ADA* FLOATING-POINT ARITHMETIC AS A BASIS FOR PORTABLE NUMERICAL SOFTWARE

PETER J L WALLIS

SCHOOL OF MATHEMATICS, UNIVERSITY OF BATH

CLAVERTON DOWN, BATH, U.K.

Abstract

Ada supports two different schemes for floating-point arithmetic portability - one based on the use of the underlying machine arithmetic and the other based on the 'model arithmetic' that the underlying machine supports. Features of both schemes are explained in the context of their suitability as bases for the production of portable numerical software.

Introduction

Although Ada is a fully-standardised high-level language for which portability of programs was a requirement, writing portable programs in Ada necessarily requires great care [1]. The need for such care is not unique to Ada, but seems to be inherent in any realistic contemporary high-level language [2].

The Ada design incorporates a serious attempt to facilitate the production of portable numerical software by the provision of suitable language features. The purpose of the present paper is to examine these features to clarify the ideas behind them and the difficulties inherent in their use. After explaining how environment parameters are provided in Ada, we present two schemes for Ada floating-point portability in detail as alternative bases for the production of portable numerical software. Finally, their relative merits are discussed.

Versions of Ada

Ada evolved through a number of versions to its recent ANSI standardisation. This paper is based on ANSI Standard Ada [3], while the November 1980 version of the language [4] was used as a basis in previous discussions of Ada arithmetic [5,6,7,8,9]. The only language changes relevant here concern the introduction of safe numbers ([4] Section 3.5.6) and a small resultant change in the handling of range constraints for floating-point types ([4] Section 3.5.7); see [7] for the previous position on these

Environment Parameters

Ada arithmetic portability ultimately relies on the well-used idea of allowing the programmer access to 'environment parameters'. There is nothing new about this - see [9] for a comprehensive survey of work relating to the parameterisation of floating-point arithmetic. However, the mechanisms provided within the Ada language for furnishing environment parameter values do deserve brief comment.

*Ada is a registered trademark of the US Government, Ada Joint Program Office.

Parameterisation in Ada

Environment parameters in Ada are supplied to programs in two different ways - in the body of package STANDARD and as attributes of types. This has the advantage that special syntax is used for them. Thus (unless their values are assigned to variables, which would be considered bad Ada programming style [10]), their use in the program is conspicuous to the reader and can lead to compiler dead-code elimination since the values of all quantities concerned are known at compile-time. This means that versions of a program tailored to different object machines could in principle be compiled from the same source text, eliminating a messy editing job sometimes needed for current numerical libraries.

Floating-Point Portability

There are two quite different methods of arranging a floating-point portability scheme based on environment parameters. The first, which may be called the 'machine-dependent' method, furnishes parameters of the arithmetic actually provided and leaves the programmer to tailor his work to this.

The second, which we call the 'model' method, relies on the provision of parameters characterising a very conservative use of the arithmetic on any particular machine in such a way that the programmer is encouraged to rely only upon characteristics common to all target machines. The provision and use of Ada attributes corresponding to the 'machine-dependent' method and to the 'model' method are now separately discussed.

'Machine-Dependent' Method

Floating-point portability in Ada can be arranged using only the machine-dependent characteristics of floating-point arithmetic. This style of Ada portable programming is analogous to Fortran portable numerical programming with the added possibility of exploiting dead-code elimination. For this purpose, access is provided (using package STANDARD) to characterisation of the available floating-point

CH1892-9/83/0000/0079$01.00 © 1983 IEEE

arithmetic. Additionally, any Ada floating-point type F has the machine-dependent attributes listed in Table 1.

Table 1. Machine-Dependent Attributes

```
F'MACHINE_RADIX
F'MACHINE_MANTISSA
F'MACHINE_EMAX
F'MACHINE_EMIN
F'MACHINE_ROUNDS
F'MACHINE_OVERFLOWS
```

Comparing the contents of Table 1 with other proposed parameterisations such as the IFIP WG 2.5 parameter list discussed in [9], it may be concluded that Ada attributes provide a plausible basis for 'machine-dependent' floating-point portability. However, the machine-dependent attributes given in Table 1 are of limited use in portable programming of this type. Availability of F'MACHINE_RADIX means that one can take advantage of wobbling precision, but the rounding action is not well specified. In practice these deficiencies are not serious, since compiler dead-code elimination based on the use of SYSTEM_NAME from package SYSTEM is the way machine dependencies at this level of detail are most likely to be handled.

Manipulative Functions

Some portable numerical software uses algorithms that treat the exponent and mantissa of a floatin-point number separately. Brown and Feldman [11], who discuss this question and illustrate it with sample algorithms, conclude that this is best done using six basic functions. These six functions, which have become known as the Brown/Feldman functions, are also discussed (and are related to similar X3J3 and IEEE proposals) in [9]. The bodies of the Brown/Feldman Functions cannot be written in a portable way using the attributes in Table 1 because none of these concern the way a floating-point number is actually represented; implementations of the Brown/Feldman Functions for the machine-dependent floating-point types would have to be written in assembly code for each target machine.

'Model' Method

As explained already, the 'model' method of Ada floating-point portability is based on encouraging programmers to use floating-point arithmetic capabilities very conservatively in the interests of guaranteed portability. The approach taken uses a model of floating-point arithmetic that is based on that of W S Brown [12].

The idea of Brown's model is to provide a formal basis for 'worst case' error analysis of portable floating-point software. Thus the model necessarily provides a pessimistic view of numerical behaviour on any particular machine. Some attempts have been made in Ada to reduce the pessimism of the model at the cost of excluding certain machines. Ada's use of supported division and a binary machine base can be seen in this light; supported division improves expected error behaviour on division and a binary base means that certain non-integral literals are model numbers, but both these assumptions exclude certain machines [6,7].

General Description

The idea of the Ada model method for floating-point portability is that the programmer specifies the precision (in decimal digits) and (optionally) the range of each floating-point quantity, and the compiler furnishes a machine-dependent type providing at least the requested range and precision. It is thus necessary for the compiler writer to know the Ada model characteristics of the floating-point arithmetic provided by the target machine hardware; the way these model characteristics may be found is explained in [7,8]. Use of the model brings with it certain guarantees about the errors inherent in the floating-point arithmetic.

Characteristics of the model arithmetic requested and provided are available as attributes. There are two sets of these - model attributes and safe attributes - which are now separately explained.

Model Attributes These attributes relate to the properties of the model arithmetic guaranteed to be the same on all machines. There are six of these as given in Table 2 for a floating-point type F, but their values are all calculated ([3] Section 3.5.8) from quantities specified by the programmer in a type definition of the form

<u>type</u> F <u>is</u> <u>digits</u> D [<u>range</u> L..R]

where D, L and R are supplied by the programmer and the range specification is optional.

Table 2. 'Model' Attributes

```
F'DIGITS
F'MANTISSA
F'EPSILON
F'EMAX
F'SMALL
F'LARGE
```

Safe Attributes As explained in [7], a consequence of having an Ada model for arithmetic that may be parameterised only by the required precision is that some extra model range to that specified by the model attributes may be available on particular hardware. In ANSI Ada [3], this extra range is made available by the three 'safe' attributes given in Table 3 - these correspond to the three attributes in Table 2 whose values are affected by the extra range.

Table 3. Safe Attributes

```
F'SAFE_EMAX
F'SAFE_SMALL
F'SAFE_LARGE
```

Comparison of Safe and Model Attributes

It remains to discuss the relative merits of the use of Safe and Model Attributes in portability using the 'model' method. Recall that Model Attributes are those which follow from the precision specified, while Safe Attributes correspond to all the model range and provision provided by the machine type allocated as a result of the precision specification. It follows that Safe Attributes should be used for full exploitation of the model arithmetic as supported by a given machine while the use of Model Attributes corresponds to making minimal demands on machine arithmetic consistent with the required portable precision. Thus it would appear that the attributes in Table 2 that correspond to those in Table 3 have little practical value.

Difficulties with the 'Model' Method

There are two different kinds of difficulty with the use of the Ada model as a basis for numerical software portability. These have been elegantly explained by Cody [9] in arguing against such use of the Ada model*. First, the model is too rigidly defined: not only are some machines excluded, but it gives an unreasonably pessimistic view of the arithmetic on many floating-point units because of the attempt at generality. Secondly, and more seriously, the model is exceedingly difficult to use properly because machine-dependent effects may arise through inadvertent use of values which are outside the range of safe numbers yet do not cause the raising of an exception. The formulation of Brown's model and its incorporation within Ada are substantial, praiseworthy efforts, but it seems most unlikely that the Ada model will find much favour among those using the language for nontrivial portable numerical software.

Apart from these particular difficulties with the Ada model, it may also be convincingly argued that the idea of using a (necessarily pessimistic) model as a basis for floating-point portability is wrong in principle. It is bound to lead to inferior performance on particular machines - for example one is prevented from taking advantage of such features as known bias in rounding and extra precision associated with a hexadecimal base ('wobbling precision'). The resulting error bounds can, in a computation of any complexity, rapidly become so wide as to be meaningless. Further, use of the model for error analysis can be seen as suggestive of a way of thinking about error behaviour that is quite inappropriate for some hardware explicitly designed to have good error behaviour. However, this does not detract from the fact that the Brown model is the best that can be done in designing a universal floating-point model that can be fitted retrospectively, so to speak, to existing floating-point hardware - only the appropriateness of rigorous model arithmetic as a basis for numerical software portability is at issue here.

Conclusion

Ada is unique among contemporary high-level languages in giving serious consideration to the perplexing problems of floating-point portability. As such, its support of floating-point arithmetic warrants sympathetic study. It seems reasonable to conclude, albeit with some regret, that the shortcomings of the Ada model outweigh its advantages for the production of serious portable numerical software; the 'machine-dependent' method is likely to continue to be used for Ada much as for contemporary portable Fortran libraries. However, the Ada design does address the issues in floating-point portability with an unwonted seriousness, and as such provides a stimulating and controversial precedent for the design of the arithmetic features of future languages.

*Cody refers to a version of Ada [4] without safe attributes, but this does not affect his arguments.

Acknowledgements For various contributions to the eventual form of this paper, I am most grateful to J M Boyle, W S Brown, W J Cody, B Ford, L D Fosdick, J B Goodenough, J K Slape, B T Smith, G T Symm, B A Wichmann and the Referees.

References

[1] J C D Nissen, B A Wichmann, P J L Wallis and others, Ada-Europe Guidelines for the Portability of Ada Programs. NPL Report DNACS 52/81 and ACM Ada Letters 1 I-3.44 - I-3.61 1982

[2] P J L Wallis, The Preparation of Guidelines for Portable Programming in High-Level Languages, Computer Journal 25 375-378, 1982

[3] U.S. Department of Defense, Reference Manual for the Ada Programming Language, ANSI/MIL-STD 1815A, January 1983

[4] U.S. Depatrment of Defense, Reference Manual for the Ada Programming Language, November 1980

[5] P J L Wallis, Portable Programming, Macmillan (London) and Wiley (New York) 1982

[6] B A Wichmann, Tutorial Material on the Real Data-Types in Ada, US Army European Research Office and ACM Ada Letters 1 I-2.15 - I-2.33 1982

[7] P J L Wallis, Ada Model Arithmetic: Costs and Benefits, IEE Proceedings Part E 127 75-80 1982

[8] Erratum to Reference [7], to appear in IEE Proceedings Part E

[9] W J Cody, Floating-Point Parameters, Models and Standards, In: The Relationship Between Numerical Computation and Programming Languages, J K Reid (ed), North-Holland Publishing Company, 1982 51-65

[10] J C D Nissen, P J L Wallis and others, Ada-Europe Guidelines for Ada Programming Style, to appear.

[11] W S Brown and S I Feldman, Environment Parameters and Basic Functions for Floating-Point Computation, ACM Transactions on Mathematical Software 6 510-523 1980

[12] W S Brown, A Simple but Realistic Model of Floating-Point Computation, ACM Transactions on Mathematical Software 7 445-480 1981