

# Discerning Quantities from Units of Measurement

Steve McKeever<sup>a</sup>

Department of Informatics and Media, Uppsala University, Sweden

Keywords: Quantities, Units of Measurement, Quantity Checking, Dimensional Analysis.

Abstract: In scientific and engineering applications, physical quantities embodied as units of measurement (UoM) are frequently used. While managing units of measurement is a fairly mature topic in software engineering, more subtle metrological concepts such as named quantities have had little traction within the scientific programming community. The loss of the Mars climate orbiter, attributed to a confusion between the metric and imperial unit systems, popularised the disastrous consequences of incorrectly handling measurement values. This has led to the development of a large number of libraries, languages and tools to ensure developers can specify and validate UoM information in their designs and codes. However these systems do not differentiate between quantities and dimensions. For instance torque and work, which share the same UoM, can not be interchanged because they do not represent the same entity. We present a named quantity layer that sits on top of a dimension checker and unit converter ensuring values of different quantities are correctly managed without undue restrictions. Our quantity algebra works alongside the unit dimensions to ensure we maintain named quantities when we perform arithmetic and function calls.

## 1 INTRODUCTION

Humans have used local units of measurement since the days of early trade, enhanced over time to fulfil the accuracy and interoperable needs of science and technology. The technical definition of a physical quantity is a “property of a phenomenon, body, or substance, where the property has a magnitude that can be expressed as a number and a reference” (Joint Committee for Guides in Metrology (JCGM), 2012) Ensuring numerical values that denote physical quantities are handled correctly is an essential requirement for the design and development of any engineering application. Infamous examples such as the Mars Climate Orbiter (Stephenson et al., 1999) or the Gimli Glider incident (Witkin, 1983) substantiate this. With ubiquitous digitalisation, and removal of humans in the loop, the need to faithfully represent and manipulate quantities in physical systems is ever increasing. Programming languages allow developers to describe how to evaluate numeric expressions but not how to detect inappropriate actions on quantities.

Dimensions are physical quantities that can be measured, while units are arbitrary labels that correspond to a given dimension to make it relative. For example a dimension is length, whereas a metre is

a relative unit that describes length. Units of measure can be defined in the most generic form as either *base quantities* or *derived quantities*. The base quantities are the basic building blocks, and the derived quantities are built from these. The base quantities and derived quantities together form a way of describing any part of the physical world (Sonin, 2001). For example length (metre) is a base quantity, and so is time (second). If these two base quantities are combined they express velocity (metre/second or  $\text{metre} \times \text{second}^{-1}$ ) which is a derived quantity. The International System of Units (SI) defines seven base quantities (length, mass, time, electric current, thermodynamic temperature, amount of substance, and luminous intensity) as well as a corresponding unit for each quantity (NIST, 2015). Some popular examples of both base and derived units are shown in Table 1. It is common for quantities to be declared as a number (the magnitude of the quantity) with an associated unit (Bureau International des Poids et Mesures, 2019).

There are many ways in which software processes and development can accommodate units of measurement in this manner (McKeever, 2021). Adding units to conventional programming languages goes back to the 1970s (Karr and Loveman, 1978) and early 80s with proposals to extend Fortran (Gehani, 1977) and


<sup>a</sup>  <https://orcid.org/0000-0002-1970-2884>

Table 1: Some SI standard base and derived units.

Name	Symbol	Quantity	Base Units
metre	<i>l</i>	length	metre
kilogram	<i>m</i>	mass	kg
second	<i>t</i>	time	second
hertz	<i>Hz</i>	frequency	second <sup>-1</sup>
newton	<i>N</i>	force, weight	metre × kg × second <sup>-2</sup>
pascal	<i>Pa</i>	pressure, stress	metre <sup>-1</sup> × kg × second <sup>-2</sup>
joule	<i>J</i>	energy, work	metre <sup>2</sup> × kg × second <sup>-2</sup>
newton metre	<i>N m</i>	torque	metre <sup>2</sup> × kg × second <sup>-2</sup>
watt	<i>W</i>	power, radiant flux	metre <sup>2</sup> × kg × second <sup>-3</sup>
square metre	<i>m</i> <sup>2</sup>	area	metre <sup>2</sup>
cubic metre	<i>m</i> <sup>3</sup>	volume	metre <sup>3</sup>
metre per second	<i>m/s</i>	speed, velocity	metre × second <sup>-1</sup>
metre per second squared	<i>m/s</i> <sup>2</sup>	acceleration	metre × second <sup>-2</sup>

then Pascal (Dreiheller et al., 1986). Hilfinger (Hilfinger, 1988) showed how to exploit Ada’s abstraction facilities, namely operator overloading and type parameterisation, to assign attributes for UoM to variables and values. The emergence of object oriented programming languages enabled developers to implement UoM either through a class hierarchy of units and their derived forms, or through the Quantity pattern (Fowler, 1997). There are a large number of libraries for all popular object oriented programming languages (Bennich-Björkman and McKeever, 2018) that support this approach (McKeever et al., 2019).

Applying UoM annotations requires an advanced checker to ensure variables and method calls are handled soundly. Two units are compatible if they both can be represented as the same derived quantity. For instance degrees Celsius is compatible with Fahrenheit. Values in Celsius can be *converted* to values in Fahrenheit. Two values can be added or subtracted only if their units are the same. Multiplication and division either add or subtract the two units product of power representations, assuming both values are compatible. Once a variable has been defined to be of a given unit, then it should remain as such. Checking that all annotated entities behave according to these rules ensures both *completeness* and *correctness* of the programme, and can be undertaken before the code is run.

However two values that share the same UoM might not represent the same ‘kinds of quantities’. For example, torque is a rotational force which causes an object to rotate about an axis while work is the result of a force acting over some distance. Surface tension can be described as newtons per meter or kilogram per second squared, and even though they equate, they represent different quantities. Our focus is to present a simple set of rules for arithmetic and

function calls that allow quantities to be named and handled *correctly*.

This paper is structured as follows, in Section 2 we describe how UoM are typically implemented and argue for a more comprehensive representation, while acknowledging the drawbacks of adding complexity to the development process. In Section 3 we introduce unit expressions and dimensional analysis. In Section 4 we describe a simple algebra of named quantities, and show how they can be maintained while evaluating unit assignments and function calls. We also discuss some of the obstacles to implementing named quantities and UoM in general. Finally, in Section 5 we summarise quantity validation and describe avenues of current research.

## 2 BACKGROUND

One can assert the physical dimension of length with the unit metre and the magnitude 10 (10m). However, the same length can also be expressed using other units such as centimetres or kilometres, at the same time changing the magnitude (1000cm or 0.01km). Although these examples are all based on the International System of Units (SI) there exists several other systems, such as the *Imperial system* where yards and miles would be used. On this basis a very simple object oriented design would entail a superclass for each dimension, such as Length, and then specific subclasses for the various units, each of which would contain overloaded operators to ensure unit based arithmetic could be performed correctly.

```
Length l1 = new LengthMetre (5.0);
Length l2 = new LengthYard (4.0);
Length l3 = l1.addlength (l2);
```

The `addlength` command would convert 12 into metres and perform the addition. We could extend our object oriented design to create a class hierarchy for each base type and use a tree structure to construct derived types. However this would result in hundreds of units and thousands of conversions.

Fortunately, a normal form exists which makes storage and comparison a lot easier. Any system of units can be derived from the base units as a product of powers of those base units:  $\text{base}^{e_1} \times \text{base}^{e_2} \times \dots \times \text{base}^{e_n}$ , where the exponents  $e_1, \dots, e_n$  are rational numbers. Thus an SI unit can be represented as a 7-tuple  $\langle e_1, \dots, e_7 \rangle$  where  $e_i$  denotes the  $i$ -th base unit; or in our case  $e_1$  denotes length,  $e_2$  mass,  $e_3$  time and so on.

This tuple can be reflected in a typical programming language as an array of integers. Although there are rare instances when fractional exponents might be required for intermediate results even when there is no SI unit that requires them. Within an object oriented class structure the array can be coupled with conversion, equality and numeric operators to form a `Unit` abstract data type which ensures only UoM correct arithmetic is undertaken. In Java this would be represented as:

```
class Unit {
    private int [7] dimension;
    private float [7] conversionFactor;
    private int [7] offset;
    ...
    boolean isCompatibleWith (Unit u);
    boolean equals (Unit u);
    Unit multiplyUnits (Unit u);
    Unit divideUnits (Unit u);
}
```

This is the basis of the Quantity pattern (Fowler, 1997) in which quantity values are represented as a pair: the numerical value,  $\{Q\}$ , and the unit of measure,  $[Q]$ , such that  $Q = \{Q\} \cdot [Q]$ .

```
class Quantity {
    private float value;
    private Unit unit;
    ....
}
```

The Quantity pattern provides a means of annotating variable declarations and method signatures with behavioural UoM specifications. Most libraries for modern programming languages implement this approach but, as was found in the survey of (Salah and McKeever, 2020), do not satisfy the core requirements of the scientific programming community. Interview subjects felt that UoM libraries were inconvenient: they did not interact well with the eco-system, had performance issues, required effort to learn and

costly rewrites to support. Increasing uptake for quantity aware code requires a language neutral interface that allows programmers to manage UoM in an indistinguishable language agnostic fashion, either as part of the core language (e.g. Swift (Apple, 2020) and F# (Microsoft, 2020)) or through the use of a separate validator (Jiang and Su, 2006; Dieterichs Henning, 2021; Hills M et al., 2012; Xiang et al., 2020)). Moreover, language based solutions enable quantity checking to be undertaken at compile-time, detecting errors early while ensuring no run-time overheads are required.

### 3 UNIT EXPRESSIONS

Performing calculations in relation to quantities, dimensions and units is often complex and can easily lead to mistakes. A dimensional analysis needs to check that (1) two physical quantities can only be equated if they have the same dimensions; (2) two physical quantities can only be added if they have the same dimensions (known as the *Principle of Dimensional Homogeneity*); (3) the dimensions of the multiplication of two quantities is given by the addition of the dimensions of the two quantities. We shall begin by defining dimensional analysis for a hypothetical programming language extended with unit variable declarations, *udecs*, and only consider the three common base dimensions of length, mass and time. Hence velocity, namely  $\text{length} \times \text{time}^{-1}$ , is represented as  $(1, 0, -1)$ . We use the standard `float` implementation to approximate for real numbers but as we are not performing arithmetic any representation would suffice.

```
udecs ::= udec1; ...; udecm
udec  ::= uv : float of (int, int, int)
```

The language has the standard statement constructs, *stmt*, and boolean expressions, *bexp*, but we will only focus on quantity variable assignments and conditionals as these affect unit variables, *uv*. We also have explicit unit expressions, *uexp*. Unit arithmetic expressions, *uexp*, impose syntactic restrictions so that their soundness can be inferred using the algebra of quantities. By creating a separate syntax for unit expressions we can distinguish between scalar values and *unitless quantities*, namely values that have the dimensions  $(0, 0, 0)$  such as moisture content.

```
stmt ::= uv := uexp | ... |
      | if bexp then stmt1 else stmt2
uexp ::= uv | uexp1+uexp2 | r*uexp | uexp1*uexp2
```

In Figure 1 we present the dimension analysis rules for declarations, assignments, conditionals and

$$\begin{array}{c}
 \langle uv : \text{real of } d, \rho \rangle \rightarrow_{udec} \rho \oplus \{uv \mapsto d\} \quad [\text{Dim Var Decl}] \\
 \\
 \frac{\langle udec_1, \rho \rangle \rightarrow_{udec} \rho_1 \quad \dots \quad \langle udec_m, \rho_{m-1} \rangle \rightarrow_{udec} \rho_m}{\langle udec_1; \dots; udec_m, \rho \rangle \rightarrow_{udecs} \rho_m} \quad [\text{Dim Var Decls}] \\
 \\
 \frac{\rho \vdash uexp \rightarrow_{uexp} (l, m, t) \quad \rho \vdash uv = (l, m, t)}{\rho \vdash uv := uexp \rightarrow_{stmt} \text{DimValid}} \quad [\text{Valid Assign Stmt}] \\
 \\
 \frac{\rho \vdash uexp \rightarrow_{uexp} (l, m, t) \quad \rho \vdash uv \neq (l, m, t)}{\rho \vdash uv := uexp \rightarrow_{stmt} \text{DimFail}} \quad [\text{Fail Assign Stmt}] \\
 \\
 \frac{\rho \vdash stmt_1 \rightarrow_{stmt} \text{DimValid} \quad \rho \vdash stmt_2 \rightarrow_{stmt} \text{DimValid}}{\rho \vdash \text{if } bexp \text{ then } stmt_1 \text{ else } stmt_2 \rightarrow_{stmt} \text{DimValid}} \quad [\text{Valid Cond Stmt}] \\
 \\
 \rho \vdash uv \rightarrow_{uexp} \rho \vdash uv \quad [\text{Dim Var Expr}] \\
 \\
 \frac{\rho \vdash uexp_1 \rightarrow_{uexp} (l, m, t) \quad \rho \vdash uexp_2 \rightarrow_{uexp} (l, m, t)}{\rho \vdash uexp_1 + uexp_2 \rightarrow_{uexp} (l, m, t)} \quad [\text{Dim Add Expr}] \\
 \\
 \frac{\rho \vdash uexp \rightarrow_{uexp} (l, m, t)}{\rho \vdash r * uexp \rightarrow_{uexp} (l, m, t)} \quad [\text{Dim Scaler Expr}] \\
 \\
 \frac{\rho \vdash uexp_1 \rightarrow_{uexp} (l_1, m_1, t_1) \quad \rho \vdash uexp_2 \rightarrow_{uexp} (l_2, m_2, t_2)}{\rho \vdash uexp_1 * uexp_2 \rightarrow_{uexp} (l_1 + l_2, m_1 + m_2, t_1 + t_2)} \quad [\text{Dim Mult Expr}]
 \end{array}$$

Figure 1: Dimension Analysis rules for declarations, assignments and expressions.

expressions. The rules for declarations build an environment,  $\rho$ , mapping variables to their dimensions. The environment will not change throughout the lifetime of the block. Thus, once a variable has been defined to be of a given quantity, then it will remain as such. Many library based systems allow programmers to change the dimensions of unit variables as they are objects of type `Quantity`, namely a mutable array. The rules for statements return either `DimValid` or `DimFail` depending on whether a given statement uses quantities correctly or not. An assignment statement is valid only if the quantity of the unit expression is dimensionally homogeneous with the unit variable that it is being assigned to, as shown in `Valid Assign Stmt`. The rule for conditionals checks the dimensional validity of both true and false statements. The rule for unit variables is just a lookup on the quantity environment  $\rho$ . The rule for addition ensures that both the left hand and right hand side subexpressions have the same quantities. Thus a proof tree cannot complete if the arguments to `+` have different dimensions, signifying a dimension error in the programme. The rules for multiplication allow constants to be applied and multiplying two unit expressions will create a combined quantity, where each dimension is summed.

Dimension analysis would be sufficient if only one unit system, such as the SI system, was required. In such cases the base units of metre, kilo-

gram and second could be implicit in implementations. Dimensionally correct unit expressions can be evaluated in much the same way as normal arithmetic expressions. As this is rarely the case in scientific applications where a myriad of unit systems and magnitudes are used, we need to perform unit conversions before evaluating the arithmetic expression. This can be undertaken at compile-time (Cooper and McKeever, 2008) or at run-time. Moreover we need to declare the units alongside their dimensions. A variable denoting torque would be stored as  $\{t \mapsto ((\text{Metre}, 2), (\text{Kilogram}, 1), (\text{Second}, -2))\}$  in the environment.

## 4 QUANTITY RULES

This section introduces named quantities, their rules and how they are supported in a typical programming language.

### 4.1 Expressions

We adopt a similar approach to (Foster, 2013; Hall, 2020) in that quantities should be represented as a 3-tuple, and not as a 2-tuple mentioned previously. Consequently we add a quantity name,  $\langle Q \rangle$ , to the nu-

$$\begin{array}{c}
\tau \vdash uv \rightarrow_{nexp} \tau uv \quad \text{[Named Var Expr]} \\
\\
\frac{\tau \vdash uexp_1 \rightarrow_{nexp} \text{Named } n \quad \tau \vdash uexp_2 \rightarrow_{nexp} \text{Named } n}{\tau \vdash uexp_1 + uexp_2 \rightarrow_{nexp} \text{Named } n} \quad \text{[Named Add Expr]} \\
\\
\frac{\tau \vdash uexp_1 \rightarrow_{nexp} \text{Named } n \quad \tau \vdash uexp_2 \rightarrow_{nexp} \text{Noname}}{\tau \vdash uexp_1 + uexp_2 \rightarrow_{nexp} \text{Named } n} \quad \text{[Named Add Expr]} \\
\\
\frac{\tau \vdash uexp_1 \rightarrow_{nexp} \text{Noname} \quad \tau \vdash uexp_2 \rightarrow_{nexp} \text{Named } n}{\tau \vdash uexp_1 + uexp_2 \rightarrow_{nexp} \text{Named } n} \quad \text{[Named Add Expr]} \\
\\
\frac{\tau \vdash uexp_1 \rightarrow_{nexp} \text{Noname} \quad \tau \vdash uexp_2 \rightarrow_{nexp} \text{Noname}}{\tau \vdash uexp_1 + uexp_2 \rightarrow_{nexp} \text{Noname}} \quad \text{[Named Add Expr]} \\
\\
\frac{\tau \vdash nexp \rightarrow_{uexp} qn}{\tau \vdash r * uexp \rightarrow_{nexp} qn} \quad \text{[Named Scaler Expr]} \\
\\
\frac{\tau \vdash uexp_1 \rightarrow_{nexp} qn_1 \quad \tau \vdash uexp_2 \rightarrow_{nexp} qn_2}{\tau \vdash uexp_1 * uexp_2 \rightarrow_{nexp} \text{Noname}} \quad \text{[Named Mult Expr]}
\end{array}$$

Figure 2: Named quantity rules for unit expression.

merical value,  $\{Q\}$ , and the unit of measure,  $[Q]$ , such that  $Q = \langle Q \rangle \cdot \{Q\} \cdot [Q]$ .

However, not all quantity variables in a programme will have a name such as Torque or Work. Some might denote an entity such as length that could be in metres or yards, while another might be a variable used to store some temporary value. Neither of these need to be named. Using an algebraic data type, we define named quantities as:

```
type quantname = Named of string | Noname
```

We are now in a position to define the rules for adding and multiplying named quantities. In both cases we assume that the unit expression is dimensionally correct, our concern is to define how named quantities conduct themselves. The operator  $\diamond$  takes two named quantities and defines how they compare: *two named quantities can be added together only if they represent the same entity*, if one quantity is named but the other is not then it is necessary for the result to be named, and if both are unnamed then the result will be too:

$$\begin{array}{l}
\text{Named } n_1 \diamond \text{Named } n_2 = \text{Named } n_1, \quad \text{if } n_1 = n_2 \\
\text{Named } n_1 \diamond \text{Named } n_2 = \text{ERROR}, \quad \text{if } n_1 \neq n_2 \\
\text{Named } n \diamond \text{Noname} = \text{Named } n \\
\text{Noname} \diamond \text{Named } n = \text{Named } n \\
\text{Noname} \diamond \text{Noname} = \text{Noname}
\end{array}$$

Our comparison rules cast upwards from Noname to Named, so as to assume a named quantity whenever possible. This is required to ensure named quantities behave correctly. If we cast downwards then we would have the alternative rule  $\text{Named } n \diamond \text{Noname} = \text{Nonamed}$ , that would allow Work to be added to

Torque through associativity:

$$\begin{array}{l}
\text{Named "Work"} \diamond (\text{Named "Torque"} \diamond \text{Noname}) \\
= \text{Named "Work"} \diamond \text{Noname} \\
= \text{Noname}
\end{array}$$

For multiplication the rules are simpler. The operator  $\Delta$  takes in two named quantities and defines how they behave over the multiplication operator. As *multiplication sums the dimensions of the two operands, the value will be different to either and so the result will always be Noname*.

$$\begin{array}{l}
\text{Named } n_1 \Delta \text{Named } n_2 = \text{Noname} \\
\text{Named } n \Delta \text{Noname} = \text{Noname} \\
\text{Noname} \Delta \text{Named } n = \text{Noname} \\
\text{Noname} \Delta \text{Noname} = \text{Noname}
\end{array}$$

The named quantity algebra can be incorporated into our language as shown in Figure 2. The language rules for scaler multiplication do not change the named quantity, the scaler value only affects the quantity value when evaluating expressions. Consider the example where  $\tau = \{t \mapsto \text{Named "Torque"}, w \mapsto \text{Named "Work"}\}$ . If we were to try to perform  $t + w$  then the rules for Named Add Expr will not succeed as upon retrieving their respective incompatible quantnames, no rule will apply.

Assignment statements have to satisfy the named quantity of the variable being assigned to, specifically the left hand side, and can therefore either succeed or fail:

```
type assignstate = Succeed | Fail
```

The  $\triangleleft$  rules specify that *one can assign a named quantity to a variable that has the same named quantity but not otherwise*. One can assign a Noname value

to a named quantity variable as it will have the same dimensions.

```

Named  $n_1 \triangleleft$  Named  $n_2$  = Succeed, if  $n_1 = n_2$ 
Named  $n_1 \triangleleft$  Named  $n_2$  = Fail, if  $n_1 \neq n_2$ 
Named  $n \triangleleft$  Noname = Succeed
Noname  $\triangleleft$  Noname = Succeed
    
```

It might seem as if we are missing a rule for assigning a named quantity to an unnamed variable but we cannot allow  $\text{Noname} \triangleleft \text{Named } n$  to Succeed as this would allow one to assign a Torque value to a Work variable through the intermediary of a local unnamed variable. This realisation has a profound effect on how we define our programming language rules to support the named quantity algebra. One *must* update the environment  $\tau$  to reflect that the named quantity assignment has taken place, as shown in the fourth Valid Assign Stmt rule of Figure 3. This ensures that the bindings of unnamed values will reflect their usage and protect the code from erroneous assignments. In order to guarantee coherence of potential changes to bindings in  $\tau$ , the environment is threaded through the rule for conditionals.

## 4.2 Function Calls

Quantity functions, *ufun*, differ from normal functions in that they can take a number of quantity arguments, and return a quantity. We must make sure that named quantities, *qn*, are passed into functions correctly. Namely that the named quantity of each argument matches that of its parameter. The quantity rules will calculate a return named quantity, if the function body satisfies the quantity algebra. Both a definition mechanism and an invocation mechanism are necessary for named quantity functions, as shown in the rules QuantFDef and QuantFCall of Figure 4.

```

ufun ::= fun ufn ( $uv_1:qn_1, \dots, uv_m:qn_m$ )
       is uexp
uexp ::= ... | ufn ( $uexp_1, \dots, uexp_m$ )
    
```

We require a second environment,  $\sigma$ , to store the relevant quantity information belonging to each quantity function. The rules for expressions and assignments will need to be extended to pass this second environment around but otherwise stay unchanged.

In order to guarantee quantity functions are handled correctly by the main programme we need to check that each invocation's named quantity arguments can be assigned to those in the function definition. For a function call *ufn* ( $uexp_1, \dots, uexp_m$ ) and corresponding definition header *fun ufn* ( $uv_1:qn_1, \dots, uv_m:qn_m$ ), we must ensure that the named quantity of each argument  $uexp_i$ , which we will infer to be  $qexp_i$  can be assigned

to each parameter, namely  $uv_i \triangleleft qexp_i$  as shown in Figure 4.

To illustrate how these collection of rules enable named quantity checking we consider a dimensionally correct assignment of  $nt := 2 * \text{addtq}(t1, t2)$  with differing named quantity definitions. In the first case we consider  $t1$  and  $t2$  to both represent torque values, so that environment  $\tau$  is  $\{nt \mapsto \text{Named "T"}, t1 \mapsto \text{Named "T"}, t2 \mapsto \text{Named "T"}\}$ . We also define *addtq* to expect two torque quantities. It will be stored in the function environment  $\sigma$  as  $\{\text{addtq} \mapsto ((x, \text{Named "T"}), (y, \text{Named "T"}), x+y)\}$ . The code fragment would look like this:

```

begin
nt : float of Named T;
t1 : float of Named T;
t2 : float of Named T;
fun addtq (x:Named T, y:Named T) = x+y
...
nt := 2 * addtq(t1, t2)
end
    
```

Quantity checking would succeed as shown in Figure 5.

Alternatively, if we try a similar assignment,  $nt := 2 * \text{addtq}(t, w)$ , but with quantities denoting torque and work:

```

begin
nt : float of Named T;
t : float of Named T;
w : float of Named W;
fun addtq (x:Named T, y:Named T) = x+y
...
nt := 2 * addtq(t, w)
end
    
```

Such that  $\tau' = \{nt \mapsto \text{Named "T"}, t \mapsto \text{Named "T"}, w \mapsto \text{Named "W"}\}$  then the proof cannot be completed as the parameter  $w$  has the named quantity *Named "W"* where a *Named "T"* was expected, as shown in Figure 6. Thus the function call cannot succeed and we would not compile this programme.

Having to explicitly name each parameter quantity is cumbersome and minimises reusability as many functions would have to be duplicated. We can use *Noname* quantities to avoid having to commit to a given name, such as torque or work:

```

begin
nt : float of Named T;
t : float of Named T;
w : float of Named W;
fun addtq (x:Noname, y:Noname) = x+y
    
```

$$\begin{array}{c}
\frac{\tau \vdash uexp \rightarrow_{nexp} \text{Named } n \quad \tau uv = \text{Named } n}{\langle uv := uexp, \tau \rangle \rightarrow_{nstmt} (\tau, \text{Succeed})} \quad [\text{Valid Assign Stmt}] \\
\\
\frac{\tau \vdash uexp \rightarrow_{nexp} \text{Named } n_1 \quad \tau uv = \text{Named } n_2 \quad n_1 \neq n_2}{\langle uv := uexp, \tau \rangle \rightarrow_{nstmt} (\tau, \text{Fail})} \quad [\text{Fail Assign Stmt}] \\
\\
\frac{\tau \vdash uexp \rightarrow_{nexp} \text{Noname} \quad \tau uv = \text{Named } n}{\langle uv := uexp, \tau \rangle \rightarrow_{nstmt} (\tau, \text{Succeed})} \quad [\text{Valid Assign Stmt}] \\
\\
\frac{\tau \vdash uexp \rightarrow_{nexp} \text{Noname} \quad \tau uv = \text{Noname}}{\langle uv := uexp, \tau \rangle \rightarrow_{nstmt} (\tau, \text{Succeed})} \quad [\text{Valid Assign Stmt}] \\
\\
\frac{\tau \vdash uexp \rightarrow_{nexp} \text{Named } n \quad \tau uv = \text{Noname}}{\langle uv := uexp, \tau \rangle \rightarrow_{nstmt} (\tau \oplus \{uv \mapsto \text{Named } n\}, \text{Succeed})} \quad [\text{Valid Assign Stmt}] \\
\\
\frac{\langle stmt_1, \tau \rangle \rightarrow_{stmt} (\tau_1, \text{Succeed}) \quad \langle stmt_2, \tau_1 \rangle \rightarrow_{stmt} (\tau_2, \text{Succeed})}{\langle \text{if } bexp \text{ then } stmt_1 \text{ else } stmt_2, \tau \rangle \rightarrow_{stmt} (\tau_2, \text{Succeed})} \quad [\text{Valid Cond Stmt}]
\end{array}$$

Figure 3: Named quantity rules for assignments and conditionals.

$$\begin{array}{c}
\frac{\sigma, \tau \vdash uexp \rightarrow_{nexp} \text{Named } n}{\sigma, \tau \vdash \langle uexp, \text{Named } n \rangle \rightarrow_{nparam} \text{Named } n} \quad [\text{Name Scaler Param}] \\
\\
\frac{\sigma, \tau \vdash uexp \rightarrow_{nexp} \text{Noname}}{\sigma, \tau \vdash \langle uexp, \text{Named } n \rangle \rightarrow_{nparam} \text{Named } n} \quad [\text{Name Scaler Param}] \\
\\
\frac{\sigma, \tau \vdash uexp \rightarrow_{nexp} \text{Noname}}{\sigma, \tau \vdash \langle uexp, \text{Noname} \rangle \rightarrow_{nparam} \text{Noname}} \quad [\text{Name Scaler Param}] \\
\\
\frac{\sigma, \tau \vdash uexp \rightarrow_{nexp} \text{Named } n}{\sigma, \tau \vdash \langle uexp, \text{Noname} \rangle \rightarrow_{nparam} \text{Named } n} \quad [\text{Name Scaler Param}] \\
\\
\frac{\langle \text{fun } ufn (uv_1 : qn_1, \dots, uv_m : qn_m) \text{ is } uexp, \sigma \rangle}{\rightarrow_{nfun} \sigma \oplus \{ufn \mapsto ((uv_1, qn_1), \dots, (uv_m, qn_m), uexp)\}} \quad [\text{Quant FDef}] \\
\\
\frac{\sigma ufn = ((uv_1, qn_1), \dots, (uv_m, qn_m), uexp) \quad \sigma, \tau \vdash \langle uexp_1, qn_1 \rangle \rightarrow_{nparam} uqn_1 \quad \dots \quad \sigma, \tau \vdash \langle uexp_m, qn_m \rangle \rightarrow_{nparam} uqn_m}{\sigma, \{uv_1 \mapsto uqn_1, \dots, uv_m \mapsto uqn_m\} \vdash uexp \rightarrow_{nexp} qn_{out}} \quad [\text{Quant FCall}]
\end{array}$$

Figure 4: Quantity Checking rules for Function Declarations and Invocation.

```

...
nt := 2 * addtq(t,w)
end

```

In this case our function `addtq` accepts `Noname` quantities so  $\sigma' = \{\text{addtq} \mapsto ((x, \text{Noname}), (y, \text{Noname}), x+y)\}$ . However, the function body will fail as both arguments to `x+y` need to follow the named quantity rules for addition, as shown in Figure 7:

Explicitly named quantity parameters protect the function body but do not allow commonalities to be exploited. Extending named quantities to include *named quantity variables*, `Quantvar`, so that

we could write generic quantity functions is recommendable:

```

fun add (x:Quantvar q, y:Quantvar q) = x+y

```

In this case the variable `q` could be assigned to any named quantity but both `x` and `y` would have the same named quantity. Our function invocation rule would need to be extended to ensure named quantity variables were uniquely assigned, and the named quantity of each argument that shared the same quantity variable were equal.

### 4.3 Discussion

Our algebra of named quantities is intended to be implemented as part of the static analysis phase of a compiler for an existing programming language or scientific domain specific language. Incorporating named quantities into software models would expand their use, and increase the robustness of designs. Although named quantity and dimension analysis are effective at discovering errors early, there are three concerns that impede their adoption (McKeever et al., 2020).

- *Lack of Awareness:* many developers are totally unaware of software solutions that deal with quantities and UoM. Inertia arises from factors like tradition, fear of change and effort of learning something new. Our approach is intended for language extensions or a pluggable type system, but could equally be included in popular UoM libraries, both of which elicit change and adaptation.
- *Technical Internal Factors:* many solutions are awkward and imprecise, introducing a loss of precision and struggling at times with dimensional consistencies. The strength of a language based solution, versus a library one, is that these issues are reduced.
- *External Factors:* modern systems are not built in a vacuum but form part of an eco-system (Lungu, 2008). It is harder to argue for quantity annotations when values pass through numerous generic components that do not support them, such as legacy systems, databases, spreadsheets, graphics tools and many other components that are unlikely to support quantities without costly updates. However, efforts are underway to address this essential issue (CODATA, 2021).

A lightweight and comprehensive language solution is fundamental to adoption. However, a relevant observation from both a survey of UoM libraries (Bennich-Björkman and McKeever, 2018) and interviews with practitioners (Salah and McKeever, 2020) was the need for quantities to be available at run-time. There are many mature and active UoM libraries for popular dynamic object oriented programming languages, such as Python and Ruby, in which no static checking will occur. If we allow dimensions to be only available at run-time then quantity checking, and also unit conversion, will have to be undertaken while the programme is executing. Faults can be avoided but testing is required to ensure annotations are congruent, a feature that a static programme analysis will uncover prior to evaluation.

## 5 CONCLUSION

We have developed a simple algebra of named quantities and shown how it can be incorporated into an existing programming language. Thereby ensuring quantities that share the same units of measure are handled separately, if required. Our algebra is safe and allows a degree of flexibility through casting upwards from unnamed quantities to named ones, thus enabling greater code reuse and a more practical implementation of the concept. Unnamed quantities can be assigned to named one's but not the other way around as this breaks substitutability and potentially reproducibility of experimental results. The algebra is distinct to that of dimension checking but they can be combined to form a single pass validator, allowing aliases such as  $\mathbb{J}$  for work or  $\mathbb{N}$  for force to be part of a prelude.

We lack an unqualified estimate of how frequently unit inconsistencies occur or their cost. Anecdotally we can glean that it is not negligible from experiments described in the literature (Cooper and McKeever, 2008; Antoniu et al., 2004; Ore et al., 2017a; Ore et al., 2017b). These focus on checking existing scientific repositories and are not representative of any quantity adhering software discipline. They have been applied post-development whereas we are seeking to support developers by ensuring their code bases model their scientific domain.

Quantity checking, much like type checking, is extremely useful when developing as errors are found before any code is run. Data on existing systems that have been extensively tested are less indicative of coding practices. However annotating all unit variables in a programme is costly. Ore (Ore et al., 2018) found subjects choose a correct UoM annotation only 51% of the time and take an average of 136 seconds to make a single correct annotation. This explains the appeal of systems that try to lower the burden while still ensuring coverage through unit type variables and a solver (Kennedy, 1994; Jiang and Su, 2006; Hills M et al., 2012; Xiang et al., 2020), or compromise coverage through a component based approach (Damevski, 2009; Ore et al., 2017a). A component based discipline means that the consequences of local unit mistakes are underestimated. On the other hand, it allows diverse teams to collaborate even if their domain specific environments or choice of quantity systems are dissimilar.

If all quantity type variables are resolved by the static checker then dimensional correctness can be shown. A corpus of quantity errors, their programming language and associated software systems is required to assess the significance of annotations and



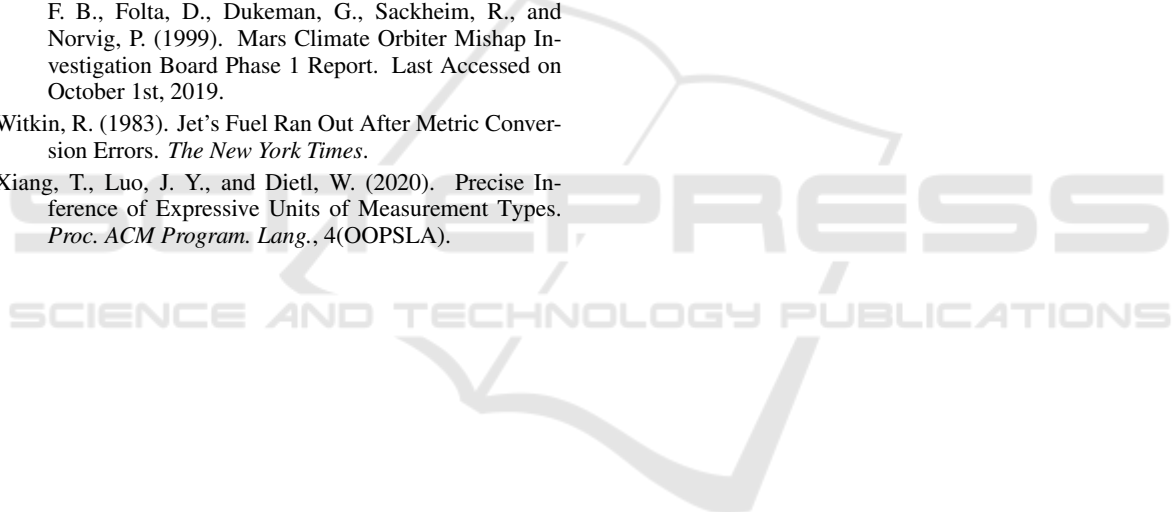
the cost of developing without. We have no idea of how much time is spent chasing incompatible quantity assignments, dimension errors or incorrect unit conversions. We do know that there are many reasons for not adopting a quantity discipline based approach (Salah and McKeever, 2020). Different stakeholders will have different robustness concerns and willingness to compromise on the proportion of quantity and unit annotations required. Addressing usability concerns is an important aspect of our research.

Our tool currently performs static named quantity and dimension analysis for a simple imperative language. It also searches for the least number of unit conversions to reduce round-off errors in generated code. Current work is looking at delaying conversions when calling functions, along with quantity and dimension variables that alleviate the annotation burden, while increasing reusability, and allowing errors to be discovered earlier in the code.

## REFERENCES

- Antoniou, T., Steckler, P. A., Krishnamurthi, S., Neuwirth, E., and Felleisen, M. (2004). Validating the unit correctness of spreadsheet programs. In *Proceedings of Software Engineering, ICSE '04*, pages 439–448, Washington, DC, USA. IEEE Computer Society.
- Apple (2020). Swift open source. Online <https://swift.org>. Last Accessed on 15th April 2020.
- Bennich-Björkman, O. and McKeever, S. (2018). The next 700 Unit of Measurement Checkers. In *Proceedings of Software Language Engineering, SLE 2018*, page 121–132, NY, USA. Association for Computing Machinery.
- Bureau International des Poids et Mesures (2019). SI Brochure: The International System of Units (SI), 9th Edition, Dimensions of Quantities. Online <https://www.bipm.org>. Last Accessed 15th April, 2020.
- CODATA (2021). Digital representation of units of measurement. Online <https://codata.org/initiatives/task-groups/drum/>. Last Accessed 25th December 2021.
- Cooper, J. and McKeever, S. (2008). A Model-Driven Approach to Automatic Conversion of Physical Units. *Software: Practice and Experience*, 38(4):337–359.
- Damevski, K. (2009). Expressing measurement units in interfaces for scientific component software. In *Proceedings of Component-Based High Performance Computing, CBHPC '09*, pages 13:1–13:8, NY, USA. ACM.
- Dieterichs Henning (2021). Units of Measurement Validator for C#. Online <https://www.codeproject.com/Articles/413750/Units-of-Measure-Validator-for-Csharp>. Last Accessed on 25th Of October.
- Dreiheller, A., Mohr, B., and Moerschbacher, M. (1986). Programming pascal with physical units. *SIGPLAN Notes*, 21(12):114–123.
- Foster, M. P. (2013). Quantities, units and computing. *Comput. Stand. Interfaces*, 35:529–535.
- Fowler, M. (1997). *Analysis Patterns: Reusable Objects Models*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Gehani, N. (1977). Units of measure as a data attribute. *Computer Languages*, 2(3):93 – 111.
- Hall, B. D. (2020). Software for calculation with physical quantities. In *2020 IEEE International Workshop on Metrology for Industry 4.0 IoT*, pages 458–463.
- Hilfinger, P. N. (1988). An Ada Package for Dimensional Analysis. *ACM Trans. Program. Lang. Syst.*, 10(2):189–203.
- Hills M, Chen Feng, and Roşu Grigore (2012). A Rewriting Logic Approach to Static Checking of Units of Measurement in C. *Electronic Notes in Theoretical Computer Science*, 290:51–67.
- Jiang, L. and Su, Z. (2006). Osprey: A Practical Type System for Validating Dimensional Unit Correctness of C Programs. In *Proceedings of the 28th International Conference on Software Engineering, ICSE '06*, pages 262–271, New York, NY, USA. ACM.
- Joint Committee for Guides in Metrology (JCGM) (2012). International Vocabulary of Metrology, Basic and General Concepts and Associated Terms (VIM). Online <https://www.bipm.org/en/about-us/>. Last Accessed 15th April 2020.
- Karr, M. and Loveman, D. B. (1978). Incorporation of Units into Programming Languages. *Commun. ACM*, 21(5):385–391.
- Kennedy, A. (1994). Dimension Types. In Sannella, D., editor, *Programming Languages and Systems—ESOP'94*, volume 788, pages 348–362, Edinburgh, U.K. Springer.
- Lungu, M. (2008). Towards reverse engineering software ecosystems. In *2008 IEEE International Conference on Software Maintenance*, pages 428–431.
- McKeever, S. (2021). From Quantities in Software Models to Implementation. In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, pages 199–206. INSTICC, SciTePress.
- McKeever, S., Bennich-Björkman, O., and Salah, O.-A. (2020). Unit of measurement libraries, their popularity and suitability. *Software: Practice and Experience*.
- McKeever, S., Paçacı, G., and Bennich-Björkman, O. (2019). Quantity Checking through Unit of Measurement Libraries, Current Status and Future Directions. In *Model-Driven Engineering and Software Development, MODELSWARD*.
- Microsoft (2020). F# software foundation. Online <https://fsharp.org>. Last Accessed on 15th April 2020.
- NIST (2015). International System of Units (SI): Base and Derived. Online <https://physics.nist.gov/cuu/Units/units.html>. Last Accessed October 2nd, 2019.

- Ore, J.-P., Detweiler, C., and Elbaum, S. (2017a). Lightweight Detection of Physical Unit Inconsistencies Without Program Annotations. In *Proceedings of International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 341–351, NY, USA. ACM.
- Ore, J.-P., Elbaum, S., and Detweiler, C. (2017b). Dimensional inconsistencies in code and ROS messages: A study of 5.9m lines of code. In *Intelligent Robots and Systems*, IROS, pages 712–718. IEEE.
- Ore, J.-P., Elbaum, S., Detweiler, C., and Karkazis, L. (2018). Assessing the Type Annotation Burden. In *Automated Software Engineering*, ASE 2018, pages 190–201, NY, USA. ACM.
- Salah, O.-A. and McKeever, S. (2020). Lack of Adoption of Units of Measurement Libraries: Survey and Anecdotes. In *Proceedings of Software Engineering in Practice*, ICSE-SEIP '20. ACM.
- Sonin, A. A. (2001). The physical basis of dimensional analysis. Technical report, Massachusetts Institute of Technology.
- Stephenson, A., LaPiana, L., Mulville, D., Peter Rutledge, F. B., Folta, D., Dukeman, G., Sackheim, R., and Norvig, P. (1999). Mars Climate Orbiter Mishap Investigation Board Phase 1 Report. Last Accessed on October 1st, 2019.
- Witkin, R. (1983). Jet's Fuel Ran Out After Metric Conversion Errors. *The New York Times*.
- Xiang, T., Luo, J. Y., and Dietl, W. (2020). Precise Inference of Expressive Units of Measurement Types. *Proc. ACM Program. Lang.*, 4(OOPSLA).



$$\begin{array}{c}
\frac{\sigma, \tau \vdash t_1 \rightarrow_{\text{next}} \text{Named } \text{"T"}}{\sigma, \tau \vdash \langle t_1, \text{Named } \text{"T"} \rangle \rightarrow_{\text{tparam}} \text{Named } \text{"T"}} \quad \frac{\sigma, \tau \vdash t_2 \rightarrow_{\text{next}} \text{Named } \text{"T"} \quad \sigma, \{x \mapsto \dots\} \vdash x \rightarrow_{\text{next}} \text{Named } \text{"T"} \quad \sigma, \{x \mapsto \dots\} \vdash y \rightarrow_{\text{next}} \text{Named } \text{"T"}}{\sigma, \tau \vdash \langle t_2, \text{Named } \text{"T"} \rangle \rightarrow_{\text{tparam}} \text{Named } \text{"T"}} \\
\frac{\sigma, \tau \vdash \text{addtq}(t_1, t_2) \rightarrow_{\text{next}} \text{Named } \text{"T"} \quad \sigma, \tau \vdash \text{addtq}(t_1, t_2) \rightarrow_{\text{next}} \text{Named } \text{"T"}}{\sigma, \tau \vdash 2 * \text{addtq}(t_1, t_2) \rightarrow_{\text{next}} \text{Named } \text{"T"}} \\
\tau \text{ nt} = \text{Named } \text{"T"}
\end{array}$$

Figure 5: Successfully Quantity checking the addition of two torque values.

$$\begin{array}{c}
\frac{\sigma, \tau' \vdash t \rightarrow_{\text{next}} \text{Named } \text{"T"} \quad \sigma, \tau' \vdash w \rightarrow_{\text{next}} \text{Named } \text{"W"}}{\sigma, \tau' \vdash \langle t, \text{Named } \text{"T"} \rangle \rightarrow_{\text{tparam}} \text{Named } \text{"T"}} \quad \frac{\sigma, \tau' \vdash \langle w, \text{Named } \text{"T"} \rangle \rightarrow_{\text{tparam}} \dots}{\sigma, \tau' \vdash \text{addtq}(t, w) \rightarrow_{\text{next}} \dots} \\
\frac{\sigma, \tau' \vdash 2 * \text{addtq}(t, w) \rightarrow_{\text{next}} \dots}{\sigma, \tau' \vdash \text{nt} := 2 * \text{addtq}(t, w) \rightarrow_{\text{next}} \dots} \quad \tau' \text{ nt} = \text{Named } \text{"T"}
\end{array}$$

Figure 6: Unsuccessfully Quantity checking a Function call with a torque value and a work value.

$$\begin{array}{c}
\frac{\sigma', \tau' \vdash t \rightarrow_{\text{next}} \text{Named } \text{"T"} \quad \sigma', \tau' \vdash w \rightarrow_{\text{next}} \text{Named } \text{"W"}}{\sigma', \tau' \vdash \langle t, \text{Noname} \rangle \rightarrow_{\text{tparam}} \text{Named } \text{"T"}} \quad \frac{\sigma', \tau' \vdash \langle w, \text{Noname} \rangle \rightarrow_{\text{tparam}} \text{Named } \text{"W"} \quad \sigma', \{x \mapsto \dots\} \vdash x \rightarrow_{\text{next}} \text{Named } \text{"T"} \quad \sigma', \{x \mapsto \dots\} \vdash y \rightarrow_{\text{next}} \text{Named } \text{"W"}}{\sigma', \tau' \vdash \langle w, \text{Noname} \rangle \rightarrow_{\text{tparam}} \text{Named } \text{"W"}} \\
\frac{\sigma', \tau' \vdash \text{addtq}(t, w) \rightarrow_{\text{next}} \dots \quad \sigma', \tau' \vdash 2 * \text{addtq}(t, w) \rightarrow_{\text{next}} \dots}{\sigma', \tau' \vdash \text{nt} := 2 * \text{addtq}(t, w) \rightarrow_{\text{next}} \dots} \quad \tau' \text{ nt} = \text{Named } \text{"T"}
\end{array}$$

Figure 7: Unsuccessfully Quantity checking the addition of a torque value with a work value.