A Package for Handling Units of Measure in Lisp

Roman Cunis*
MAZ GmbH
Karnapp 20, D-2100 Hamburg 90

February 7, 1992

1 Introduction

The ability to handle units of measure in a programming environment together with numerical quantities in scientific and engineering programs helps greatly in achieving computational safety as well as code and data readability. It allows for dimensional analysis, thus safeguarding against erroneous combination of values of different dimensionality, e.g. adding distances to masses, or against scaling errors, e.g. assigning distances measured in inch to variables supposed to hold centimeters. Associating values and variables with units of measure explicitly clarifies code and data and—given a clever compiler—allows for the above-mentioned dimensional analysis to be performed at compile time. If extended to input and output of user data, it might free the user from the burden to perform proper scaling himself when entering data conforming to program-defined units of measure. And it might allow for easy customization of output in order to present computational results to users familiar with different sets of units of measure, e.g. British and American use of units versus metric units.

Since the late seventies several proposals have been made to incorporate units of measure as data attributes [Gehani 77, Karr&Loveman 78,

*The work reported here has been done during my affiliation with the Laboratory for AI, Dept. of Computer Science, University of Hamburg.

Hilfinger 88], most of them in connection with Ada's typing capabilities, but none of them has it—as least to my knowledge—yet made it into Lisp. All proposals that I know of ultimately strived to incorporate units of measure as *declarative* attributes to be utilized by the compiler in order to perform consistency checks, generate proper conversion code where necessary, and "compile away" any computational overhead associated with handling dimensional information at run-time.[1] On the other hand there are some strong arguments in favour of actually *incorporating* units of measure information with numeric data objects in a dynamic and interactive programming environment like Lisp.

Section 2 discusses these arguments in some detail. Section 3 briefly describes the main features of a package for handling units of measure in CommonLisp, thereby focussing on an efficient representation of numbers with dimensional attributes (henceforth called dim-numbers) in order to reduce the computational overhead mentioned above as far as possible. Section 4 gives a short summary.

2 Explicit representation of dimensionality information

The first argument in favour of explicit representation of dimensionality within dim-number objects in Lisp draws on the Lisp specific tradition of having all data-type information at the object itself rather than associating it with variables intended to hold those objects. Because of this explicit unit of measure representation is the natural way of integrating dim-numbers into Lisp. Generic functions[2] might be written, that handle dim-numbers in general without having to know anything about the concrete measures involved. This allows for pattern matchers, constraint propagators, or other general purpose inference mechanisms to handle dim-numbers as an additional data-type.

[1] However, to my knowledge none of them actually reached this goal. They all include explicit dimensionality information in the number representation and perform consistency checks at run-time only. [Hilfinger 88] discusses in some detail compiler modifications (in Ada) that would be necessary to reach the ultimate goal.

[2] Not only in today's CLOS-sense of the word but also in the traditional sense of genericity in functions that process e.g. numbers or sequences.

The second argument is concerned with user comfort in handling dim-numbers in the interactive environment of Lisp. By extending the explicit dimensionality information to reader and printer syntax of Lisp, entering and displaying of dim-numbers—in a format chosen by the user and not by the programmer—can happen anywhere in the Lisp environment. This includes inspectors, browsers and debuggers of a Lisp environment that have no prior knowledge of their own about dim-number objects.

My third argument is in fact an extension of the second with respect to a typical problem often found in AI systems: Building knowledge bases for technical and engineering applications can greatly benefit from leaving to the user the choice which units of measure to use. Because application knowledge is often entered into a dynamic knowledge acquisition environment, again no pre-compilation dimensionality information would be available in order to handle units of measure statically.

3 Efficient representation of unit of measure information

Having thus argued that keeping unit of measure information at numbers explicitly is not only a deplorable burden but may actually have advantages of its own, I will now turn to the representation of dim-numbers. This ought to be as efficient as possible with respect to space and computational time. Moreover, it ought to support easy consistency checks in additive and comparative operations as well as easy unit conversion in multiplicative operations.

All authors in the field agree that the representation that is best suited for these tasks, is based on exponent vectors over elementary units of measure. Elementary measures are e.g. length, time, and weight. These will be referred to by their basic units of measure: e.g. m (meter), s (second), and g (gram). (Scaled units like feet, hour, or pound are not relevant for consistency and unit conversion and will be handled separately.)

More complex measures (like e.g. volume (m^3), speed ($\frac{m}{s}$), or force ($\frac{gm}{s^2}$)) can be expressed as products of elementary measures taken to some power: $\frac{m}{s} = m^1 s^{-1} g^0$, $\frac{gm}{s^2} = m^1 s^{-2} g^1$. Associating a unique vector index with each elementary measure allows for representing every unit of measure as an

exponent vector: m is thus represented by [1 0 0], $\frac{m}{s}$ by [1 -1 0], and $\frac{gm}{s^2}$ by [1 -2 1]. Commeasurability checks can thus be reduced to equality tests between vectors; multiplication of dim-numbers is done by adding vectors, and resulting vectors can immediately be reconverted to resulting units.

This is common usage among all approaches to dimensional analysis so far. However, the vector representation has two disadvantages: First, vectors consume space. Second, vectors have fixed length and can thus handle only a fixed number of elementary measures.[3] Our approach to unit of measure representation reduces exponent vectors to a single rational number. The central idea is to associate each elementary unit of measure with a *prime number* instead of a vector index. Complex measures can the be represented as ratios by multiplying these prime numbers according to the underlying exponent vectors. Associating m with 2, s with 3, and g with 5 the exemplarily given units of measures for volume, speed, and force can then be identified by ratios as follows: $m^3 \sim 8$, $\frac{m}{s} \sim \frac{2}{3}$, $\frac{gm}{s^2} \sim \frac{10}{9}$. Handling of these ratios in computations and comparisons is as easy as for vectors. Moreover, given the ability of CommonLisp to handle ratios, computation is even faster and the computational and spatial complexity of the representation is independent of the number of elementary measures involved.

The rest is fairly straightforward. Scaling for numbers with scaled units is done at input time, so that all dim-numbers are internally represented with respect to their base units of measure. Thus, no burden is added to any computation by using scaled units of measure. A special reader syntax is provided that reads any number immediately followed (i.e. without whitespace) by a defined unit specification as dim-number. (Thus `90min` = `5400s` = `1.5h` = `#<Dim-Number :value 5400.0 :ratio 3>`.) Special print-functions for dim-number objects support output of dim-numbers using any desired unit and even fancy formatting like `1h:30min:0s` for the example given above. Moreover, decomposing a resulting unit-ratio into its constituting prime numbers allow it to be displayed even if an explicit unit definition is missing: e.g. dividing a speed quantity by a time quantity will be displayed as `m/s2` even if acceleration is not a defined measure.

The following piece of code illustrates the definition syntax for measures and units:

[3] This problem might easily be avoided in Lisp by using variable width vectors, padding them with trailing zeros if necessary—and adding some complexity to the calculations.

```
(defmeasure speed ; name of measure
             "m/s" ; base unit
             :units
             ("km/h" ; implicit unit definition
                     ; will be automatically decomposed and
                     ; scaled if km and h are known.
              ("mph" 1.6km/h) ; explicit unit definition
              )
             :output-format
             (:unit "mph") ; all speed dim-numbers will be
                           ; printed scaled to mph
                           ; (unless explicitly converted)
             )
```

4 Summary

Traditionally, packages for unit of measure handling have been developed with the aim of performing dimensional analysis at compile-time in languages like Ada. However, arguments have been given that representing dimensionality information explicitly with a numeric quantity (and thus supporting dimensional analysis at run-time) is not only natural in an interactive environment like Lisp but may actually have advantages of its own.

Units of measure are typically represented as exponent vectors over elementary units. Associating prime numbers with elementary units allows for compact and efficient encoding of units of measures as ratios. Furthermore, a CommonLisp package has been described that makes use of this representation technique and integrates numbers with dimensionality information smoothly into Lisp by providing suitable reader and printer extensions.

Acknowledgment

I owe special thanks to my colleague Thorsten von Stein at the University of Hamburg who provided the original idea of replacing exponent vectors by ratios constructed from prime numbers.

References

[Gehani 77] Gehani, N.: *Units of Measure as a Data Attribute.* Computing Languages 2, 3(1977), 93–111.

[Hilfinger 88] Hilfinger, Paul N.: *An Ada Package for Dimensional Analysis.* ACM Transactions on Programming Languages and Systems 10, 2(1988), 189–203.

[Karr&Loveman 78] Karr, M., and Loveman, D.B.: *Incorporation of Units into Programming Languages.* Communications of the ACM 21, 5(1978), 385–391.