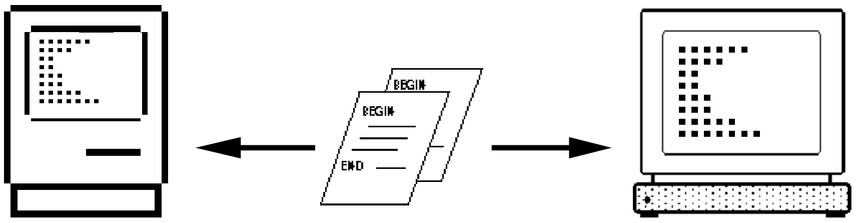
SYSTEMÖKOLOGIE ETHZ

SYSTEMS ECOLOGY ETHZ

Bericht / Report Nr. 20

Practical Considerations on Writing Portable Modula-2 Code

J. Thöny



January 1995

Eidgenössische Technische Hochschule Zürich ETHZ
Swiss Federal Institute of Technology Zurich
Departement für Umweltnaturwissenschaften / Department of Environmental Sciences
Institut für Terrestrische Ökologie / Institute of Terrestrial Ecology

Adresse des Autors / Addresse of the author:

J. Thöny
Systemökologie ETH Zürich
Institut für Terrestrische Ökologie
Grabenstrasse 3
CH-8952 Schlieren/Zürich
SWITZERLAND

e-mail: sysecol@ito.umnw.ethz.ch

Practical considerations on writing portable Modula-2 code

J. Thöny
Systems Ecology, ETHZ

1 Introduction

This paper discusses some aspects of portability of Modula-2 code and gives some rules which helps in writing portable code. It is meant for all persons which programs in Modula-2.

2 What is portability

Portability means, that a system written using the language Modula-2 is easy transferable and shares the same behaviour on every platform which conforms a defined standard.

We consider several aspects of portability; portability on Modula-2 code level, portability on compiler implementation level and portability on application level.

The standard for Modula-2 of this report is "Programming in Modula-2, Third, corrected edition" [PIM3] (Wirth, 1985) with some restrictions.

3 Portability on language and compiler implementation level

Modula-2 is not fully standardized in every aspect. Especially the handling of LONG types are not well defined. The various compiler implementations don't fully implement PIM3, or add several extensions to it.

3.1 Type transfer

Modula-2 performs a strong type checking. It isn't possible to use every variable or expression in every context. We have to distinguish between expression and assignment compatibility. Type transfers are easier in assignments than in expressions.

It is sometimes necessary to use a variable or expression in contexts which differ in their type. For Example:

```
VAR
  i : INTEGER;
  r : REAL
BEGIN
  r := 0.0;
  FOR i := 1 TO 100 DO
    r := r + FLOAT(i);   (* REAL and INTEGER inside one expression *)
  END;(*FOR*)
```

For such situations, Modula-2 offers type transfers. They can be safe or unsafe.

A safe type transfer respects the semantic of the source type. FLOAT(1) yields 1.0. Semantic checks, like range checks are performed. TRUNC(1.0e20) produces a range error.

An unsafe type transfer transfers the source type, without respecting its semantics, to the destination type.

Rules for more portable type transfers:

1 Use type transfers only as expressions. This prohibits the usage of type transfers as a VAR parameter in a procedure call.

2 Whenever possible, take advantage of assignment compatibility. Together with rule 1, 90% of the type transfers can be accomplished by this method in a fully portable way.

3 Whenever possible use the standard, safe type transfers: FLOAT, ORD, TRUNC.

4 Never use VAL, although it was defined in PIM3 as the inverse function of ORD. Since VAL has a different definition and semantics in the various compiler implementations we should not use it. Use T(x) or write your own transfer function instead.

3.2 Dynamic Memory

Allocate allows to connect every data structure to a pointer type. Therefore we have to use Allocate with great care.

Rule:

5 Always use the form: Allocate(pT, SIZE(T)) where pT is a pointer which points to type T.

Since Modula-2 doesn't' support arrays of variable size, this rule cannot be applied in every situation. If you simulate variable size arrays by partial allocated array types, never export a pointer to such arrays. Be aware of pitfalls. The biggest pitfall is p^ without index (p^[index]).

Example to illustrate this pitfall :

```
  TYPE
    RealArray = ARRAY [0..8000] OF REAL;
    RealArrayPtr = POINTER TO RealArray;

  VAR
    myOpenArray1, myOpenArray2 : RealArrayPtr;
    myOpenArraySize : INTEGER;

BEGIN
  myOpenArraySize := 100;
  Allocate(myOpenArray1, myOpenArraySize*SIZE(REAL)); (* =ARRAY[0..99]OF REAL *)
  Allocate(myOpenArray2, myOpenArraySize*SIZE(REAL)); (* =ARRAY[0..99]OF REAL *)
  myOpenArray1^ := myOpenArray2^; (* This copies 8001 REALS ! *)
END Test.
```

This example will be executed on a machine without protected memory, but it will produce garbage. It will copy 8001 REALS to the memory location starting at ADDRESS(myOpenArray1), where only 100 REAL are reserved. It is likely that this will overwrite other data structures. On machines with protected memory, you will get a protection error.

3.3 Module SYSTEM

Whenever possible avoid imports from SYSTEM. This is not always possible for ADDRESS or ADR.

Rule:

6 Whenever possible avoid imports from SYSTEM.

3.4 Type size, Type range, memory alignment

It is not the responsibility of a programmer to know about the memory size of a type. It is wrong to assume that SIZE(LONGINT) > SIZE(INTEGER) or that MAX(LONGINT) > MAX(INTEGER).

It is the compilers responsibility to perform address arithmetic. If you implement your own address arithmetic, your code may run on the computer and compiler where you have developed it. It is likely that it will fail on another computer or compiler. The reason is, that you never know the memory alignment and the order of variables on every potential target host.

Rules:

7 Never make any implicit or explicit assumption about the size or the range of a type. Let the compiler perform address arithmetic and use the MIN/MAX functions if you have to use range information.

8 Don't perform your own address arithmetic.

9 Never use variant records to map one data structure into an other. If you have to do it, you have to perform an integrity check on the data structure.

3.5 INTEGER - CARDINAL

INTEGER and CARDINAL are assignment compatible, but not expression compatible. Some compilers implementations return INTEGER, some CARDINAL on HIGH and ORD.

Rule:

10 Use HIGH and ORD only as the sole factor within an expression which is used for an assignment or as a procedure's value parameter.

Example:

```
PROCEDURE DoIt(str : ARRAY OF CHAR);
  VAR
    i, highStr : INTEGER
BEGIN
  highStr := HIGH(str);
  FOR i := 0 TO highStr DO (* and not FOR i := 0 TO HIGH(str) DO *)
  ...
```

3.6 LONG types

The LONG types were introduced late (Wirth, 1988). The relationship between the LONG types and other types were never officially defined. In the mean-time, most compiler builders have introduced their own LONG extension. This resulted in portability problems.

Rules:

11 Define LONG constants with type transfers. i.e. LongOne = LONGINT(1). Never use literal constants inside your code. e.g. IF long < LongOne THEN and not IF long < 1D THEN.

12 TRUNCD, LTRUNC, LONGTRUNC rsp. FLOATD, LFLOAT, LONGFLOAT are used by the various compiler implementation for the same standard type transfer function. Avoid using them. If this is not possible, try to use assignment compatibility or encapsulate the transfer function inside a function.

3.7 Module file names

The naming conventions of module file names are compiler and platform dependent. If we consider Unix, Mac, and MS-DOS platforms, we can define the following rules:

13 The file name has to be in the form: ModuleName.Extension. The extension identifies whether it is an (implementation) module or an definition module. ModuleName must match the case of the name of the module.

14 The module names of an application have to be distinguishable inside the first eight, capitalised characters.

15 The extension cannot be used for naming, i.e. MyModule.mod2.

3.8 ASCII character set

Most compilers allows the use of non ASCII characters (CHAR(128)-CHAR(255)). Since they are highly platform dependent, it is recommended to use only ASCII characters.

Rules:

16 Use only printable characters in the range CHAR(32) to CHAR(126).

17 Use the EOL character supported by the corresponding file and terminal library. Never use CR and/or LF.

3.9 Conditional Compilation

It is impossible to write portable code in any circumstances. The best work around would be to use conditional compilation. Since the syntax of conditional compilation is highly compiler depended, it is not possible to use these technique. As an alternative, we can use several versions of modules. To prevent an unnecessary tree of versions we define the following rules:

18 These rules only apply if the differences between the versions are small. If the modules differ to a large extent, we have to maintain separate versions for every desired compiler/platform.

19 Declare a master module. This module unites all versions. They are all commented out, but the one of the main platform. By changing the comments, you can generate any platform-specific version. Never edit the derived versions, use only the master module instead.

20 Every alternating part starts with the comment (* IF VERSION_NAME *) and is closed by (* ENDIF VERSION_NAME *). (This rule is not fully defined yet)

21 All versions are listed inside a comment at the beginning of a module.

3.10 Modula-2 Libraries

There exists no standard Modula-2 library. Most compilers adapt more or less the libraries mentioned in PIM3. Therefore, a fully portable library or application must not directly rely on any of these library modules. It is recommended to define the smallest possible interface to the operating system and build the own library on top of this. An example are the modules SysDep and Portab of RASS (Thoeny *et al.*, 1994).

4 Portability on application level

Even when we achieve to write fully portable source code, an application written in Modula-2 doesn't need to be fully portable. An application may differ in the behaviour while executed on different platforms.

4.1 File handling

An application consists not only of the program itself, but often also of auxiliary data files. The file handling has also to be portable to achieve a portability on the application level. In addition to the rules 16 and 17 we can define the following rules:

22 Filenames must not exceed eight characters. In addition to that, an extension of maximum three characters are allowed.

23 Filenames are case sensitive, but don't depend on the case. The reason is, that some platforms are case sensitive (Unix) and some not (Mac).

24 Never hard-code a path name. If you need path names, you should use a mechanism like described in "Modula-2 Libraries".

25 Determine the end of file using the content of a file. This is more secure than relaying on the EOF mechanism.

4.2 Linking an application

It is impossible to achieve a fully portable solution of the linking process. A partial portability may be achieved by using make files. The make tools where introduced by the Unix systems. Most platforms supply such a tool. . They are more or less compatible. Use the macro facility of make to define the tools (compilers, linker etc.) and their options.

5 Literature

Thoeny, J., Fischlin, A. & Gyalistras, D., 1994. *Introducing RASS - The RAMSES Simulation Server*. Internal Report # 21, Systems Ecology Group, ETHZ, .

Wirth, N., 1985 (Third, corrected edition ed.). *Programming in Modula-2*. (Texts and monographs in computer science), Gries, D., (ed.);Springer:

Wirth, N., 1988 (Fourth, corrected edition ed.). *Programming in Modula-2*. (Texts and monographs in computer science), Gries, D., (ed.);Springer:

BERICHTE DER FACHGRUPPE SYSTEMÖKOLOGIE
SYSTEMS ECOLOGY REPORTS
ETH ZÜRICH

[*] Out of print