Technical Report

CMU/SEI-87-TR-15
ESD-TR-87-116

The Use of Representation Clauses and Implementation-Dependent Features in Ada:

IIA. Evaluation Questions

B. Craig Meyers
Andrea L. Cappellini

July 1987

Technical Report
CMU/SEI-87-TR-15
ESD/TR-87-116
July 1987

The Use of Representation Clauses and Implementation-Dependent Features in Ada: IIA. Evaluation Questions

B. Craig Meyers
Andrea L. Cappellini
Ada Embedded Systems Testbed Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

The Use of Representation Clauses and Implementation-Dependent Features in Ada: IIA. Evaluation Questions

Abstract: This report is the second in a series on the use of representation clauses and implementation-dependent features in Ada. It is the purpose of this document to specify a set of questions relevant to the assessment of the support of representation clauses and implementation-dependent features provided by an Ada compiler. The questions identified are categorized according to functionality and address both qualitative and quantitative aspects.

1. Introduction

The Ada language was developed as a general-purpose language with specific application to mission-critical systems for the Department of Defense (DoD). The language is now mandated for use in real-time, mission-critical systems, as specified in reference [1].

In spite of the attempt to define Ada as a general-purpose language, a need to support implementation-dependent functionality remained. This amounted to the language providing for a coupling to the specific underlying architecture. Much of this support is defined in terms of representation clauses and implementation-dependent features which are discussed in Chapter 13 of the *Reference Manual for the Ada Programming Language* [2].

The support for a particular type of representation clause is an implementation-dependent issue. In other words, the degree to which a type of representation clause is supported, if at all, is left to the discretion of a compiler developer. In view of the fact that many systems may need to implement the functionality supported through representation clauses, it is clearly advantageous to assess different compilers from this perspective.

It is the purpose of this document to specify a set of issues relevant to the evaluation of the support of representation clauses and implementation-dependent features for a given compiler. The questions identified are categorized according to functionality and address both qualitative and quantitative aspects.

This report is one of a series dealing with the use of representation clauses and implementation-dependent features in Ada. The first volume in the series, reference [3], provides an overview of the use of representation clauses and implementation-dependent features. A number of case study examples, drawn from existing systems, illustrate the application of the machine-dependent characteristics of Ada. The following volume in the series deals with methodology and experimental procedures for the assessment of representation clauses and implementation-dependent features [4]. A qualitative examination of the VAX Ada compiler, based on the framework developed here, has been reported [5].

This report has been prepared by the Ada Embedded Systems Testbed Project at the Software Engineering Institute (SEI). The SEI is a federally funded research and development center (FFRDC) sponsored by the Department of Defense and established and operated by Carnegie Mellon University. This report is based on work performed by the authors while they were on sabbatical leave at the SEI.

2. Discussion

A characteristic of many mission-critical systems is that they manifest a need to implement functional capabilities which are directly related to the underlying machine architecture. This includes, for example, the ability to process "packed" data structures, access data of specific lengths, and format internal data representations in machine-dependent terms. Another example is provided by requirements to conform to constraints imposed by external hardware devices. Thus, some external device may directly map data into memory which is available at some fixed address.

In the development of the Ada language, the developers recognized that there was a need to provide a coupling between the language and the underlying machine architecture. This coupling is affected through the use of representation clauses and other implementation-dependent features in Ada. For example, pragma PACK may be specified which minimizes storage allocation for arrays and records. There are also representation clauses for specifying the maximum amount of storage to be allocated for objects of a particular type, specifying the values associated with enumerated data types, and defining the precise layout of data within a record structure. Additionally, representation clauses may be used to allow a program to access a specific address. Representation clauses also exist which apply to information relative to tasking; however, discussion of this latter type of representation clause is beyond the scope of this report.

We emphasize that the support for representation clauses and implementation-dependent features is left to the developer of a particular compiler. The language does not require that a compiler provide support for all of the representation clauses or implementation-dependent features which are defined as part of the language. This has several implications, including the following:

1. The expected support for representation clauses may vary widely across different compilers. Some compilers may provide a wide range of support, while others may provide little support.

2. The manner of implementation for machine-dependent features may also be expected to vary between different compilers. Note that it is in the area of representation clauses where the ability to couple to the underlying architecture may be most noticeable. Therefore, some compilers may provide more support for these language elements than others; this may be an inherent reflection of the underlying machine. As an example, a virtual machine architecture may not provide the support for a representation clause to access a specific address.

The preceding clearly has an impact on application designers of mission-critical systems who are required to use machine-dependent features. This impact may be manifest in either or both of the following ways:

1. It is required that the application developer be clearly aware of the support provided for representation clauses and implementation-dependent features. It is important for the developer to understand the compiler support, as well as the limitations of a particular compiler. In fact, one may go so far as to say that the choice of a particular compiler may, in some cases, depend on the support provided for representation clauses and implementation-dependent features.

2. It is expected that the application developer will need to examine alternative ap-

proaches to the use of representation clauses for a particular problem. That is, if the chosen compiler does not implement some language-defined aspect of representation clauses, and if the associated functional capability is needed, the developer is forced to consider alternative means in finding a solution to the problem.

It is recognized, therefore, that a need exists for the assessment of support provided for representation clauses and implementation-dependent features by various compilers. As implied above, this assessment is especially needed since the language does not require any compiler to implement the support for representation clauses and implementation-dependent features as defined in reference [2]. The ability to assess a set of compilers can be of benefit to those involved in the development of mission-critical systems.

As one element of an assessment process, those issues relevant to the assessment of representation clauses and implementation-dependent features needed to be defined. These issues have been identified in terms of a set of questions which appear in Appendix I. The questions specified in Appendix I have been grouped in terms of functional support relating to the various aspects of representation clauses and implementation-dependent features defined by the language. The areas are the following:

1. general
2. data types supported
3. pragma PACK
4. length clauses
5. enumeration representation clauses
6. record representation clauses
7. address clauses
8. data conversion and assignment
9. representation attributes
10. miscellaneous

The questions listed in Appendix I may be grouped into two basic categories. On the one hand, there are questions which are principally qualitative in nature. Characteristically, this type of question may be answered by reference to the documentation provided by a particular compiler. In some cases, these questions may involve a limited analysis of generated code as well. On the other hand, there are questions which are essentially quantitative in nature. For this class of questions, a complete evaluation is expected to require a possibly large amount of experimentation. It is this latter type of question which provides details of execution timing and code size, for example. It is noted that in the context of this work, a broad interpretation of quantitative is taken. That is, not only are typical performance issues included, but so too are evaluations of the manner in which a particular feature is implemented.

The questions delineated in Appendix I refer to the assessment of a particular compiler; they are not, for example, general issues dealing with a particular assessment methodology. However, one should equally recognize that the development of a particular methodology also has certain associated issues. Discussions along the preceding lines are provided in another document in this series [4].

The questions specified in Appendix I form a basic starting point for the application of experimental procedures to assess the support of representation clauses and implementation-dependent features by any given compiler. In some sense, the questions appearing in Appendix I are an implicit set of guidelines which an experimental procedure may be expected to follow. The details of an appropriate methodology, and the associated experimental procedures, are found in reference [4].

The questions appearing in Appendix I apply to the larger issue of compiler assessment. It is to be recognized, however, that either qualitative or quantitative assessments may be conducted. A start has been made in this direction. Thus, reference [5] reports on a qualitative assessment for a particular compiler, and it is expected that others will follow. The qualitative assessments are based on a subset of the questions appearing in Appendix I. The performance of quantitative assessments is a considerably larger problem, although a methodology has been formulated in some detail and appears in reference [4].

It has been the purpose of this document to delineate those issues relevant to the assessment of compilers from the perspective of support provided for representation clauses and implementation-dependent features. This document is necessarily short to maintain focus on the issues relevant to assessment. Other volumes in the series incorporate the issues delineated here into the larger framework of a methodology.

3. Summary

Although the Ada language was developed as a general-purpose language with application to mission-critical systems, a need to provide a coupling between the language and the underlying machine was recognized. This coupling is accomplished, in part, through the use of representation clauses and implementation-dependent features. However, the amount of support provided for these language features is left to the compiler developer.

The recognized need for representation clauses and implementation-dependent features by many systems, as well as the expected variability of support among compilers, motivates the need for assessment. This document delineates those issues which are believed germane to the assessment of support provided by a given compiler for representation clauses and implementation-dependent features. The issues are qualitative, as well as quantitative. This document, in addition to a discussion of experimental procedures [4], provides a framework for conducting assessments.

References

1. *DoD Instruction 5000.31*, July 1986. *DoD Directive 3405.2*, March 30, 1987.

2. *Reference Manual for the Ada Programming Language*, Department of Defense MIL-STD-1815, 1983.

3. B. Craig Meyers and Andrea L. Cappellini, *The Use of Representation Clauses and Implementation-Dependent Features in Ada: I. Overview*, CMU/SEI-87-TR-14, ESD-TR-87-115, July 1987.

4. B. Craig Meyers and Andrea L. Cappellini, *The Use of Representation Clauses and Implementation-Dependent Features in Ada: IIB. Experimental Procedures,* CMU/SEI-87-TR-18, ESD-TR-87-126, July 1987.

5. B. Craig Meyers and Andrea L. Cappellini, *The Use of Representation Clauses and Implementation-Dependent Features in Ada: IIIA. Qualitative Results for VAX Ada, Version 1.3,* CMU/SEI-87-TR-17, ESD-TR-87-118, July 1987.

Appendix I: Questions Relevant to the Use of Representation Clauses and Implementation-Dependent Features

In the following, we list questions which are pertinent to the considered use of representation clauses and implementation-dependent features. These questions apply both to the support provided by the Ada compiler as well as the run-time environment. The questions below have been grouped into categories.

A. General:

1. What is the basic unit of SYSTEM.STORAGE_UNIT? (This is useful when defining record layouts.)

2. What is the ordering of allocation for storage units? Is it left-to-right or right-to-left with respect to each other? How are bits numbered within storage units? Is it left-to-right or right-to-left? Does the numbering always begin with zero? (This is useful when defining record layouts and verifying the actual allocation of record layouts.)

3. It is also appropriate to consider the role of the underlying architecture, particularly regarding data conversions from representation clauses to other formats. Does the machine include instructions for inserting and extracting bit-length fields? What are the restrictions on the use of such instructions (for example, what is the maximum field size to which an instruction may be applied)?

4. Is pragma OPTIMIZE supported? If so, are there any restrictions on its use?

5. The use of representation clauses may present unusual problems throughout design and coding. What facilities exist for verifying results when representation clauses are used? We are speaking here of the debugger; thus, are there restrictions on the use of the debugger when representation clauses are used?

6. Does the compiler provide a load map that contains sufficient details to identify the location of quantities specified using representation clauses?

7. Are there any restrictions on representation clauses?

8. Compiler implementors currently have the option as to what degree, if any, the features in Chapter 13 of the *Reference Manual for the Ada Programming Language* will be supported. It is conceivable that upgraded versions of an implementation will enhance the support originally available for such features as representation clauses. How is the documentation upgraded? Is it by release notes or page changes? The manner in which this is accomplished can affect the ease with which documentation can be used.

B. Data Types Supported:

1. What are the basic implementations of integer types?

2. What are the basic implementations of fixed-point types?

3. What are the basic implementations of floating-point types?

4. Does the compiler provide predefined, unsigned data types? If not, is it permissible for a user to define these types? For example, is the following legal:

```
type Unsigned_Small_Int is range 0 .. 7;
for Unsigned_Small_Int'SIZE use 3;
```

C. Pragma PACK:

1. Does the compiler support the use of pragma PACK?

2. What restrictions are placed on the use of pragma PACK? For example, are there certain types that may or may not be packed?

D. Length Clauses:

1. Does the compiler support the use of length clauses? What are the restrictions on their use?

2. Are there restrictions on the use of the SIZE attribute designator in a length clause?

3. Are there restrictions on the use of the STORAGE_SIZE attribute designator in a length clause?

4. Are there restrictions on the use of the SMALL attribute designator in a length clause?

5. When using a SIZE attribute designator in a length clause, the *Reference Manual for the Ada Programming Language* states that the value of the expression specifies an upper bound for the number of bits to be allocated. The presence of a range constraint or the use of a predefined type implicitly defines the maximum number of bits required to allocate objects. If extra bits are specified in the length clause, are these extra bits allocated by the compiler?

6. Suppose a type, with associated length clause, has been specified storage where the number of bits is not sufficient to store the specified range of values. For example, suppose an integer type with range 10 .. 13 is defined, and three bits of storage are allocated for that type. Is an error generated for this case? If no error is generated by the compiler, how is a case such as this treated?

7. What impact does the length clause have on the packing algorithm of composite types?

8. What is the effect of pragma OPTIMIZE (TIME) on storage allocation when length clauses are used?

9. What is the effect of pragma OPTIMIZE (SPACE) on storage allocation when length clauses are used?

10. What is the effect of pragma PACK on storage allocation when length clauses are used?

E. Enumeration Representation Clauses:

1. Does the compiler support the use of enumeration representation clauses? What are the restrictions on their use?

2. Consider an enumeration type and associated enumeration representation clause where the enumerated values specified are not contiguous integers, such as:

   ```
   type Name is (Name_1, Name_2, Name_3, Name_4);
   for Name use
     ( Name_1 => 1, Name_2 => 5, Name_3 => 12, Name_4 => 163 );
   ```

   The enumeration type may not be efficiently implemented because of the noncontiguous nature of the integers specified in the enumeration representation clause, illustrated above. Hence, how are enumeration types represented internally, particularly in the case where enumeration clauses are specified with noncontiguous values?

3. What is the effect of pragma PACK on storage allocation when enumeration representation clauses are used?

4. What is the effect of pragma OPTIMIZE (TIME) on storage allocation when enumeration representation clauses are used?

5. What is the effect of pragma OPTIMIZE (SPACE) on storage allocation when enumeration representation clauses are used?

F. Record Representation Clauses:

1. Does the compiler support the use of record representation clauses? What are the restrictions on their use?

2. What are the restrictions on the use of an alignment clause in a record representation clause?

3. What are the restrictions on the use of component clauses in a record representation clause?

4. Are there restrictions on the overlap of record components with respect to the basic machine storage unit? For example, if a machine has a SYSTEM.STORAGE_UNIT equal to 16 bits, is it permitted to have components of a record that are larger than this value?

5. Consider the case when a record is specified with a record representation clause. Where is a record component placed that has no associated component clause?

6. What is the effect of pragma OPTIMIZE (TIME) on storage allocation when record representation clauses are used?

7. What is the effect of pragma OPTIMIZE (SPACE) on storage allocation when record representation clauses are used?

8. What is the effect of pragma PACK on storage allocation when record representation clauses are used?

G. Address Clauses:

1. Does the compiler support the use of address clauses? What are the restrictions on their use?

2. What is the type SYSTEM.ADDRESS?

3. What is the effect of pragma OPTIMIZE (TIME) on storage allocation when address clauses are used?

4. What is the effect of pragma OPTIMIZE (SPACE) on storage allocation when address clauses are used?

5. Does the compiler enforce strong typing in the presence of address clauses? For example, is the following recognized as erroneous by the compiler:

```
type T_1 is range 0 .. 100;
O_1 : T_1;
for O_1 use at 16#1000#;

type T_2 is digits 2 range 0.0 .. 100.0;
O_2 : T_2;
for O_2 use at 16#1000#;
```

6. Does the compiler or linker recognize potential conflicts when address clauses are used? For example, suppose an address clause is present that references some address, say X. Assume that the address X is such that it lies within the address space of generated code. How is this case treated by the compiler and/or linker?

H. Data Conversion and Assignment:

1. How is conversion accomplished between values of a type specified by the default representation and a type specified with a representation clause? (This refers to the use of a ***new*** (derived) type that is defined in terms of a representation clause.)

2. For conversions between objects of different types, does the compiler produce in-line code or generate a call to a library routine to accomplish the conversion?

3. Is support of the generic function UNCHECKED_CONVERSION provided?

4. Are there any restrictions on the use of UNCHECKED_CONVERSION? For example, are there any restrictions on the source and target types for UNCHECKED_CONVERSION? Do they have to be of the same size?

I. Representation Attributes:

1. What are the restrictions on the use of the ’ADDRESS representation attribute? How does the compiler interpret the use of this attribute?

2. What are the restrictions on the use of the ’SIZE representation attribute? How does the compiler interpret the use of this attribute?

3. What are the restrictions on the use of the ’POSITION representation attribute for a record component?

4. What are the restrictions on the use of the ’FIRST_BIT representation attribute for a record component?

5. What are the restrictions on the use of the ’LAST_BIT representation attribute for a record component?

6. What is the effect of pragma OPTIMIZE (TIME) on the values of the representation attributes?

7. What is the effect of pragma OPTIMIZE (SPACE) on the values of the representation attributes?

J. Miscellaneous:

1. Suppose an object has been allocated storage where the number of bits is not sufficient to store the specified range of values. For example, suppose an object has been allocated three bits of storage, but is specified to be in the range 10 through 13. Is an error generated for this case? If no error is generated by the compiler, how is a case such as this treated?

2. Does the compiler support the use of pragma SUPPRESS?

3. What restrictions are placed on the use of pragma SUPPRESS? For example, can every check be suppressed?

4. Is pragma STORAGE_UNIT supported? If so, are there any restrictions on the argument?

5. Is pragma INTERFACE supported? If so, are there any restrictions on the allowable forms and places of parameters and calls?

6. Is pragma SHARED supported? If so, are there any restrictions on its use?

7. Are there other implementation-dependent features supported such as pragmas or attributes?

Table of Contents