Editors: Konrad Hinsen, konrad.hinsen@cnrs-orleans.fr | Matthew Turk, matthewturk@gmail.com

Units-of-Measure Correctness in Fortran Programs

Mistral Contrastin, Andrew Rice, and Matthew Danish | University of Cambridge
Dominic Orchard | Imperial College London

Much of mathematics' use in science revolves around measurements of physical quantities, both abstractly and concretely. Such measurements are naturally classified by their *dimension*, that is, whether the measurement is of distance, energy, time, and so on. Dimensionality is further refined by a measurement's units-of-measure (or units, for short), such as meters, Joules, seconds, and so on. Units-of-measure distinguish magnitudes from each other, giving additional meaning, but despite their extensive use in the practice of science, units-of-measure don't see widespread adoption in tools for scientific computing. Here, we demonstrate how our freely available, open source tool, CamFort, provides a low-effort and automated way of detecting mismatched units-of-measure in code. This feature of CamFort is an example of a lightweight, nonbinding specification and analysis tool that can help find bugs in programs before they strike. We hope that, in general, these kinds of program analysis tools will become more widely used by scientists to save time and reduce grief during the development process, as well as increase confidence in results of numerical models.

Ensuring the consistent use of units is an important sanity check in scientific computing. For example, adding a value in kilograms to a value in liters is a nonsensical operation. As trivial as that error may seem, in large enough projects, such errors can go unnoticed by human eyes. One such famous incident was the Mars Climate Orbiter spacecraft, which disintegrated in the Martian atmosphere because one part of the critical mission-control software provided values in imperial units whereas another part expected values

 1521-9615/16/$33.00 © 2016 IEEE Copublished by the IEEE CS and the AIP

in metric units.[1] This seemingly simple error cost nearly US$330 million and delayed the scientific mission of Martian exploration.

That isn't to say that scientific programmers neglect units in their source code. In fact, we see many pieces of scientific computing code using comments to record information for major variables, equations, and functions. These comments, however, are of limited efficacy because developers must manually check the consistency of every variable's units with respect to their use—and repeat this process every time a change is made. What if it were possible to use these same comments to automatically check the entire system's consistency?

Checking the consistency of units-of-measure is akin to type checking, and solutions to the problem have been developed on this basis. Type checking systems ensure that certain illegal operations, such as dividing a Boolean value by a string value, don't occur. This class of illegal operations is similar in nature to that of adding a kilogram to a liter (an inherently meaningless statement) and thus should be automatically prohibited by the programming language.

Arguably, the most established language in scientific computing is Fortran, which contains a large body of libraries and expertise. To date, however, Fortran has lacked a system for checking units. CamFort is a multifaceted tool that provides various kinds of analysis, verification, and refactoring techniques for Fortran code aimed at scientific computing. CamFort operates on Fortran programs compatible with at least the Fortran 66 standard or newer. In this article, we demonstrate CamFort's automated units-of-measure consistency checking system. We designed this system with existing codebases in mind: CamFort minimizes the effort to introduce units to existing code, uses comments to record annotations (short specifications on variable declarations), and uses only static checks to validate units and so imposes no performance overhead at runtime.

The rest of this article consists of a brief introduction to computing total mechanical energy, followed by its implementation in Fortran 90. We then give a tutorial on how CamFort can be used to add units-of-measure specifications, check unit consistency, and eliminate units-of-measure bugs, using the total mechanical energy program as an example.

Example: Total Mechanical Energy Computation

Total mechanical energy is the sum of a point object's kinetic and potential energy. The relevant equations to calculate total mechanical energy for a point object in free fall are as follows:

$$E_{potential} = mass \times gravity \times height,$$

$$E_{kinetic} = 1/2\ mass \times velocity^2, \text{ and}$$

$$E_{mechanical} = E_{potential} + E_{kinetic}.$$

An example Fortran program using these formula might be as follows:

```
1 program energy
2     real, parameter :: mass = 3.00, gravity =
      9.81, height = 4.20
```

Units Support in Other Languages

Tools for checking units-of-measure consistency in modern languages can be broadly categorized as either static or dynamic analyses.

Static analysis of units-of-measure means that consistency is guaranteed at compilation time. Hence, if the tool statically confirms that units are used consistently, with respect to annotations in the source code, then unit consistency won't be violated by any execution path the program may take. For example, F# allows variables to be annotated and statically checked for units-of-measure consistency. C++ programmers can use the Boost Units library, which uses C++ templates to statically ensure that units are used appropriately. The Osprey project provides another solution with an external tool.[1] Haskell has various forms of units-of-measure typing provided internally by building on Haskell's rich type system.[2,3]

The other option is *dynamic analysis*, in which consistent use of units is checked when the program runs. This means that no guarantees can be made before running the program. Instead, safety violations are caught as they happen. This has the additional downside of adding runtime overhead. Python's Pint library is an example of a dynamic units-of-measure analysis.

Libraries and tools also exist for assisting with the management or conversion of units. The udunit C library provides support for converting between units and has a large database of standard units-of-measure. This library is also available to R programmers through the udunits2 package. The difference here is that these libraries aim to provide routines for helping with the conversion of units rather than for detecting their misuse.

References

1. L. Jiang and Z. Su, "Osprey: A Practical Type System for Validating Dimensional Unit Correctness of C Programs," *Proc. Int'l Conf. Software Eng.* (ICSE), 2006, pp. 262–271.
2. A. Gundry, "A Typechecker Plugin for Units of Measure," *Proc. ACM SIGPLAN Symp. Haskell*, 2015, pp. 11–22.
3. T. Muranushi and R.A. Eisenberg, "Experience Report: Type-Checking Polymorphic Units for Astrophysics Research in Haskell," *Proc. ACM SIGPLAN Symp. Haskell*, 2014, pp. 31–38.

```
3     real, parameter :: half = 0.5, velocity =
      4.00
4     real :: kinetic_energy, potential_energy,
      total_mechanical_energy

5
6     potential_energy = mass * gravity * height
7     kinetic_energy = half * mass * (velocity**2)
8
9     total_mechanical_energy = potential_energy +
      kinetic_energy
10 end program energy
```

This program calculates the total mechanical energy for a point object of 3 kg mass in free fall, traveling at 4 meters per second toward the Earth from a height of 4.2 meters. The resulting total mechanical energy is 148 Joules.

Building a Unit Annotated Program for the Task

It's easy to integrate CamFort with an existing codebase. CamFort can first analyze a codebase and report a minimal set of variables that can be annotated by the programmer to gain the maximum amount of unit information in the rest of the program. Once the programmer inserts the initial annotations, further unit annotations can be added as developers write more code. In our experiments, a typical program requires roughly only 18 percent of its declared variables to be annotated before CamFort can automatically infer the rest.[2]

We consider first a subset of the mechanical energy program for just the potential energy:

```
1 program energy
2     real, parameter :: mass = 3.00, gravity =
      9.81, height = 4.20
3     real :: potential_energy
4
5     potential_energy = mass * gravity * height
6 end program energy
```

We find the minimal set of variables to annotate by invoking the `criticalUnits` analysis feature of CamFort (passing the source directory as an argument):

```
$ camfort criticalUnits energy_src
     Inferring critical variables for units
inference in directory "energy_src"
     energy_src/energy.f90: Critical variables:
potential_energy,gravity,mass
```

The output from CamFort tells us we need to annotate the variables `potential_energy`, `gravity`, and `mass`. CamFort can then deduce the units for other variables in the program. Annotations are similar to Fortran 90 variable declaration syntax but inside of a comment (that is, preceded by an exclamation point `!`). The syntax is of the form

```
!= unit <unit_name> [:: variable_name]
```

Starting a comment with an exclamation point `!` has been valid only since Fortran 90, so for earlier Fortran standards, we replace `!` with `c` or `C`.

The advantage of using comments for program annotations is that they don't interfere with compilation in any way. This means CamFort can be added or removed from the development process at any point with no need to alter the source code. Possible unit annotations for the above program are then

```
1 program energy
2     != unit kg :: mass
3     != unit m/s**2 :: gravity
4     real, parameter :: mass = 3.00, gravity =
      9.81, height = 4.20
5     != unit kg m**2/s**2
6     real :: potential_energy
7
8     potential_energy = mass * gravity * height
9 end program energy
```

This highlights some properties of unit annotations:

- The declaration of unit names is implicit (lines 2, 3, and 5 implicitly declare noncompound units `kg`, `m`, and `s` for use in the program).
- A variable name in an annotation is optional (line 5) in which case any variable declarations after the annotation are assigned that unit (`potential_energy` on line 6 in this case).
- Annotation comments must always precede the related variable declaration (line 3) but can be separated from a variable declaration by other unit annotations (line 2 separated from the `mass` variable declaration (line 4) by the line 3 annotation).

We can now apply CamFort's automatic units-of-measure inference, which will check the consistency of the units in the program and insert any inferred units—in this case, for `height`, which was unspecified. This is invoked by the `units` feature (which requires an output directory to be specified for the updated source files):

```
$ camfort units energy_src energy2
     Inferring units for "energy_src"
     energy_src/energy.f90: Added 1 unit
       annotation: m
     energy_src/energy.f90: Checked/inferred 4
       user variables
     Writing refactored files to directory: energy2/
     Writing energy2/energy.f90
```

CamFort generates a modified program in the output directory. Source code is changed only to insert the annotation comment, while the rest of the source code lines—including spacing and indentation—are preserved. In the resulting program, we can see on line 4 that CamFort has inferred that `height` must have units `m` to be consistent and has thus inserted this annotation:

```
1 program energy
2     != unit kg :: mass
3     != unit m/s**2 :: gravity
4     != unit m :: height
5     real, parameter :: mass = 3.00, gravity =
      9.81, height = 4.20
6     != unit kg m**2/s**2 :: potential_energy
7     real :: potential_energy
8
9     potential_energy = mass * gravity * height
10 end program energy
```

We can extend the program to compute the total mechanical energy and demonstrate how a unit error is caught by CamFort:

```
1 program energy
2     != unit kg :: mass
3     != unit m/s**2 :: gravity
4     != unit m :: height
5     real, parameter :: mass = 3.00, gravity =
      9.81, height = 4.20
6     != unit kg m**2/s**2 :: potential_energy
7     real :: potential_energy
8
9     != unit 1 :: half
10    != unit m/s :: velocity
11    real, parameter :: half = 0.5, velocity = 4.00
12    real :: kinetic_energy, total_energy
13
14    potential_energy = mass * gravity * height
15    kinetic_energy = half * mass * velocity
16
17    total_energy = potential_energy + kinetic_energy
18 end program energy
```

Note the use of the special unit `1` for the variable `half`. When `1` is used for type annotation, it signifies a unitless (or scalar) quantity.

Our implementation of kinetic energy contains a programming error: velocity should have been squared on line 15. CamFort detects this bug as a unit error. If we attempt to run either the `units` or `criticalUnits` analysis, CamFort will display a warning that the system is inconsistent because we're adding `potential_energy` to `kinetic_energy`, which have different units in the existing implementation:

```
$ camfort units energy energy_out
     Inferring units for "energy"
     energy/energy.f90: inconsistent units of
       measure:
           line 17: cannot match units 'kg
             m**2/s**2' and 'kg m/s'
  energy/energy.f90: checked/inferred 8 user
    variables
```

The fix is thus to square the `velocity`. For further illustration, we do this with a user-defined squaring function, rather than just the built-in `**` operator:

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

Unitless vs. Polymorphic Units

A unitless value is a scalar quantity. Some operators require a unitless value—for example, exponentiation requires that the exponent is a unitless quantity (it doesn't make sense to raise a value to the power of 2 meters). For other operators, a unitless operand gives flexibility: it's valid to multiply a value of any unit by a unitless scaling factor.

A polymorphic unit represents a generalization over all units—for example, the `abs` intrinsic function takes a number and returns its absolute (positive) value. This function can be applied to a value of any unit and is thus described as being *polymorphic* in its unit. Furthermore, if the input is of some unit α, then the output value is of the same unit α.

Almost all constant values must have a particular unit (perhaps inferred by CamFort), or they must be unitless. The exception to this rule is zero, which can be treated polymorphically in its unit. To see why this is the case, consider the addition operation: adding zero to any value preserves the unit of that value because zero is the additive identity, that is $x + 0 = x$ for all x. However, it doesn't make sense to add 1 (or any nonzero constant) to any other value unless the units match exactly. In the former case, we can declare a zero constant and safely add it to *any* value with any unit. In the latter case, addition of a nonzero constant with a *different* unit would cause units-of-measure inconsistency.

Although adding unit annotations is an additional task for the programmer, the overall effort is reduced (less time debugging) and programmers can be more confident in their code's correctness.

```
7     real :: potential_energy
8
9     != unit 1 :: half
10    != unit m/s :: velocity
11    real, parameter :: half = 0.5, velocity = 4.00
12    != unit kg m**2 / s**2
13    real :: kinetic_energy, total_energy
14
15    potential_energy = mass * gravity * height
16    kinetic_energy = half * mass * square(velocity)
17
18    total_energy = potential_energy + kinetic_energy
19
20    contains
21
22    real function square(x)
23          real x
24          square = x * x
25    end function square
26 end program energy
```

The user-defined `square` function is inferred to be polymorphic in its units due to the lack of any annotations inside the function body. This means that `square` can be reused on values of different units, where given an input `x` of unit α, then the output has units α^2. Its use on line 16 means that `square(velocity)` has units `(m/s)**2`.

The energy unit $kg\ m^2\ s^{-2}$ is quite long and complicated; instead, Joule is the preferred name for this unit, abbreviating the more complicated, derived form. Such complicated unit terms can be tedious to write and understand. CamFort provides a solution to this via *unit aliases*, which allow names to be given to derived units. In this case, we can declare `!= unit :: Joule = kg m**2 / s**2`, which aliases the more complex unit term to the name `Joule`. Energy annotations can now be expressed more succinctly and clearly in terms of `Joule`.

Moving toward Modern Fortran with Units

The 2015 Fortran standard includes a proposal for extending Fortran with unit annotations in the N1969 report.[3] The new proposal suggests introducing a `UNIT` attribute for variable annotations and a `UNIT` statement for unit declarations. Unlike CamFort, unit declarations are obligatory. The potential energy computation would look like this, according to N1969:

```
1 program energy
2     UNIT kg, m, s
3     UNIT a = m/s**2
4     real, parameter, unit(kg) :: mass = 3.00
5     real, parameter, unit(a) :: gravity = 9.81
6     real, parameter, unit(m) :: height = 4.20
7     UNIT Joule = kg * m**2 / s**2
8     real, unit(Joule) :: potential_energy
9
10    potential_energy = mass * gravity * height
11 end program energy
```

We originally considered the N1969 syntax for CamFort, but we later decided on the comment notation to enable support for older versions of Fortran without maintaining both the annotated and the compilable versions of source code. The other important differences are that N1969 features compulsory explicit unit declaration, lack of derived unit annotations, and conversion units.

Every unit used in an annotation in N1969 needs to be declared explicitly. Further derived (composite) units such as `m/s**2` can't be used in annotations directly; they must first be declared and given a name. CamFort similarly allows names to be given to composite (derived) units as this can increase clarity (discussed earlier). Always requiring this naming, however, can also hinder clarity. For example, in the code above, acceleration is declared as a composite unit `a` to be used in the annotations; however, $m^2\ s^{-2}$ is more familiar to scientists and programmers. Therefore it's more natural to use it directly in annotations rather than introducing a unit name.

Conversion units allow units in the same dimension to be exchanged automatically. For example, variables annotated with the units "meters," "centimeters, and "inches" all belong to the same dimension. Conversion units, regardless of their original declaration unit, can be used alongside other variables in the same dimension inside any expression. This is achieved through the insertion of implicit conversion functions. Although this is a useful tool, it doesn't fall into the scope of the CamFort project or its design philosophy. Such an extension requires additional compiler support to implicitly insert

conversions. Furthermore, automated conversion between units often requires a chain of conversions such as meters to centimeters and then centimeters to inches. Lack of transparency in this conversion means any floating-point error introduced during the conversion is hidden from the programmer.

Our earlier paper on units-of-measure offers a more thorough comparison of features between CamFort and N1969, and demonstrates the support for N1969 syntax in CamFort.[2]

Although adding unit annotations is an additional task for the programmer, the overall effort is reduced (less time debugging) and programmers can be more confident in their code's correctness. The units-of-measure annotations we showed here are a kind of lightweight specification, which, coupled with the CamFort automated checker, provide a simple form of program verification.

Formal specification and verification techniques are increasingly important to the practice of scientific computing[4,5] by increasing confidence in the program's correctness, reducing the amount of time spent debugging, aiding understanding and communication of ideas, and avoiding embarrassing errors in publications. Lightweight formal specification mechanisms, such as CamFort's `units` feature, don't require a complete specification beforehand, making them easier than ever to deploy. Nor do they hinder the development cycle, as unit information is often already included in the source code for documentation. In general, these kinds of local and incremental specifications can help prevent common classes of errors from occurring without unduly burdening the programmer. Thus, we hope that computational scientists will embrace current and future means of lightweight specification and verification to help produce software that's quicker to write, easier to understand, and with fewer bugs than ever before. Furthermore, we hope for increased interaction between scientists and computer scientists, such that new lightweight verification techniques can be developed to aid computational science programming.

For more information, including the CamFort source code, please visit www.cl.cam.ac.uk/research/dtg/naps. The source code and data supporting this article are free to download and available at https://www.repository.cam.ac.uk/handle/1810/251380.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (EP/M026124/1). Dominic Orchard additionally thanks the Software Sustainability Institute for its support.

References

1. A.G. Stephenson et al., "Mars Climate Orbiter Mishap Investigation Board Phase I Report," tech. report, NASA, 1999.
2. D. Orchard, A. Rice, and O. Oshmyan, "Evolving Fortran Types with Inferred Units-of-Measure," *J. Computational Science*, vol. 9, 2015, pp. 156–162.
3. W.V. Snyder, "ISO/IEC JTC1/SC22/WG5 N1969," tech. report, Int'l Organization for Standardization, 2013; ftp://ftp.nag.co.uk/sc22wg5/N1951-N2000/N1969.pdf.
4. K. Hinsen, "Writing Software Specifications," *Computing in Science & Eng.*, vol. 17, no. 3, 2015, pp. 54–61.
5. D. Orchard and A. Rice, "A Computational Science Agenda for Programming Language Research," *Procedia Computer Science*, vol. 29, 2014, pp. 713–727.

Mistral Contrastin is a research assistant at the University of Cambridge. His research interests include (de)obfuscation of binaries, static and dynamic malware analysis, and static analysis of source code. Contact him at mojpc2@cam.ac.uk.

Andrew Rice is a senior lecturer at the University of Cambridge. His current research is on the application of programming language design and theory to scientific computing. Rice works in the Computing for the Future of the Planet framework. Contact him at acr31@cam.ac.uk.

Matthew Danish is a research associate at the University of Cambridge. He's interested in program verification using type systems of programming languages. Danish has a PhD in computer science from Boston University. Contact him at mrd45@cam.ac.uk.

Dominic Orchard is a research associate at Imperial College London and a fellow of the Software Sustainability Institute. He's interested in the intersection between semantics, program analysis, and type theory. Orchard has a PhD in computer science from University of Cambridge, where he was also a postdoctoral researcher on the CamFort project. Contact him at d.orchard@imperial.ac.uk.