Efficient Scalars in Scala

Russell A. Paielli

Summary-- The `Scalar` class represents physical scalars in the Scala programming language. The standard arithmetic operators are overloaded to provide a syntax identical to that for basic numeric types. An operation with inconsistent units (e.g., adding a time to a length) causes an exception to be thrown. The `Scalar` class itself does not define any units but comes with a complete implementation of the standard SI metric system of units and many common non-metric units. The design also allows users to define a specialized or reduced set of physical units for any particular application or domain. Once an application has been developed and tested, the `Scalar` class can be easily disabled or bypassed (with no change required to client code) to achieve the execution efficiency of operations on basic numeric types, which are more than an order of magnitude faster. The source code is available from http://RussP.us.

Introduction

Physical units and scalars are fundamental to scientific and engineering calculations and computations. Scientists and engineers learn to add, subtract, multiply, divide, and convert units, and to keep track of them with pencil and paper. When they program a computer, however, they usually drop the explicit units and leave them implicit in the actual numerical calculations, with perhaps a comment to document the units. They do that for two basic reasons: (1) software that automatically tracks and checks units, although available, is not widely standardized and well known, and (2) that software can, in some cases, drastically reduce computational speed. The software presented here solves the latter problem and could eventually solve the former as well.

As a consequence of the lack of explicit units in most scientific and engineering software, mistaken units are a common source of error. Perhaps the most common such error involves passing an angle in degrees to a trigonometric function that takes it in radians. Humans tend to think in degrees, but standard trig functions take arguments in radians, and the conversion is often forgotten until its omission is discovered after time-consuming debugging. In one case that this author is aware of, aircraft simulation results over a period of six months were corrupted by such an error. In another case, confusion between seconds and minutes of time was found in a critical section of research software three years after results from it were published.

In air traffic management (ATM), horizontal distance is usually specified in nautical miles, whereas altitude is specified in feet, hundreds of feet, or thousands of feet. Horizontal speed is usually given in knots, whereas vertical speed is usually given in feet per minute. Such units can easily get confused if comments in the code are the only mechanism for enforcing consistency. Permitted units can be restricted by software coding standards, but draconian restrictions on permitted units can be inelegant and inconvenient, and they can force error-prone conversions on input and output.

The approach taken here to prevent such confusion is to allow the user to select units that are appropriate for the job, then to track those units explicitly in software just as an engineer or scientist would do on paper. The `Scalar` class itself does not specify any units, but a unit definition file can be used to define the preferred units and the relevant conversion factors for the particular application or domain. Two such unit definition files are included with the scalar package, one for metric and other common units, and one for traditional ATM units. These files also serve as examples for defining other sets of units.

When run in the Scala interpreter, the scalar package can serve as an interactive calculator that tracks units and checks consistency. No longer must engineers enter only the numbers into a calculator and manipulate the units separately in their heads or on paper. But the larger benefit is in automatically catching unit errors in Scala software. For computationally intensive applications, the `Scalar` class can be disabled or bypassed for efficient production runs that produce the same results. Thus, the `Scalar` class can be used during development and testing to guarantee unit correctness, then turned off to obtain the execution efficiency associated with basic numeric types. The resulting improvement in speed, which will be discussed later, is more than an order of magnitude.

At least one other Scala software package is also currently available for representing and manipulating physical scalars with units, but the scalar package presented here was developed independently. Any resemblance to other classes or packages is coincidental. The other Scala package for scalars checks units a compile time rather than run time, which is desirable but causes it to run some 300-400 times slower than operations with basic numeric types! The scalar package presented here is believed to be the first to provide a simple option to revert to basic numeric types (with no change required to client code) for efficient production runs after unit consistency is verified. That capability could be a key to widespread adoption, because the large computational overhead involved with tracking and checking units is no longer a reason to avoid using them.

Basic Usage

The scalar package is based on the `Scalar` class, which represents a physical scalar as a `Double` numeric type and a `Map[String, Double]` hash map that maps base units to their exponents. However, the `Scalar` constructor is not intended to be used directly. Instead, a "factory" function called "`unit,`" which returns a `Scalar` object, is used to define units. And the user need not even call the `unit` function directly unless a new unit is needed that is not already available in the unit definition file included with the package. The `units.scala` file defines a comprehensive set of standard units, including the complete SI metric system and many common non-metric units. To access those units, import the `Scalar` and `units` modules as follows:

```
import Scalar._
import units._
```

The predefined units in the `units` module are as consistent as possible with standard metric unit abbreviations such as "`s`" for seconds and "m" for meters. Thus, for example, the scalar 23 m/s^2 can be constructed with

```
val accel = 23 * m/sqr(s)
```

In addition to the `units` module, another smaller module called "`ATMunits`" is also included. It was designed for air traffic management. It provides an example of a simplified unit definition file for a particular application or domain without the many unneeded units in the `units` module. The smaller number of units and the lack of single-letter unit names makes the `ATMunits` module more manageable for large projects. Users are also free to copy the `units` module and strip out what they don't need, of course.

As in the standard metric system of units, the base unit for length in the `units` module is the meter, and several common scaled variations of it are defined:

```
val m  = unit("m", "meter", "length") // base unit

val mm = unit("mm", m/1000, "millimeter", "length")
val um = unit("um", mm/1000, "micrometer", "length")
val cm = unit("cm", m/100, "centimeter", "length")
val km = unit("km", 1000*m, "kilometer", "length")
```

The meter definition in the first line shows the creation of a base unit using the `unit` function. This version of the overloaded `unit` function takes three `String` arguments,

the last two of which are optional. The first argument, "`m`", is the abbreviated name of the unit, which is used internally as a key in a hash map of the exponents for each base unit, and it is also used for output representation. The second argument, "`meter`", is the (optional) full name of the unit, and the third argument, "`length`", is the (optional) unit type. The remaining four lines above show the creation of derived units based on the base meter unit. This version of the `unit` function is the same as the base-unit version except that it takes a `Scalar` type as the second argument to define the derived unit. Only one base unit should be defined for each unit type, such as time or length. When inconsistent units are added, subtracted, or compared, as in "`4*m + 3*s`," an exception is thrown.

Outputs will be printed by default in the base unit, such as meters for length, unless otherwise specified using the "`format`" function. For example:

```
scala> dist = 5.2 * m
scala> dist += 27 * mm
scala> println(dist)
5.227 m
scala> println(format(dist, "ft", "%2.2f")) // feet
17.15 ft
```

The third argument for numeric formatting is optional. The `format` function raises an exception if the unit specified in the second argument is not of the correct type for the scalar passed in the first argument.

No function is provided to extract the numerical coefficient of a scalar independent of the units, because that would depend on the base units chosen, and the output should not depend on the choice of base units. To print or write numerical data to a file without the units, or to pass data to a third-party function or application without the units, simply divide the scalar by the required unit to convert to a `Double` type for output. For example:

```
scala> println(dist/ft)
17.1489501312
```

A `Scalar` object cannot be converted to a `Double` unless it is dimensionless. The predefined units of "`rad`" (radians) and "`deg`" (degrees) in the `units` module are dimensionless, and the base unit is radians. This convention guarantees that standard trigonometric functions will be passed arguments in terms of radians, as required, and it prevents any other unit from being erroneously passed to a trigonometric function. For example, "`sin(30*deg)`" returns 0.5 as expected rather than being converted to "`sin(30)`", which would be wrong.

The convention for printing units is to show multiplication with asterisks, division with slashes, and exponents with carats. For example, "kilogram-meters/second-squared" would be shown as "`kg*m/s^2`." Only one slash is used, and if the denominator has multiplied units they are placed in parentheses, as in "`kg/(m*s^2)`."

To allow the `Scalar` class to be disabled or bypassed for efficiency (to be discussed in the next section), it has very few public member functions (other than the overloaded arithmetic operators). The reason is that some calls of member functions using standard "dot" notation do not work on built-in numeric types. Exceptions are "`toString`" and other conversion functions that also work on basic numeric types. The `Scalar` module redefines the functions "`sqrt`," "`hypot`," and "`atan2`" so they work correctly for both `Double` and `Scalar` argument types.

For convenience, an object called "`zero`" in the `Scalar` companion object is defined as the `Scalar` version of zero if the `Scalar` class is in use or the `Double` version of zero otherwise. This allows the use of normal Scala type inference conventions that work for variables in both cases. For example, the line "`var x = zero`" is equivalent to "`var x: Scalar=0`". A common error when first using the `Scalar` class is to write "`var x = 0.0`" when x should be a `Scalar`, and tracking down this error can waste a significant amount of time.

Disabling the Scalar Class for Efficiency

The `Scalar` class can be used as an interactive units-based calculator in the Scala shell, and it is also computationally efficient enough for many applications. However, it may be too slow for some computationally intensive applications. The computational "overhead" of the `Scalar` class can make arithmetic operations more than an order of magnitude slower than corresponding operations on the basic numeric types such as `Double`.

The source of that overhead is twofold. First, the character-string manipulation involved with tracking and checking the units obviously takes some time. But just disabling the unit tracking cannot increase the efficiency to the level of the `Double` type. The `Scalar` class "boxes" a basic numeric type, which adds substantial overhead compared to using a basic numeric type directly even if unit checking were disabled within the `Scalar` class.

Fortunately, a simple method has been devised to eliminate both sources of overhead if performance is or becomes an issue for a particular application. After an application has been tested and its unit consistency verified, the `Scalar` class can be disabled at compile time for production runs. During testing and development, the `Scalar` class can be enabled, but if the tests themselves are computationally intensive, it needs to be enabled only occasionally to detect the vast majority of unit errors.

Two versions of the file `Scalar.scala` are provided, one that implements the Scalar class, and the other that bypasses it. The latter file replaces the `unit` function with a function of the same name that simply returns the `Double` 1.0 rather than an instance of the `Scalar` class, for base units. Thus, the expression "`25*m`" would be replaced with "`25*1.0`," or "`25*km`" would be replaced with "`25*1000.0`". This replacement of the `Scalar` class eliminates its overhead. It may leave a few unnecessary multiplications and divisions by 1.0, but that should not cause a significant performance penalty unless they occur in high-rate, numerically intensive loops. If they do, the multiplication by the unit should be moved ahead of the loop.

Enabling or disabling the `Scalar` class can be done a couple of different ways. One way is to simply copy (or link) into your application the (enabled or disabled) version of `Scalar.scala` that you currently wish to use, then recompile. Another way is to maintain both versions in compiled form and use your CLASSPATH environment variable to select the one you currently wish to use. The latter way allows the `Scalar` class to be enabled or disabled without recompiling. The directory tree that `Scalar.scala` comes in is compatible with the standard "simple build tool" (sbt) directory structure and provides directories called "`scalar-on`" and "`scalar-off`" for that purpose.

If you are willing to recompile to switch modes, one convenient approach is to copy the two versions of the `Scalar.scala` file into files called "`Scalar-off.scala.`" and "`Scalar-on.scala.`" (note the periods at the end of the file names to distinguish them from regular source files). Then use aliases or short scripts called "`Scalar-on`" and "`Scalar-off`" to switch between the two versions. Those aliases or scripts should execute the commands "`ln -sf Scalar-on.scala. Scalar.scala`" or "`ln -sf Scalar-off.scala. Scalar.scala`", respectively.

In practice, explicit multiplication by units tends to occur mainly on input, and explicit division by units tends to occur mainly on output, so neither typically occur in high-rate loops. Assuming no unit inconsistencies and the observance of a few basic rules, all outputs with the `Scalar` class disabled should be identical to what they were with it enabled. The main rule is that output of `Scalar` objects should be done using the

`format` function rather than `println`, or they should be made dimensionless for output by dividing by the appropriate unit, as explained earlier (e.g., `dist/ft`).

To quantify the speedup resulting from disabling the `Scalar` class, a basic timing test was done in which scalars were added one hundred million times. With the `Scalar` class enabled, the time required on a Sun Ultra 24 Linux workstation was approximately 47.4 seconds. When the `Scalar` class was switched off, the computation time dropped to approximately 1.38 seconds, for a speedup factor of approximately 34. Similar results occurred for multiplication.

Note that most of the benefits of the scalar package are still available even when the `Scalar` class is disabled for efficiency. The only thing missing with the `Scalar` class disabled is the automatic checking for addition, subtraction, or comparison of scalars with incompatible units. With the `Scalar` class disabled, seconds can be added to meters without raising an exception, for example. But with the `Scalar` class disabled, the scalar package still automatically converts everything to standard base units, so the user need not worry about base units or conversion factors. Thus, confusion between different units representing the same physical quantity will still be avoided, and those kinds of errors are probably much more common than adding or subtracting incompatible units. Confusion between degrees and radians will still be avoided, for example, as will confusion between seconds and minutes or feet and meters.

Dynamic Type Checking

The `Scalar` class can also be used to implement a form of dynamic type checking that augments the static type checking done by the Scala compiler or interpreter. Suppose, for example, that a count of “bars” needs to be maintained and perhaps passed to a function or object. The user can simply define a unit for it and use it like any other unit:

```
bar = unit("bar")
count = 0
...
count += bar
```

If a count of type “`bar`” is then erroneously added to an incompatible type, the error will be detected and flagged immediately, saving debugging time.

About the Author

Russ Paielli is an aerospace research engineer in the Silicon Valley area of Northern California. He has an MS degree in Aeronautics and Astronautics from Stanford University (1987). He has worked on flight control theory and precision landing guidance using GPS integrated with inertial navigation. For the past several years, he has worked in the field of air traffic management. Russ is currently developing a research prototype of an aid for air traffic controllers to alert them to imminent conflicts and to compute maneuvers to resolve those conflicts when necessary. He maintains a website at http://RussP.us. The scalar package is free and is available from http://RussP.us/scalar-scala.htm.

This document was last revised on 2011-03-13.