Physical Units with GNAT

Christoph K W Grein
email: christ-usch.grein@t-online.de

Abstract

There has often been a demand to be able to compute with physical items where dimensional correctness is checked. However, methods working at compile-time suffered from the combinatorial explosion of the number of operations required for mixing units and thus could be used with a set of only very few units like e.g. distance, time, and speed. The full SI system with seven base dimensions evaded all such attempts. On the other hand, methods working at run-time were not really applicable because of the memory and calculation overhead.

Ada with its newest generation of 2012 has introduced so-called aspect clauses to allow, among others, specifying additional type properties like type invariants. AdaCore's GNAT uses these aspects in an implementation-specific way to handle physical units at compile-time. This paper presents an overview of the achievements and shortcomings of this method.

With some modification, AdaCore's invention, in the author's opinion, might be apt to standardization in a future Ada generation.

Keywords: physical units, aspect clause, Ada enhancement.

1 Introduction

In the Ada Europe conference in Toulouse 2003, the present author, with co-authors Dmitry Kazakov and Fraser Wilson, gave an overview of methods used to handle physical units in Ada [3, 4]. Unfortunately, the Ada 95 issue 324 [5] dealing with a proposal by the *Ada Rapporteur Group* had not been able to be included in the proceedings because the final submission date for papers had just expired when the ARG proposal was published, so it was mentioned only orally. The conclusion to be drawn from all those attempts was that neither compile-time nor run-time methods were satisfactory for general use.

With the Ada 2012 aspects, things might turn out different. AdaCore's [1] GNAT compiler handles physical dimensions with implementation defined aspects at compile-time in such an ingenious way that it might be apt to be standardized with the next Ada generation (whenever this might be). However, for this to occur, proper demand from the Ada community must be shown to the ARG and the method must prove itself free from pitfalls. Otherwise, ARG would view any such request with utmost reluctance.

This paper presents the author's personal view on the achievements and shortcomings of the GNAT method (as of GNAT GPL 2013 [2]). As you will see, the notation is very natural; any combination of units is possible without the dreaded combinatorial explosion. Some problems have already been solved since the method's first release a few years ago due to user input. It is the author's hope that this paper will induce further discussions among physicists and help to optimize the method so that its chances of standardization will be increased.

2 Shortcomings of hitherto used methods

Compile-time methods using separate types for each dimension and overloading for operators mixing types are well-known to suffer from the combinatorial explosion of the number of operators needed. Thus also the ARG proposal [5] was doomed to fail, which was heavily based on a very clever use of generics. Hence those methods are only applied for a small set of dimensions.

Run-time methods, on the other hand, store dimension information for each item in additional components and thus suffer from the vast additional time and space demands for storing and calculating them. It is unknown to the present author whether these methods have found any application at all.

3 GNAT's use of aspects

GNAT uses an implementation-specific language extension of the new Ada 2012 aspects to define a type and appropriate subtypes for any physical dimension in such a way that dimensional correctness can be checked at compile-time.

The type to be used for physical items is defined with the GNAT-specific aspect Dimension_System, a kind of record aggregate, specifying the seven base dimensions together with the base unit names and symbols. The symbols may be either characters or strings. (The dimension symbols are used for error messages in case of dimensional errors only.) This is done in a package called System.Dim.MKS (see next page). Conceptually, the Dimension_System aggregate declares a record with components

```
record is
   Meter, Kilogram, Second, Ampere,
   Kelvin, Mole, Candela: Fraction;
end record;
```

which the compiler invisibly affixes to any object of the type MKS_Type during compile-time, where the type of each record component is a fraction, i.e. either an integer or a rational number; the Unit_Symbol might be seen as a shortcut for an aggregate like e.g.

```
'm' := (Meter => 1, others =>0):
```

```
package System.Dim.MKS is
type MKS_Type is new Long_Long_Float with
    Dimension_System =>
    ((Unit_Name => Meter   , Unit_Symbol => 'm'  ,
                             Dim_Symbol => 'L'),
     (Unit_Name => Kilogram, Unit_Symbol => "kg" ,
                             Dim_Symbol => 'M'),
     (Unit_Name => Second  , Unit_Symbol => 's'  ,
                             Dim_Symbol => 'T'),
     (Unit_Name => Ampere  , Unit_Symbol => 'A'  ,
                             Dim_Symbol => 'I'),
     (Unit_Name => Kelvin  , Unit_Symbol => 'K'  ,
                             Dim_Symbol => "Θ"),
     (Unit_Name => Mole    , Unit_Symbol => "mol",
                             Dim_Symbol => 'N'),
     (Unit_Name => Candela , Unit_Symbol => "cd" ,
                             Dim_Symbol => 'J'));
```

Other such types may be defined, using at most seven base dimensions, but less are tolerated like for instance the outdated Gaussian CGS with only three dimensions (centimeter, gram, second; the electric and magnetic items use combinations of fractional powers thereof):

```
type CGS_Gauss is new Long_Long_Float with
    Dimension_System =>
   ((Unit_Name =>Centimeter, Unit_Symbol => "cm",
                             Dim_Symbol => 'L'),
    (Unit_Name => Gram     , Unit_Symbol => 'g' ,
                             Dim_Symbol => 'M'),
    (Unit_Name => Second   , Unit_Symbol => 's' ,
                             Dim_Symbol => 'T'));
```

From this type, GNAT creates subtypes via another aspect Dimension, again in the form of a kind of record aggregate:

```
subtype Length is MKS_Type with
  Dimension => (Symbol => 'm',
                Meter =>1,
                others =>0);
```

Here, no connection is present to any of the dimensions defined before, although the aggregate component's name Meter seems to indicate so, because no check is made that the symbol does not conflict with the unit symbol defined above. Any nonsense is possible like Symbol => 's' or even Symbol => "XYZ".

```
subtype Speed is MKS_Type with
   Dimension => (Symbol => "m/s",
                 Meter  => 1,
                 Second =>-1,
                 others => 0);
```

Again no check is performed that the symbol is compatible with the exponents as long as it is composed from basic symbols. Silly lapses like Symbol => "m/s**2" will remain undetected.

Of course, new names may be defined for further subtypes' dimension symbols. We even can use fractional powers:

```
subtype Charge is CGS_Gauss with
   Dimension => (Symbol     => "esu",
                 Centimeter =>3/2,
                 Gram       =>1/2,
                 Second     =>-1,
                 others     =>0);
```

For sure, no check can be performed here that this is correct, as the symbol "esu" has no connection to the unit symbols. This in fact is an implicit declaration of the unit g**(1/2)*cm**(3/2)/s.

In this way, GNAT defines all other SI units with names and symbols by further subtypes:

```
subtype Pressure is MKS_Type with
  Dimension => (Symbol   => "Pa",
                Meter    => -1,
                Kilogram =>  1,
                Second   => -2,
                others   =>  0);

subtype Thermodynamic_Temperature is
   MKS_Type with  Dimension =>
                 (Symbol   => 'K',
                  Kelvin   =>1,
                  others   => 0);

subtype Celsius_Temperature is
   MKS_Type with   Dimension =>
                  (Symbol   => "°C",
                   Kelvin   =>1,
                   others   =>0);
```

The declaration of Celsius_Temperature in the author's opinion is a bad mistake, since temperatures in Kelvin are not simply compatible with those in Celsius.

In effect, the GNAT method is very similar to one of those described in the Toulouse Ada Europe conference 2003 [3], except that the type is not private and the dimension record is present only during compile-time.

4 Notation

The package System.Dim.MKS goes on to declare constants for all named SI units with names reflecting the symbols in order to be able to write values with units:

```
m : constant Length := 1.0;
s : constant Time   := 1.0;
g : constant Mass   := 1.0e-3;
A : constant Electric_Current := 1.0;
Si: constant Electric_Conductance :=  1.0;
dC: constant Celsius_Temperature :=  273.15;

Dist := 5.0 * m;
Dist := 5.0 * M;
```

Whether this is a good idea is questionable, because symbols (and prefixes, see below) are case sensitive whereas Ada is not.

So please note here that the correct symbol S(upper case) for *Siemens* is impossible (and therefore replaced by the invention Si) because of the s (lower case) for *Second* defined before. Also note that in the last line, M for *Meter* in upper case is legal, but actually in the wrong casing,

misleading the reader to think of some other item like a mass.

And GNAT's declaration of dC looks like a severe error in the author's opinion since 5°C is anything but 5.0 * dC! Most probably this constant is meant as the conversion factor between Kelvin and Centigrade, but the latter should not have been declared as a subtype in the first place.

On the other hand, short names should be avoided in any case in software. In physics literature, items are written in italics, units in straight face, so that 5g is 5 grams, but 5*g* could be 5 times the earth acceleration. With short names, mistakes are probable if not inevitable.

A better idea could be to do without these declarations and use the original symbol names instead, since this would easily solve the casing problem, e.g.:

```
Dist := 5.0 * 'm';
Dist := 5.0 * 'M';  -- illegal
Pres := 2.1 * "Pa";
```

Here, the symbol M would be wrong and illegal since there is no such symbol.

An even better idea could be to define a *dimensioned literal*, i.e. a new kind of numeric literal with unit suffixes (in a similar way as C defines literals with suffixes describing the length like 1L for a long integer):

```
Time_of_Travel := 5.0's';
Conductance := 4.2'S';
```

Prefixes

Prefixes pose the same case sensitivity problem – mS is Milli-Siemens, Ms is Mega-Second. You often find such wrong casings in software, and the author, being a physicist, finds this abhorrent.

GNAT defines prefixes only for a few of the base units (meter, kilogram, second, ampere), and only for some powers (milli to mega) of all those defined for the SI system (from 10^{-24} to 10^{+24}). Case sensitivity hits back here – names different from the SI ones have to be used for some prefixes like Meg instead of Mg.

```
mg : constant Mass := 1.0E-06;  -- milli
Meg: constant Mass := 1.0E+03;  -- mega
```

To define prefixes in this way for all named units and all powers is of course feasible, but introduces names that will never be used because of inappropriate size (like GF, gigafarad, whereas capacities generally lie in the pico- respectively nanofarad range; or kT, kilotesla, a magnetic field strength which would tear apart any matter).

A better proposal could be: Again use dimensioned literals like 5.0"ms", and the casing problem is solved. The author has no proposal how these prefixed units could be defined. Ideas are welcome.

As a preliminary conclusion, we see that, apart from some problematic cases, the notation is very natural.

```
Grav: constant Accelaration := 9.81*m/s**2;
  -- 9.81"m/s**2";  -- author's proposal
T: Time;
D: Length;

D := 0.5*Grav*T**2;
```

As was mentioned above, even fractional powers are provided, so that the author's pet equation, the Schottky-Langmuir equation, may be solved for any item, the current density *j*, the voltage *U*, the distance *d*.

$$j = \frac{4}{9}\varepsilon_0\sqrt{\frac{2e_0}{m_0}}\frac{U^{\frac{3}{2}}}{d^2}$$

When you declare constants of unknown dimension like intermediate values, the dimension is taken from the initial value as has always been the case with indefinite subtypes:

```
Material_Const: constant MKS_Type :=
      4.0/9.0 * Eps0 * (2.0 * E0/M0)**(1/2);
```

Unfortunately, GNAT does not allow this for variables.

Let us deal with some more fractional exponents.

```
Dist: constant Length := (8.0*cm)**(1/3+2/3);
```

This fails with dimension mismatch because of preference of integer division in the exponent (1/3+2/3=0+0). However, this works:

```
Eight_cm: constant Length := (8.0*cm)**((1+2)/(5-2));
```

Admittedly, who would write such nonsense, but the rational arithmetics package (there is no documentation) seems inconsistent.

You have to be very careful with fractional powers because of the preference of integer division. The reason for this behaviour lies in the very base of Ada and is partly unavoidable:

```
 8.0**(1/3) = 1.0  -- (a)
 8.0**(1/3)*cm  = 2.0*cm  -- (b)
(8.0*cm)**(1/3) = 2.0*cm**(1/3)  -- (c)
```

Case (a) is "classical" Ada: 1/3=0; (b) and (c) use fractional arithmetic with the GNAT invention because the item is dimensioned. You also need a dimensioned value to give the expected result for expression (a) (by the way: what *is* expected here?):

```
One: constant MKS_Type := 1.0;
8.0**(1/3)*One = 2.0  -- GNAT invention
8.0**(1/3) = 1.0  -- classical Ada
```

Here, the constant One also has *dimension* 1, i.e. in normal parlance it is "dimensionless". This behaviour might lead to very difficult to find problems, to say the least.

Exponents must be known at compile time, so X**N may be written as long as N is a static integer constant, but the following is illegal when X is not dimensionless:

```
for N in A_Range loop
   … X**N …  -- illegal
end loop;
```

This is not really a limitation since variable exponents turn up normally only in power series, and these can be reformulated so that the dimension is extracted to a common factor.

Fractional constants cannot be written at all since GNAT does not disclose their type:

```
One_Third: constant ?:= 1/3;
```

5 Mathematical functions

We have seen that fractional powers are possible. This means that the exponentiation operator is triply overloaded:

```
function "**"  -- Standard
  (Left: MKS_Type'Base; Right: Integer)
  return MKS_Type'Base;
function "**"  -- GNAT invention
   (Left: MKS_Type'Base;
    Right: Rational)
   return MKS_Type'Base;
function "**"  -- gen.elem.functions
  (Left: MKS_Type'Base;
   Right: MKS_Type'Base)
  return MKS_Type'Base;
```

where the first two have dimensioned arguments and return another dimensioned value, whereas the last requests all parameters dimensionless and also returns a pure number.

(The second declaration is in fact a lie, there is no type named *Rational*, but a function with such a profile must exist somewhere, albeit hidden. And it is absolutely not clear how GNAT manages to resolve the overloading of an expression like a**(1/3), since 1/3=0 if the literals in the fraction are of type Integer; see the cases (a) and (b) above in the previous section. So of which type are the literals if 1/3 is a *Rational* and not an Integer.)

This leads us to the question which dimensions are allowed for arguments of mathematical functions. From physics, we know the answer: They must all be dimensionless except for the square root (i.e. the rational exponent 1/2) and some of the trigonometric functions, e.g.:

```
function Sin (X, Cycle: MKS_Type'Base)
      return MKS_Type'Base;
```

Both arguments here must have the same dimension, the result is dimensionless, a pure number. The corresponding rule holds for the inverse function:

```
function Arcsin (X, Cycle: MKS_Type'Base)
      return MKS_Type'Base;
```

must request X dimensionless and return a value that is dimensioned like Cycle.

Similar rules apply to the arctangent as the reverse of the tangent:

```
function Arctan (Y: MKS_Type'Base;
                 X: MKS_Type'Base := 1.0;
                 Cycle: MKS_Type'Base])
      return MKS_Type'Base;
```

must request X and Y to have the same dimension (the quotient Y/X must be dimensionless) and the return value must be dimensioned like Cycle.

On the other hand, an expression like Exp(5.0*m) or Sin(42.0*kg) is complete nonsense.

GNAT requires all mathematical functions except SQRT of an instantiation of the package Ada.Numerics.Generic_Elementary_Functions for a dimensioned type to have dimensionless parameters.

6 Vectors and records

An instantiation of package Ada.Numerics.Generic_Real_Arrays provides arrays and matrices, which may serve as vectors and tensors. Providing a dimension is possible, but is partly ignored:

```
subtype Axis is Integer range 1 .. 3;
subtype Vector is Real_Vector (Axis);
  A: Vector := (1=> 1.0, 2 => 0.0, 3 => -9.8) * cm/s**2;
  D: Vector := (Axis => 0.0) * m**2;
  T: Mass   := 10.0*kg;
  D := A * T**2 / 2.0;
```

The result of this equation with nonsense units is computed numerically correct, i.e. the factor 10^{-2} (because of the unit cm) is taken into account in the acceleration vector A. The result, when output (see IO below), is without unit indication. It seems that GNAT takes the units into account when computing the values, but then ignores dimensions.

While a matrix and a vector may have a dimension as a whole, individual components with different dimensions are not allowed. This is in best order, since using the method for linear algebra (see below) is more than can be expected.

On the other hand, records may serve for instance as a collection of particle properties, so each component may indeed have a different dimension like mass, charge, location, speed, etc.

GNAT allows those multidimensional components:

```
type Particle is record
   M: Mass;
   Q: Electric_Charge;
   R: Vector := (Axis => 0.0) * m;
  -- Darn, this conflicts with M!
   V: Vector := (Axis => 0.0) * m/s;
   -- Same conflict.
end record;
```

We see here another reason why short names especially for units are evil.

7 Input and output

There is a generic package System.Dim.Float_IO, which however is a plain lie – only output exists, however with unit indication; input of dimensioned items is still an open issue. Also the output facility leaves a lot of wishes open.

```
Q: Electric_Charge := 40.0 * C;
R: Length := 10.0 * cm;
Put (Q**2/R**2, Aft => 2, Exp => 0);
```

This results in

```
160000.00 m**(-2).s**2.A**2
```

The opinions about the dot as a unit separator may vary. However, the fact that there is a blank character between the number and the unit makes it difficult to read the value back in. How can a potential Get operation discriminate between a pure value (with dimension 1) and a dimensioned value when there is no indication how far to read? Also the dot separator makes the integer number 2 in a sequence like s**2.A look like a floating point number 2.0 (remember that upon input, the decimal digits after the dot may be omitted).

The specification of Put is

```
procedure Put
  (Item  : Num_Dim_Float;
   Fore  : Field  := Default_Fore;
   Aft   : Field  := Default_Aft;
   Exp   : Field  := Default_Exp;
   Symbol: String := "");
```

There is some description given in the package specification how the Symbol could be used, but when used the compiler complains (as of GNAT GPL 2013):

```
Symbol parameter should not be provided
reserved for compiler use only
```

In former compiler versions, the symbol could be any string, which, when given, replaced the unit output. There was not any check that the string was appropriate, so any nonsense could be supplied.

What is expected when the symbol string is given, is at least a check that the symbol be appropriate or else an exception be raised. Far better would be an adaptation to the magnitude requested. So for instance the expected output for

```
Put (12.0*m, Symbol => "km");
```

must be something like 0.012 km.

The compiler developers are well aware of the missing input facility. On a personal note to the author they said they were waiting for user requirements.

Compare [3] for a better solution of IO.

8 Type conversions

Since GNAT's dimensional types are numeric types, the Ada type conversion is available, e.g. with the two types shown in this paper, we could write:

```
H: MKS_Type := CGS_Gauss (1.0 * Oe);
```

This is utter nonsense! The H-field is measured in Oersted in the Gaussian system, in A/m in SI. The correct conversion is

$$1\ \mathrm{Oe} = \frac{1000}{4\pi}\ \mathrm{A/m}$$

A type conversion like this cannot handle the unit conversion, so the best would be that this be illegal.

GNAT allows such conversions!

Of course, for being able to provide correctly dimensioned conversions, some form of the Ada type conversion must be available. Ideas are welcome again. One possible way would be to allow type conversions only for dimensionless values, so that the original dimension would first have to be stripped, the value type-converted, last the new dimension added together with the necessary conversion factors.

9 Linear algebra

There is one further application which in the author's view need not be handled: linear algebra. This means that physical dimensions need not be included when linear equations are solved like e.g. for linear partial differential equations. Thus, vectors (like velocity or force) (represented as arrays with three components) and tensors (3 by 3 matrices) just have one physical dimension, whereas in linear algebra, arrays and matrices may have any number of components and each component may have a different dimension. This is, in the author's opinion, way beyond what this method can (and should be able to) handle.

10 Conclusion

GNAT's use of Ada's new aspects, despite its present shortcomings, for physical dimensions is indeed ingenious and deserves attention and thoughtfulness by the Ada community. It has been much improved over the years, and most of the problems mentioned in this paper can easily be solved. Also a few improvement proposals have been presented.

Thus the author again wants to express his hope that widespread use of this method will persuade Ada programmers to further improve the method by communicating their findings to AdaCore and eventually ask the Ada Rapporteur Group to consider incorporation into the next Ada standard.

References

[1] AdaCore, http://www.adacore.com/

[2] GNAT GPL 2013, http://libre.adacore.com/tools/gnat-gpl-edition/

[3] C. Grein, D. A. Kazakov and F. Wilson, *A Survey of Physical Unit Handling Techniques in Ada,* In Jean-Pierre Rosen, Alfred Strohmeier (Eds), Reliable Software Technologies - Ada-Europe 2003, Lecture Notes in Computer Science, Vol. 2655, Springer-Verlag.

[4] C. Grein, *Handling Physical Dimensions in Ada*, http://www.christ-usch-grein.homepage.t-online.de/Ada/Dimension.html

[5] AI95-00324-01 *Physical Units Checking* http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ais/ai-00324.txt?rev=1.3.