PUC

Series: Monographs in Computer Science
and Computer Applications
Nº 7/75

ON TYPE DEFINITION FACILITIES IN VERY HIGH LEVEL PROGRAMMING LANGUAGES

by

Sergio Carvalho

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 7/75

ON TYPE DEFINITION FACILITIES IN VERY HIGH LEVEL PROGRAMMING LANGUAGES *

by

Sergio Carvalho

Series Editor: Larry Kerschberg

November, 1975

* This work was supported in part by the Brazilian Government Agency FINEP under contract Nº 244/CT, and developed by the Information Systems Group, Departamento de Informática, PUC/RJ.

ABSTRACT:

Very high level languages are being developed to serve as a tool in the production of reliable software. One of the features of such languages in the ability programmers have of defining new data types. In this paper we define one such type, the domain, which, when coupled with a generalized FOR statement, can be a valuable aid to programming in high levels of abstraction.

KEY WORDS:

Very high level language, data types, FOR statements, program correctness .

RESUMO:

Linguagens de especificação estão sendo usadas na produção de sistemas de computação confiáveis. Uma das características destas linguagens é a possibilidade dada aos programadores de definir novos tipos de dados. Nesta monografia é definido o tipo "domain", que, quando usado em conjunto com um comando "FOR" generalizado, pode ser um instrumento valioso na programação em níveis de especificação.

PALAVRAS CHAVE:

Linguagem de especificação, tipo de dado, comando iterativo, correção de programas.

CONTENTS

1. INTRODUCTION AND PREVIEW

Recently it has been recognized that abstractions are the main tool available for disciplined problem solving [DIJ 72, HOA 72a, WIR 71a, WIR 74]. It has been pointed out that the human mind has a very limited capability for handling large problems. We can then use abstractions to decompose and "solve" the original problem; each abstraction becomes a new problem to be solved, until we are left with an organized sequence of steps, which can be easily coded in some programming language. This is the principle of Dijkstra's structured programming [DIJ 72] and Wirth's stepwise program development [WIR 71a]. It has been argued that programs resulting from the application of the technique above are more reliable, in an informal sense. Programmers can assure themselves that at any time during the solution process, the partial solution at hand is " correct ", and hence the final solution is also correct. We would like,however, to obtain formal proofs of program correctness. This requires the use of a set of formally defined tools which would allow us to encode abstactions (either functional or data abstractions, as recognized by Liskov [LIS 75]); each partial solution could then be formally proved correct.

To define this set of tools, two main (and interrelated) approaches are being taken. The extensibility approach [CHE 69, SCH 71] consists in allowing programmers to incorporate to a " base language " the features he feels are important to the solution of his particular problem. Thus, for instance, programmer defined data types and control structures would add to the power of the base language at hand. Standish [STA 75] recently surveyed this field.

On the other hand, the acknowledged inadequacy of most programming languages available today to serve as a tool for describing partial solutions has led many to develop " very high level " or " specification " languages [SCH 73, LIS 74, MOW 74, ASW 75] . Common features of those languages are, for instance, more powerful control structures and programmer defined data types. Although several such languages have been implemented, more research is still needed in this area. In particular, Zahn [ZAH 75] points out that " serious consideration should be given to extensions of the for statement to cater for progressions of values defined by more general successor functions ".

In this paper we propose one such extension. The results that follow were obtained during the definition of the specification language PEP [CLRS 75], which is now under implementation.

In section 2 the type "domain" is introduced and defined, and the operations allowed on domain values are explained. In section 3 an iterative statement which takes advantage of domain types is presented.

2. THE DOMAIN TYPE

2.1. MOTIVATION

From the data structuring point of view, Pascal [WIR 71b] can be considered a language of a higher level than, say, PL/1 [IBM 68]. Pascal users have the ability to define new types in their programs, and, in particular, a "scalar type" can be so introduced. A scalar type in Pascal in defined by enumerating the identifiers which are the type constants. For example, the type definition below introduces a new type "div 12", whose constants are identifiers for the divisors of 12.

```
type div 12 = (one, two, three, four, six, twelve);
```

A left to right ordering is assumed among the constants of a scalar type. The functions succ, pred and ord can be applied to constants of such types, yielding respectively the successor, the predecessor and the ordinal number (in the enumeration) of a given constant.

One drawback of this form of type definition is that the programmer has to list all his type constants. Clearly in some situations this would be cumbersome. Another problem is that the type constants are identifiers, and sometimes what the programmer wants is to define a new type whose values are a particular group of integers. For example, suppose the programmer has a way of defining as a type a certain group of prime numbers, say between 2 and 100. Such a facility would be very useful in a very high level language environment where the programmer could use, for instance, the following iterative statement:

```
for i in prime2-100 repeat S f-end
```

Such a statement could be used in some level of abstraction during the development of the solution of the problem at hand, thus relieving the programmer from the burden of, at this early stage, worrying about programming details.

2.2. DEFINITION

In this section we show a way in which unstructured types whose constants (integers) are generated according to powerful formation rules can be defined. Such types in the language PEP are called domains, and are syntactically defined as follows:

```
domain_type → DOMAIN (domain_former  { ; domain_former }* )
domain_former → domain_expression
                {, ident_values_spec }*
                [ : condition ]

domain_expression → integer_expression

ident_values_spec → identifier  IN  values

values → type_identifier
       | integer_subrange

integer_subrange → (initial_value.. final_value [, step ] )

initial_value → integer_expression
final_value   → integer_expression
step  →  integer_expression
```

The values of a domain are generated by a sequence of domain formers (similar to the set formers of SETL [SCH 73]). Each domain former consists of an integer expression, followed by a (possibly absent) list of specifications of identifier values, followed by a (possibly absent) condition. Each and every integer variable appearing in a domain expression must have the corresponding set of values stated in an identifier values specification. In the trivial case a domain expression is an integer number, there is no corresponding list of identifier values specification, and no condition. The possible values for variables in a domain expression can be stated either by a type identifier (in the sense of Pascal) corresponding to another (previously defined) domain type or by an integer subrange.

The values belonging to a domain type are those generated by all domain formers when the integer variables appearing in the corresponding domain expressions take on their corresponding values, provided:

(i) the conditions (whenever present) are "true" ;

(ii) the value just generated is not already in the domain.

The identifiers in the list of identifier values specification in a domain former are assumed to be nested, the rightmost one in the list being the most deeply nested one, and so on. Domain former values are ordered according to this nested priority, as in a nested group of iterations. Domain values are ordered from left to right, obeying each domain former internal order.

As an example, consider another way of representing the divisors of 12:

```
DOMAIN (X, X IN (1..12)   :   REM (12,X)= 0)
```

A condition in a domain former can be either a Boolean expression or a quantifier expression:

```
condition → Boolean_expression
          | quantifier_expression
quantifier_expression → quantifier  ident_values_spec
                      {, quantifier ident_values_spec }*
                                  : Boolean_expression

quantifier  →  [ ¬ ] ∃
            |  ∀
```

Note that the identifiers in quantifier expressions are not necessarily those appearing in the domain former expression, as is the case when the condition is a Boolean expression. The domain below contains all prime numbers between and including 1 and 100:

```
DOMAIN (1 ;
        X, X  IN (2..100)
            : ¬ ∃ Y  IN (2..X-1)
            :   REM  (X,Y) = 0)
```

2.3. OPERATIONS ON DOMAIN TYPE CONSTANTS

As mentioned before, domain values are ordered. Functions like Pascal's succ, pred and ord are thus naturally applicable to domains. However, a slight modification from the Pascal definitions is needed to avoid a possible ambiguity with respect to type constants. This can arise if a certain constant is present in two different domains, say the domain of the prime numbers from 1 to 100 and the domain of odd numbers from 1 to 99. To solve this problem we add to the function specification the name (type_identifier) of the desired domain. For example, if "prime" and "odd" are the names of the domains mentioned above, we have:

succ (7, prime) = 11

and succ (7, odd) = 9.

Note that since no value can appear in a domain more than once, no ambiguity can arise from the use of the functions above.

Domains were designed mainly to be used in conjunction with FOR statements (section 3), but since domain constants are integer values, integer arithmetic could be applied to them, provided:

(i) in an arithmetic expression, all variables are of the same domain type, and all constants belong to this particular domain;

(ii) the value of the arithmetic expression is a constant belonging to that particular domain. Note that this is run time type checking, and as such inefficient.

3. DOMAINS AND FOR STATEMENTS

With the help of domains, a class of FOR statements more general than that in most programming languages can be defined. This is so because control variables in those statements may range through values which do not necessarily form an arithmetic progression. In the language PEP, FOR statements are defined as follows:

```
for_statement → FOR control_variable IN range_spec
                REPEAT statement { statement }*  F_END

control_variable → identifier

range_spec → type_identifier
           | integer_subrange
```

The sequence of allowed values for the control variable is expressed by a range specification, which can be either the name of a previously defined domain or an integer subrange. As is usual [HOA 72b], the control variable can be assumed as being declared in the scope of the FOR statement, of the type determined by the range specification. As an example, consider the FOR statement below:

```
FOR  i  IN  prime  REPEAT  S  F_END
```

A useful extension can be easily added to the statement defined above. Suppose one must use a sequence of integer values which is present in a domain, but which does not constitute the whole domain (say the sequence of odd numbers between 51 and 99). The following syntax could be used instead of defining a new domain :

```
for_statement → FOR  control_variable
                IN range_spec

                      : Boolean_expression
                REPEAT statement { statement }*
                F_END
```

Then it would be possible to write

```
FOR i IN odd  :   i > 51 AND i < 99 REPEAT  S  F_END
```

4. CONCLUSIONS

Very high level languages are being used as a means towards the obtention of reliable software. One of the features in such languages is the ability programmers have of defining new types. In this paper it was shown a way in which programmers can construct new types (domains) whose values are generated according to powerful formation rules. Domains, coupled with a more general FOR statement, can be a very useful construct in a very high level language aimed at disciplined programming.

REFERENCES

ASW 75 ASHCROFT, E.A. & WADGE, W.W. Lucid - a formal system for writing and proving programs. Waterloo, Univ. of Waterloo, Department of Computer Science, 1975. CS - 01.

CLRS 75 CARVALHO, S. et alii. PEP: a language for provability, efficiency and portability. Rio de Janeiro, PUC, Departamento de Informática, 1975. ISG - 2.

CHE 69 CHEATHAM, T.E. Motivation for extensible languages. SIGPLAN Notices, 4 (8) : 45-8, Aug. 1969.

DIJ 72 DIJKSTRA, E.W. Notes on structured programming. In: DAHL,O.J. et alii. Structured programming. London, Academic Press, 1972. p.1 - 72.

HOA 72a HOARE, C.A.R. Notes on data structuring. In: DAHL, O.J. et alii. Structured programming. London, Academic Press, 1972. p. 83 - 174.

HOA 72b _____. A note on the FOR statement. BIT, 12 : 334-41, 1972.

IBM 68 IBM. System / 360 PL/1 reference manual. 1968. GC 28-8201 - 1.

LIS 74 LISKOV, B. A note on CLU. Cambridge, Mass., MIT Project MAC, 1974. Group Memo 112.

LIS 75 _____. Data types and program correctness. SIGPLAN Notices, 10 (7) : 16 - 7, July 1975.

MOW 74 MORRIS, J.B. & WELLS, M. Progress report on the language MADCAP - 6. Univ. of California, Los Alamos Laboratory, 1974.

SCH 71 SCHUMAN, S., ed. Proceedings of the International Symposium on extensible languages. SIGPLAN Notices, 6 (12) Dec. 1971.

SCH 73 SCHWARTZ, J.T. On programming: an interim report on the SETL project - installment I: generalities. New York, New York Univ., Computer Science Department, 1973.

STA 75 STANDISH, T.A. Extensibility in programming language design. SIGPLAN Notices, 10 (7) : 18 - 21, July 1975.

WIR 71a WIRTH, N. Program development by stepwise refinement. Commun. ACM, 14 (4) : 221-7, Apr. 1971.

WIR 71b _____. The programming language PASCAL. Acta Informatica, 1 : 35-63, 1971.

WIR 74 _____. On the composition of well structured programs. Computing Surveys 6 (4) : 247 - 59, Dec. 1974.

ZAH 75 ZAHN, C.T. Structured control in prgramming. SIGPLAN Notices, 10 (7) : 13 - 5, July 1975.