A Sound Type System for Physical Quantities, Units, and Measurements

Simon Foster
University of York, York, UK

Burkhart Wolff
LMF, Université Paris-Saclay, Paris, France

Abstract

We present a theory in Isabelle/HOL [8] that builds a formal model for both the *International System of Quantities* (ISQ) and the *International System of Units* (SI), which are both fundamental for physics and engineering [2]. Both the ISQ and the SI are deeply integrated into Isabelle's type system. Quantities are parameterised by *dimension types*, which correspond to base vectors, and thus only quantities of the same dimension can be equated. Since the underlying "algebra of quantities" from [2] induces congruences on quantity and SI types, specific tactic support is developed to capture these. Our construction is validated by a test-set of known equivalences between both quantities and SI units. Moreover, the presented theory can be used for type-safe conversions between the SI system and others, like the British Imperial System (BIS).

2012 ACM Subject Classification Author: Please fill in 1 or more \ccsdesc macro

Keywords and phrases

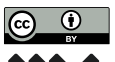
Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Modern Physics is based on the concept of quantifiable properties of physical phenomena such as mass, length, time, current, etc. These phenomena, called *quantities*, are linked via an *algebra of quantities* to derived concepts such as speed, force, and energy. The latter allows for a *dimensional analysis* of physical equations, which had already been the backbone of Newtonian Physics. In parallel, physicians developed their own research field called "metrology" defined as a scientific study of the *measurement* of physical quantities.

The relevant international standard for quantities and measurements is distributed by the *Bureau International des Poids et des Mesures* (BIPM), which also provides the *Vocabulaire International de Métrologie* (VIM) [2]. The VIM actually defines two systems: the *International System of Quantities* (ISQ) and the *International System of Units* (SI, abbreviated from the French 'Système international dunités'). The latter is also documented in the *SI Brochure* [3], a standard that is updated periodically, most recently in 2019. Finally, the VIM defines concrete reference measurement procedures as well as a terminology for measurement errors.

Conceived as a refinement of the ISQ, the SI comprises a coherent system of units of measurement built on seven base units, which are the metre, kilogram, second, ampere, kelvin, mole, candela, and a set of twenty prefixes to the unit names and unit symbols, such as milli- and kilo-, that may be used when specifying multiples and fractions of the units. The system also specifies names for 22 derived units, such as lumen and watt, for other common physical quantities. While there is still nowadays a wealth of different measuring systems such as the *British Imperial System* (BIS) and the *United States Customary System*



(USC), the SI is more or less the de-facto reference behind all these systems. [1]

The present Isabelle theory builds a formal model for both the ISQ and the SI, together with a deep integration into Isabelle's order-sorted polymorphic type system [10]. Quantities and units are represented in a way that they have a *quantity type* as well as a *unit type* based on its base vectors and their magnitudes. Since the algebra of quantities induces congruences on quantity and SI types, specific tactic support has been developed to capture these. Our construction is validated by a test-set of known equivalences between both quantities and SI units. Moreover, the presented theory can be used for type-safe conversions between the SI system and others, like the British Imperial System (BIS).

As a result of our theory development[2], it is possible to express "4500.0 kilogram times metre per second squared" has the type $\mathbb{R}[kg \cdot m \cdot s^{-3}]$. This type means that the magnitude `4500.0` (which by lexical convention is considered as a real number) of the dimension $M \cdot L \cdot T^{-3}$ is a quantity intended to be measured in the SI-system, which means that it actually represents a force measured in Newtons. Via a type synonym, the above type expression gets the type $\mathbb{R}$ `newton`.

In the example, the *magnitude* type part of this type is the real numbers $\mathbb{R}$. In general, however, magnitude types can be more general. If the term above is presented slightly differently as "4500 kilogram times metre per second squared", the inferred type will be $'\alpha[kg \cdot m \cdot s^{-3}]$ where $'\alpha$ is a magnitude belonging to the type-class numeral. This class comprises types like $\mathbb{N}$, $\mathbb{Z}$, 32 bit integers (`32word`), IEEE-754 floating-point numbers, as well as vectors belonging to the three-dimensional space $\mathbb{R}^3$, etc. Thus, our type-system allows to capture both conceptual entities in physics as well as implementation issues in concrete physical calculations on a computer.

As mentioned before, it is a main objective of this work to support the quantity calculus of ISQ and the resulting equations on derived SI entities (cf. [3]), both from a type checking as well as a proof-checking perspective. Our design objectives are not easily reconciled, however, and so some substantial theory engineering is required. On the one hand, we want a deep integration of dimensions and units into the Isabelle type system. On the other, we need to do normal-form calculations on types, so that, for example, the units $'\alpha[s \cdot m \cdot s^{-2}]$ and $'\alpha[m \cdot s^{-1}]$ can be equated.

Isabelle's type system follows the Curry-style paradigm, which rules out the possibility of direct calculations on type-terms (in contrast to Coq-like systems). However, our semantic interpretation of ISQ and SI requires the foundation of the heterogeneous equivalence relation $\cong_Q$ in semantic terms. This means that we can relate quantities with syntactically different dimension types, yet with same dimension semantics. This paves the way for derived rules that do computations of terms, which represent type computations indirectly. This principle is the basis for the tactic support, which allows for the dimensional type checking of key definitions of the SI system inside Isabelle/HOL, i. e. without making use of code-generated reflection. The resulting proof-support allows for an automatic check of physical equations such as those defining SI units.

1.1 The Plan of the Theory Development

In the following we describe the overall theory architecture in more detail. Our ISQ and SI model provides the following fundamental concepts:

---

[1] See also `https://en.wikipedia.org/wiki/International_System_of_Units`.

[2] The sources can be found in the Isabelle Archive of Formal Proofs at `https://www.isa-afp.org/entries/Physical_Quantities.html`

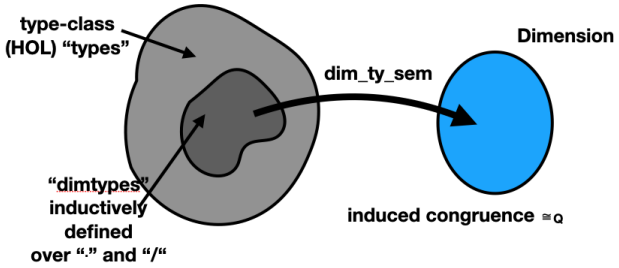


Figure 1 The "Inductive" Subset of *dim_types*-types interpreted in the *Dimension*-Type

1. the definition and theory of a vector space *dimensions* and the base vector terms **L**, **M**, **T**, **I**, **Θ**, **N**, **J** and their products and inverses as in the expression **M** · *L* / *T*.
2. the extension of dimensions by magnitudes to a structure of *quantities*, together with its terms ⦇ *mag, dim,* ... ⦈, products and inverses.
3. the extension of quantities to a structure of *measurement systems* with terms ⦇ *mag, dim, unit_sys* ⦈, scalar products, products and inverses.
4. the type definitions abstracting dimensions, quantities, and measurement systems, providing ISQ conform type symbols such as *L*, *M*, *T* and type expressions $L \cdot T^{-1} \cdot T^{-1} \cdot M$ as well as type expressions for measurement systems such as ℝ*[M · L / T, 's]*.
5. a *quantity calculus* consisting of *quantity equations*, i. e. rules resulting from the algebraic structure of dimensions, quantities, and measurement systems.
6. the abstraction of dimensions, quantities, and measurement systems induces an isomorphism on types: thus, $L \cdot T^{-1} \cdot T^{-1} \cdot M$ is isomorphic to $M \cdot L \cdot T^{-2}$ is isomorphic to *F* (the first type equals mass times acceleration which is equal to *force*). The isomorphism on types is established by a semantic interpretation in dimensions (c.f. Figure 1).
7. an instance of measurement systems providing types such as ℝ*[SIPRINT(M · L / T)]* together with syntax support for standardised SI-unit symbols such as *m*, *kg*, *s*, *A*, *K*, *mol*, and *cd*.
8. a standardised set of symbols of SI-prefixes for multiples of SI-units, such as *giga* ($=10^9$), *kilo* ($=10^3$), *milli* ($=10^{-3}$), etc.

2 Background: Some Advanced Isabelle Specification Constructs

This work uses a number of features of Isabelle/HOL and its meta-logic Isabelle/Pure, that are not necessarily available in another system of the LCF-Prover family and that needs therefore some explanation. These concepts are in particular:

2.1 Type-Classes

Type-classes and order-sorted parametric polymorphism [9, 10] in Isabelle provide Haskell-like type-classes that allow for types depend on constants and represent therefore a restricted form of dependent types (c.f. `https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/classes.pdf`). For example, it is possible to define a type-class *preorder* which carries syntactic as well as semantic requirements on type-instances:

```
class ord =
  fixes less_eq :: "'a ⇒ 'a ⇒ bool"
begin notation  less_eq  ("'(≤')")  end
```

```
class preorder = ord +
  assumes order_refl : "x ≤ x"
  and     order_trans : "x ≤ y ⟹ y ≤ z ⟹ x ≤ z"
begin ... end
```

An instantiation of this class with the concrete type ℕ has then the format:

```
instantiation nat :: preorder
begin
definition less_eq_nat :: "nat ⇒ nat ⇒ bool" where "x ≤ y =  ... "
instance   <instance proof establishing order_refl and order_trans>
end
```

Note that instantiations are also possible for parametric type constructors such as *'α list*: an instantiation may use the preorder of the argument class *'α* both in the definition as well as the proof part. Note further, that we will use *nat* and ℕ as type synonyms in this paper.

2.2 Types in Type-abstractions and as Terms

The meta-logic `Isabelle/Pure` provides mechanisms to denote types of sort 'types' in the type language as well in the term language: *'α itself* denotes an unspecified type and *TYPE* a constructor that injects the language of types into the language of terms. It is therefore possible to abstract a type as such from a term and mimick the effect of a type instantiation well-known in higher type-calculi by an ordinary application. For example, some function *f :: 'a itself ⇒ τ* may be instantiated via *f (TYPE(ℕ))*.

2.3 Types as Parameters and as Terms

`Isabelle` provides a powerful code generation framework that is both extremely versatile and reconfigurable as well as well supported by many specification constructs (c.f. `https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/codegen.pdf`). For example, datatype definitions or recursive function definitions automatically produce the necessary setups for the code-generator, which can be configured with formally proven coding-rules to produce efficient SML (and other) code. In our context, it is of major importance since the normalisation process of dimension types is computable, which paves the way for efficient decision procedures establishing the isomorphism of, e. g., dimension types. Note that the command **value** *"E"* makes direct use of the code-generator, as well as the *eval*-proof method.

2.4 The lifting package

Since type-definitions define new types τ_A in terms of a subset of values of an implementing (concrete) type τ_C along the isomorphism represented by functions *(Abs,Rep)*, a particular construction called *lifting* is a recurrent scheme for which Isabelle offers support by an own specification construct (see [6] for the theoretical foundation). The construction has the pattern for the binary case (cf. Figure 2):

```
lift_definition f_A :: "τ_A × τ_A ⇒ τ_A"  is "f_C"
```

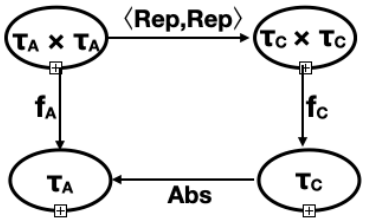


Figure 2 Lifting a binary operation f_C to f_A

Besides compactness and conciseness, liftings have the advantage to setup the code-generator appropriately, such that calculations on τ_A can be mapped automatically on τ_C.

3 Preliminaries: Basic Algebraic Structures

At the core, the multiplicative operation on physical dimension results in additions of the exponents of base vectors:

$$\begin{aligned}&(M^{\alpha 1}.\ L^{\alpha 2}.\ T^{\alpha 3}.\ I^{\alpha 4}.\ \Theta^{\alpha 5}.\ N^{\alpha 6}.\ J^{\alpha 7}) * (M^{\beta 1}.\ L^{\beta 2}.\ T^{\beta 3}.\ I^{\beta 4}.\ \Theta^{\beta 5}.\ N^{\beta 6}.\ J^{\beta 7})\\ =\ &(M^{\alpha 1+\beta 1}\ .\ L^{\alpha 2+\beta 2}\ .\ T^{\alpha 3+\beta 3}\ .\ I^{\alpha 4+\beta 4}\ .\ \Theta^{\alpha 5+\beta 5}\ .\ N^{\alpha 6+\beta 6}\ .\ J^{\alpha 7+\beta 7})\end{aligned}$$

This motivates type classes that represent this algebraic structure. We chose to represent it for the case of vectors of arbitrary length. We define the classes *group_mult* and the abelian multiplicative groups as follows:

```
notation times (infixl "·" 70)

class group_mult = inverse + monoid_mult +
  assumes left_inverse: "inverse a · a = 1"
  assumes multi_inverse_conv_div [simp]: "a · (inverse b) = a / b"
...
class ab_group_mult = comm_monoid_mult + group_mult
...
abbreviation(input) npower::"'α::{power,inverse} ⇒ nat ⇒ 'α" ("(_⁻¹⁻)"[1000,999]999)

  where "npower x n ≡ inverse (x ^ n)"
```

... and derive the respective properties:

```
lemma div_conv_mult_inverse : "a / b = a · (inverse b)" ...
lemma diff_self             : "a / a = 1" ...
lemma mult_distrib_inverse  : "(a * b) / b = a" ...
lemma mult_distrib_inverse' : "(a * b) / a = b" ...
lemma inverse_distrib       : "inverse (a * b)  =  (inverse a) * (inverse b)" ...
lemma inverse_divid         : "inverse (a / b) = b / a" ...
```

.

On this basis we define *dimension vectors* of arbitrary size via a type definition as follows:

```
typedef ('β, 'ν) dimvec = "UNIV :: ('ν::enum ⇒ 'β) set"
  morphisms dim_nth dim_lambda ..
```

.

Here, the functions *dim_nth* and *dim_lambda* represent the usual function pair that establish the isomorphism between the defined type *('β, 'ν) dimvec* and an implementing domain, in this case the universal set of type *('ν ⇒ 'β) set*. Note that the index-type *'ν* is restricted to be enumerable by type class *enum*.

Via a number of intermediate lemmas over types, we can finally establish the desired result in Isabelle compactly as follows:

```
instance dimvec :: (ab_group_add, enum) ab_group_mult  by (<proof omitted>)
```

If *'β* is an abelian additive group, and if the index type *'ν* is enumerable, *('β, 'ν) dimvec* is an abelian multiplicative group.

4 The Domain: ISQ Dimension Terms and Calculations

In the following, we will construct a concrete semantic domain as instance of *('β, 'ν) dimvec*. This is where the general model of the dimension vector space of Section 3 becomes a specific instance of the current ISQ standard as defined [2]; should physicians discover one day a new physical quantity, this would just imply a change of the following enumeration. Moreover, we will define the ISQ standards dimensions as *base vectors* in this vector space; historically, there had been alternative proposals of a quantity system that boil down to the choice of another eigen-vector set in this vector space.

The definition of an enumeration and the proof that it can be accommodated to the required infrastructure of the *enum*-class is straight-forward, and the construction of our domain *Dimension* follows immediately:

```
datatype sdim = Length | Mass | Time | Current | Temperature | Amount | Intensity

instantiation sdim :: enum
begin
  definition "enum_sdim = [Length,Mass,Time,Current,Temperature,Amount,Intensity]"
  definition "enum_all_sdim P  ⟷ P Length ∧ P Mass ∧ P Time ∧ ..."
  definition "enum_ex_sdim P   ⟷ P Length ∨ P Mass ∨ P Time ∨ ..."
  instance <proof omitted>
end

type_synonym Dimension = "(ℤ, sdim) dimvec"
```

Note that the *enum*-class stems from the Isabelle/HOL library and is intended to present sufficient infrastructure for the code-generator. Note, further, that [2] discusses also the possibility of rational exponents, but finally defines them as integer numbers $\mathbb{Z}$.

A base dimension is a dimension where precisely one component has power 1: it is the dimension of a base quantity. Here we define the seven base dimensions. For the concrete definition of the seven base vectors we define a constructor:

```
definition mk_BaseDim :: "sdim ⇒ Dimension" where
          "mk_BaseDim d = dim_lambda (λ i. if (i = d) then 1 else 0)"
```

which lets us achieve a first major milestone on our journey: a *term* representation of base vectors together with the capability to prove and to compute dimension-algebraic equivalences. We introduce the ISQ dimension symbols defined in [2]:

```
abbreviation LengthBD      ("L") where "L ≡ mk_BaseDim Length"
abbreviation MassBD        ("M") where "M ≡ mk_BaseDim Mass"
...
abbreviation "BaseDimensions ≡ {L, M, T, I, Θ, N, J}"

lemma BD_mk_dimvec [si_def]:
  "L = mk_dimvec [1, 0, 0, 0, 0, 0, 0]"
  "M = mk_dimvec [0, 1, 0, 0, 0, 0, 0]"
  ...
```

A demonstration of a computation [3] and a proof is shown in the example below:

```
value "L·M·T⁻²"

lemma "L·M·T⁻² = mk_dimvec [1, 1, - 2, 0, 0, 0, 0]"  by (simp add: si_def)
```

Note that the multiplication operation `(·)` is inherited from the fact that the `Dimension`-type is a proven instance of the `ab_group_mult`-class. So far, the language of dimensions is represented by a shallow embedding in the `Dimension` type.

5 Dimension Types and their Semantics in Terms of the `Dimension`-Type

The next section on our road is the construction of a sub-language of type-expressions. To this end, we define a *type class* by those type-terms for which we have an interpretation function `dim_ty_sem` into the values of the `Dimension`-type. For our construction it suffices that the type-symbols of this class have a *unitary*, i. e., one-elementary, carrier-set.

```
class dim_type = unitary +
  fixes   dim_ty_sem :: "'α itself ⇒ Dimension"

class basedim_type = dim_type +
  assumes is_BaseDim: "is_BaseDim (dim_ty_sem (TYPE('α)))"
```

Recall that the the type constructor `'α itself` from Isabelle/Pure denotes an unspecified type and `TYPE` a constructor that injects the language of types into the language of terms. We also introduce a sub-type-class `basedim_type` for base-dimensions.

[3] The command **value** compiles the argument to SML code and executes it

The definition of the basic dimension type constructors is straightforward via a one-elementary set, *unit set*. The latter is adequate since we need just an abstract syntax for type expressions, so just one value for the `dimension`-type symbols. We define types for each of the seven base dimensions, and also for dimensionless quantities.

```
typedef Length = "UNIV :: unit set" .. setup_lifting type_definition_Length
type_synonym L = Length
typedef Mass   = "UNIV :: unit set" .. setup_lifting type_definition_Mass
type_synonym M = Mass
...
```

The following instantiation proof places the freshly constructed type symbol *L* in the class *basedim_type* by setting its semantic interpretation to the corresponding value in the *Dimension*-type.

```
instantiation Length :: basedim_type
begin
definition [si_eq]: "dim_ty_sem_Length (α::Length itself) = L"
instance <proof omitted>
end
```

Note that Isabelle enforces a convention to name the definition of an operation assumed in the interface of the class to be the concatenation of the interface name (e. g. *dim_ty_sem*) and the name of the class intantiation (e. g. *Length*). For the other 6 base-types we proceed analogously.

Dimension type expressions can be constructed by multiplication and division of the base dimension types above. Consequently, we need to define multiplication and inverse operators at the type level as well. On the class of dimension types (in which we have already inserted the base dimension types), the definitions of the type constructors for inner product and inverse is straightforward.

```
typedef ('α::dim_type, 'β::dim_type) DimTimes (infixl "·" 69) = "UNIV :: unit set"
..
setup_lifting type_definition_DimTimes
```

The type $'\alpha \cdot '\beta$ is parameterised by two types, $'\alpha$ and $'\beta$ that must both be elements of the *dim_type* class. As with the base dimensions, it is a unitary type as its purpose is to represent dimension type expressions. We instantiate *dim_type* with this type, where the semantics of a product dimension expression is the product of the underlying dimensions. This means that multiplication of two dimension types yields a dimension type.

```
instantiation DimTimes :: (dim_type, dim_type) dim_type
begin
  definition dim_ty_sem_DimTimes :: "('α · 'β) itself ⇒ Dimension" where
  [si_eq]: "dim_ty_sem_DimTimes x = (dim_ty_sem TYPE('α)) · (dim_ty_sem TYPE('β))"
  instance <proof>
end
```

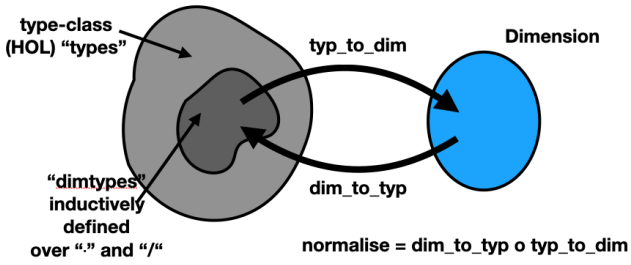


Figure 3 The "Inductive" subset of *dim_types* interpreted in SML Lists

Thus, the semantic interpretation of the product of two *dim_type*'s is a homomorphism over the product of two dimensions. Similarly, we define the inversion of dimension types and prove that dimension types are closed under this operation.

```
typedef 'α DimInv ("(_⁻¹)" [999] 999) = "UNIV :: unit set" ..
setup_lifting type_definition_DimInv
instantiation DimInv :: (dim_type) dim_type
begin
  definition dim_ty_sem_DimInv :: "('-¹) itself ⇒ Dimension" where
  [si_eq]: "dim_ty_sem_DimInv x = inverse (dim_ty_sem TYPE('α))"
  instance <proof>
end
```

Finally, we introduce some syntactic sugar such as $'\alpha^4$ for $'\alpha{\cdot}'\alpha{\cdot}'\alpha{\cdot}'\alpha$ or $'\alpha^{-4}$ for $('\alpha^4)^{-1}$.

By the way, we also implemented two morphisms on the SML-level underlying Isabelle, which is straight-forward and omitted here (C.f. Figure 3). These functions yield for:

```
ML⟨ Dimension_Type.typ_to_dim @{typ "L⁻²·M⁻¹·T⁴·I²·M"};
    Dimension_Type.dim_to_typ [1,2,0,0,0,3,0];
    Dimension_Type.normalise @{typ "L⁻²·M⁻¹·T⁴·I²·M"}⟩
```

the system output:

```
val it = [~2, 0, 4, 2, 0, 0, 0]: int list
val it = "L · M² · N³": typ
val it = "L⁻² · T⁴ · I²": typ
```

6 ISQ Quantity and SI Types

6.1 The Semantic Domain of Physical Quantities

Here, we give a semantic domain for particular values of physical quantities. A quantity is usually expressed as a number and a measurement unit, and the goal is to support this. First, though, we give a more general semantic domain where a quantity has a magnitude and a dimension.

```
record ('α) Quantity =
  mag  :: 'α           — Magnitude of the quantity.
  dim  :: "Dimension" — Dimension of the quantity – denotes the kind of quantity.
```

The magnitude type is parametric as we permit the magnitude to be represented using any kind of numeric type, such as $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$, though we usually minimally expect a field.

By a number of class instantiations, we lift the type *'α Quantity* into the class *comm_monoid_mult*, provided that the magnitude is of that class. The following homomorphisms hold:

```
lemma mag_times  [simp]: "mag (x · y) = mag x · mag y" <proof>
lemma dim_times  [simp]: "dim (x · y) = dim x · dim y" <proof>
lemma mag_inverse [simp]: "mag (inverse x) = inverse (mag x)" <proof>
lemma dim_inverse [simp]: "dim (inverse x) = inverse (dim x)" <proof>
```

```
record ('α, 's::unit_system) Measurement_System = "('α) Quantity" +
  unit_sys  :: 's — The system of units being employed
```

where *unit_system* again forces the carrier-set of its instances to have cardinality 1.

6.2 Dimension Typed Measurement Systems

We can now define the type of parameterized quantities *'α['d, 's]* by *('α, 's) Measurement_System*'s, which have a dimension equal to the semantic interpretation of the *'d*:

```
typedef (overloaded) ('α, 'd::dim_type, 's::unit_system) QuantT
                     ("_[_, _]" [999,0,0] 999)
                     = "{x :: ('α,'s)Measurement_System.
                             dim x = dim_ty_sem TYPE('d)}"
  morphisms fromQ toQ <non-emptyness proof omitted>

setup__lifting type_definition_QuantT
```

where *'s* is a tag-type characterizing the concrete measuring system (e. g., SI, BIS, UCS, ...). Via the class *unit_system* these tag-types are again restricted to carrier-sets of cardinality 1.

Intuitively, the term *x* can be read as "x is a quantity with magnitude of type *'α*, dimension type *'d*, and measured in system *'s*.

6.3 Operators in Typed Quantities

We define several operators on typed quantities. These variously compose the dimension types as well. Multiplication composes the two dimension types. Inverse constructs and inverted dimension type. Division is defined in terms of multiplication and inverse.

```
lift_definition
  qtimes :: "('α::comm_ring_1)['τ1::dim_type, 's::unit_system]
             ⇒ 'α['τ2::dim_type, 's] ⇒ 'α['τ1 ·'τ2, 's]"
             (infixl "·" 69)
  is "(*)" <proof>

lift_definition
  qinverse :: "('α::field)['τ::dim_type,'s::unit_system] ⇒'α['τ-¹,'s]"
              ("(_⁻¹)"[999]999)
  is "inverse" <proof>
```

Additionally, a scalar product $(*_Q)$ and an addition on the magnitude component is introduced that preserves the algebraic properties of the magnitude type:

```
lift_definition
   scaleQ::"'α⇒'α::comm_ring_1['d::dim_type,'s::unit_system]⇒'α['d,'s]"
           (infixr"*Q"63)
   is "λ r x. ⦇ mag = r * mag x, dim = dim_ty_sem TYPE('d), unit_sys = unit ⦈"
       <proof>

instantiation QuantT :: (plus, dim_type, unit_system) plus
begin
lift_definition
  plus_QuantT :: "'α['d, 's] ⇒ 'α['d, 's] ⇒ 'α['d, 's]"
  is "λ x y. ⦇ mag = mag x+mag y, dim = dim_ty_sem TYPE('d),unit_sys = unit ⦈"
     <proof>
instance ..
end
```

6.4 Predicates on Typed Quantities

The standard HOL order $(\leq)$ and equality $(=)$ have the homogeneous type `'a ⇒ 'a ⇒ bool` and so they cannot compare values of different types. Consequently, we define a heterogeneous order and equivalence on typed quantities. Both operations were defined as lifting of the core operations, and exploit the semantic interpretation of `'d` as shown in Figure 1 and were constructed as liftings over the equality and order on `Dimension`'s.

```
lift_definition
    qless_eq :: "'α::order['d::dim_type,'s::unit_system] ⇒ 'α['d,'s] ⇒ bool"
                (infix "≲Q" 50) is "(≤)" .

lift_definition
    qequiv ::  "'α['d1::dim_type, 's::unit_system]⇒'α['d2::dim_type,'s] ⇒ bool"

                (infix "≅Q" 50) is "(=)".
```

These are both fundamentally the same as the usual order and equality relations, but they permit potentially different dimension types, `'d1` and `'d2`. Two typed quantities are comparable only when the two dimension types have the same semantic dimension.

The equivalence properties on $(\cong_Q)$ hold and even a restricted form of congruence inside the *dim_type*'s can be established.

6.5 SI as Typed Quantities

It is now straight-forward to define an appropriate tag-type *SI* and to introduce appropriate syntactic abbreviations that identify the type *'α[L,SI]* with *'α[m]*, *'α[M,SI]* with *'α[kg]*, *'α[T,SI]* with *'α[s]*, etc, i. e. the standard's symbols for measurements in the 'système international des mesurements' (SI). Since these are just syntactic shortcuts, all operations and derived propertied in this section also apply to the SI system, as well as equivalent presentations of the British Imperial System (BIS) or the US-customer system (UCS). If needed, type-safe conversion operations between these systems can be defined, whose precision will depend on the underlying magnitude types, however.

6.6 SI Prefixes

The VIM defines in its SI part aa number of prefixes are simply numbers that can be composed with units using the scalar multiplication operator $(*_Q)$. These are in particular:

```
definition deca :: "'a" where [si_eq]: "deca = 10^1"
definition hecto:: "'a" where [si_eq]: "hecto = 10^2"
definition kilo :: "'a" where [si_eq]: "kilo = 10^3"
definition mega :: "'a" where [si_eq]: "mega = 10^6"

...

definition deci  :: "'a" where [si_eq]: "deci = 1/10^1"
definition centi :: "'a" where [si_eq]: "centi = 1/10^2"
definition milli :: "'a" where [si_eq]: "milli = 1/10^3"
definition micro :: "'a" where [si_eq]: "micro = 1/10^6"
...
```

For example, it is therefore possible to represent and prove the following scalar calculations directly by *si_simp*:

```
lemma "2.3 *Q (centi *Q metre)^3 = 2.3 · 1/10^6 *Q metre^3"
  by (si_simp)

lemma "1 *Q (centi *Q metre)^-1 = 100 *Q metre^-1"
  by (si_simp)
```

7 Validation by the VIM and the 'Brochure'

Of course, the question arises if our construction actually captures the SI standard, as described in the VIM and the 'Brochure'. The question can be answered in the sense that both as well as accompanying documents mention a number of equations and consequences

of the 'algebra of quantities' and the 'algebra of SI'. It is therefore possible to just represent the catalogue of equalities of the VIM inside our theory and check if the expected results turn out to be provable on our theory.

Note that the most general magnitude types we support must form a field into which the natural numbers can be injected.

Another large group of equations are just definitions that can be copied out of the book:

- *minute = (60::'a) $*_Q$ second*
- *hour = (60::'a) $*_Q$ minute*
- *day = (24::'a) $*_Q$ hour*
- *astronomical_unit = (149597870700::'a) $*_Q$ metre*
- *degree = (2::'a) · of_real pi / (180::'a) $*_Q$ radian*
- *n° ≡ n°*
- *litre = (1::'a) / (1000::'a) $*_Q$ metre^3*
- *tonne = (10::'a)^3 $*_Q$ kilogram*
- *dalton = (166053906660::'a) / (10::'a)^{11} · ((1::'a) / (10::'a)^{27}) $*_Q$ kilogram*
- etc.

On this basis, another group of unit equations can be recalculated in our theory just by unfolding these definitions:

lemma "*1 $*_Q$ hour = 3600 $*_Q$ second*" **by** *(si_simp)*

lemma "*watt · hour $\cong_Q$ 3600 $*_Q$ joule*" **by** *(si_calc)*

lemma "*25 $*_Q$ metre / second = 90 $*_Q$ (kilo $*_Q$ metre) / hour*" **by** *(si_calc)*

The Cs frequency *(9192631770::'a) $*_Q$ second^{-1}*, the speed of light **c**, the Planck constant **h** etc. are similar abbreviations:

abbreviation *caesium_frequency::* "*'a[T^{-1},SI]*" ("*$\Delta\nu_{Cs}$*") **where**
"*caesium_frequency ≡ 9192631770 $*_Q$ hertz*"

abbreviation *speed_of_light ::* "*'a[L · T^{-1},SI]*" ("**c**") **where**
"*speed_of_light ≡ 299792458 $*_Q$ (metre·second^{-1})*"

abbreviation *Planck ::* "*'a[M · L^2 · T^{-2} · T,SI]*" ("**h**") **where**
"*Planck ≡ (6.62607015 · 1/(10^34)) $*_Q$ (joule·second)*"

On this basis, we can finally re-check the foundational equations of the SI System:

theorem *second_definition:*
"*1 $*_Q$ second $\cong_Q$ (9192631770 $*_Q$ 1) / $\Delta\nu_{Cs}$*"
by *si_calc*

theorem *metre_definition:*
"*1 $*_Q$ metre $\cong_Q$ (**c** / (299792458 $*_Q$ 1))·second*"
"*1 $*_Q$ metre $\cong_Q$ (9192631770 / 299792458) $*_Q$ (**c** / $\Delta\nu_{Cs}$)*"
by *si_calc+*

```
theorem kilogram_definition:
  "   ((1 *Q kilogram)::'a kilogram)
   ≅Q (h / (6.62607015 · 1/(10^34) *Q 1))·metre⁻²·second"
  by si_calc
```

As equations give the concrete definitions for the metre and kilogram in terms of the physical constants **c** and **h**, we can be fairly that even typos in the numerical constants appearing in the definitions are excluded.

8 Conclusion and Related Work

We have presented a substantial theory development of about 2500 lines of definitions and proofs that captures the ISQ and SI as defined in the international standard of the VIM [2]. The theory that is generally available on the Isabelle/HOL Archive of Formal Proofs provides a type system for physical quantities and measurements that is by construction sound and complete. Given the fact that Isabelle's type-system is far from being trivial, we believe that this is both significant and useful for applications in the hybrid system domain. We provided a validation of our theory by checking the mandatory definitions and described corollaries in the VIM and the SI-Brochure[3]. The provided proof support allows to do this smoothly.

8.1 Related Work

There had been numerous direct implementations of ISQ and SI for programming languages: early implementations by Jean Goubault-Larrecq have been worked out in the early 90ies and is still implemented in GimML. At about the same time works on Dimension Types has been presented by Andrew Kennedy [7], and a more recent account along this line is by Garrigue and Ly [4]. All these works attempt to directly implement a type-system in an ML-like language, while our approach *formally derives* such type inference inside the framework of parameterized polymorphism and the theoretical framework of HOL.

As a formal development, this work has drawn inspiration from some previous formalisations of the ISQ and SI, notably Hayes and Mahoney's formalisation in Z[5] and Aragon's algebraic structure for physical quantities[1]. However, to the best of our knowledge, our mechanisation represents the most comprehensive account of ISQ and SI in a theory prover.

References

1 S. Aragon. The algebraic structure of physical quantities. *Journal of Mathematical Chemistry*, 31(1), May 2004.

2 Bureau International des Poids et Mesures and Joint Committee for Guides in Metrology. Basic and general concepts and associated terms (vim) (3rd ed.). Technical report, BIPM, JCGM, 2012. Version 2008 with minor corrections.

3 Bureau International des Poids et Mesures and Joint Committee for Guides in Metrology. The International System of Units (SI). Technical report, BIPM, JCGM, 2019. 9th edition.

4 J Garrigue and D Ly. Des unités dans le typeur. In *28ièmes Journées Francophones des Langages Applicatifs*, Gourette, France, January 2017. URL: `https://hal.archives-ouvertes.fr/hal-01503084`.

5 Ian J. Hayes and Brendan P. Mahony. Using units of measurement in formal specifications. *Formal Aspects of Computing*, 7(3):329–347, 1995. `doi:10.1007/BF01211077`.

6 Brian Huffman and Ondrej Kuncar. Lifting and transfer: A modular design for quotients in isabelle/hol. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs - Third*

International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings, volume 8307 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2013. URL: http://dx.doi.org/10.1007/978-3-319-03545-1_9, doi:10.1007/978-3-319-03545-1_9.

7 Andrew Kennedy. Dimension types. In Donald Sannella, editor, *Programming Languages and Systems — ESOP '94*, pages 348–362, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

8 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. doi:10.1007/3-540-45949-9.

9 Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *J. Funct. Program.*, 5(2):201–224, 1995. doi:10.1017/S0956796800001325.

10 Tobias Nipkow and Gregor Snelting. Type classes and overloading resolution via order-sorted unification. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 1991. doi:10.1007/3540543961_1.