What is 1 kg + 0.1 m? Handling dimensions and units of measure in a program

Arjen Markus, Brad Richardson

September 15, 2021

Abstract

Most, if not all, programs that perform numerical computations of some kind or other need to deal with the units of measure in which their variables are expressed. Many solutions for the general problem have been proposed, but the problem is more difficult than it looks at first sight. In this article we aim to give insight in the various aspects and what the proposed solutions can and cannot do.

1 TODO

TODO: add SI units package to tex-file

TODO: more descriptions – comments as annotations are rather non-committal

TODO: examples in "Existing solutions" – also preprocessors as category

2 Introduction

In almost all programs that perform numerical computations the question arises whether the units of measure have been treated correctly. For instance: the sum of a variable that represents a mass and a variable that represents a length is undefined, yet for most computer languages, the variables would probably be simple floating-point numbers and it is the burden of the programmer to make sure that they are used in the right way, that is, two masses can be added and you can multiply or divide a mass by a length, but the sum or the difference has no meaning and so should not pop up in the source code.

Various solutions have been proposed for many different programming languages, but there does not seem to be a silver bullet (cf. Section 5). Examination of the publications on these solutions shows several classes of solutions:

- Variables in the source code are decorated with static features that represent the units of measure [Pucci and Schonberg(s.a.), Moy et al.(s.a.)Moy, Becker and Regnath, Richardson(2020), Snyder(2019)]. The burden is on the programmer

to provide these features. Depending on the language they are extensions of simple floating-point numbers or specific additions to the language itself, requiring support from the language tools (notably compilers and linkers).

- A preprocessor/static analyser is developed that helps identify violations of the arithmetic rules for units of measure [SimCon(2015), Contrastin et al.(2016)Contrastin, Rice and Danish]. Typically, the programmer annotates the variables via special comments, so that the analysis can be refined.
- Rather than a static checker in the form of a compiler or a preprocessor, all (relevant) variables are defined to have a dynamic set of dimensions and the checks are made at run-time [Petty(2001)]. This clearly impacts the performance, but makes for a light-weight solution, because the developer of the package can use the standard facilities offered by the programming language.

Surprisingly, however, while the publications discuss in more or less detail the techniques used and how the user can use the solutions to their benefit, none seem to discuss the characteristics of units of measure and how these play a role in the average program. In fact, the distinction between *dimension* and *unit* is seldom made explicitly. This article is meant to explore precisely these characteristics and how they influence the design of a programming solution.

3 Dimensions and units

First of all, we need to be clear about what a dimension is and what a unit is. In science and engineering we typically rely on the *Système International* for defining dimensions and units, but in finance or economics other systems are of importance. But let us begin with the SI units – or dimensions.

A dimension is the general aspect of some measure. Here is an arbitrary definition found on the Internet: "a measurable extent of a particular kind, such as length, breadth, depth, or height." The SI distinguishes seven basic dimensions: mass (M), length (L), absolute temperature (Θ), time (T), amount of substance (N), electric current (I) and luminous intensity (J). Dimensions can be combined, for instance a surface area would be a squared length – L^2, or a density would be mass divided by length to the power three – M/L^3.

Many solutions that help verifying that the program code does the right thing as far as units in the expressions are concerned limit themselves to *dimensional analysis*. Small wonder, because the *units of measure* in which the actual value of a quantity is expressed, are legion: common units for length are meter, centimeter, inch, foot, kilometer, mile and so on. Each has the dimension of length, but each is numerically different. So, whether the solution allows you to work with expressions like $1\ m + 10\ inch$ very much depends on whether it does *unit conversions* or not.

4 Design questions

The requirements we can deduce from available use cases (Section 6) and from general considerations can be divided into two categories. On the one hand we have requirements that are necessary for the feature to be at all useful, and on the other hand we have requirements that are nice to have but are not actually essential or might hinder some usages.

Here is an overview of both types:

Requirement 1 (Diagnostics and error reporting): It is essential that violations of the rules that gouvern units and dimensions be clearly identified in the source code. Suppose the program calls a routine that expects an acceleration as argument but gets a velocity. When this mistake can be detected *statically*, the reported error can pinpoint the exact location. If a variable receives its dimension/unit as a result of some calculation, it can be very hard to identify that location.

Error reporting should be as accurate as possible, both concerning the character and the location in the program.

Requirement 2 (Flexibility in units): The feature must be able to deal with different units for the same dimension. A classical example is the use of input in feet as the unit of length, where the program itself (or the library it uses) prefers meters. Similarly, it will be important to produce output in suitable units.

Requirement 3 (Flexibility in dimensions): The programmer should be able to define their own dimensions, not only units. This is important, because the *SI* dimensions are but a subset of the dimensions that are really in use. This could be within the context of physical and chemical applications – think of "milliliter oxygen per liter water" but also of financial or other dimensions.

Requirement 4 (Arrays with units): If the feature does not support arrays whose elements have different dimensions/units of measure, then certain use patterns are not possible. That may or may not be a breaking requirement for the feature.

Requirement 5 (Static or dynamic units for variables): If the feature allows variables to have different units, depending on where they are used in the program, this may hinder static analysis and it may also hide mistakes. If that is not allowed, variables will have to be declared to have certain units and this puts an extra burden on the programmer.

Requirement 6 (Static or dynamic units for function results and arguments): If the feature does not allow functions to take arguments with different units and produce results with units derived from the arguments, it will be difficult to write a generally applicable library that can handle units consistently. For instance, a library that solves differential equations will need to handle any conceivable units for the dependent and independent variables. Writing a version for each unit separately is prohibitively

expensive.

Requirement 7 (Performance): There should not be any noticeable effect on the performance of a program that handles units versus the same program that does not handle units explicitly, with a possible exception for input and output unit conversions. Of course, this requirement is met much easier by solutions that use static analysis than by solutions that check the units at run-time.

Requirement 8 (Programming effort): While explicitly declaring the units of variables in a program will save a lot of debugging effort, the extra effort should not get in the way of using the feature. This may mean that a predefined set of units be available, so that a programmer does not need to start from scratch for each new program. Declaring the unit for a variable is inevitable, but defining the units and possibly conversions between them ought to be a matter of supplying a suitable units library.

5 Existing solutions

A survey of existing solutions finds that library and language built-in solutions fall broadly into four categories. Each solution offers its own positive and negative aspects and trade-offs. We will explore the strengths and weaknesses of each from the perspectives of a library writer, an application developer, and an end user.

5.1 Define the units associated with a variable (i.e. meters, inches, etc.)

This is the solution commonly used when such a facility is built-in to a language. For example F# has such a facility [Anonymous(2020b)], as does Ada [Pucci and Schonberg(s.a.)]. The main benefit to such an approach is that once type-checking (or other static analysis) has been completed, the dimensional information is no longer needed, and the information can be ignored at run-time. This allows the possibility that such a solution incur no run-time overhead. There are implementations in this category which are not built into the language, for example the C++ library mp-units [Anonymous(2020a)], the Rust library Dimensioned [Rust(s.a.)], or the Nim library Unchained [Keller et al.(2021)Keller, Granström, Vindaar and Caillaud], which generally require more advanced templating or type system capabilities, and can begin to blur the lines between this and the next category of solutions.

From the perspective of a library writer, there are two stances that one might take. They could provide interfaces in only the units of their choice and be confident that they do not have any dimensional inconsistencies and incur no runtime overhead, or they could be quite frustrated that in order to provide an interface to a wide range of applications they must either duplicate their algorithms for all conceivably useful combinations of units, or work around the units system and lose the compile time safety

it provides. The proposal by [Snyder(2016)] solves this situation by explicitly allow so-called *abstract* units, units that take their identity from the arguments of the function or subroutine in which they are used.

An application developer will generally find this solution quite agreeable. It provides compile-time guarantees that their dimensional analysis is correct. If a library provides interfaces in the units they desire, they can be confident that the library will not incur unnecessary run-time overhead, and if it does not, the places in which they must do explicit unit conversions will usually be fairly limited in scope. It is also a natural extension of how many application developers (usually domain experts) think about their solutions, i.e. that the numbers in their formulas may change, but the units do not.

End users of some applications may find this solution somewhat inflexible. It may be inconvenient for them to input values to the program in the units dictated by the application developer, but it would also be quite difficult for the application to deal with values input in different units, as it is impossible to declare a variable that does not know its units until run-time. Similarly it would be difficult for the program to allow the user to request outputs in different units.

Defining the units associated with a variable can provide a large amount of compile-time checking, incurs little to no run-time overhead, and can be a natural solution for application developers. However, it can often be quite inflexible in ways that make development and maintenance difficult for library writers, and can result in programs with inconvenient input specifications. Solutions of this kind are therefore most suitable for prototype development, but can be a long term maintenance burden for larger scale, long lived applications.

5.2 Define the dimension of a variable (i.e. length, force, etc.)

This category of solutions is not taken as frequently as the other two. It is usually implemented as a library, but can be an expensive library to develop and maintain. The development expense comes from needing to define types and units for all possible dimensions, as well as the mathematical operations allowed between them. The main benefit of this approach is that it provides additional flexibility sufficient for most library writers and application end users, without a large amount of run-time overhead. The run-time overhead comes in the form of either keeping track of a value's units, or implicitly performing unit conversions at the time of value creation. Example libraries that would fit this category are quaff [Richardson(2020)] and UnitsNet [Larsen(2021)].

From the perspective of a library writer, this solution strikes a nice balance. It relieves most, if not all, of the burden of providing interfaces to their algorithms for all conceivably useful combinations of units without needing to work around the units system or the compile time safety it provides. It does not completely eliminate duplication for algorithms that work for different dimensions however, for example linear solvers which could be used to solve for stress or temperature depending on the

dimensions of the inputs.

The abstracts units proposed by [Snyder(2016)] again solve this situation explicitly.

From the perspective of an application developer, this solution generally strikes the ideal balance. It still provides compile-time guarantees that their dimensional analysis is correct, does not incur significant run-time overhead, and makes it easy to provide end users the flexibility of providing inputs and requesting outputs in whatever units are most convenient. However, this solution will not be sufficient for more general purpose expression evaluators or calculators, because even the dimension of a value will not be known until run-time.

End users will find this solution sufficient for most applications. It will be easy for application developers to allow them to provide inputs in whatever units are most convenient, and request outputs in whatever units they desire. However, general expression evaluation and calculators with units will still not be feasible with this solution.

5.3 Track the dimensions of variables at run time

This category of solution is frequently used in library implementations. The main benefit of the approach is its high degree of flexibility. However, it generally comes at the expense of a high amount of run-time overhead, because all dimensions and units are tracked, checked, and simplified at run-time. Example libraries that fit this category are more common in interpreted languages, for example the Python library pint [Grecco(2021)] or the Ruby library Ruby Units [Olbrich(2021)].

From the perspective of a library writer, this solution eliminates any extra development and maintenance overhead, but still ensures that users of the library have consistent units. Unfortunately, it comes at the expense of significant run-time overhead.

From the perspective of an application developer, it practically eliminates all of the units checking they were after. Any errors in the units will be found, but only at run-time, possibly far from the cause of the error. It does however allow for the flexiblity to develop applications which do not even know the dimensionality of their inputs until run-time.

From the perspective of an end user, this solution is necessary for applications that perform general purpose calculations and expression evaluation with units.

5.4 Preprocessing the source code

The fourth category of solutions consists of preprocessors: these tools analyse the source code and deduce the use of units and dimensions from the actual source. Two such tools are: *fpt* [SimCon(2015)] and *Camfort* [Contrastin et al.(2016)Contrastin, Rice and Danish]. From the point of view of a library writer and an application developer, the use is very much the same. Via an extra step in the build process they check the source code (in much the same way as a compiler does) and report anything that is not correct, that is, violations against the rules by which values with units can be used. With respect to libraries, the same problem as with

the first category occurs, there is no mechanism to support different unit systems automatically.

To help the tool the programmer can annotate the variables with unit information, mostly via specific comments to the declaration. This means very little support from the programming language is needed, unlike in the first category, but the comments do not form an integral part of the declaration and thus may get forgotten about.

Here is an example taken from [Contrastin et al.(2016)Contrastin, Rice and Danish] (the *fpt* tool does not support such annotations):

```
program energy
  != unit kg :: mass
  != unit m/s**2 :: gravity
  real, parameter :: mass = 3.00, gravity = 9.81, height = 4.20
  != unit kg m**2/s**2
  real :: potential_energy

  potential_energy = mass * gravity * height
end program energy
```

In this fragment the unit of the parameter `height` was not specified and instead it is inferred from the usage in the calculation of the potential energy.

The end user, though, does not notice anything – there is no mechanism provided to deal with different units than the ones built in to the application in the first. That burden lies with the programmer.

6 Use cases

Within a typical program, be it engineering or economical, you will find different usage patterns and the question is which of these can and cannot be supported by the current set of solutions:

Input/output – flexibility in units

The user should be able to provide the data in a convenient form, including a convenient unit. The program may work in meters, but to specify the size of an atom in meters is not the most friendly possibility. Likewise for output.

As an example, let's consider a hypothetical calculator for determining average speed. Given a list of distances covered and times taken for each, we can calculate an average speed. Such a program would be valid for calculating the average speed of anything from the random walk of a particle in a fluid, to the average speed of a contestant on an obstacle course, to the average speed of an airplane on it's daily route. The units we would like to input and receive from such a program are different, but the algorithm is the same. The example program might look something like the following, with different input files as shown after.

```
program speed_calculator
  use units ! some hypothetical module providing the types and operators used below
```

```
  implicit none
  integer :: num_legs
  type(length_unit) :: distance_units
  real, allocatable :: distances(:)
  type(time_unit) :: time_units
  real, allocatable :: times(:)
  type(speed) :: average_speed
  type(speed_unit) :: output_units
  ! determine the input file name and open it
  read(input_file, *) num_legs, distance_units, time_units, output_units
  allocate(distances(num_legs), times(num_legs))
  read(input_file, *) distances
  read(input_file, *) times
  average_speed = sum(distances*distance_units) / sum(times*time_units)
  print *, average_speed.in.output_units
end program
```

Then the input files for calculating a particle's average speed could look like,

```
5, "nm", "ns", "m/s"
11.4, 12.2, 9.7, 13.1, 12.5
8.6, 7.9, 9.1, 10.2, 9.8
```

for calculating a runner's average speed could look like,

```
6, "ft", "s", "mile/hr"
11.4, 12.2, 9.7, 13.1, 12.5, 8.4
8.6, 7.9, 9.1, 10.2, 9.8, 10.3
```

and for calculating a plane's average speed could look like,

```
4, "mile", "hr", "km/d"
275, 315, 395, 245
1.5, 2.5, 2.75, 2.0
```

In this case one can see how specifying a variable's units is too restrictive, but not specifying it's dimensionality leaves the code too vague.

General calculator – flexibility in units and dimensions

As an extension of the requirement on flexible input and output, we may also want the program to support interactive use, as in a general calculator. Feed it data with different units of length and it prints the sum in the unit of your choice.

Typical expressions – flexibility in units and dimensions, static or dynamic units for function results and arguments

Expressions in which the variables and constants have dimensions and consequently units of measure come in various categories. Here are a few:

Mathematical formulae:
Consider the area of a circle:

$$A = \pi R^2 \tag{1}$$

In this formula the radius R has the dimension L (length) and the area A has the dimension L^2 (length squared). The constant π is essentially unitless. To deal with this formula, the system must either recognise that dimensions can be multiplied, forming new combinations, or it must be told that both L and L^2 are allowed.

But if you were to calculate the mass of a cylinder from its radius R, height H and the density of the material ρ via the slightly more complicated formula:

$$M = \rho \pi R^2 L \tag{2}$$

the system now has to deal with a dimension ML^{-3} for the density ρ as well as L^2 and L. Depending on the way the formula is programmed and the way the compiler (or the executable program) handles this expression, we may encounter the following intermediate results:

Density multiplied by length	$ML^{-3} \cdot L$
Density multiplied by length squared	$ML^{-3} \cdot L^2$
Density multiplied by length cubed	$ML^{-3} \cdot L^3$
Length squared multiplied by length	$L^2 \cdot L$
Density times length squared multiplied by length	$(ML^{-1}) \cdot L$

and possibly others. In short: expressions that involve variables and constants with units of measure can generate new combinations of dimensions. With a naïve implementation this can easily result in tens of thousands combinations [Snyder(2016)].

An obvious seeming solution is to define a specific type or an attribute representing the desired unit or dimension:

```
type(length_dim)  :: radius, length
type(density_dim) :: density
type(mass_dim)    :: mass
real, parameter   :: pi = 3.1415926 ! Approximately

mass = pi * density * radius ** 2 * length
```

At least the operations *multiply* and *exponentiation* should be overloaded. For the above formula you would also need to represent *density* * *length* because a *density* gets multiplied by a *length*, as well as any of the other combinations.

While in a given program there will be a finite number of combinations, some support from the compiler or analyser would help: the tool in question would then gather all the units/dimensions for each term, rather than for each combination separately and create the final unit/dimension. This is essentially what the package by Petty does [Petty(2001)]: for each variable of the type `type(preal)` the exponents of the basic SI dimensions are traced, so that the multiplication of a *density* [M L^{-3}] and a length [L]

results in an intermediate variable with dimension [M L^{-2}]. Thus there is only a single user-defined type, which stores the exponents of the seven fundamental units.

It may seem that the dimensions are raised to integer powers, but rational powers can also arise, as in the period T for a harmonic pendulum:

$$T = 2\pi\sqrt{\frac{l}{g}} \tag{3}$$

Empirical formulae:
While mathematical formulae are usually rather "crisp", with clear definitions of exponents and constants, that is not so for empirical or semi-empirical formulae. Here is an example, a relation between the water level h and the flow rate Q in some unspecified river:

$$Q = 5.3 \cdot h^{2.7} \tag{4}$$

The water level must be expressed in meters relative to a reference level and the flow rate is in m^3/s. If you were to express the water level in fathoms instead, the result would be completely wrong. It would even be wrong if you expressed the water level relative to another reference level than implied in the formula.

This type of formula can only be used with a lot of background information, information that may be found in the comments or the documentation or simply in the heads of the users of the program, because they have been working with it for years.

A correct way to write this formula is:

$$Q = Q_0 \cdot \left(\frac{h}{h_{norm}}\right)^{2.7} \tag{5}$$

where $Q_0 = 5.3m^3/s$ and $h_{norm} = 1\ m$, a normalisation factor, rendering the value to exponentiated dimensionless. Still, this leaves implicit the reference level against which the water level is measured.

Chemical formulae:
According to the Système Internationale (SI) amounts of a substance are to be expressed in *moles*, the number of molecules or atoms or ions divided by Avogrado's number. This is a very convenient way to do calculations regarding chemical reactions. For instance, this equilibrium between CO_2 and HCO_3^- in water:

$$CO_2 + H_2O \rightleftharpoons HCO_3^- + H^+ \tag{6}$$

The reaction constant is (since the amount of water normally exceeds the amounts of the other substances by many orders of magnitude, it is essentially constant and can be absorbed in the reaction constant):

$$K = \frac{[CO_2]}{[HCO_3^-][H^+]} \rightarrow \text{mol}^{-1} \tag{7}$$

but there are three molecules/ions involved, so actually three different molar units.

Other units that need to be considered are:

- Using *ml/l* as the unit for gas dissolved in water. In particular, the concentration of dissolved oxygen is sometimes expressed in this unit. It should be interpreted as "milliliter of dissolved oxygen per liter of water".
- The concentration of salt in sea water, salinity, is expressed in units like *ppt* (parts per thousand), *psu* (practical salinity units) or *1e-3*. This latter is the official unit prescribed by the "CF conventions" organisation [Anonymous(2021)]. Since it looks very much like a plain number, supporting something like that requires extra care, also on the part of the programmer, one would think.

Financial/economic expressions:
As engineers with physical or chemical background we are used to thinking in terms of the SI dimensions and units, or perhaps imperial units like inches and ounces. In other areas of computation very different dimensions are involved. Consider the gross domestic product, GDP. It is expressed as the amount of currency per year. And the exchange rates of dollars to euros or yens involve the ratio of two currencies. Formally, that may be a dimensionless value, but you should not use the wrong way around. A unit checking mechanism therefore might be required to distinguish the exchange rate dollar/euro from the rate euro/dollar.

But even seemingly dimensionless numbers require some care: the number of inhabitants of a city is a mere count without a dimension, but it does not make sense to add a bare number like 100 to it, unless that number also represents some number of people.

And what to think of percentages? Combining two percentages α and β should not be done by adding them, but by a formula like:

$$\gamma = ((1 + \frac{\alpha}{100}) \cdot (1 + \frac{\beta}{100}) - 1) \cdot 100\% \tag{8}$$

Dimensional analysis – diagnostics and error reporting

The very least we may require from a solution for the problem we consider here is that it checks the dimensional correctness of expressions. This means that the arithmetic for dimensions is checked and verified [Wikipedia(2021)]:

- Multiplication and division is possible with any combination of dimensions, but leads to a new combination.
- Addition and subtraction is possible only when the dimensions of the terms are identical.
- Exponentiation is – in general – limited to rational powers. One may argue whether, like in the case of empirical formulae, exponentiation to some arbitrary power implicitly means the unit is ignored.
- Elementary functions as *sin* and *log* do not take arguments which have a dimension, though for the trigonometric functions they are often interpreted in *radians* and alternatives may exist for arguments expressed in *degrees*.

Things become much more complicated if we also require conversion of units be taken into account. Quite apart from the vast number of units that is in use for the various basic dimensions or the combined dimensions that are derived from them, some units of measure do not adhere to the above general rules.

The first example is units of temperature other than *kelvin*. Suppose you were to add a temperature of $10°C$ and $20°C$ – or the same values in degrees Fahrenheit. The naïve result is $30°C$, or about $303\ K$, but if this was an allowable operation, we could also convert the two terms first to kelvin and then the result would be quite different: $283\ K + 293\ K = 776\ K$. You can, however, subtract two temperatures and you can add a temperature and a temperature difference.

As a second example, take the *decibel*, used among other things as a unit of sound. It is defined as ten times the logarithm of the ratio of two values. Adding two values with the unit of decibel is quite ambiguous, as can be seen by writing out the definitions:

$$dB_a = 10\ \log\Big(\frac{V_1}{V_2}\Big) \tag{9}$$

$$dB_b = 10\ \log\Big(\frac{V_3}{V_4}\Big) \tag{10}$$

$$dB_{a+b} = 10\ \log\Big(\frac{V_1 \cdot V_3}{V_2 \cdot V_4}\Big) \tag{11}$$

The interpretation becomes awkward, unless there is a relation between the various values, like V_2 identical to V_3, in which case the summation means an *increase* of the original value.

Ideally, a programming solution for checking the dimensions (and units) used in a program would take all such considerations into account.

State vector – arrays with units

Some of the solutions that have been suggested define a variable to have a fixed unit or dimension. But that presents a problem if you have an array of which each element can have a different dimension. Here is a simple example, a (forced) harmonic oscillator:

$$\frac{d^2x}{dt^2} - kx = F(t) \tag{12}$$

The system has two state variables, the position (L) and the velocity (LT^{-1}). It is natural to put these two state variables into an array and solve the system of equations via a standard ODE solver:

```
real :: x(2)

x(1) = 1.0   ! Start position in [m]
x(2) = 0.0   ! Start velocity in [m/s]
dt   = 1.0   ! Time step in [s]

do i = 0,times
```

```
    t = i * dt
    call solve( x, t, dt, func )
    write(*,*) t, x
enddo
```

A library such as described in [GrantPettyPhysUnits] provides an easy solution:

```
use SI_units ! this module uses "physunits", so no need to get that explicitly

type(preal) :: x(2), t, dt

x(1) = 1.0 * u_meter             ! Start position in [m]
x(2) = 0.0 * u_meter / u_second  ! Start velocity in [m/s]
t    = 0.0 * u_second            ! Time in [s]
dt   = 1.0 * u_second            ! Time step in [s]

do i = 0,times
    t = i * dt
    call solve( x, t, dt, func )
    write(*,*) real(t), real(x)
enddo
```

To preserve the unit/dimension information the `solve` and `func` subroutine and function must adopt the same approach.

Dynamic units of measure – static or dynamic units for variables

Another question that arises in every day programming practice is whether dimensions (or units) are static properties of a variable or not. [Farrimond and Collins(s.a.)] explicitly consider this possibility, as illustrated by the following code fragment:

```
! Sort the tables by weight
change_f = .TRUE.
DO WHILE (change_f)
    DO i = 2, n
        IF (weight(i) < weight(i-1)) THEN
            temp = weight( i )            !<-- unit of temp: mass
            weight( i ) = weight( i-1 )
            weight( i-1 ) = temp
            temp = height( i )            !<-- unit of temp: length
            height( i ) = height( i-1 )
            height( i-1 ) = temp
            change_f = .TRUE.
        ENDIF
    ENDDO
ENDDO
```

The variable `temp` is used first as a weight (M) and then as a height (L). If the dimension of each variable were a static property – which

Table 1: Overview of the various possible solutions in relation to the requirements

Requirement	*Static definition* [Snyder(2016)]	*Library approach* [Richardson(2020)]	*Dynamic checking* [Petty(s.a.)]	*Preprocessor* [SimCon(2015)] [Contrastin et al.(2016)C
Diagnostics (1)	Exact location	Possibly later	Later	Exact location
Flexible units (2)	Yes	Yes	No	Yes
Flexible dimensions (3)	Yes	Yes	No	Yes
Arrays with units (4)	No	No	Yes	No (difficult)
Units of variables (5)	Static	Static	Dynamic	Dynamic (lifetime)
Units of arguments (6)	Derived	Static (?)	Dynamic	Derived
Performance (7)	Compile-time	Compile-time	Affected	Compile-time
Programming effort (8)	Medium	Large	Medium	Medium

allows checking at compile-time – then this usage pattern would be prohibited. In the proposal by V. Snyder [Snyder(2016), Snyder(2019)] a similar situation is described: here it is suggested to allow dimensionally polymorphic functions that inherit the dimension of the result from their arguments. One example, mentioned in [SimCon(2015)], is the *maximum* function – all of its arguments should have the same dimension (unit) and the result will be that dimension (or, more precisely, unit).

7 Suitability of the existing solutions

Given the wide range of usage patterns and the requirements that can be distilled from them, what solutions fit best? To answer this question let us examine how the requirements are met. We concentrate on packages suitable for use with Fortran programs.

TODO: Refine the table

TODO: Extend the text

The solutions that facilitate static definitions of the variables require support in one way or another from the programming language. The proposal for Fortran [Snyder(2016)] relies on a new keyword `unit` and the associated semantics. This means that checking can be done fully or almost fully at compile-time, thus reducing effects on the performance. Via the mechanism of *abstract* units it allows functions and subroutines to be generic in their way of handling units. The only use cases/requirements that are not supported are those where the variable should change its unit or where the variable represents an array of different quantities, each with its own unit.

The preprocessing solutions are perhaps the most flexible, as they do not require built-in language support and do support variables whose units depend on the context. However, they are commonly implemented as separate tools and thus complicate the build process.

For library solutions like [Richardson(2020)] supporting a particular

set of units means that suitable unit conversion routines are needed. While this is not impossible, and in fact it adds to the flexibility of the package, it does represent a significant burden, if these routines have not been defined already. This is the reason for marking the programming effort as "large".

In all cases the burden for specifying the correct units is on the programmer, even if some tools can deduce the units from the source code but the rewards for using consistent and verifiable units far outweigh that burden.

References

[Anonymous(2020a)] Anonymous, 2020a. mp-units: A units library for C++. URL: https://github.com/mpusz/units.

[Anonymous(2020b)] Anonymous, 2020b. Units of measure. URL: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/units-of-measure.

[Anonymous(2021)] Anonymous, 2021. CF Standard Name Table. URL: http://cfconventions.org/Data/cf-standard-names/77/build/cf-standard-name-table.html.

[Contrastin et al.(2016)Contrastin, Rice and Danish] Contrastin, M., Rice, A., Danish, M., 2016. Units-of-measure correctness in fortran programs. Computing in Science , 102–107URL: https://www.cl.cam.ac.uk/~acr31/pubs/contrastin-units.pdf.

[Farrimond and Collins(s.a.)] Farrimond, B., Collins, J., s.a. Dimensional inference using symbol lives. URL: http://www.simconglobal.com/farrimond_and_collins_2007_dimensional_inference_using_symbol_lives.pdf.

[Grecco(2021)] Grecco, H.E., 2021. Pint: makes units easy. URL: https://pint.readthedocs.io/en/stable/.

[Keller et al.(2021)Keller, Granström, Vindaar and Caillaud] Keller, F., Granström, H., Vindaar, Caillaud, R., 2021. Unchain - compile time only units checking. URL: https://github.com/SciNim/Unchained.

[Larsen(2021)] Larsen, A.G., 2021. Units.net. URL: https://github.com/angularsen/UnitsNet.

[Moy et al.(s.a.)Moy, Becker and Regnath] Moy, Y., Becker, M., Regnath, E., s.a. Physical units pass the generic test. URL: https://blog.adacore.com/physical-units-pass-the-generic-test.

[Olbrich(2021)] Olbrich, K.C., 2021. Ruby units. URL: http://github.com/olbrich/ruby-units.

[Petty(s.a.)] Petty, G., s.a. Physunits module. URL: http://rime.aos.wisc.edu/gpetty/?page_id=684.

[Petty(2001)] Petty, G.W., 2001. Automated computation and consistency checking of physical dimensions and units in scientific programs. Software practice and experience 31, 1067–076. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.401, doi:10.1002/spe.401.

[Pucci and Schonberg(s.a.)] Pucci, V., Schonberg, E., s.a. Implementation of a simple dimensionality checking system in Ada 2012. URL: https://blog.adacore.com/uploads/dc.pdf.

[Richardson(2020)] Richardson, B., 2020. quaff - quantities for Fortran. make math with units more convenient. URL: https://gitlab.com/everythingfunctional/quaff.

[Rust(s.a.)] Rust, s.a. Crate dimensioned. URL: https://docs.rs/dimensioned/0.7.0/dimensioned/.

[SimCon(2015)] SimCon, 2015. Research - physical units and dimensions. URL: http://www.simconglobal.com/units_and_dimensions.html.

[Snyder(2016)] Snyder, V., 2016. Units of measure for numerical quantities URL: http://vandyke.mynetgear.com/Fortran/Units-TR-19.pdf.

[Snyder(2019)] Snyder, V., 2019. Physical and engineering units of measure. URL: https://github.com/j3-fortran/fortran_proposals/issues/50.

[Wikipedia(2021)] Wikipedia, 2021. Dimensional analysis. URL: https://en.wikipedia.org/wiki/Dimensional_analysis.