a um uso disciplinado de indicadores. Nós alcançamos este objetivo através da introdução do conceito de valores do tipo tipo .

PALAVRAS E FRASES CHAVE

Módulo de Programa, Programação Modular, Projeto de Linguagens de Programação, Indicadores, Identificação de Espaços de Dados, Valores de Tipo tipo.

CONTENTS

1 INTRODUCTION

Some of the desirable features of program modules are: context independence and syntactic non-interference [Dennis 73], strength and low degree of coupling [Myers 73], and transparency (property by which modules offer no insight into their implementations [Parnas 71]).

Modules should be able to implement special forms of memories such as stacks, lists, arrays, etc. (clusters [Liskov 74]). In this case, a *data space identification* must be provided explicitly or implicity to an *access function* to be contained in the module that enables the access to the specific data space which is internal to the module. In a sense, this amounts to making internal data spaces (the memory) available to the exterior (the user of the module).

We used the expression access function in the sense of a mechanism by means of which we gain access (read or write) to some space on some storage medium.

The object of our study will be to determine conditions under which making internal data spaces available to the exterior does agree with the above mentioned properties of modules. As we proceed we will point out that this objective may be attained through a disciplined use of pointers. We will not cover in this paper the difficulties arising from sharing data spaces (or access paths) among several modules.

2 THE CONCEPT OF PROGRAM MODULE

Our characterization of the concept of a *program module*, or simply *module*, is similar to the one introduced by a SIMULA 67 *class* [Dahl 68], or by a *cluster* as in CLU [Liskov 74].

Specifically a program module defines:

i. internal data spaces, internal type descriptors and internal functions which are not known externally to the module;

ii. accessible type descriptors and functions which can be made explicit to

the exterior of the module, i.e. are within its outside scope.

Modules may act as type descriptors, or simply as types, e.g. integers, stacks, lists, trees. For example, the type integer can be described by a module defining an internal data space, a word, and the accessible operations copy, fetch, add, multiply, etc. An integer variable declaration is then nothing more than the creation of an instance of the module describing this type, i.e. an "object" of a SIMULA 67 class.

Type descriptors are information and, consequently, are values of the type type. It is conceivable then to allow the existence of variables and parameters of this type. Such variables and parameters allow the practice of modular programming, where the implementation characteristics of a given type descriptor are effectively hidden from its users.

```
type queue of  (type  user_type) is
     begin queue;
          type node is struct (ref node next; user_type info);
          ref node head, tail;
          head: = tail: = null
          outside scope functions;
               user_type function dequeue is ...
               function enqueue (user_type data) is ...
          end functions;
     end queue;
```

Figure 1. Example of a type descriptor with a parameter of type type

In figure 1 we show an example of a type descriptor where we use a parameter of type type . Due to this parameter we may implement the module "queue" without having to define completely the information stored within the nodes of this module. In this paper we are not concerned with efficiency, type identification, and parameter association. The interested reader is directed to [Staa 74] for a discussion about these problems.

It should be clear from the example shown in figure 1 that "user_type" does not need any information about "queue" and, conversely, "queue" does not need any information about "user_type" but for the size of the space occupied by an instance of "user_type".

3 ENCAPSULATED ACCESS

When making internal data spaces available to the exterior of a module some form of identification of these spaces is necessary. This identification can be the address of the data space, as in the case of references, or some information provided to a function which performs the access, as in the case of indices belonging to the index set of an array *.

Before proceeding, we will introduce the concept of an access typed value. Such values designate data spaces where the information to be stored or retrieved is located. Thus access typed values are special forms of references. They differ from references, since they cannot be read explicitly. Using ALGOL68 terminology [Winjgaarden 69], access typed values are always dereferenced inside expressions. Intuitively access typed values act as if they where dynamically bound in the same way as a formal parameter, or a dummy argument called by reference. In ALGOL68 terminology an access typed value is a "ref type" value rather than a "ref ref type" value. The latter kind is a reference in our terminology.

It is not difficult to visualize functions which return access typed values. When using such access functions, we can effectively substitute external accesses to internal data spaces by calls to the corresponding access function. The specific data space to be accessed is then determined by adequate actual parameters. Notice that we have not restricted anything yet since such an access function could be the identity function receiving a pointer or a reference as parameter.

---

* For the sake of completeness, a pointer is simply an address (e.g. PL/I), whereas a reference is an address to a data space of some fixed predefined type.

In many cases access functions do not require any actual parameter, since the data space to be accessed is already implied, e.g. accesses to record fields, stacks queues, etc. We say that such functions establish an encapsulated environment since the user has no means at all to violate internal data spaces, as long as the access functions are correct.

Ideally all accessible access functions (see our characterization of the concept of modules) provided by a module should establish an encapsulated environment since this would allow us to get away with data space identifications and, thus, with pointers, references and the like. Unfortunately, however, we cannot do so, since there are modules which implement sort of "random access" memories and consequently the corresponding access functions need parameters defining the data space to be accessed, e.g. lists, arrays.

4 ACCESS TYPED VARIABLES

It may be costly to repeatedly evaluate access typed functions. It would then be desirable to assign an access typed variable. This is impossible, however, due to the restriction imposed on this kind of value. The problem can be solved by means of a construct which makes the attempted assignment explicit and that also states a textual scope within which the target access type variable is bound. This is similar to a parameter called by reference, only that, in this case, no activation record is created.

```
integer array   A[1:10, 1:10],   B[1:10, 1:10];
access integer pivot;
with pivot = A[i, i] do;
     ... B[i, j] /pivot ...
od;
```

Figure 2. Example of the with statement.

In figure 2 we show an example of the with statement. This statement is a generalization of the PASCAL [Wirth 72] with statemente or the SIMULA67 inspect statement. It is a generalization since our version of the with statement allows the nesting of with "blocks".

The with statement is a double edged sword though, since two distinct access paths are established and, thus, synchronization problems may arise. As mentioned earlier, we will not deal with these problems here, since this would be beyond the scope of this paper.

5 CONTROLLED ENVIRONMENT

We will examine now the problems that arise when parameters are passed to access functions.

Definition 5.1 - We say that module M existe within a controlled environment if:

a. all external accesses to data spaces internal to M are made only by means of some accessible access function defined by M;

b. access functions can have an arbitrary complexity and must successfully elaborate for any access request;

c. access functions yield access to at most one data space for each individual access request.

It must be noted that access functions which return erros conditions, or well defined interrupts for incorrect parameters, elaborate successfully for any input parameter. A parameter list can be visualized as a single compound (vector) parameter, thus we are not imposing any restriction by considering only single parameters. Finally, a data space does not have to be a contiguous physical memory space [Staa 74]. Again, generality is not lost by referring only to accesses to data spaces as if they were single spaces.

Definition 5.2 - Let α be a data space which is internal to some program module M. Furthermore, let F be an accessible access function defined by M. An F-identification of α is a value V of some type τ such that:

i - α and only α is accessed when V is presented to F;

ii - F produces an error exit for any value V' of type τ presented to F and for which there is no corresponding data space;

iii - externally to M, values of type τ are indivisible and cannot be obtained by computational means;

iv - externally to M, values of type τ can access data spaces internal to M only by means of F.

Several observations are due here. Conditions (i) and (ii) say only that data space identifications are bound to at most one data space, thus avoiding possible ambiguities of access. Condition (iii) determines that data space identifications may not be forged externally to M. Finally, condition (iv) forbids the direct use of references since externally to M we are unable to access information contained within M unless we use some accessible access function defined by M. This is true even if the access function is the identity function, i.e. a function whose parameter is the reference to the data space to be accessed.

Theorem 5.3 - A program module M exists within a controlled environment, iff M makes internal data spaces available to its exterior only by means of F-identifications.

Outline of the Proof - Suppose that conditions (i) thru (iv) of F-identifications hold. Condition (a) of controlled environment is trivially met by condition (iv). Condition (b) is met by condition (iii) since any specific F-identification V may carry as much additional information as necessary, where this information can neither be modified externally nor can it be produced externally, thus allowing the access function to be of any complexity. Finally, condition (c) is met by conditions (i) and (ii) since accesses are

gained only to the data space bound to the F-identification if this binding exists.

If conditions (i) or (ii) do not hold, condition (c) cannot possibly be met, since accesses are either ambiguous or are made to undefined data spaces. If condition (iii) does not hold, condition (b) cannot be met, since part of the information contained within an F-identification could be modified, thus restricting the set of tests which could be performed by the access function F and consequently its complexity. Finally, if condition (iv) does not hold trivially, condition (a) cannot be met.

A by-product of this theorem is that references may still exist, but only within the module which defines them. Thus, for example, we may define modules which perform list manipulation without being able to interfere with these lists from the exterior, even when the F-identifications are disguised references.

In order to activate an accessible access function, the module instance within which it acts must be given. By inclusion of adequate information into F-identifications, it is possible to completely separate accesses relative to different instances of the same module even if these accesses are relative to data spaces contained within some dynamic memory shared by all these instances. We conclude then that the use of F-identifications is even more restrictive than the substitution of references by indices into an array.

Modules may define accessible type descriptors. For each instance of such a module, the type descriptor is identified in a different manner, thus corresponding to a different type (see [Staa74]). Therefore the separation of the data spaces by module instances may be tested at compile time in most cases, making our model quite efficient.

It follows from all the above that references must not necessarily be thrown away; it is sufficient to hide them. Of course the control (or protection) we gain is weak, since if any of the conditions is not followed rigorously, the control is lost. The control is particularly weak since it is based on syntactic aspects of the language, enabling a loss of control when different language processors are combined.

Access Functions

```
type tree if (type user_type) is
   begin tree;
      type node=struct(user_type info;
                integer node_id; ref node left, right);
      type reference=struct(ref node pointer; integer id);
      outside scope tree_ops;
         type node_ref=shield reference;
               ...
         user_type access function get(node_ref what) is
            if what.id=what.pointer->node_id
               then what.pointer->info;
               else fail;
               ...
   end tree;
```

Example of use:

```
tree a (integer);
type pointer=a.node_ref;  pointer point;
with node=a.get( point ) do ... node:=10; ... od;
```

Figure 3 - Example of the use of type shield.

In figure 3 we show an example of a module (type) implementing a forest. This module assures that there is a strict separation between different trees of this forest even when nodes migrate between trees. That is, accesses are negated when attempting to access a node of a tree with an

obsolete data space identification. Due to the indivisibility condition, the user of this module is unaware that this safeguard exists and, if he notices it he is unable to determine how it is enforced and / or how to circumvent it.

The type operation shield in figure 3 generates a copy of the shielded type descriptor where all operations but copy and declare have been eliminated. A shielded type is thus indivisible and not " producible " externally to the module where it is defined. Within this module though , there is an identity conversion between values of the original type and the shielded type allowing direct use of shielded values .

5. CONCLUSION

We have shown in this paper how to use references in a disciplined way. To allow us to do this, we departed from the conventional form of languages and introduced the concept of types as modules and values of type type. We have shown that in such a programming environment we are able to effectively hide the existence of references externally to the module which defines them, without having to forbid their existence within this module.

Several details have been omitted, specially with regard to type identification and shielded types. The interested reader is directed to [Staa74] for an extensive description of these details.

Some problems remain open, such as synchronization of the execution of programs which define several access paths to a same data space. Such problems occur for instance when inadvertedly dividing the pivot element by itself in a matrix inversion program.

REFERENCES

[Dahl 68] DAHL, O.J. et alii. Simula 67 common base language. Norway, Norwegian Computing Center, 1968. S-22.

[Dennis 73] DENNIS, J.B. Modularity. In: BAUER, F.L., ed. Advanced course on software engineering. Berlin, Springer - Verlag, 1973. p. 128-82.

[Liskov 74] LISKOV, B. et alii. CLU design notes. Massachusets, Massachusets Institute of Technology, 1974.

[Hoare 73] HOARE, C.A.R. Hints on programming language design. Stanford, Computer Science Dept., 1973. CS-403.

[Myers 73] MYERS, G.J. Characteristics of composite design. Datamation, 19 (9) : 100-2, Sept. 1973.

[Parnas 71] PARNAS, D.L. Information distribution aspects of design methodology. In : INTERNATIONAL FEDERATION FOR INFORMATION PROCCESSING. IFIP Congress, 1971. Amsterdam, North Holland, 1971. p. 26-30 TA-3.

[Staa 74] STAA, A.V. Data transmission and modularity aspects of programming language. Waterloo, Univ. of Waterloo, Dept. of Computer Science, 1974.

[Winjgaarden 69] WINJGAARDEN, A.V. et alii. Report on the algorithmic language Algol 68. Numerische Mathematik, 14 : 79-218, 1969.

[Wirth 72] WIRTH, N. The programming language Pascal (rev. Report) Zuerich, Eidgenoessische Technische Hochschule, 1972. TR-5.