Units of Measure in Fortran

N1970

Van Snyder
van.snyder@jpl.nasa.gov

Caltech Jet Propulsion Laboratory

22 April 2013

The Problem

Incorrect use of physical units is the third most common error in scientific or engineering software for which assistance from the programming language might be expected, coming immediately after mismatched, missing or excess actual arguments in procedure references, and out-of-bounds array references. Explicit interfaces largely solve the second problem and help the third problem, but do nothing directly for the first.

Expensive example

On September 23, 1999, the NASA/JPL Mars Climate Orbiter spacecraft arrived at Mars 67 kilometers lower than expected. As a result, it was destroyed by atmospheric stresses. The cost of the mission was $US 300 million.

The cause of the accident was improper units of measure.

Proposal

Define a language-based system that shall

- allow to define a system of units upon an arbitrary foundation,
- check units in expressions, assignment, and procedure references at compile time,
- distinguish different measures of the same fundamental quantity, and provide for explicit conversion between them,
- not increase execution time, except where conversion is explicitly requested,
- not require additional storage,
- allow to output units, and to check and convert units during input,
- require minimal labor for its use, and
- support abstract units, which specify the relationship of units of dummy arguments and function results, and thereby the relationship of corresponding actual arguments and function results, without requiring specific units.

Alternatives based upon derived types

There are at least three methods based upon derived types that can provide some support for units. Each would have a real component to represent the value of the object.

- Specify exponents of fundamental units using kind type parameters.
- Specify exponents of fundamental units using integer components. Different measures of the same unit could be distinguished and converted by providing for scale and offset using additional real components.
- Use a different type for each unit.

General drawbacks of methods based upon derived types

- All such methods increase execution time.
- All such methods have larger labor cost than the proposed method.

Kind type parameters to represent exponents

Assume one wishes to check and compose only the exponents of units in a system based only upon SI units. The meaningful ranges of the exponents of SI units in most applications are:

Unit	Measure	Exponent Range	Number of exponents
length	meter	$-3 \cdots +3$	7
time	second	$-3 \cdots +1$	5
mass	kilogram	$-1 \cdots +1$	3
temperature	Kelvin	$-1 \cdots +1$	3
electric current	Ampere	$-1 \cdots +1$	3
quantity of a substance	mole	$-1 \cdots +1$	3
luminous intensity	candela	$-1 \cdots +1$	3
angle[1]	radian	$-2 \cdots +2$	5

These aren't enough: Vacuum permittivity is $\ell^{-3} \times m^{-1} \times t^4 \times I^2$.

[1] Not an SI unit

Kind type parameters (cont.)

Compile-time checking and composition of exponents of these units could be provided by a type with eight kind type parameters, one for each exponent, and a real component to represent the value of the object. This scheme cannot support abstract units.

If one wishes to provide procedures in advance for all possible combinations of exponents, one needs $10 \times (7 \times 5 \times 3^5 \times 5) = 425,250$ procedures for identity, negation, addition, subtraction, and comparison, for each REAL kind. For multiplication and division, one needs an additional $2 \times \prod_{i=1}^{8} f(a_i, b_i)$ procedures, where $f(a, b) = (b - a + 1)^2 + a(1 - a)/2 - b(1 + b)/2$ and a_i, b_i are the exponent bounds in the table, or $425,351,556$ procedures, assuming one wants not to produce any exponents outside the above ranges.

Defined formatted I/O, to output units or check input units automaticaly, requires an additional 85,500 procedures, for each REAL kind.

Exponentiation by an integer requires an additional type with one kind type parameter, and about 150,000 more procedures, for each REAL kind.

Altogether, about 426 million procedures, for each REAL kind.

Kind type parameters (cont.)

It is clearly out of the question to produce, in advance, a library of more than 425 million procedures, or 1.7 billion for all possible combinations of only two real kinds.

Fortran does not yet provide for macros, parameterized modules, or parameterized procedures.

One would therefore need either to write a preprocessor to generate the procedures needed for a specific application, use a macro processor to do so, or write them manually. Each method has substantial labor cost.

Using kind type parameters, one gets compile-time checking of exponents of units, but no checking or conversion, at compile time or run time, of different measures of the same unit, such as feet and meters.

Unless procedures are inlined and optimized, execution time would be increased.

Additional units for different applications, such as decibels, currency, safety rates. . . , would require additional kind type parameters, and additional procedures.

Support for fractional exponents would require enormously more procedures.

Components to represent exponents

Grant Petty described a system to check and compose exponents of units using a derived type with integer components for the exponents. A description appeared in **Software–Practice and Experience 31**, pp 1067–1076 (2001), and a module to implement it is available online at `http://sleet.aos.wisc.edu/~gpetty/wp/?page_id=684`. The module defines 116 procedures.

The number of exponent components is specified by a named constant, with a value of eight in the distributed module.

The values of exponent components of arguments are compared in procedures for addition, subtraction, comparison, and assignment. The program stops if any are unequal. Pointer assignment is not checked.

The values of exponent components of the function result are computed in procedures for multiplication, division, and exponentiation.

Components to represent exponents (cont.)

There is significant internal overhead in Petty's type and procedures. In some applications this is not important. In others, it is intolerable.

Petty suggests removing the exponent components, and simplifying the procedures to eliminate checking and composition of exponents, when the program is deployed. Even so, execution time would be increased unless procedures are inlined and optimized.

Edsgar Dijkstra likened this to being required to fasten your seat belt while the airplane taxies about the airport, but then the flight attendant takes it away before the airplane takes off.

Components to represent exponents (cont.)

Petty's scheme does not distinguish measures of the same fundamental unit, such as feet and meters. Doing so would require additional components for scale and offset, which would mostly not do anything, and could not be removed in production.

If one of Petty's procedures discovers an error, it executes a STOP statement. Most processors do not provide an indication where in a program this occurs.

Petty's scheme provides no compile-time checking.

Different type for each measure

One could provide complete compile-time checking of exponents of units, and distinguish and convert between different measures of the same unit, by using a different type for each measure.

The number of types would depend upon the application.

The number of procedures would be on the order of the square of the number of types.

The amount of labor required would depend upon the application.

A library hoping to provide all meaningful types and procedures for a broad area of applications, such as mechanics, electronics, or thermodynamics, could be very large.

Execution time would be increased, unless procedures are inlined and optimized.

Cannot support abstract units.

Proposed language-based method

The proposed language-based method adds syntax to define units and attributes of units, and to specify the UNIT attribute for real variables, components, and named constants.

A unit can be an atomic, composite, conversion, or abstract unit.

Specifications of how units are used in expressions, assignment, ASSOCIATE and SELECT TYPE constructs, procedure arguments and function results, input/output, generic resolution, and intrinsic procedures, are included, but require no new syntax beyond one edit descriptor.

Unit definition statement

Syntax for a UNIT definition statement is specified:

`UNIT [[` *unit-attr-list*`] ::]` *unit-name* ■
`[ =` *unit-expression* `]`

Unit attributes are ABSTRACT, EXCLUDE_ARITHMETIC, or an access specification.

If no *unit-expression* is specified, the unit is atomic.

If a *unit-expression* is specified, it can either specify

- composition involving multiplication and division of previously-defined units, and exponentiation of a unit by a rational exponent, or
- a conversion from one previously-defined unit, boiling down to an expression of the form $a \times$ *unit-name* $+ b$, where a and b are unitless constant expressions, and a is nonzero.

Intrinsic atomic units UNITLESS and RADIAN are defined.

Unit definition statement examples

```
!  Atomic units:
UNIT, PUBLIC :: METER, SECOND, KILOGRAM, KELVIN
UNIT, PUBLIC, EXCLUDE_ARITHMETIC :: DECIBEL

!  Composite units:
UNIT :: CENTARE = METER**2
UNIT :: STERE = METER**3
UNIT :: METER_PER_SECOND = METER / SECOND
UNIT :: KG_PER_STERE = KILOGRAM / STERE

!  Conversion units:
UNIT :: CM = 100 * METER
UNIT :: HECTARE = CENTARE / 10000.0
UNIT :: INCH = CM / 2.54
UNIT :: IPS = 100 * METER_PER_SECOND / 2.54
UNIT :: CELSIUS = KELVIN - 273.15
UNIT :: FAHRENHEIT = 9.0 * CELSIUS / 5.0 + 32.0
```

Unit attribute declaration

A real variable, component, or named constant, can be declared to have the UNIT attribute.

A UNIT(*unit-name*) attribute specification can appear in a type declaration statement and a component declaration.

Alternatively, a real variable or named constant may be declared to have the UNIT attribute using a UNIT(*unit-name*) statement.

Examples:

```
REAL, UNIT(KELVIN) :: K
REAL :: F
UNIT(FAHRENHEIT) :: F
```

Units in expressions and assignment

Where operands are added or subtracted, their units shall be equivalent.

Where operands are multiplied, the result units are the product of their operand's units.

Where operands are divided, the result units are the units of the first operand divided by the units of the second operand.

Where an operand is raised to a constant integer power, the units of the result are the units of the operand raised to the same constant integer power.

Where an operand is raised to a power that is not constant or not an integer, the operand shall be unitless and the result is unitless.

In intrinsic assignment or pointer assignment, the units of the left- and right-hand sides shall be equivalent.

Atomic form of a unit and unit equivalence

The atomic form of an atomic or conversion unit is that unit.

Atomic units are equivalent if their local names refer to the same unit definition.

Conversion units are equivalent if their local names refer to the same unit definition, or they are synonyms.

The atomic form of a composite unit is obtained by replacing each composite unit in its definition by that unit's atomic form.

The atomic form of a unit is an algebraic expression of the form $\prod_{i=1}^{n} a_i^{e_i}$, where a_i is a unit name, no two a_i are equivalent, and e_i is a rational number.

If the atomic forms of two units are $\prod_{i=1}^{n} a_i^{e_i}$ and $\prod_{j=1}^{m} b_j^{e'_j}$, the units are equivalent if $m = n$, there is a one-to-one equivalence between a_i and b_j, and where a_i and b_j are equivalent, $e_i = e'_j$.

Unit conversion

Definition of a conversion unit defines an elemental generic function of the same name, with one real argument that is the unit that appears in the definition, and a real result with the specified units. The function evaluates the defining expression.

An elemental generic function that has the name and real result units of the unit that appears in the definition, and an argument that is the unit being defined, is also defined. The function evaluates the inverse of the defining expression. This is trivial: Because the unit definition is of the form $y = a \cdot x + b$, where y and x are unit names and a is nonzero, $x = (y - b)/a$.

Conversion in expressions only occurs were explicitly invoked. Conversion functions are automatically composed as necessary. Example:

```
K = 273.15
print *, FAHRENHEIT(K) ! prints 32.0
F = 32.0
print *, KELVIN(F) ! prints 273.15
```

Units as arguments and function results

Units of arguments and function results, whether they are abstract, and if they are abstract their relationship to other units, are characteristics of a procedure.

If a dummy argument has nonabstract units, corresponding actual arguments shall have equivalent units.

Nonabstract units of arguments participate in generic resolution.

If a function result variable has nonabstract units, the units of the function invocation are the units of the result variable.

Units as arguments and function results (cont.)

If a dummy argument has atomic abstract units, the corresponding actual argument can have any unit.

If a dummy argument has composite abstract units, the corresponding actual argument shall have units related to units of other actual arguments in the same way that the abstract units of dummy arguments corresponding to those other actual arguments are related.

If a function result variable has abstract units, the units of the function invocation are related to the units of the actual arguments in the same way as the units of the function result are related to the units of the dummy arguments.

Abstract units example

```
interface
  real function CBRT ( X ) ! cube root of X
   unit, abstract :: A, R = RATIONAL_POWER(A, 1, 3 )
   real, unit(A) :: X
   unit(R) :: CBRT
  end function CBRT
end interface

real, unit(CENTARE) :: B
B = CBRT(METER(2.0)**3) ** 2 !  B = CENTARE ( 4.0 )
```

Units of actual and dummy argument are METER^3.

Units of result of CBRT are $(\mathrm{METER}^3)^{\frac{1}{3}} = \mathrm{METER}$.

Units of RHS are $(\ (\ \mathrm{METER}^3\ )^{\frac{1}{3}}\ )^2 = \mathrm{METER}^2$, which are equivalent to CENTARE, the units of B.

This is impossible using a derived type with kind type parameters for exponents, or a different derived type for each unit.

Units in I/O

Units can be output. Units can be checked on input, and converted if necessary.

A U[*w*][.*s*] edit descriptor suffix is defined for D, E, EN, ES, F, or G edit descriptors.

Where a U[*w*][.*s*] suffix appears for output, the local unit name of the list item is output, using the same rules as for the A[*w*] edit descriptor, separated from the value by *s* spaces if .*s* appears, and one space otherwise. The item shall be real and not unitless.

For list-directed or namelist output, no unit is produced for a unitless item.

Units in I/O (cont.)

For list-directed or namelist output, If a scalar item has units other than unitless, the local name of the unit is produced, separated from the value by one space. If an array item has units other than unitless, the local name of the unit is produced only after the value of the first array element, separated from the value by one space.

If an output item is not unitless, it shall have a unit name. For example, a list item of `X**3` where `X` has units `METER` is prohibited, while `STERE(X**3)` is permitted (`STERE` in this case is a units confirmation function).

Where a U[*w*][.*s*] suffix appears for input, the list item shall be real and not unitless, and the unit name that appears in the input shall be the same as the local unit name of the list item, or related to it by a sequence of conversions. If the input unit is not the same as the list item's unit name, conversion is applied, as specified by the appropriate unit definitions.

Units in I/O (cont.)

For list-directed or namelist input of scalars that are not unitless, a unit name shall appear after the value and be separated from it by one or more spaces. The unit name that appears in the input shall be the same as the local unit name of the input item, or related to it by a sequence of conversions.

For list-directed or namelist input of arrays that are not unitless, a unit name shall appear after the first value, may appear after other values, and be separated from the preceding value by one or more spaces. If no unit name appears, the unit of the value is assumed to be the same as the previous one.

If the input unit is not the same as the list item's unit name, conversion is applied, as specified by the appropriate unit definitions.

Intrinsic functions

Intrinsic functions UNITLESS and RATIONAL_POWER are defined.

UNITLESS has a real argument with any units, and a real result of the same kind as the argument, without units.

RATIONAL_POWER has three arguments X, N and D, where X is real, and N and D are integer. If X is not unitless, N and D shall be constants. The result value is X raised to the rational power N/D. X has an abstract unit A. The units of the result are RATIONAL_POWER(A,N,D).

Intrinsic functions having real arguments and result type other than real have unitless arguments.

Intrinsic functions (cont.)

With the following exceptions, the units of real arguments of an intrinsic function with real result are the same abstract unit, and the units of the result are that abstract unit.

The arguments of ATAN2, or ATAN with two arguments, have the same abstract unit. The result is unitless.

Trigonometric functions have arguments with units either UNITLESS or RADIAN, and the results are unitless.

The arguments of DOT_PRODUCT, DPROD, and MATMUL have abstract units A and B, and the units of the result are A*B.

Where the arguments of MOD or MODULO are real, they have different abstract units, and the units of the result are those of the first actual argument.

Intrinsic functions (cont.)

The argument of RANDOM_NUMBER has an abstract unit.

The argument of SQRT has an abstract unit A and the unit of the result is A**(1/2).

The arguments and results of the following functions are unitless.

Hyperbolic functions	Inverse hyperbolic functions	Bessel functions
Error functions	EXP	FRACTION
GAMMA	LOG	LOG_GAMMA
LOG10	PRODUCT	RRSPACING
TRANSFER		

Inverse trigonometric functions other than ATAN2, or ATAN with two arguments