Implementation of a simple dimensionality checking system in Ada 2012

Vincent Pucci and Edmond Schonberg

Adacore Inc., New York NY 10011, USA,
pucci@adacore.com, schonberg@adacore.com,
http://www.adacore.com

Abstract. We present the design and implementation of a dimensionality checking system in Ada 2012. The system is implemented in the GNAT compiler, and performs compile-time checks to verify the dimensional consistency of physical computations. The system allows the user to define his own system of units, and imposes no run-time changes nor multiple compilation passes on the user.

Keywords: Dimensionality Checking, System of units, Ada 2012

1 Introduction

Dimensionality checking (DC) is a common practice in Physics and Engineering: formulae are checked to verify that meters are assigned to meters and not kilograms, that an empirical formula for viscosity yields a force per unit area, etc. From the earliest days of Ada and the introduction of derived types, there have been proposals to use the Ada type system to perform dimensionality checking as part of the static analysis of a program.

A comprehensive summary of the history of the problem is given in [1]. Most proposals involve discriminants for records that wrap a numeric quantity. These discriminants are either assumed to be stored apart from the record, or else the system requires two compilation passes, one with discriminants to perform the check, and one without them for actual execution. Conceptually, all models (including ours) consist in associating some attribute (in general a vector of numbers) to existing types, and associating the required checks to arithmetic manipulations of such annotated types.

We have implemented in the GNAT compiler a simple DC system (dubbed GeoDesiC) that relies on the Aspect constructs of Ada 2012, and on some simple extensions to the semantic analyzer of the front-end of GNAT. The system allows the user to define his own system of units.

We present an example MKS (for meter-kilogram-second) package and its use (see appendix, section 9). The MKS system of units is the international standard (SI) sytem of units based on seven base physical units: meter, kilogram, second, ampere, kelvin, mole and candela.

2 Ada 2012 Aspects for Dimensional Checking

2.1 Ada 2012 Aspects

Aspects constitute a new feature of the Ada 2012 language [2]. Aspects generalize earlier Ada notions of attributes, but present a number of advantages over them. Aspects are associated with the declaration of an entity and denote some specifiable characteristics of that entity. The aspects associated wih a given entity declaration are provided by an *Aspect_Specification* whose grammar is [2]:

```
with ASPECT_MARK [=> ASPECT_DEFINITION] {,
    ASPECT_MARK [=> ASPECT_DEFINITION] }

ASPECT_MARK ::= aspect_IDENTIFIER['Class]

ASPECT_DEFINITION ::= NAME | EXPRESSION | IDENTIFIER}
```

Aspect specifications can be provided for such familiar attributes as Size, Address, Convention, Inline, etc. The can also specify Ada 2012 characteristics such as Pre and Post-conditions, type invariants, etc. Finally, an implementation can provide additional aspects, for declarations that can legally received them.

2.2 Aspects for Dimensional Checking

We introduce two implementation-defined aspects: **Dimension_System** and **Dimension**.

Aspect Dimension_System The form of an aspect Dimension_System is as follow:

```
with Dimension_System => (
     DIMENSION
  [, DIMENSION]
  [, DIMENSION]
  [, DIMENSION]
  [, DIMENSION]
  [, DIMENSION]
  [, DIMENSION]);

DIMENSION ::= (
  [Unit_Name   =>] IDENTIFIER,
  [Unit_Symbol =>] SYMBOL,
  [Dim_Symbol  =>] SYMBOL)

SYMBOL ::= STRING_LITERAL | CHARACTER_LITERAL
```

An aspect Dimension_System applies only to a numeric type, typically a floating point type of the appropriate precision (any numeric type can be used as a base). The aspect specifies the unit names, the unit symbols and the dimension symbols to be used when performing formatted output on dimensioned quantities. The aspect value is thus an aggregate of some internal array type. In an SI system, this aggregate will have seven components, but this is not required by GeoDesiC.

Illustration of the syntax

```
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 => ""),
     (Unit_Name => Mole,      Unit_Symbol => "mol", Dim_Symbol => 'N'),
     (Unit_Name => Candela,   Unit_Symbol => "cd",  Dim_Symbol => 'J'));
```

A type (more accurately a first subtype) to which the aspect Dimension_System applies is a ***dimensioned type***.

Aspect Dimension The form of an aspect Dimension is as follow:

```
with Dimension => (
  [[Symbol =>] SYMBOL,]
               DIMENSION_VALUE
  [,           DIMENSION_VALUE]
  [,           DIMENSION_VALUE]
  [,           DIMENSION_VALUE]
  [,           DIMENSION_VALUE]
  [,           DIMENSION_VALUE]
  [,           DIMENSION_VALUE]);

SYMBOL ::= STRING_LITERAL | CHARACTER_LITERAL

DIMENSION_VALUE ::=
    RATIONAL
  | others => RATIONAL
  | DISCRETE_CHOICE_LIST => RATIONAL

RATIONAL ::= [-] NUMERAL [/NUMERAL]
```

An aspect Dimension applies only to a subtype of a dimensioned type. The aspect specifies a string [1](or character literal) to be used for output, and the rational

[1] Noted that the string here is optional. When not present, a special treatment is performed by GeoDesic for output facilities (section 5).

numbers that determine the dimensions of the physical entity represented by the subtype:

Illustration of the syntax

```
subtype Frequency is Mks_Type
  with
   Dimension => (Symbol => "Hz",
     Second => -1,
     others => 0);
```

A subtype to which the aspect Dimension applies is a ***dimensioned subtype***.

3 Design Constraints

GeoDesiC is a compile-time system: it does not modify run-time structures and must be able to perform dimensional checking on all numeric expressions. This imposes on the user the requirement that all variables and constants be of a dimensioned subtype. Constants whose type declaration lacks an aspect Dimension, as well as named numbers, are treated as dimensionless quantities.

- A special programming obligation applies to exponentiation: the exponent must be a dimensionless static rational constant, otherwise the base must be dimensionless and then the result is dimensionless as well. This is consistent with Physics practice: variable exponents do not appear in formulae, unless the base of the exponentiation is dimensionless. (they are more exponents in geometric formulae, e.g. in n-dimensional computations, but these are invariably dimensionless). Exponents in general must be rational numbers, not just signed integers. Fractional exponents are common in physical computations (e.g. the period of a pendulum as a function of length). Arithmetic computations on rational are exact, which is a requirement for proper dimensionality checking. Thus, a rational arithmetic package is provided in GNAT for GeoDesiC use. *Rational grammar in GeoDesiC:*
  `RATIONAL ::= [-] NUMERAL [/ NUMERAL]`[2]

- Because of its frequent appearance in physical formulae, the square root function (defined in Ada.Numerics.Elementary_Functions) is recognized by GeoDesiC, and treated specially, so the dimensions of the result can be set to half the dimensions of the operand.

  Example of the Square root function in GeoDesic:

  ```
  subtype Area is Mks_Type
    with
     Dimension => ("m**2",
       Meter =>  2,
  ```

[2] Note that an integer is a rational.

```
        others => 0);

   A : constant Area := 2.0;
   L : Length;
begin
   L := Sqrt (A);
```

– Moreover all the other elementary functions are also recognized, and GeoDesic ensures that any parameters for these functions, as well as their results, are dimensionless.

Example of the Sin function in GeoDesic:

```
   A : constant Area := 2.0;
   R : Mks_Type;
begin
   R := Sin (A);
```

Incorrect call to Sin is rejected with the following diagnoses:

```
parameter of "Sin" must be dimensionless
parameter has dimension [L**2]
```

4 Implementation

Arithmetic operations, assignment, object declarations with initialization expressions (component declarations with default expressions), exponentiation, function calls and return statements are operations that verify and/or compute the dimension vectors of the constituents of the operation at compile-time. (each one of the examples used below is followed by the corresponding compilation error messages).

– For addition operators, both operands must have the same dimension. Unary operations propagate the dimension of their operand.

```
   Result : Mks_Type;
begin
   Result := cm + kg;
```

Incorrect addition is rejected with the following diagnoses:

```
both operands for operation "+" must have same dimensions
left operand has dimension [L]
right operand has dimension [M]
```

– For multiplication operators (* and /) the operands are unconstrained, and the components of the resulting dimension vector are the sum (resp. the difference) of the components dimensions of the operands.

– For assignment the dimension vectors of name and expression must agree. A similar rule applies to parameter passing in calls.

```
   Result : Mks_Type;
begin
   Result := cm * kg;
```

Incorrect assignment is rejected with the following diagnoses:

```
dimensions mismatch in assignment
left-hand side is dimensionless
right-hand side has dimension [L.M]
```

– For object declarations with initialization expressions, both the expression and the type mark must have same dimensions, unless the expression is a literal.

```
   Correct_1 : constant Length := 2.0;
   Correct_2 : constant Length := 2.0 * m;
   Wrong     : constant Length := 2.0 * kg;
```

Incorrect object declaration is rejected with the following diagnoses:

```
dimensions mismatch in object declaration
object type has dimension [L]
object expression has dimension [M]
```

– For exponentiation, the exponent must be a static rational constant. The components of the resulting dimension vector are obtained by multiplying the corresponding components of the dimensions of the base by the value of the exponent.

– Both operands in a relational operation must have the same dimensions.

```
   Result : Boolean;
begin
   Result := kg**(5/6) >= kg;
```

Incorrect relational operation is rejected with the following diagnoses:

```
both operands for operation ">=" must have same dimensions
left operand has dimension [M**(5/6)]
right operand has dimension [M]
```

– For functions (except elementary functions, see section 3), the resulting dimension vector is the dimension vector of the type entity.

– For return statements, the returned expression and return type must have the same dimensions.

```
   Result : Time;

   function Correct (L : Length) return Time;
   function Wrong (X : Mks_Type) return Length;

   ---------------
   -- Correct --
   ---------------

   function Correct (L : Length) return Time is
   begin
      return L * s / m;
   end Correct;

   -------------
   -- Wrong --
   -------------

   function Wrong (X : Mks_Type) return Length is
   begin
      return X * kg;  --  X * kg is not a length
   end Wrong;

begin
   Result := Correct (cm);
   --  Both Result and Correct (cm) denote a time.
```

Incorrect return statement is rejected with the following diagnoses:

```
dimensions mismatch in return statement
returned type has dimension [L]
returned expression has dimension [M]
```

The modifications to the GNAT semantic analyzer are straightforward and need not be discussed here. The interested reader will find them in the GNAT sources.

5 Input-Output

The current system provides formatted output thanks to two generic packages for integer and float dimensioned types. If the argument of the output routine is a variable and if a symbol is provided in the dimension Aspect of its subtype declaration, then the symbol string is used as a suffix to the value. Otherwise the dimension vector is output in standard notation, e.g. *"m*kg*s** (-2)"*. Engineering conventions differ in the display of negative values for dimensions, and some may prefer to see *"m*kg/s**2"*.

Similarly to the elementary functions, GeoDesiC recognizes the put routines for dimensioned types, so it can output the dimension symbols as a suffix of the value.

6 Examples

This section illustrates two well-known physical systems.

6.1 The simple pendulum

In this section, a simple example is used in order to illustrate the syntax of GeoDesic.

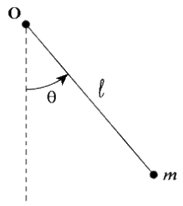

Fig. 1. Pendulum oscillation

Question: Determine the length l of a simple pendulum (figure 1) whose period T_o equals 2 seconds (consider the case of small amplitude oscillations).
Data: $g = 9.81ms^{-2}$.

Resolution:

$$T_o = 2\pi\sqrt{\frac{l}{g}} \tag{1}$$

From equation 1, we deduce the expression of l:

$$l = \frac{gT_o^2}{4\pi^2} \tag{2}$$

```
with Ada.Text_IO;              use Ada.Text_IO;
with System.Dim.Mks;           use System.Dim.Mks;
with System.Dim.Mks_IO;        use System.Dim.Mks_IO;
--  System.Dim.Mks_IO provides Mks output routines

procedure Pendulum is
   --  New dimensioned subtype for Earth gravity g

   subtype Gravity is Mks_Type
     with
      Dimension => ("m.s**(-2)",
```

```
        Meter => 1,
        Second => -2,
        others => 0);

   -- Data

   g   : constant Gravity := 9.81;
   T_0 : constant Time := 2.0;
   l   : Length;
begin
   -- Evaluate l

   l := g * T_0**2 / (4.0 * Pi**2);

   -- Output l

   Put ("l = ");
   Put (l, 1, 3, 0);
end Pendulum;
```

Result:

```
l = 0.994 m
```

6.2 The Venturi effect in constricted flow

This common fluid statics problem is a more ellaborate example and deals with various dimensioned formulae.

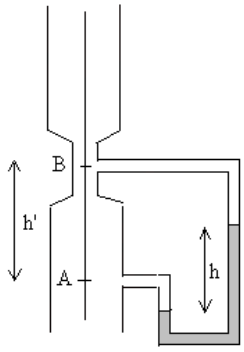

Fig. 2. Venturi

Questions:

- Determine the expression of the water flow in the Venturi Q_V (figure 2) in terms of pressure difference between points A and B and of the distance $h^{'}$. Consider the perfect fluid case.

- Evaluate the flow knowing that diameters of the neck and of the pipe are respectively 15 and 30 cm.

- Calculate both water average speeds v_A and v_B.

Data: $h = 75.0cm$ and mercury density $\nu_{Hg} = 13.6$.

Solution:

- Flow conservation law and Bernouilli's theorem set:

$$v_B = v_A \frac{D^2}{d^2} \tag{3}$$

$$\frac{1}{2}v_A^2 + \frac{p_A}{\nu_{H_2O}} + g \cdot z_A = \frac{1}{2}v_B^2 + \frac{p_B}{\nu_{H_2O}} + g \cdot z_B \tag{4}$$

Equation 3 yields:

$$v_B^2 - v_A^2 = v_B^2 \left(1 - \beta^4\right) \tag{5}$$

where $\beta = \frac{d}{D}$.

And equation 4 leads to:

$$v_B^2 - v_A^2 = 2\frac{p_A - p_B}{\nu_{H_2O}} - 2 \cdot g \cdot h^{'} \tag{6}$$

where $h^{'} = z_A - z_B$

Hence, knowing that $Q_V = v_B S_B$ (where $S_B = \pi \frac{d^2}{4}$ is the venturi internal area at point B):

$$Q_V = \pi \frac{d^2}{4} \sqrt{\frac{2\left(p_A - p_B\right)/\nu_{H_2O} - 2 \cdot g \cdot h'}{1 - \beta^4}} \tag{7}$$

- In order to evaluate the flow, we introduce points 1 and 2 setting: $z_2 - z_1 = h$. By definition:

$$\begin{aligned} p_1 - p_2 &= \nu_{Hg} \cdot g\left(z_2 - z_1\right) \\ &= \nu_{Hg} \cdot g \cdot h \\ &= p_A + \nu_{H_2O} \cdot g\left(z_A - z_1\right) - p_B - \nu_{H_2O} \cdot g\left(z_B - z_2\right) \end{aligned} \tag{8}$$

Thus:

$$p_A - p_B = \left(\nu_{Hg} - \nu_{H_2O}\right) g \cdot h + \nu_{H_2O} \cdot g \cdot h^{'} \tag{9}$$

Replacing expression 9 in equation 6:

$$v_B^2 - v_A^2 = 2\frac{(\nu_{Hg} - \nu_{H_2O})\, g \cdot h}{\nu_{H_2O}} \tag{10}$$

Finally, we deduce the new expression of Q_V:

$$Q_V = \pi\frac{d^2}{4}\sqrt{\frac{2\,(\nu_{Hg} - \nu H_2O)\, g \cdot h}{\nu_{H_2O}\,(1-\beta^4)}} \tag{11}$$

```
with Ada.Text_IO;          use Ada.Text_IO;
with System.Dim.Mks;       use System.Dim.Mks;
with System.Dim.Mks_IO;    use System.Dim.Mks_IO;
--  System.Dim.Mks_IO provides Mks output routines

procedure Venturi is
   --  New Dimensioned subtypes

   subtype Area is Mks_Type
     with
      Dimension => (
        Meter =>  2,
        others => 0);
   subtype Flow is Mks_Type
     with
      Dimension => (
        Meter =>  3,
        Second => -1,
        others => 0);
   subtype Gravity is Mks_Type
     with
      Dimension => (
        Meter =>  1,
        Second => -2,
        others => 0);
   subtype Speed is Mks_Type
     with
      Dimension => (
        Meter =>  1,
        Second => -1,
        others => 0);

   --  Data

   d_1     : constant Length := 15.0 * cm;
   d_2     : constant Length := 30.0 * cm;
```

```
   beta    : constant Mks_Type := d_1 / d_2;
   g       : constant Gravity := 9.81;
   h       : constant Length := 75.0 * cm;
   nu_Hg   : constant Mks_Type := 13.6;
   nu_H2O  : constant Mks_Type := 1.0;
   Q_v     : Flow;
   S_B     : Area;
   v_A     : Speed;
   v_B     : Speed;

begin
   --  Evaluation of Q_v

   S_B := Pi * (d_1 / 2.0)**2;
   Q_v := S_B *
            (2.0 * (nu_Hg - nu_H2O) * g * h /
              (nu_H2O * (1.0 - beta**4)))**(1/2);
   --  Use of exponent 1/2 instead of Sqrt (same behavior for
   --  dimensioned operand)

   --  Output Q_v

   Put ("Q_v = ");
   Put (Q_v, 0, 3, 0);
   New_Line;

   --  Evaluation of v_A and v_B

   v_B := Q_v / S_B;
   v_A := beta * v_B;

   --  Output v_A and v_B
   Put ("v_A = ");
   Put (v_A, 1, 2, 0);
   New_Line;

   Put ("v_B = ");
   Put (v_B, 2, 1, 0);
end Venturi;
```

Results:

```
Q_v = 0.249 m**3.s**(-1)
v_A = 7.03 m.s**(-1)
v_B = 14.1 m.s**(-1)
```

7 Conclusions

Apart from the use of Ada2012 features (which could even be transformed into Ada 2005 pragmas) and its full compile-time behavior, GeoDesiC makes no claims on originality, and is close in spirit to previous DC proposals for Ada. A dimension aspect is simply a set of discriminants stored away from a discriminated value. This allows the implementation to share discriminant vectors during semantic analysis, and garbage-collect them when analysis is complete. A substantial part of the implementation (trivial from an algorithmic point of view, but vital for usability) deals with output of dimensioned quantities. We hope that the relative ease of use of the system will make it attractive in Physics and Engineering applications, and we welcome feedback and suggestions for improvement from users.

Related works

The work of Christof Grein on the topic has been invaluable to us, and we have freely borrowed from the work described in [3], including the manipulation of rational and integer constants. The earliest proposal for dimensionality checking in Ada is due to Paul Hilfinger [4].

The Units of Measure technique in F# described in [5] has also been a source of inspiration to us, as well as the implementation of DC in Parasail (T.Taft, private communication).

We hope that our simple model will finally make dimensionality checking widespread in scientific programming.

8 Aknowledgements

Many thanks are due to Hirstian Kirchev, Thomas Quinot, Bob Duff, Tucker Taft and Yannick Moy for numerous comments and productive discussions.

References

1. C. Grein, D.A Kazakov, and Fraser Wilson: A survey of Physical Units Handling Techniques in Ada. Ada-Europe 2003, LNCS Vol. 2655, p.258-270. Springer, Heidelberg (2003)
2. Ada Reference Manual, 13.3.1 Aspect Specifications. (2011) http://www.ada-auth.org/standards/12rm/html/RM-13-3-1.html
3. C.Grein: Handling Physical Dimensions in Ada. (28 April 2008) http://www.christ-usch-grein.homepage.t-online.de/Ada/Dimension.html
4. Paul N. Hilfinger: An Ada package for dimensional analysis. (1988)
5. Microsoft Developer Network, Units of Measure (F#). (May 2010) http://msdn.microsoft.com/en-us/library/dd233243.aspx

9 Appendix: The MKS package

Gnat provides a predefined MKS package with the following outline:

```
package System.Dim.Mks is
   --  Dimensioned type Mks_Type

   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 => ""),
        (Unit_Name => Mole,      Unit_Symbol => "mol", Dim_Symbol => 'N'),
        (Unit_Name => Candela,   Unit_Symbol => "cd",  Dim_Symbol => 'J'));

   --  SI Base dimensioned subtypes

   subtype Length is Mks_Type
     with
      Dimension => (Symbol => 'm',
        Meter  => 1,
        others => 0);
   subtype Mass is Mks_Type
     with
      Dimension => (Symbol => "kg",
        Kilogram => 1,
        others =>   0);
   subtype Time is Mks_Type
     with
      Dimension => (Symbol => 's',
        Second => 1,
        others => 0);
   subtype Electric_Current is Mks_Type
     with
      Dimension => (Symbol => 'A',
        Ampere => 1,
        others => 0);
   subtype Thermodynamic_Temperature is Mks_Type
     with
      Dimension => (Symbol => 'K',
        Kelvin => 1,
        others => 0);
   subtype Amount_Of_Substance is Mks_Type
```

```
  with
   Dimension => (Symbol => "mol",
     Mole =>    1,
     others => 0);
subtype Luminous_Intensity is Mks_Type
  with
   Dimension => (Symbol => "cd",
     Candela => 1,
     others =>  0);

--  SI Base units

m   : constant Length                      := 1.0;
kg  : constant Mass                        := 1.0;
s   : constant Time                        := 1.0;
A   : constant Electric_Current            := 1.0;
K   : constant Thermodynamic_Temperature   := 1.0;
mol : constant Amount_Of_Substance         := 1.0;
cd  : constant Luminous_Intensity          := 1.0;

--  SI common prefixes for Meter

um  : constant Length := 1.0E-06;  -- micro
mm  : constant Length := 1.0E-03;  -- milli
cm  : constant Length := 1.0E-02;  -- centi
dm  : constant Length := 1.0E-01;  -- deci
dam : constant Length := 1.0E+01;  -- deka
hm  : constant Length := 1.0E+02;  -- hecto
km  : constant Length := 1.0E+03;  -- kilo
Mem : constant Length := 1.0E+06;  -- mega
--  SI prefixes for Kilogram, Second, and other units

--  SI Derived dimensioned subtypes

subtype Angle is Mks_Type
  with
   Dimension => (Symbol => "rad",
     others => 0);
subtype Solid_Angle is Mks_Type
  with
   Dimension => (Symbol => "sr",
     others => 0);
subtype Frequency is Mks_Type
  with
   Dimension => (Symbol => "Hz",
```

```
        Second => -1,
        others => 0);
   subtype Force is Mks_Type
     with
      Dimension => (Symbol => 'N',
        Meter =>    1,
        Kilogram => 1,
        Second =>  -2,
        others =>   0);
   --  etc

   --  SI derived units

   rad : constant Angle       := 1.0;
   sr  : constant Solid_Angle := 1.0;
   Hz  : constant Frequency   := 1.0;
   N   : constant Force       := 1.0;
   --  etc
end System.Dim.Mks;
```