

Torque not Work, Representing Kinds of Quantities

Steve McKeever ^a

Department of Informatics and Media, Uppsala University, Sweden

Keywords: Kind of Quantity, Dimensional Analysis, Units of Measurement, Unit Conversion.

Abstract: A system of units, such as the SI system, will have a number of fundamental units representing observable phenomena and a means of combining them to create compound units. In scientific and engineering disciplines, a quantity would typically be a value with an associated unit. Managing quantities in software systems is often left to the programmer, resulting in well-known failures when manipulated inappropriately. While there are a large number of tools and libraries for validating expressions denoting units of measurement, none allow the kind of quantity to be specified. In this paper we explore the problem of quantities that might share the same units of measurement but denote different kinds of quantities, such as work and torque. We develop a data type that represents compound units in a tree structure rather than as a tuple. When performing arithmetic, this structure maintains the compound definition allowing for a richer static analysis, and a complete definition of arithmetic on kinds of quantities.

1 INTRODUCTION


Humans have developed systems of measurement since the early days of trade, enhanced over time to fulfil the accuracy and interoperable needs of science and technology. In the 19th century, James Clerk Maxwell (Maxwell, 1873) introduced the concept of a system of quantities with a corresponding system of units. This generalisation allowed scientists working with different measurement systems to communicate more easily, as unit names (such as inch or metre) are treated as numeric variables and can be interchanged through multiplication.

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 measurement (UoM) can be defined in the most generic form as either *base quantities* or *compound quantities*. Fundamental physical quantities are the basic quantities that require no other physical quantities to express. Physical quantities that can be derived from the combination of two or more fundamental physical quantities are called compound physical quantities. For instance, velocity (m/s or $\text{m} \cdot \text{s}^{-1}$) can be derived from the base quantities of metre and second. 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 (Bureau International des Poids et Mesures, 2019). It is common for quantities to be declared as a number (the magnitude of the quantity) with an associated UoM (NIST, 2015).

Two values that share the same UoM might not represent the same *kinds of quantities* (KOQ) (Lodge, 1888; Foster and Tregeagle, 2018). The relationship between quantities and unit names is *many-to-one*. 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. They both share the same UoM, namely $\text{kg} \cdot \text{m}^2 \cdot \text{s}^{-2}$, but torque is usually written as $\text{N} \cdot \text{m}$, while work is conventionally expressed as J. Other examples are heat capacity and entropy ($\text{kg} \cdot \text{m}^2 \cdot \text{s}^{-2} \cdot \text{K}^{-1}$), and electric current and magnetomotive force.

With digitalisation affecting most facets of our lives, the need to faithfully represent and manipulate quantities in physical systems is ever increasing (Wilkinson et al., 2016). Popular programming languages allow developers to describe how to evaluate numeric expressions but not how to detect inappropriate actions on quantities. Consequently there have been infamous examples, such as the Mars Climate Orbiter (Stephenson et al., 1999), where UoM conversion omissions led to dire outcomes. Develop-

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

ers can choose to use tools or libraries to ensure UoM are managed correctly. However, KOQ analysis has only recently been formalised (McKeever., 2022). In this paper we present a more refined version that connects both dimensional and kind of quantity analysis through an alternative representation of UoM that allows quantities in arithmetic expressions to be named and handled *safely*. Explicitly naming quantities also allows known conversions to be applied to particular values, such as a becquerel for measuring radioactivity. In doing so we can ensure separate quantities (such as torque and work) are not combined but also that dimensionless quantities (such as radians or steradians) are distinct.

The paper is structured as follows. In Section 2 we discuss efforts to include UoM into conventional programming languages and modeling platforms. In Section 3 we introduce a very simple assignment language to show how values are defined and evaluated. In Section 4 we describe how conventional dimensional analysis would ensure that arithmetic proceeds correctly. We introduce a notion of kinds of quantities in Section 5, based on an algebraic data type that captures the structure of compound quantities and its operations. In Section 6 we show how this data type can detect errors that conventional UoM checkers cannot. Finally, in Section 7 we summarise quantity validation and describe extensions of our approach.

2 BACKGROUND

To a physicist or applied mathematician, it is taken for granted that a quantity is used in the same manner as a unit-independent value, and that all arithmetic and comparative operators can be applied to it. However, it does not make sense to multiply scales of intelligence or personality traits. Stevens (Stevens, 1946) identified four categories of scale that places limits on the type of measurement that can be used to construct valid terms: *nominal*, *ordinal*, *interval* and *ratio*. We shall focus on quantities that belong to the ratio scale as these are used to model the physical world and include all the usual arithmetic operators.

Adding UoM to programming languages was first undertaken in the 1970s (Karr and Loveman, 1978) and early 80s with proposals to extend Fortran (Gehani, 1977) and Pascal (Dreiheller et al., 1986). These efforts were mostly syntax based, requiring modifications to the underlying languages, reducing backwards compatibility and thus usage. The operator overloading and type parameterisation of Ada allowed for a more versatile approach (Hilfinger, 1988) to labelling variables with UoM fea-

tures. As practical object oriented programming languages started to emerge in the early 90s, such as C++ and Java, developers began to implement UoM via the Quantity pattern (Fowler, 1997). This has led to a veritable explosion in the number of UoM libraries available for all popular programming languages (Bennich-Björkman and McKeever, 2018) based on this pattern (McKeever et al., 2019). However, a survey of scientific coders (Salah and McKeever, 2020) revealed that these libraries have significant limitations that impedes their adoption. Even C++ with a de facto UoM library based on the template meta-programming feature, that ensures efficient run-time computation, suffers from both accuracy and usability issues. The F# programming language has excellent native support for UoM but lacks popularity. An alternative compile-time approach is to define UoM through comments or attributes and to build a tool that attempts to perform as much inspection as possible (Jiang and Su, 2006; Xiang et al., 2020; Hills M et al., 2012). These approaches are lightweight and scalable but they need to be maintained.

It is customary for software development to begin at a higher level of abstraction through diagrams and rules that focus on the conceptual model that is to be constructed. Adding UoM to software modeling languages has been successful but unless the workflow has been created specifically, declaring quantities in a system specification language offers no guarantee that the UoM information is supported in the eventual implementation. Extensions to the Unified Modeling Language (UML) have been proposed to support quantities. SysML¹, for instance, is defined as an extension of a subset of the UML to support systems engineering activities and has extensive support for quantities. The Event-B modelling language (Gibson and Méry, 2017) provides UoM and leverages the Rodin theorem prover to detect errors before translating to Java. Comparable techniques have been proposed for more formal notations such as Z (Hayes and Mahony, 1995) and Maude (Chen et al., 2003). Unit checking and conversion of UML can be undertaken before code is generated, either through a compilation workflow that leverages Object Constraint Language (OCL) expressions (Mayerhofer et al., 2016) or staged computation (Allen et al., 2004).

The key aspect of all these systems is that, by only representing dimensions, they cannot coherently distinguish between kinds of quantities and units of measure. We lack a definitive understanding of how frequently any form of quantity error occurs in practice: be it a naming, dimensional or unit conversion error.

¹<https://sysml.org>

However, it would seem careless in the current software development context to model complex systems in terms of floating point numbers without further delineation or annotation.

3 EVALUATING EXPRESSIONS

Performing calculations in relation to quantities, dimensions and units is subtle and can easily lead to mistakes. We shall begin by looking at a very simple language of declarations and assignments so that we can focus on the key aspects involved in managing quantities correctly. A program is a *block*, namely one or more quantity variable declarations, *udec*, followed by one or more statements, *ustmt*. Unit variables, *uv*, are represented as floating-point numbers, *float*. Enriching the language with more structure such as conditionals, while-loops and function calls does not affect the core of the analysis. Unit arithmetic expressions, *uexp*, impose syntactic restrictions so that their soundness can be inferred using the algebra of quantities.

$$\begin{array}{l}
 \mathcal{E} : uexp \rightarrow (uv \rightarrow value) \rightarrow value \\
 \mathcal{E}[[uv]]_{\rho} = \rho uv \\
 \mathcal{E}[[r * uexp]]_{\rho} = r \times \mathcal{E}[[uexp]]_{\rho} \\
 \mathcal{E}[[uexp_1 + uexp_2]]_{\rho} = \mathcal{E}[[uexp_1]]_{\rho} + \mathcal{E}[[uexp_2]]_{\rho} \\
 \mathcal{E}[[uexp_1 - uexp_2]]_{\rho} = \mathcal{E}[[uexp_1]]_{\rho} - \mathcal{E}[[uexp_2]]_{\rho} \\
 \mathcal{E}[[uexp_1 * uexp_2]]_{\rho} = \mathcal{E}[[uexp_1]]_{\rho} \times \mathcal{E}[[uexp_2]]_{\rho} \\
 \mathcal{E}[[uexp_1 / uexp_2]]_{\rho} = \mathcal{E}[[uexp_1]]_{\rho} \div \mathcal{E}[[uexp_2]]_{\rho}
 \end{array}$$

Figure 1: Rules for evaluating expressions.

```

block ::= begin udec in ustmt end
udec  ::= uv : float | uv : float is f
        | udec1; udec2
ustmt ::= uv := uexp
uexp  ::= uv | r * uexp
        | uexp1 + uexp2 | uexp1 - uexp2
        | uexp1 * uexp2 | uexp1 / uexp2

```

By creating a separate syntax for unit expressions we can distinguish between scalar values, such as *r*, and *unitless quantities* in which all the dimensions are zero, such as moisture content. Consider a simple program to calculate Newton's second law of motion:

```

begin
  f : float;
  m : float is 5.7;
  a : float is 3.2
in
  f := m * a
end

```

We can use the evaluate function, \mathcal{E} of Figure 1, with an environment consisting of values for *m* and *a* to calculate *f*.

4 DIMENSIONAL ANALYSIS AND UNIT CONVERSION

As is the case in nearly all programming languages, users have to assume that the mass (*m*) is given in kilograms, and the acceleration (*a*) is given in metres per second per second for the assignment to be correct.

$$\begin{array}{l}
 \mathcal{D} : uexp \rightarrow (uv \rightarrow dims) \rightarrow dims \\
 \mathcal{D}[[uv]]_{\sigma} = \sigma uv \\
 \mathcal{D}[[r * uexp]]_{\sigma} = \mathcal{D}[[uexp]]_{\sigma} \\
 \mathcal{D}[[uexp_1 + uexp_2]]_{\sigma} = \mathcal{D}[[uexp_1]]_{\sigma} \cong \mathcal{D}[[uexp_2]]_{\sigma} \\
 \mathcal{D}[[uexp_1 - uexp_2]]_{\sigma} = \mathcal{D}[[uexp_1]]_{\sigma} \cong \mathcal{D}[[uexp_2]]_{\sigma} \\
 \mathcal{D}[[uexp_1 * uexp_2]]_{\sigma} = \mathcal{D}[[uexp_1]]_{\sigma} \hat{\times} \mathcal{D}[[uexp_2]]_{\sigma} \\
 \mathcal{D}[[uexp_1 / uexp_2]]_{\sigma} = \mathcal{D}[[uexp_1]]_{\sigma} \hat{\div} \mathcal{D}[[uexp_2]]_{\sigma}
 \end{array}$$

Figure 2: Dimensional Analysis rules for Expressions.

A dimensional analysis needs to ensure 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. If we only consider the three common dimensions of *length*, *mass* and *time*, a tuple of integers can be used to represent these dimensions.

```

udec ::= uv : float of dims | udec1; udec2
dims ::= (int, int, int)

```

Dimensional homogeneity can be used to check for equality:

$$\begin{aligned}
 (l_1, m_1, t_1) &\cong (l_2, m_2, t_2) \\
 &= (l_1, m_1, t_1), \text{ if } l_1 = l_2 \wedge m_1 = m_2 \wedge t_1 = t_2
 \end{aligned}$$

While the rules for multiplication and division on as follows:

$$\begin{aligned}
 (l_1, m_1, t_1) \hat{\times} (l_2, m_2, t_2) &= (l_1 + l_2, m_1 + m_2, t_1 + t_2) \\
 (l_1, m_1, t_1) \hat{\div} (l_2, m_2, t_2) &= (l_1 - l_2, m_1 - m_2, t_1 - t_2)
 \end{aligned}$$

This allows us to rewrite the rules of Figure 1 by replacing the arithmetic operators to create a dimensional checker, shown in Figure 2. Scalar multiplication does not affect the dimensions of a quantity. The dimension of mass, *m*, is described as $(0, 1, 0)$, while acceleration, *a*, is $m \cdot s^{-2}$ or $(1, 0, -2)$ as a tuple. Our dimensional checker will compute with dimensions and attempt to ensure all assignments are correct. Consider our example program:

```
begin
  f : float of (1,1,-2);
  m : float of (0,1,0);
  a : float of (1,0,-2)
in
  f := m * a
end
```

The assignment is safe as $(0, 1, 0) \hat{\times} (1, 0, -2)$ yields $(1, 1, -2)$. Most UoM checkers adopt this approach, extending the checking into the statements and function calls of typical programming language constructs. For instance, all branches of conditionals and case statements must have the same dimensions, while comparison operators can only operate on quantities of the same dimension. If the dimensions of all variables are known at compile-time then this checking can be undertaken before the program is executed, incurring no run-time cost. Furthermore, if UoM annotations are used then unit conversions can be inserted into the run-time code (Cooper and McKeever, 2008; McKeever, 2023).

5 KINDS OF QUANTITIES

Using a tuple representation of dimensions, torque and work are both encoded as $(1, 2, -2)$. In order to distinguish them we need a richer datatype to represent units of measure, a table that maps quantity names onto their compound representations, and a means of equating them. Using an algebraic data type, we define named quantities as follows:

```
type quant = Name of string | Dimless
           | Qmul of (quant * quant)
           | Qdiv of (quant * quant)
```

Base quantities are represented as a unit name, for instance a length quantity can be represented as `Name "metre"`, while velocity would be described as `Qdiv (Name "metre", Name "sec")`. Dimensionless quantities, `Dimless`, have no physical dimension. They are often obtained as ratios resulting from the division of quantities of the same kind.

Definition 5.1 (Converting KOQ into UoM). A compound quantity can be converted into the *length*, *mass* and *time* form as follows:

$$\begin{aligned}
 C : \text{quant} &\rightarrow \text{dims} \\
 C (\text{Name "metre"}) &= (1, 0, 0) \\
 C (\text{Name "kg"}) &= (0, 1, 0) \\
 C (\text{Name "sec"}) &= (0, 0, 1) \\
 C \text{Dimless} &= (0, 0, 0) \\
 C (\text{Qmul } (p, q)) &= (C p) \hat{\times} (C q) \\
 C (\text{Qdiv } (p, q)) &= (C p) \hat{\div} (C q)
 \end{aligned}$$

$$\begin{aligned}
 \text{Qmul } (p, \text{Dimless}) &\Rightarrow p \\
 \text{Qmul } (\text{Dimless}, p) &\Rightarrow p \\
 \text{Qdiv } (p, \text{Dimless}) &\Rightarrow p \\
 \text{Qdiv } (\text{Dimless}, p) &\Rightarrow p \\
 \text{Qmul } (p, q) &\Rightarrow \text{Qmul } (q, p) \\
 \text{Qmul } (\text{Qmul } (p, q), r) &\Rightarrow \text{Qmul } (p, \text{Qmul } (q, r)) \\
 \text{Qmul } (p, \text{Qmul } (q, r)) &\Rightarrow \text{Qmul } (\text{Qmul } (p, q), r) \\
 \text{Qmul } (\text{Qdiv } (p, q), q) &\Rightarrow p \\
 \text{Qmul } (p, \text{Qdiv } (q, p)) &\Rightarrow q \\
 \text{Qdiv } (\text{Qmul } (p, q), q) &\Rightarrow p \\
 \text{Qdiv } (\text{Qmul } (p, q), p) &\Rightarrow q \\
 \text{Qdiv } (p, p) &\Rightarrow \text{Dimless}
 \end{aligned}$$

Figure 3: Named Quantity Expression Simplification.

Owing to the properties of the tuple arithmetic operators (identity, associative, commutative) there are many ways of describing the same compound quantity using the `quant` data type.

Proposition 5.2 (UoM Preserving Simplification). For given quantities p and q , the rules of Figures 3 maintain the UoM of a given compound quantity.

Proof. By structural induction on p and q . We assume p and q can be safely simplified, and apply the tuple conversion function C to both sides of the \Rightarrow arrow, resulting in equivalent tuples for all cases. \square

In order to construct *named compound quantities* we create a pair binding the name of a compound quantity to its quant form:

```
("metre_per_sec",
 [Qdiv (Name "metre", Name "sec")])
```

The `quant form` is modelled as a list because some quantities have more than one compound form. Consider the quantity `joule` that has two compound forms:

```
("joule",
 [Qmul (Name "metre", Name "newton");
 Qmul (Name "watt", Name "sec")])
```

A system of kinds of quantities can be created by building a table τ that maps names to their equivalent compound forms.

```
 $\tau = \{$  ("meter_per_sec",
 [Qdiv (Name "metre", Name "sec")]);
 ("acc", [Qdiv (Name "metre",
 Qmul (Name "sec", Name "sec")]);
 ("newton",
 [Qmul (Name "kg", Name "acc")]);
 ("joule",
 [Qmul (Name "metre", Name "newton");
 Qmul (Name "watt", Name "sec")]);
 ("watt", [Qdiv (Name "joule", Name "sec")]);
 ("rad", [Qdiv (Name "metre", Name "metre")]);
 ("newton_metre",
 [Qmul (Name "newton", Name "metre")]);
 ... }
```

A named quantity can be *expanded* through repeated lookups in the table. However, the table can be cyclic so we need to be careful when accessing a name not to end up in an infinite loop. While replacing compound names one can also apply the simplification rules of Figure 3 such that there may be many more ways of representing a compound quantity than just those in the table. For instance, a watt has 18 representations:

```
Name "watt"
⇒ Qdiv(Name "joule",Name "sec")
⇒ Qdiv(Qmul(Name "newton",Name "metre"),
        Name "sec")
⇒ Qdiv(Qmul(Name "metre",Name "newton"),
        Name "sec")
⇒ Qdiv(Qmul(Qmul (Name "kg",Name "metre"),
                Qdiv(Name "metre",
                    Qmul(Name "sec",Name "sec"))),Name "sec")
⋮
```

Even though all the representations denote the same UoM, there is a hierarchy. We therefore need a way of capturing the representation that is closest to the conceptual description of the quantity.

Proposition 5.3 (Most Abstract Named Quantity). If we have two quant trees denoting the same UoM, then the one with the shortest height, $\min(\text{height } p, \text{height } q)$, will be the *most abstract* named quantity, which we shall write as $\text{maq}(p, q)$.

Proof. A named quant entity is the most abstract as it describes explicitly what kind of quantity the entity is, and is unique. A table lookup will replace a Name by a compound quant with greater complexity, and a greater height. Thus, the quant tree with the shortest height will represent the most concise version of the given entity. It will denote the quantity whose names can be retrieved and simplified to the greatest number of representations. \square

Definition 5.4 (Structural Equality of Named Quantities). Two quantities, p and q are considered structurally equal if they have the same representation in terms of the algebraic data type quant. If the two quantities are structurally equal then $p \triangleq q$ holds:

```
Name n      △ Name m      = (n = m)
Dimless     △ Dimless     = true
Qmul(p1, q1) △ Qmul(p2, q2) = (p1 △ p2) ∧ (q1 △ q2)
Qdiv(p1, q1) △ Qdiv(p2, q2) = (p1 △ p2) ∧ (q1 △ q2)
-           △ -           = false
```

Definition 5.5 (Equality of Named Quantities). Given a system of kinds τ , two quantities p and q are considered equal, if they share a representation. This is achieved by expanding and simplifying p into the

set of forms $\{p_1, \dots, p_n\}$ and q into the set of forms $\{q_1, \dots, q_m\}$, performing the cartesian product of both sets and comparing the pairs for structural equality, $p_i \triangleq q_j$. If there is at least one identical pair then we say that $p =_{\tau} q$ holds.

For example:

```
Name "newton" =τ Qmul(Name "acc",Name "kg")
```

When we expand Name "newton", we have the following forms:

```
Name "newton"
⇒ Qmul(Name "kg",Name "acc")
⇒ Qmul(Name "kg",Qdiv(Name "metre",
                        Qmul(Name "sec",Name "sec")))
⇒ Qmul(Qdiv(Name "metre",
              Qmul(Name "sec",Name "sec")),Name "kg")
⇒ Qmul(Name "acc",Name "kg")
```

while expanding $\text{Qmul}(\text{Name "acc"}, \text{Name "kg"})$ creates the forms:

```
Qmul(Name "acc",Name "kg")
⇒ Qmul(Qdiv(Name "metre",
              Qmul(Name "sec",Name "sec")),Name "kg")
⇒ Qmul(Name "kg",Qdiv(Name "metre",
                        Qmul(Name "sec",Name "sec")))
⇒ Qmul(Name "kg",Name "acc")
```

The cartesian product of both simplifications has at least one identical pair, shown as underlined. Thus, the two named quantities are equal (dimensionally). This will yield the same result as expanding p to create p' , expanding q to create q' , and performing $C p' \cong C q'$.

Definition 5.6 (Homogeneity of Named Quantities). Given a system of kinds τ , two quantities, p and q are considered to represent the same named quantity if they share the same name, or if they they are equal. The operator $p \equiv_{\tau} q$ will return the most abstract quant representation of two equal named quantities. It is defined as follows:

```
Name n  ≡τ Name m  = Name n,    if n = m
Name n  ≡τ q       = Name n,    if Name n =τ q
p       ≡τ Name m  = Name m,    if p =τ Name m
p       ≡τ q       = maq(p, q) if p =τ q
```

Using our previous example:

```
Name "newton" ≡τ Qmul(Name "acc",Name "kg")
```

will return Name "newton", as both quantities are equal and a newton represents the most abstract representation of the two. However, attempting to equate work and watt will not succeed:

```
Name "joule" ≠τ Name "newton_metre"
```

as even though they are dimensionally equal, they do not share the same name.

6 KIND OF QUANTITY ANALYSIS

We are now in a position to define the arithmetic rules for named quantities. We extend our definition of *udec* to include names rather than dimensions:

udec ::= *uv* : float called *string* | ...

The rules of Figure 4 calculate the *quant* representation of an arithmetic expression given an environment γ mapping variables to their named quantities. The rules for multiplication and division build compound quantities, while the rules for addition and subtraction will check the homogeneity of both operands. In other words, only quantities that have the same dimensions can be added or subtracted. The rules of \mathcal{K} will return the most abstract *quant* representation for the given expression.

To ensure the rules for addition and subtraction are *safe* in terms of named quantity arithmetic, we need to demonstrate that the \equiv_{τ} operator is both commutative and associative. This is necessary to show that the KOQ of subexpressions are not subsequently overridden when applying \mathcal{K} .

Proposition 6.1 (Commutativity of \equiv_{τ}). For given quantities p and q that represent equal quantities, $p \equiv_{\tau} q$ yields the same result as $q \equiv_{\tau} p$.

Proof. By case analysis of the function \equiv_{τ} , the first case will hold as string equality is commutative, case two and three can be reordered, while the *min* function is independent on the ordering of the arguments. Hence, changing the order of the operands of the homogeneity operator does not change the result. \square

Proposition 6.2 (Associativity of \equiv_{τ}). For given quantities p , q and r that represent equal quantities, we need to show that $(p \equiv_{\tau} q) \equiv_{\tau} r \triangleq p \equiv_{\tau} (q \equiv_{\tau} r)$.

Proof. By structural induction on *quant* entities that are either named or unnamed, resulting in 8 cases which all hold true. \square

The proof ensures that *addition* and *subtraction* maintains the property that named subexpressions must represent the same entity for evaluation to succeed. In essence, one always casts upwards from unnamed entities to known named ones.

As homogeneity does not guarantee structural equality we provide a second assignment operator that allows one to match on a given compound quantity:

ustmt ::= *uv* := *uexp*
 | *uv* := *uexp* of *quant*

Assignments either succeed or fail depending on whether the expression derives a quantity that

matches the variable being assigned to. Thus, named quantity rules for assignments will return a status tag:

type *stmtstate* = Succeed | Fail

The rules of Figure 5 allow the user to describe two forms of assignment. In the first instance we check that the expression can be assigned to the named quantity variable, and in the second we perform an additional check to make sure it has a specified compound form.

Using our motivating example of wanting to prohibit the addition of a torque value to that of a work value, consider the following program segment:

```
begin
  wk : float called "joule";
  tq : float called "newton_metre"
in
  wk := tq + wk
end
```

We can check the assignment as follows:

```
S[[wk := tq + wk]]{wk → Name "joule", tq → Name "newton_metre"}
⇒ Success, if Name "joule" ≡τ K[[tq + wk]]{...}
  Fail, otherwise
⇒ Success, if Name "joule" ≡τ
  (K[[tq]]{...} ≡τ K[[wk]]{...})
  Fail, otherwise
⇒ Success, if Name "joule" ≡τ
  (Name "newton_metre" ≡τ Name "joule")
  Fail, otherwise
⇒ Fail
```

Resulting in our system not allowing the assignment to take place.

However, the analysis is not able to recognise incorrect unnamed multiplication. Our solution is to break the calculation into smaller parts and structurally examine the quantity using the *of* keyword:

```
begin
  n : float called "newton";
  m : float called "metre";
  tq : float called "newton_metre"
in
  tq := m * n of Qmul(Name "newton",
                      Name "metre")
end
```

In this case, we will be checking:

$\mathcal{K}[[m * n]]_{\gamma} \triangleq \text{Qmul}(\text{Name "newton"}, \text{Name "metre"})$

which will not succeed as the multiplication expected is $n * m$. Finally, our system is able to manipulate dimensionless quantities whereas a tuple representation treats all dimensionless quantities as identical.

$$\begin{aligned}
\mathcal{K} &: uexp \rightarrow (uv \rightarrow \text{quant}) \rightarrow \text{quant} \\
\mathcal{K}[\![uv]\!]_{\gamma} &= \gamma uv \\
\mathcal{K}[\![r * uexp]\!]_{\gamma} &= \mathcal{K}[\![uexp]\!]_{\gamma} \\
\mathcal{K}[\![uexp_1 + uexp_2]\!]_{\gamma} &= \mathcal{K}[\![uexp_1]\!]_{\gamma} \equiv_{\tau} \mathcal{K}[\![uexp_2]\!]_{\gamma} \\
\mathcal{K}[\![uexp_1 - uexp_2]\!]_{\gamma} &= \mathcal{K}[\![uexp_1]\!]_{\gamma} \equiv_{\tau} \mathcal{K}[\![uexp_2]\!]_{\gamma} \\
\mathcal{K}[\![uexp_1 * uexp_2]\!]_{\gamma} &= \text{Qmul}(\mathcal{K}[\![uexp_1]\!]_{\gamma}, \mathcal{K}[\![uexp_2]\!]_{\gamma}) \\
\mathcal{K}[\![uexp_1 / uexp_2]\!]_{\gamma} &= \text{Qdiv}(\mathcal{K}[\![uexp_1]\!]_{\gamma}, \mathcal{K}[\![uexp_2]\!]_{\gamma})
\end{aligned}$$

Figure 4: Named Quantity rules for Expressions.

$$\begin{aligned}
S &: ustmt \rightarrow (uv \rightarrow \text{quant}) \rightarrow \text{stmtstate} \\
S[\![uv := uexp]\!]_{\gamma} &= \text{Succeed, if } (\gamma uv) \equiv_{\tau} \mathcal{K}[\![uexp]\!]_{\gamma} \\
&= \text{Fail, otherwise} \\
S[\![uv := uexp \text{ of } q]\!]_{\gamma} &= \text{Succeed, if } (\gamma uv) \equiv_{\tau} \mathcal{K}[\![uexp]\!]_{\gamma} \wedge \mathcal{K}[\![uexp]\!]_{\gamma} \triangleq q \\
&= \text{Fail, otherwise}
\end{aligned}$$

Figure 5: Named Quantity rules for Statements.

7 SUMMARY

Global and existential challenges, from infectious diseases to environmental breakdown, require high-quality data (Hanisch, et al., 2022). Ensuring software systems support quantities explicitly is becoming ever more important. While there are solutions that allow UoM to be specified at both the model and code level, adoption is challenging due to the annotation burden and also the lack of perceived benefits.

We have presented a conceptual model for *kinds of quantities*, and a means of ensuring safe expression evaluation, that extends the traditional dimensional analysis through the use of a recursive data type. Our model allows compound units to be constructed, and a table which enables named quantities to be mapped to one or more compound forms. We provide ways of comparing and constructing such quantities to ensure that (1) only values of the same kind can be added or subtracted; and (2) that multiplication and division create new compound forms that represent the addition or subtraction of their dimensions. As unit variables are all assigned a named quantity, our current system effectively combines dimensional analysis with the naming scheme presented in (McKeever, 2022), and formalises the quantity calculus presented by Lodge (Lodge, 1888). It allows us to distinguish between quantities that share the same UoM, even if that UoM denotes a dimensionless entity, and subsequently apply conversions that only apply to that particular kind.

Hall (Hall, 2022; Hall, 2023) has developed a more comprehensive notation for modeling quantities

that includes both scale and aspect. The notion of aspect is more general than our formulation of KOQ as it can be used to denote measurable properties. However, the intention of Hall is to enable effective exchange of information, where aspect and scale tables are stored in a central register, rather than the robust evaluation of numeric expressions.

Our methodology can be implemented natively or as a pluggable-type system for modeling or programming languages. It allows for a richer static analysis than dimensional analysis, and a complete definition of arithmetic on kinds of quantities. It is less suited to run-time checking due to the additional overhead of supporting the `quant` data type. Quantity annotations are initially costly for the developer but relatively stable to program evolution. Therefore scalability and maintainability within potentially safety-critical code is assured. Moreover, the cost of our KOQ annotations is equivalent to most UoM annotations as they are both written in terms of multiplication on base units and then converted into internal representations. Our naming scheme improves the scope of existing dimensional analysis.

At present, our system lacks two essential features. It needs to support synonyms that allow compound quantities to have the same status as their named counterpart. Synonyms would allow users to replace particular compound forms by their names, such as `newton_metre`, if required. More importantly, the addition of quantity variables to the `quant` data type is pivotal in order to support polymorphic functions, and also variables whose units are unknown, or that represent values of intermediate cal-

culations. Thereby reducing the annotation burden. Polymorphic quantity variables incur an additional computational overhead as many quant forms will be assignable for a given scope within a program.

REFERENCES

- Allen, E., Chase, D., Luchangco, V., Maessen, J.-W., and Steele, Jr., G. L. (2004). Object-oriented units of measurement. In *Proceedings of Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '04*, pages 384–403, NY, USA. ACM.
- 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. Last Accessed 19th May, 2022.
- Chen, F., Rosu, G., and Venkatesan, R. P. (2003). Rule-Based Analysis of Dimensional Safety. In *RTA*.
- Cooper, J. and McKeever, S. (2008). A Model-Driven Approach to Automatic Conversion of Physical Units. *Software: Practice and Experience*, 38(4):337–359.
- Dreiheller, A., Mohr, B., and Moerschbacher, M. (1986). Programming pascal with physical units. *SIGPLAN Notes*, 21(12):114–123.
- Foster, M. and Tregaele, S. (2018). Physical-type correctness in scientific python.
- 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.
- Gibson, J. P. and Méry, D. (2017). Explicit modelling of physical measures: from Event-B to Java. In *International Workshop on Handling IMPLICIT and EXPLICIT knowledge in formal system development*.
- Hall, B. (2022). The problem with ‘dimensionless quantities’. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, pages 116–125, Portugal. INSTICC, SciTePress.
- Hall, B. (2023). Modelling expressions of physical quantities. In *Proceedings of the 15th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management IC3K*, Portugal. INSTICC, SciTePress.
- Hanisch, R. et al. (2022). Stop squandering data: make units of measurement machine-readable. *Nature*, 605:222–224.
- Hayes, I. J. and Mahony, B. P. (1995). Using Units of Measurement in Formal Specifications. *Formal Aspects of Computing*, 7(3):329–347.
- 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.
- Karr, M. and Loveman, D. B. (1978). Incorporation of Units into Programming Languages. *Commun. ACM*, 21(5):385–391.
- Lodge, A. (1888). The Multiplication and Division of Concrete Quantities. *General Report (Association for the Improvement of Geometrical Teaching)*, 14:47–70.
- Maxwell, J. C. (1873). *A treatise on electricity and magnetism*, volume 1. Oxford: Clarendon Press.
- Mayerhofer, T., Wimmer, M., and Vallecillo, A. (2016). Adding uncertainty and units to quantity types in software models. In *Software Language Engineering, SLE 2016*, pages 118–131, NY, USA. ACM.
- McKeever, S. (2022). Discerning Quantities from Units of Measurement. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, pages 105–115, Portugal. INSTICC, SciTePress.
- McKeever, S. (2023). Acknowledging implementation trade-offs when developing with units of measurement. In Pires, L. F., Hammoudi, S., and Seidewitz, E., editors, *Model-Driven Engineering and Software Development*, pages 25–47, Switzerland. Springer Nature.
- 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*.
- NIST (2015). International System of Units (SI): Base and Derived. Last Accessed May 19th, 2022.
- 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*, NY, USA. ACM.
- 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 May 19th, 2022.
- Stevens, S. S. (1946). On the theory of scales of measurement. *Science*, 103(2684):677–680.
- Wilkinson, M. D., Dumontier, M., Aalbersberg, I. J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.-W., da Silva Santos, L. B., Bourne, P. E., et al. (2016). The fair guiding principles for scientific data management and stewardship. *Scientific data*, 3.
- Xiang, T., Luo, J. Y., and Dietl, W. (2020). Precise Inference of Expressive Units of Measurement Types. *Proc. ACM Program. Lang.*, 4(OOPSLA).