PUC

Series: Monographs in Computer Science
and Computer Applications
Nº 5/76

ON THE TREE ORGANIZATION OF ABSTRACT TYPES

by

Carlos H. Lauterbach

and

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 5/76

ON THE TREE ORGANIZATION OF ABSTRACT TYPES *

DIVISÃO DE INFORMAÇÕES
BIBLIO
código/registro
2363 29, 6, 76
RIO DAT

by

Carlos H. Lauterbach **

and

Arndt von Staa

M 2085
RIO DATACENTR
...ÃO DE INFORMA
BIBLIOTECA

Series Editor: Sérgio E.R. Carvalho March, 1976

* This work was supported in part by the National Science Foundation (NSF-USA) and the Conselho Nacional do Desenvolvimento Científico e Tecnológico (CNPq-Brasil) and developed by the Information Systems Group of the Departamento de Informática - PUC/RJ.

** Centro de Ciencias de Computacion, Universidad Catolica, Santiago - Chile.

Copies may be requested from:

Rosane T.L. Castilho, Head
Setor de Documentação e Informação
Deptº de Informática - PUC/RJ
Rua Marquês de São Vicente, 209 - Gávea
20000 - Rio de Janeiro - RJ - BRASIL

ABSTRACT:

The concept of an abstract type tree is presented. By means of this tree a well defined mechanism to automatically perform type conversions is presented. It is shown how the existence of this type tree allows achieving data generality.

KEY WORDS:

Modules, modular programming, types, abstract type trees, conversion, transfer, type refinement, type extension.

RESUMO:

É apresentado o conceito de árvore de tipos abstratos. Por intermédio dessa árvore é apresentado um mecanismo bem definido para o estabelecimento automático de conversões. É mostrado também como esta árvore de tipos abstratos apoia a generalidade de tipos de dados.

PALAVRAS CHAVE:

Módulos, programação modular, tipos, árvores de tipos abstratos, conversão, transferência, refinamento de tipos, extensão de tipos.

CONTENTS

1 - INTRODUCTION 1

2 - THE TYPE TREE 1

2.1 - The definition of type tree 2

2.2 - Coersions 6

2.3 - Operations on the type tree 7

2.4 - Examples of a type declaration 11

3 - CONCLUSIONS 13

REFERENCES 14

1 - INTRODUCTION

In the present days, much attention has been given to modular programming. One of the properties which modules must possess in order to permit modular programming is Data Generality. Data Generality can be defined intuitively as being the capability to combine modules interchanging information via abstract data types [Dennis 73]. Since modules must interchange information via concrete implementations of such data types, a need for interfacing modules via conversion functions arrises [von Staa & Lucena 75].

We present in this report the concept of an abstract type tree, where this tree defines the relations which exist among all types of a given program. By means of this type tree, a clean mechanism for automatically choosing conversions can be defined.

The abstract type tree is infinite. We present thus a mechanism which turns explicit the specific subtrees (type refinement) necessary to allow compilation to procede.

We also present mechanisms by means of which users may define new types within the type tree (type extension). This mechanism allows users to adapt the type tree to their specific needs and thus contributes to the ability of writing programs in a modular way.

An abstract data type is a conceptual entity. Abstract data types define characteristics which values of such types possess, e.g. valid operations, valid values (value set axioms). It is completly machine and implementation independent.

2 - THE TYPE TREE

In this section we will introduce the concept of an abstract data type tree. Several examples are presented, which are based on MADCAP-S [Melkanoff et alli 74]. The section

is divided into the following points:

- The definition of the abstract type tree
  the concepts of type and objects are introduced, and the organization of types is discussed.

- Coercions
  the concept of coercion is discussed, and the relation between abstract type and type realization is introduced.

- Operations on the type tree
  operations modifying the type tree are introduced, and the mechanism to associate an implementation with an abstract type is presented.

2.1 - The definition of type tree

Types are hierarchically organized in a data type tree. This type tree is a tree of abstract types, each node defining abstract selectors, operators, etc. The meaning of these selectors and operators is identical for all the subtypes of a type. Furthermore, each subtree is considered to be a proper subset of the root.

If a node N possesses the subtrees ST1, ST2,...,STn, the type of N is the union of these n types. If n is greater than 1, each of the STi is clearly a subset of N. It should be noted that the node N may define values that are non existent in any of the subtypes. In the case that there is only a single subtree, this subtree must define fewer elements than the root, since otherwise it would not be a proper subset. In figure 1 we show the abstract type tree which is predefined in MADCAP-S.

The universe of discourse in such a tree organization can be viewed as being the root of the data type tree. We will call this root node UNIVERSE. Its main characteristic is that any predefined type or user defined type used by a given program

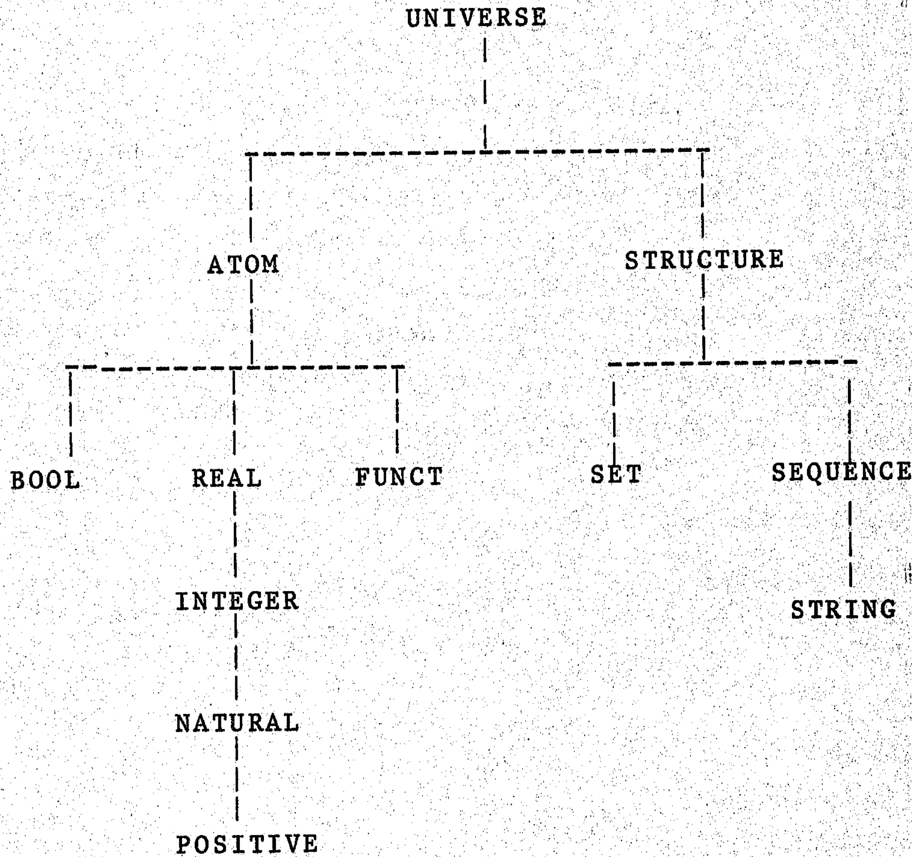


Figure 1.

is a subset of UNIVERSE and that a variable of type UNIVERSE can carry any object.

Several different implementations of an abstract data type are possible. All these implementations are associated with the node where the abstract type is defined. The implementations may each possess different names. In the case of multiple implementations the selection of a specific implementation may be done based on efficiency considerations, as well as by means of user defined selection.

The type tree is a conceptual tree, and it is infinite. Each compilation will cause the creation of a subtree of this type tree, where this subtree contains all types effectively used during compilation. The type tree is infinite since the definition of types may be based on other types (possibly UNIVERSE). Let N be such a type, and T be a type necessary to define N. N defines then a subtree which contains a copy of the type subtree with root T. For example, the type SEQUENCE is predefined as

$$\underline{\text{type}}\ \text{SEQUENCE} = \text{UNIVERSE}^{<\ \text{infinity}}$$

consequently there are an infinite number of possible refinements of the type sequence, such as REAL^2, $\text{INTEGER}^{<\ \text{infinity}}$, SEQUENCE^{24}, etc.

Subset relationships

The subset relationship of data types has very important consequences in the interpretation of the type tree:

1) Operators defined over a given data type (in general over a given domain) are automatically inherited by the subdomains. For instance, division is defined in MADCAP-S as

   "/" : <<REAL, REAL; -> REAL>>

   and it is also inherited by the integers with the same definition.

2) Operators can be restricted over a subdomain (or subtype of a type), as long as the meaning of the operators remains unchanged. This feature has the following implecations:

   a. it gives more information about the ranges of operators operating over such restricted domains, and, therefore, it allows the compiler to make assumptions about the

results. E.g, in MADCAP-S the operator "+" for reals is restricted in the following way when applied to integers:

"+" : <<INTEGER, INTEGER; -> INTEGER>>

b. it allows for more efficient operator implementations, possibly involving different representations for objects. This is possible due to the requirement that the restricted operator has exactly the same meaning as the unrestricted operator.

c. in the presence of multiple definitions of the same operator within the same arc of the type tree, the operator chosen is the one whose domain is the closest to the actual domain of the parameters used.

3) New operators can be defined over subdomains. These operators are not defined over types that contain the given subdomain. Furthermore, these new operators must not conflict with existing ones, i.e. they must possess different textual names from all the operations occuring within the arc from UNIVERSE to the node defining this new operator. If not so, semantic equivalence is assumed possibly resulting in erroneous execution due to inadequate operator choice, viz. paragraph 2 above.

Two disjoint subtrees lead to mutualy disjoint data types. This disjointness requirement is based on the meaning of the values of a given type and not on the representations of these values. Thus the types

type T1 = {50..500}, and

type T2 = {10..100}

are two disjoint types since the meaning can not possibly be the same for both. However, both could be subsets of the type

"sets of integers", and both could have value representations in common, e.g. the set {50..100}.

2.2 - Coercions

Two kinds of conversions may be introduced automaticaly:

a) Conversion from one implementation to another one of the same abstract data type. Such conversions are permissible since they preserve the meaning of the abstract object. Conversions of this kind may fail, however, due to implementation restrictions [von Staa & Lucena 75] , e.g. converting from a "list representation" of a STACK to an "array representation" of a STACK may fail due to the restricted size of the array. Furthermore, conversions should not reduce precision. For example, when converting integers to reals precision may be lost due, in general, to that the fraction part of a floating point number permits fewer significant digits than an integer. Consequently the conversion from integer to real must fail whenever a loss in precision occurs.

b) Conversion from a subtype to a supertype. Such conversions are possible since, by definition, subtypes are effectively subsets of the supertype and, furthermore, all operations applied to the supertype are automaticaly inherited by its subtypes.

These two conversions may be thought of as being simple changes to the representation. From a purely abstract point of view, the meaning of the values converted are preserved. Thus, theoreticaly at least, a transfer back to the original representation is always possible and yields exactly the original value. It follows then that these conversions are always permissible.

From the above discussion it follows that, within a set of given implementations, it is possible to have subtypes

containing elements not contained in the supertype. This is a purely technical problem and does not occur at the abstract level. The problem can be countered by "disjoining" the types where it occurs, or by giving better implementations to the types.

The "broadening" operation (i.e. converting to a supertype) of an object should always be assumed, but the converse ("narrowing", i.e. converting to a subtype) must be requested explicitly, except in assignments (see below). Narrowing consists of

1) testing if the object really belongs to the subtype (2.0 is integer but 2.4 is not), and
2) if it belongs, in mapping the value to the subtype's representation. (Notice that this mapping could be the identity mapping).

The only place where narrowing should automaticaly be assumed is in assignments, where the resulting value of the expression is converted to the type of the target object. Assignments involving disjoint types can be detected at compile time, and must be forbidden.

Conversions among disjoint types are the user's responsibility, and may be performed by means of explicity invoked transfer functions. For instance, in order to convert an integer into a singleton set, whose only element is this same integer, an explicit call to an appropriate transfer function must be issued. Such transfers should never be introduced automatically.

2.3 - Operations on the type tree

In order to achieve data generality, the user must be able to define implementations of abstract data types, as well as abstract data types themselves. Following are the operations on the data type tree to accomplish this:

a) type refinement and

b) type extension

a) Type refinement

Type refinement consists in declaring more precisely some abstract data type. Type refinement can be viewed as an inclusion of a particular type into the compile time subtree of the abstract type tree. Recall that the type tree is conceptual, a subtree of this type tree being constructed at compile time based on the particular needs of the compilation. Thus type refinement does not create new types. It only fixes a type which already exists conceptually. This allows the compiler to produce code based on the characteristics of the refined type.

Type refinement is also used to associate a given implementation with an abstract type. By means of this mechanism, the compiler will, hopefully, be enabled to generate efficient code.

b) Type extension

Type extension consists in creating new sets of values and new operators to operate upon them, or in defining subsets of existing types and allowing the old operators to operate on these new values. Type extension actually modifies the data type tree. In the sequel we present the operators found in MADCAP-S which implement type extension:

type T under T' ...; or

type T above T' ...;

where the elipsis represent the rest of the declaration which involves the names, domains and ranges of the operators, the names and types of the abstract field selectors, names and ranges of travelers (generator functions), etc.

a) _Type_ T _under_ T' means that the new abstract type T is a subset of T' (fig. 2). All the operators valid on T' are also defined on T, and conversely, _new_ operators defined over T do not have the same meaning for T'. Recall that by definition operators in T which redefine equally named operators in T' must have the same meaning. The operators defined for T' can be used explicitly (are valid) on the new type (T), provided that

1) either T has the same representation as T' (default assumption), or

2) if T does not have the same representation as T' then the system must be able to perform the necessary conversions in the representation. For example, in the declaration

   type T _under_ T' _like_ T''

   (the _like_ T'' clause indicates that the representation to be used the one of T''. This clause is explained in [Melkanoff et alii 74]).

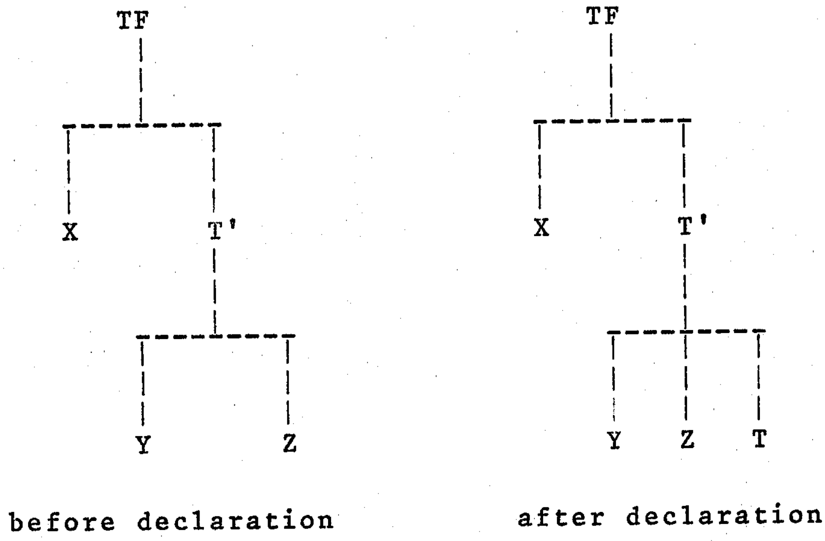
The programmer must provide the compiler with two mapping functions involving the types T and T', such as

mapping T' _in_ T = << _formal_ x in T' ... >> and

mapping T _in_ T' = << _formal_ x in T ... >>

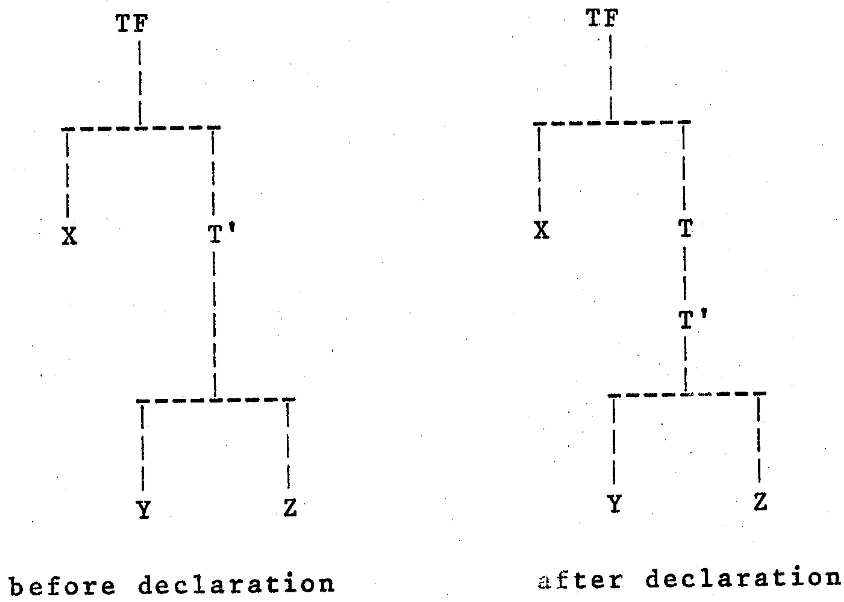
where, for example, W _in_ REAL denotes that W must be converted to the representation of the type real.

b) _type_ T _above_ T' places T above T' in the data type tree: it restructures the previously defined data type tree, and places the newly created abstract data type between T' and its "father", say TF (fig.3.).



Type T under T'

Fig. 2



Type T about T'

Fig. 3

All the operators valid on TF are now also valid on the new type T, and new operators defined on T that did not apply to T' are now also valid on T'. It is necessary, however, to consider the consequences of this declaration when several data representations are involved.

1) The default representation for T is the same as that of T'. In this case, the mapping from T to TF and vice-versa is the same as from T' to TF.

2) Given the declaration

   type T above T' like T'';

   T does not have the same representation as T'. In this case four mapping functions involving respectively the types T and T', and the types T and TF have to be defined. This is not necessary if the declaration has the form

   type T above T' like TF;

   since in this case the compiler can deduce the conversion routines to be used. (i.e. the ones involving T' and TF).

For example, a new type declaration could involve the creation of the complex numbers as a superset (above) the type REAL. At this point, the set of pure imaginary numbers could be defined as a subset (under) the complex numbers, yielding a type immaginary which is disjoint from the type real, although the representations of both may be the same.

2.4 - Example of a type declaration

We now consider the declaration of the abstract type stack and its implementation. The stack constructor takes a parameter of type type, and an integer, the first indicating what objects the stack will operate upon, and the second indicating the number of elements the stack can hold. The notation used is that of MADCAP-S.

Notice that TOP has been defined as a selector

(rather than an operator). This was done to ilustrate the concept of abstract selector as a pair of access functions, one for assignment and the second for selection.

```
 1) type STACK of (type(type1), INTEGER) under UNIVERSE =
 2) selector "TOP" <-> type (type1): << STACK >> ,
 3) operator "PUSH" : << STACK itself, type(type1) >>
 4)          "POP" : << STACK itself >>
 5)          "EMPTY" : << STACK -> BOOLEAN >>
 6) constructor STACK of(type(type1) , POSITIVE) =
 7) <<
 8) shared formal size in POSITIVE;
 9) type T = type(type1) <infin;
10) shared local x in T <- <NULL: size items>;
11) shared local top in NATURAL <- 0
12) >>;
13) selector TOP(STACK) -> type(type1) =
14) <<
15) formal a in STACK;
16) -> NULL if a.top = 0 or a.x [a.top] otherwise;
17) >>;
18) selector TOP(STACK) <- type(type1) =
19) <<
20) formal a in STACK, t in type(type1);
21) if a.top = 0 then fail else a.x [a.top] <- t;
22) >>;
23) operator PUSH (STACK itself, type(type1)) =
24) <<
```

```
25) formal (s in STACK, t in type(type1));
26) if s.top > s.size - 1 then fail fi;
27) s.top <- s.top + 1;
28) s.x [s.top] <- t;
29) >>;
30) operator POP (STACK itself) =
31) <<
32) formal s in STACK;
33) s.top <- max (s.top - 1, 0);
34) >>
35) operator EMPTY (STACK) -> BOOLEAN =
36) <<
37) formal (s in STACK)
38) -> true if s.top = 0 or false otherwise;
39) >>;
40) end type STACK;
```

3 - CONCLUSIONS

We have shown in this report a clean mechanism for automatically selecting coercions among abstract data types and their implementations. We have also shown how data type extensibility can be obtained without violating the general consistency rules.

REFERENCES

DENNIS, J.B. - Modularity. In: LECTURE NOTES IN ECONOMICS AND MATHEMATICAL SYSTEMS, 81. Berlin, Springer Verlag, 1973, p. 128-82.

LAUTERBACH, C.M.; MELKANOFF, M.; MOSZKOWSKY, B. Abstract data types in MADCAP-S. Los Angeles, University of California, Computer Science Department, 1975, Internal Memorandum nº 30.

MELKANOFF, M.; MOSZKOWSKY, B.; LAUTERBACH, C.M. Implementation of MADCAP-S on UCLA'S 360-91 IMLAC System. Los Angeles, University of California, Computer Science Dept., 1974, UCLA-ENG-7438.

VON STAA, A.; LUCENA, C.J. - On the implementation of data generality. Rio de Janeiro, Pontifícia Universidade Católica, Departamento de Informática, 1975. MCSCA, nº 5/75.