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

Simon Foster       Burkhart Wolff

October 28, 2020
Abstract

The present Isabelle theory 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).
# Contents

1 ISQ and SI: An Introduction .................................................. 7

2 Preliminaries ........................................................................... 11
   2.1 Integer Powers ................................................................. 11
   2.2 Enumeration Extras ......................................................... 12
     2.2.1 First Index Function .................................................. 12
     2.2.2 Enumeration Indices .................................................. 13
   2.3 Multiplication Groups ....................................................... 14

3 International System of Quantities ........................................... 17
   3.1 Quantity Dimensions ....................................................... 17
     3.1.1 Preliminaries ............................................................. 17
     3.1.2 Dimension Vectors ..................................................... 18
     3.1.3 Code Generation ........................................................ 19
     3.1.4 Dimension Semantic Domain ....................................... 21
     3.1.5 Dimension Type Expressions ....................................... 23
     3.1.6 ML Functions ............................................................ 27
   3.2 Quantities ........................................................................ 28
     3.2.1 Quantity Semantic Domain .......................................... 28
     3.2.2 Measurement Systems ............................................... 32
     3.2.3 Dimension Typed Quantities ...................................... 34
     3.2.4 Predicates on Typed Quantities .................................... 35
     3.2.5 Operators on Typed Quantities .................................... 36
   3.3 Proof Support for Quantities ............................................... 40
   3.4 Algebraic Laws .................................................................. 42
     3.4.1 Quantity Scale ........................................................... 42
     3.4.2 Field Laws ................................................................. 43
   3.5 Units .............................................................................. 44
   3.6 Conversion Between Unit Systems ....................................... 44
     3.6.1 Conversion Schemas .................................................... 44
     3.6.2 Conversion Algebra .................................................... 45
     3.6.3 Conversion Functions ................................................ 47
   3.7 Meta-Theory for ISQ ......................................................... 49
4 International System of Units

4.1 SI Units Semantics ........................................ 51
4.1.1 Example Unit Equations ................................. 52
4.1.2 Metrification ............................................ 52

4.2 Centimetre-Gram-Second System ............................ 53
4.2.1 Preliminaries ........................................... 53
4.2.2 Base Units ............................................... 53
4.2.3 Conversion to SI ......................................... 53
4.2.4 Conversion Examples .................................... 54

4.3 Physical Constants ......................................... 54
4.3.1 Core Derived Units ..................................... 54
4.3.2 Constants ............................................... 55
4.3.3 Checking Foundational Equations of the SI System .... 55

4.4 SI Prefixes .................................................. 56
4.4.1 Definitions ............................................... 56
4.4.2 Examples ................................................ 57
4.4.3 Binary Prefixes ......................................... 57

4.5 Derived SI-Units ........................................... 58
4.5.1 Definitions ............................................... 58
4.5.2 Equivalences ............................................ 59
4.5.3 Properties ............................................... 60

4.6 Non-SI Units Accepted for SI use .......................... 60
4.6.1 Example Unit Equations ................................. 60

4.7 Imperial Units via SI Units .................................. 61
4.7.1 Units of Length .......................................... 61
4.7.2 Units of Mass ............................................ 61
4.7.3 Other Units ............................................. 62
4.7.4 Unit Equations ........................................... 62

4.8 Meta-Theory for SI Units ................................... 62

4.9 Astronomical Constants .................................... 62

4.10 Parsing and Pretty Printing of SI Units .................... 63
4.10.1 Syntactic SI Units ..................................... 63
4.10.2 Type Notation .......................................... 64
4.10.3 Value Notations ........................................ 65

4.11 British Imperial System (1824/1897) ....................... 66
4.11.1 Preliminaries .......................................... 66
4.11.2 Base Units ............................................. 66
4.11.3 Derived Units .......................................... 66
4.11.4 Conversion to SI ....................................... 67
4.11.5 Conversion Examples .................................... 67
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 (d’unités)). 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.

The present Isabelle theory builds a formal model for both the ISQ and the SI, together with a deep integration into Isabelle’s type system [5]. Quan-
tities 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).

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

1. dimensions represented by a type \( (\text{int}, \ 'd::\text{enum}) \ dimvec \), i.e. a \('d\)-index vector space of integers representing the exponents of the dimension vector. \('d\) is constrained to be a dimension type later.

2. quantities represented by type \((\ 'a, \ 'd::\text{enum}) \ Quantity\), which are constructed as a vector space and a magnitude type \('a\).

3. quantity calculus consisting of quantity equations allowing to infer that \(LT^{-1}T^{-1}M = MLT^{-2} = F\) (the left-hand-side equals mass times acceleration which is equal to force).

4. a kind of equivalence relation \(\cong_Q\) on quantities, permitting to relate quantities of different dimension types.

5. base quantities for length, mass, time, electric current, temperature, amount of substance, and luminous intensity, serving as concrete instance of the vector instances, and for syntax a set of the symbols \(L, M, T, I, \Theta, N, J\) corresponding to the above mentioned base vectors.

6. (Abstract) Measurement Systems represented by type \((\ 'a, \ 'd::\text{enum}, \ 's::\text{unit system}) \ Measurement\_System\), which are a refinement of quantities. The refinement is modelled by a polymorphic record extensions; as a consequence, Measurement Systems inherit the algebraic properties of quantities.

7. derived dimensions such as volume \(L^3\) or energy \(ML^2T^{-2}\) corresponding to derived quantities.

Then, through a fresh type-constructor \(SI\), the abstract measurement systems are instantiated to the SI system — the British Imperial System (BIS) is constructed analogously. Technically, \(SI\) is a tag-type that represents the fact that the magnitude of a quantity is actually a quantifiable entity in the sense of the SI system. In other words, this means that the magnitude 1 in quantity \(1/L\) actually refers to one metre intended to be measured according to the SI standard. At this point, it becomes impossible, for example, to add to one foot, in the sense of the BIS, to one metre in the SI without creating a type-inconsistency.
The theory of the SI is created by specialising the Measurement_System-type with the SI-tag-type and adding new infrastructure. The SI theory provides the following fundamental concepts:

1. measuring units and types corresponding to the ISQ base quantities such as metre, kilogram, second, ampere, kelvin, mole and candela (together with procedures how to measure a metre, for example, which are defined in accompanying standards);

2. a standardised set of symbols for units such as m, kg, s, A, K, mol, and cd;

3. 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.; and a set of

4. unit equations and conversion equations such as \(J = kgm^2/s^2\) or \(1km/h = 1/3.6 m/s\).

As a result, it is possible to express “4500.0 kilogram times metre per second squared” which has the type \(\mathbb{R} [M \cdot L \cdot T^{-3} \cdot SI]\). This type means that the magnitude 4500 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. In the example, the magnitude type of the measurement unit is the real numbers (\(\mathbb{R}\)). In general, however, magnitude types can be arbitrary types from the HOL library, so for example integer numbers (\(int\)), integer numbers representable by 32 bits (\(int32\)), IEEE-754 floating-point numbers (\(float\)), or, a vector in the three-dimensional space \(\mathbb{R}^3\). 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 \(m\) and \(ms^{-1}\)s 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 allows for the foundation of the heterogeneous equivalence relation \(\equiv_Q\) 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. Some examples are given below.

**theorem** metre-definition:
\[
1 \ast_Q \text{metre} \equiv_Q (c / (299792458 \ast_Q 1)) \cdot \text{second}
\]
by *si-calc*

**theorem** kilogram-definition:
\[
1 \ast_Q \text{kilogram} \equiv_Q (h / (6.62607015 \cdot 1/(10^\text{34}) \ast_Q 1)) \cdot \text{metre}^{-2} \cdot \text{second}
\]
by *si-calc*

These equations are both adapted from the SI Brochure, and give the concrete definitions for the metre and kilogram in terms of the physical constants \(c\) (speed of light) and \(h\) (Planck constant). They are both proved using the tactic *si-calc*.

This work has drawn inspiration from some previous formalisations of the ISQ and SI, notably Hayes and Mahoney’s formalisation in Z [4] and Aragon’s algebraic structure for physical quantities [1]. To the best of our knowledge, our mechanisation represents the most comprehensive account of ISQ and SI in a theory prover.
Chapter 2

Preliminaries

2.1 Integer Powers

theory Power-int
  imports HOL.Real
begin

The standard HOL power operator is only for natural powers. This operator allows integers.

definition intpow :: 'a::{linordered_field} ⇒ int ⇒ 'a (infixr ^ Z 80) where
  intpow x n = (if (n < 0) then inverse (x ^ nat (-n)) else (x ^ nat n))

lemma intpow-zero [simp]: x ^ Z 0 = 1
  by (simp add: intpow-def)

lemma intpow-spos [simp]: x > 0 ⇒ x ^ Z n > 0
  by (simp add: intpow-def)

lemma intpow-one [simp]: x ^ Z 1 = x
  by (simp add: intpow-def)

lemma one-intpow [simp]: 1 ^ Z n = 1
  by (simp add: intpow-def)

lemma intpow-plus: x > 0 ⇒ x ^ Z (m + n) = x ^ Z m * x ^ Z n
  apply (simp add: intpow-def field-simps power-add)
  apply (metis (no-types, hide-lams) abs-ge-zero add-commute add-diff-cancel-right'
    nat-add-distrib power-add uminus-add-conv-diff zabs-def)
  done

lemma intpow-mult-combine: x > 0 ⇒ x ^ Z m * (x ^ Z n * y) = x ^ Z (m + n)
  * y
  by (simp add: intpow-plus)

lemma intpow-pos [simp]: n ≥ 0 ⇒ x ^ Z n = x ^ nat n
by (simp add: intpow-def)

lemma intpow-uminus: \( x \cdot z^{-n} = \text{inverse } (x \cdot z^n) \)
by (simp add: intpow-def)

lemma intpow-uminus-nat: \( n \geq 0 \implies x \cdot z^{-n} = \text{inverse } (x \cdot \text{nat } n) \)
by (simp add: intpow-def)

lemma intpow-inverse: \( \text{inverse } a \cdot z^n = \text{inverse } (a \cdot z^n) \)
by (simp add: intpow-def power-inverse)

lemma intpow-mult-distrib: \( (x \cdot y) \cdot z^m = x \cdot z^m \cdot y \cdot z^m \)
by (simp add: intpow-def power-mult-distrib)

end

2.2 Enumeration Extras

theory Enum-extra
imports HOL-Library.Cardinality
begin

2.2.1 First Index Function

The following function extracts the index of the first occurrence of an element in a list, assuming it is indeed an element.

fun first-ind :: 'a list ⇒ 'a ⇒ nat ⇒ nat where
first-ind [] y i = undefined |
first-ind (x # xs) y i = (if (x = y) then i else first-ind xs y (Suc i))

lemma first-ind-length:
\( x \in \text{set}(xs) \implies \text{first-ind } xs \cdot x \cdot i < \text{length}(xs) + i \)
by (induct xs arbitrary: i, auto, metis add-Suc-right)

lemma nth-first-ind:
\[ \text{distinct } xs; x \in \text{set}(xs) \implies xs ! (\text{first-ind } xs \cdot x \cdot i - i) = x \]
apply (induct xs arbitrary: i)
apply (auto)
apply (metis One-nat-def add.right-neutral add-Suc-right add-diff-cancel-left diff-diff-left empty-iff first-ind.simps(2) list.set(1) nat.simps(3) neq-Nil-conv nth-Cons' zero-diff)
done

lemma first-ind-nth:
\[ \text{distinct } xs; i < \text{length } xs \implies \text{first-ind } xs \cdot (xs ! i) \cdot j = i + j \]
apply (induct xs arbitrary: i j)
apply (auto)
apply (metis less-Suc-eq-le nth-equal-first-eq)
2.2. ENUMERATION EXTRAS

using less-Suc-eq-0-disj apply auto
done

2.2.2 Enumeration Indices

syntax
-ENUM :: type ⇒ logic (ENUM('·'))

translations
-ENUM('a) => CONST Enum.enum :: ('a::enum) list

Extract a unique natural number associated with an enumerated value by using its index in the characteristic list enum-class.enum.

definition enum-ind :: 'a::enum ⇒ nat
enum-ind (x :: 'a::enum) = first-ind ENUM('a) x 0

lemma length-enum-CARD: length ENUM('a) = CARD('a)
by (simp add: UNIV-enum distinct-card enum-distinct)

lemma CARD-length-enum: CARD('a) = length ENUM('a)
by (simp add: length-enum-CARD)

lemma enum-ind-less-CARD [simp]:
enum-ind (x :: 'a::enum) < CARD('a)
using first-ind-length[of x, OF in-enum, of 0] by (simp add: enum-ind-def CARD-length-enum)

lemma enum-nth-ind [simp]:
Enum.enum ! (enum-ind x) = x
using nth-first-ind[of Enum.enum x 0, OF enum-distinct in-enum] by (simp add: enum-ind-def)

lemma enum-distinct-conv-nth: 
assumes i < CARD('a) j < CARD('a) ENUM('a) ! i = ENUM('a) ! j
shows i = j
proof –
have (∀ i<length ENUM('a). ∀ j<length ENUM('a). i ≠ j −→ ENUM('a) ! i ≠ ENUM('a) ! j)
using distinct-conv-nth[of ENUM('a), THEN sym] by (simp add: enum-distinct)

with assms show ?thesis
by (auto simp add: CARD-length-enum)
qed

lemma enum-ind-nth [simp]:
assumes i < CARD('a::enum)
shows enum-ind (ENUM('a) ! i) = i
using assms first-ind-nth[of ENUM('a) i 0, OF enum-distinct]
by (simp add: enum-ind-def CARD-length-enum)

lemma enum-ind-spec:
enum-ind (x :: 'a::enum) = (THE i. i < CARD('a) ∧ Enum.enum ! i = x)
proof (rule sym, rule the-equality, safe)
  show \(\text{enum-ind } x < \text{CARD}(\text{a})\)
    by (simp add: enum-ind-less-CARD[of x])
  show \(\text{enum-class.enum ! enum-ind } x = x\)
    by simp
  show \(\forall i. i < \text{CARD}(\text{a}) \Longrightarrow x = \text{ENUM}(\text{a}) ! i \Longrightarrow i = \text{enum-ind}(\text{ENUM}(\text{a})! i)\)
    by (simp add: enum-ind-nth)
qed

lemma \(\text{enum-ind-inj}: \text{inj}(\text{enum-ind} :: 'a::enum \Rightarrow \text{nat})\)
  by (rule inj-on-inverseI[of - (\lambda i. \text{ENUM}(\text{a}) ! i), simp])

lemma \(\text{enum-ind-neq [simp]}: x \neq y \Longrightarrow \text{enum-ind } x \neq \text{enum-ind } y\)
  by (simp add: enum-ind-inj inj-eq)

end

2.3 Multiplication Groups

theory Groups-mult
  imports Main
begin

The HOL standard library only has groups based on addition. Here, we build one based on multiplication.

notation \(\text{times} (\text{infixl} \cdot 70)\)

class \text{group-mult} = \text{inverse} + \text{monoid-mult} +
  assumes \(\text{left-inverse}: \text{inverse } a \cdot a = 1\)
  assumes \(\text{multi-inverse-conv-div [simp]}: a \cdot (\text{inverse } b) = a / b\)
begin

lemma \(\text{div-conv-mult-inverse}: a / b = a \cdot (\text{inverse } b)\)
  by simp

sublocale \text{mult: group times 1 inverse}
  by standard (simp-all add: left-inverse)

lemma \(\text{diff-self [simp]}: a / a = 1\)
  using mult.right-inverse by auto

lemma \(\text{mult-distrib-inverse [simp]}: (a * b) / b = a\)
  by (metis local.mult-1-right local.multi-inverse-conv-div mult.right-inverse mult-assoc)

end

class \text{ab-group-mult} = \text{comm-monoid-mult} + \text{group-mult}
begin
2.3. MULTIPLICATION GROUPS

lemma mult-distrib-inverse [simp]: \((a * b) / a = b\)
using local.mult-distrib-inverse mult-commute by fastforce

lemma inverse-distrib: inverse \((a * b)\) = \((inverse a) * (inverse b)\)
by (simp add: local.mult.inverse-distrib-swap mult-commute)

lemma inverse-divide [simp]: inverse \((a / b)\) = \(b / a\)
by (metis div-conv-mult-inverse inverse-distrib mult.commute mult.inverse-inverse)

end

abbreviation (input) npower :: 'a::{power,inverse} ⇒ nat ⇒ 'a ((−−) [1000, 999]
999)
where npower x n ≡ inverse \((x ^ n)\)

end
Chapter 3

International System of Quantities

3.1 Quantity Dimensions

theory ISQ-Dimensions
  imports Groups-mult Power-int Enum-extra
       HOL.Transcendental
       HOL-Eisbach.Eisbach
begin

3.1.1 Preliminaries

class unitary = finite +
  assumes unitary-unit-pres: card (UNIV::'a set) = 1
begin

definition unit = (undefined::'a)

lemma UNIV-unitary: UNIV = {a::'a}
proof –
  have card(UNIV :: 'a set) = 1
    by (simp add: local.unitary-unit-pres)
  thus ?thesis
    by (metis (full-types) UNIV-I card-1-singletonE empty-iff insert-iff)
qed

lemma eq-unit: (a::'a) = b
  by (metis (full-types) UNIV-unitary iso-tuple-UNIV-I singletonD)

end

lemma unitary-intro: (UNIV::'s set) = {a} \implies OFCLASS('s, unitary-class)
apply (intro-classes, auto)
using finite.simps apply blast
using card-1-singleton-iff apply blast done

named-theorems si-def and si-eq

instantiation unit :: comm-monoid-add begin
definition zero-unit = ()
deinition plus-unit (x::unit) (y::unit) = ()
instance proof qed (simp-all)
end

instantiation unit :: comm-monoid-mult begin
definition one-unit = ()
definition times-unit (x::unit) (y::unit) = ()
instance proof qed (simp-all)
end

instantiation unit :: inverse begin
definition inverse-unit (x::unit) = ()
definition divide-unit (x::unit) (y::unit) = ()
instance ..
end

instance unit :: ab-group-mult by (intro-classes, simp-all)

3.1.2 Dimension Vectors

Quantity dimensions are used to distinguish quantities of different kinds. Only quantities of the same kind can be compared and combined: it is a mistake to add a length to a mass, for example. Dimensions are often expressed in terms of seven base quantities, which can be combined to form derived quantities. Consequently, a dimension associates with each of the base quantities an integer that denotes the power to which it is raised. We use a special vector type to represent dimensions, and then specialise this to the seven major dimensions.

typedef ('n, 'd) dimvec = UNIV :: ('d::enum ⇒ 'n) set
morphisms dim-nth dim-lambda ..

declare dim-lambda-inject [simplified, simp]
declare dim-nth-inverse [simp]
declare dim-lambda-inverse [simplified, simp]

instantiation dimvec :: (zero, enum) one begin
### 3.1. QUANTITY DIMENSIONS

**definition** `one-dimvec` :: \((a, 'b)\) `dimvec` where `one-dimvec = dim-lambda (\lambda i. 0)`

**instance** ..

**end**

**instantiation** `dimvec` :: `(plus, enum) times`

**begin**

**definition** `times-dimvec` :: \((a, 'b)\) `dimvec` \Rightarrow \((a, 'b)\) `dimvec` where `times-dimvec x y = dim-lambda (\lambda i. dim-nth x i + dim-nth y i)`

**instance** ..

**end**

**instance** `dimvec` :: `(comm-monoid-add, enum) comm-monoid-mult` by `(intro-classes; simp add: times-dimvec-def one-dimvec-def fun-eq-iff add.assoc, simp add: add.commute)`

We also define the inverse and division operations, and an abelian group, which will allow us to perform dimensional analysis.

**instantiation** `dimvec` :: `{plus,uminus}, enum) inverse` **begin**

**definition** `inverse-dimvec` :: \((a, 'b)\) `dimvec` \Rightarrow \((a, 'b)\) `dimvec` where `inverse-dimvec x = dim-lambda (\lambda i. - dim-nth x i)`

**definition** `divide-dimvec` :: \((a, 'b)\) `dimvec` \Rightarrow \((a, 'b)\) `dimvec` \Rightarrow \((a, 'b)\) `dimvec` where [code-unfold]: `divide-dimvec x y = x * (inverse y)`

**instance** ..

**end**

**instance** `dimvec` :: `(ab-group-add, enum) ab-group-mult` by `(intro-classes, simp-all add: inverse-dimvec-def one-dimvec-def times-dimvec-def divide-dimvec-def)`

#### 3.1.3 Code Generation

Dimension vectors can be represented using lists, which enables code generation and thus efficient proof.

**definition** `mk-dimvec` :: `\'n\ list` \Rightarrow `(\'n::ring-1, \'d::enum) dimvec` where `mk-dimvec ds = (if (length ds = CARD(\'d)) then dim-lambda (\lambda d. ds ! enum-ind d) else 1)`

**code-datatype** `mk-dimvec`

**lemma** `mk-dimvec-inj`: inj-on (\`mk-dimvec :: \'n\ list \Rightarrow (\'n::ring-1, \'d::enum) dimvec\) \{xs. length xs = CARD(\'d)\}

**proof** (rule inj-onI, safe)
fix \( x \) \( y \) :: 'n list

assume \( a \): (mk-dimvec \( x \) :: ('n, 'd) dimvec) = mk-dimvec \( y \) length \( x \) = CARD('d)

length \( y \) = CARD('d)

have \( \forall i. \ i < \text{length} \ x \implies x \![i] = y \![i] \)

proof -
  fix \( i \)
  assume \( i < \text{length} \ x \)
  with \( a \) have enum-ind (ENUM('d) ![i]) = \( i \)
    by (simp)
  with \( a \) show \( x \![i] = y \![i] \)
    by (auto simp add: mk-dimvec-def fun-eq-iff, metis)
qed

then show \( x = y \)
  by (metis \( a(2) \) \( a(3) \) nth-equalityI)
qed

lemma mk-dimvec-eq-iff [simp]:
assumes length \( x \) = CARD('d)
length \( y \) = CARD('d)

shows ((mk-dimvec \( x \) :: ('n::ring-1, 'd::enum) dimvec) = mk-dimvec \( y \)) \iff (x = y)

by (rule inj-on-eq-iff [OF mk-dimvec-inj], simp-all add: assms)

lemma one-mk-dimvec [code, si-def]: (1::('n::ring-1, 'a::enum) dimvec) = mk-dimvec (replicate CARD('a) 0)

by (auto simp add: mk-dimvec-def one-dimvec-def)

lemma times-mk-dimvec [code, si-def]:
(mk-dimvec \( xs \) * mk-dimvec \( ys \) :: ('n::ring-1, 'a::enum) dimvec) =
(if (length \( xs \) = CARD('a) \& length \( ys \) = CARD('a))
then mk-dimvec (map \( \lambda (x, y). x + y \) (zip \( xs \) \( ys \)))
else if length \( xs \) = CARD('a) then mk-dimvec \( xs \) else mk-dimvec \( ys \))

by (auto simp add: times-dimvec-def mk-dimvec-def fun-eq-iff one-dimvec-def)

lemma power-mk-dimvec [si-def]:
infect (power (mk-dimvec \( xs \)) \( n \) :: ('n::ring-1, 'a::enum) dimvec) =
(if (length \( xs \) = CARD('a)) then mk-dimvec (map ((*) (of-nat \( n \))) \( xs \)) else mk-dimvec \( xs \))

by (induct \( n \), simp add: one-dimvec-def mk-dimvec-def)
(auto simp add: times-mk-dimvec zip-map-map[where \( f = \text{id} \), simplified] comp-def split-beta' zip-same-cone-map distr-right mult.commute)

lemma inverse-mk-dimvec [code, si-def]:
infect (mk-dimvec \( xs \) :: ('n::ring-1, 'a::enum) dimvec) =
(if (length \( xs \) = CARD('a)) then mk-dimvec (map uminus \( xs \)) else 1)

by (auto simp add: inverse-dimvec-def one-dimvec-def fun-eq-iff)

lemma divide-mk-dimvec [code, si-def]:
3.1. QUANTITY DIMENSIONS

\[(\text{mk-dimvec } xs / \text{mk-dimvec } ys :: (\text{'n::ring-1, 'a::enum}) \text{ dimvec}) = \]

\[
(\text{if } (\text{length } xs = \text{CARD('a)} \land \text{length } ys = \text{CARD('a)})
\text{ then mk-dimvec (map } (\lambda (x, y). x - y) \text{ (zip } xs \text{ ys))}
\text{ else if length } ys = \text{CARD('a) then mk-dimvec (map uminus } ys \text{ else mk-dimvec} \]

\[
\text{xs)}
\] by (auto simp add: divide-dimvec-def inverse-mk-dimvec times-mk-dimvec zip-map-map[where \( f=\text{id, simplified; comp-def split-beta} \])

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.

\[\text{definition mk-BaseDim :: 'd::enum} \Rightarrow (\text{int, 'd}) \text{ dimvec where}
\]

\[
\text{mk-BaseDim } d \equiv \text{dim-lambda } (\lambda i. \text{ if } (i = d) \text{ then 1 else 0)}
\]

\[\text{lemma mk-BaseDim-neq [simp]: } x \neq y \Rightarrow \text{mk-BaseDim } x \neq \text{mk-BaseDim } y
\]

\[\text{by (auto simp add: mk-BaseDim-def fun-eq-iff)}\]

\[\text{lemma mk-BaseDim-code [code]: mk-BaseDim } (d::'d::enum) = \text{mk-dimvec (list-update}
\]

\[
(\text{CARD('d) 0}) \text{ (enum-ind d) 1)}
\]

\[\text{by (auto simp add: mk-BaseDim-def mk-dimvec-def fun-eq-iff)}\]

\[\text{definition is-BaseDim :: (int, 'd::enum)} \text{ dimvec } \Rightarrow \text{bool where}
\]

\[
\text{is-BaseDim } x \equiv (\exists i. x = \text{dim-lambda } ((\lambda x. 0)(i := 1)))
\]

\[\text{lemma is-BaseDim-mk [simp]: is-BaseDim } (\text{mk-BaseDim } x)
\]

\[\text{by (auto simp add: mk-BaseDim-def is-BaseDim-def fun-eq-iff)}\]

3.1.4 Dimension Semantic Domain

We next specialise dimension vectors to the usual seven place vector.

\[\text{datatype sdim } = \text{Length} | \text{Mass} | \text{Time} | \text{Current} | \text{Temperature} | \text{Amount} | \text{Intensity}\]

\[\text{lemma sdim-UNIV [simp]}: \text{(UNIV :: sdim set)} = \{\text{Length, Mass, Time, Current, Temperature, Amount, Intensity}\}
\]

\[\text{using sdim.exhaust by blast}\]

\[\text{lemma CARD-sdim [simp]: CARD(sdim) = 7}
\]

\[\text{by (simp add: sdim-UNIV)}\]

\[\text{instantiation sdim :: enum begin}
\]

\[\text{definition enum-sdim = [Length, Mass, Time, Current, Temperature, Amount, Intensity]}\]

\[\text{definition enum-all-sdim P } \longleftrightarrow P \text{ Length } \land P \text{ Mass } \land P \text{ Time } \land P \text{ Current } \land P \text{ Temperature } \land P \text{ Amount } \land P \text{ Intensity}
\]

\[\text{definition enum-ex-sdim P } \longleftrightarrow P \text{ Length } \lor P \text{ Mass } \lor P \text{ Time } \lor P \text{ Current } \lor P \text{ Temperature } \lor P \text{ Amount } \lor P \text{ Intensity}
\]
instance
  by (intro-classes, simp-all add: sdim-UNIV enum-sdim-def enum-all-sdim-def enum-ex-sdim-def)
end

instantiation sdim :: card-UNIV
begin
  definition finite-UNIV = Phantom(sdim) True
  definition card-UNIV = Phantom(sdim) 7
  instance by (intro-classes, simp-all add: finite-UNIV-sdim-def card-UNIV-sdim-def)
end

lemma sdim-enum [simp]:
  enum-ind Length = 0 enum-ind Mass = 1 enum-ind Time = 2 enum-ind Current = 3
  enum-ind Temperature = 4 enum-ind Amount = 5 enum-ind Intensity = 6
  by (simp-all add: enum-ind-def enum-sdim-def)

type-synonym Dimension = (int, sdim) dimvec

abbreviation LengthBD (L) where L ≡ mk-BaseDim Length
abbreviation MassBD (M) where M ≡ mk-BaseDim Mass
abbreviation TimeBD (T) where T ≡ mk-BaseDim Time
abbreviation CurrentBD (I) where I ≡ mk-BaseDim Current
abbreviation TemperatureBD (Θ) where Θ ≡ mk-BaseDim Temperature
abbreviation AmountBD (N) where N ≡ mk-BaseDim Amount
abbreviation IntensityBD (J) where J ≡ mk-BaseDim Intensity

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]
  T = mk-dimvec [0, 0, 1, 0, 0, 0, 0]
  I = mk-dimvec [0, 0, 0, 1, 0, 0, 0]
  Θ = mk-dimvec [0, 0, 0, 0, 1, 0, 0]
  N = mk-dimvec [0, 0, 0, 0, 0, 1, 0]
  J = mk-dimvec [0, 0, 0, 0, 0, 0, 1]
  by (simp-all add: mk-BaseDim-code eval-nat-numeral)

The following lemma confirms that there are indeed seven unique base dimensions.

lemma seven-BaseDimensions: card BaseDimensions = 7
  by simp

We can use the base dimensions and algebra to form dimension expressions. Some examples are shown below.

term L·M·T⁻²
term M·L⁻³

The following lemma confirms that there are indeed seven unique base dimensions.

lemma seven-BaseDimensions: card BaseDimensions = 7
  by simp

We can use the base dimensions and algebra to form dimension expressions. Some examples are shown below.

term L·M·T⁻²
term M·L⁻³

The following lemma confirms that there are indeed seven unique base dimensions.

lemma seven-BaseDimensions: card BaseDimensions = 7
  by simp

We can use the base dimensions and algebra to form dimension expressions. Some examples are shown below.

term L·M·T⁻²
term M·L⁻³
3.1. QUANTITY DIMENSIONS

value $L \cdot M \cdot T^{-2}$

lemma $L \cdot M \cdot T^{-2} = \text{mk-dimvec} [1, 1, -2, 0, 0, 0, 0]$
  by (simp add: si-def)

3.1.5 Dimension Type Expressions

Classification

We provide a syntax for dimension type expressions, which allows representation of dimensions as types in Isabelle. This will allow us to represent quantities that are parametrised by a particular dimension type. We first must characterise the subclass of types that represent a dimension.

The mechanism in Isabelle to characterize a certain subclass of Isabelle type expressions are type classes. The following type class is used to link particular Isabelle types to an instance of the type Dimension. It requires that any such type has the cardinality $1::'a$, since a dimension type is used only to mark a quantity.

```plaintext
class dim-type = unitary +
  fixes dim-ty-sem :: 'a itself ⇒ Dimension
```

syntax

- $QD :: type ⇒ logic (QD(''))$

translations

$QD('a) == CONST \ dim-ty-sem \ TYPE('a)$

The notation $QD('a)$ allows to obtain the dimension of a dimension type $'a$.

The subset of basic dimension types can be characterized by the following type class:

```plaintext
class basedim-type = dim-type +
  assumes is-BaseDim: is-BaseDim QD('a)
```

Base Dimension Type Expressions

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.

```plaintext
typedef Length = UNIV :: unit set .. setup-lifting type-definition-Length
typedef Mass = UNIV :: unit set .. setup-lifting type-definition-Mass
typedef Time = UNIV :: unit set .. setup-lifting type-definition-Time
typedef Current = UNIV :: unit set .. setup-lifting type-definition-Current
typedef Temperature = UNIV :: unit set .. setup-lifting type-definition-Temperature
```
typedef Amount = UNIV :: unit set .. setup-lifting type-definition-Amount
typedef Intensity = UNIV :: unit set .. setup-lifting type-definition-Intensity
typedef NoDimension = UNIV :: unit set .. setup-lifting type-definition-NoDimension

type-synonym M = Mass
type-synonym L = Length
type-synonym T = Time
type-synonym I = Current
type-synonym Θ = Temperature
type-synonym N = Amount
type-synonym J = Intensity
type-notation NoDimension (1)

translations
(type) M <= (type) Mass
(type) L <= (type) Length
(type) T <= (type) Time
(type) I <= (type) Current
(type) Θ <= (type) Temperature
(type) N <= (type) Amount
(type) J <= (type) Intensity

Next, we embed the base dimensions into the dimension type expressions by instantiating the class basedim-type with each of the base dimension types.

instantiation Length :: basedim-type
begin
definition [si-eq]: dim-ty-sem-Length (::Length itself) = L
instance by (intro-classes, auto simp add: dim-ty-sem-Length-def, (transfer, simp)+)
end

instantiation Mass :: basedim-type
begin
definition [si-eq]: dim-ty-sem-Mass (::Mass itself) = M
instance by (intro-classes, auto simp add: dim-ty-sem-Mass-def, (transfer, simp)+)
end

instantiation Time :: basedim-type
begin
definition [si-eq]: dim-ty-sem-Time (::Time itself) = T
instance by (intro-classes, auto simp add: dim-ty-sem-Time-def, (transfer, simp)+)
end

instantiation Current :: basedim-type
begin
definition [si-eq]: dim-ty-sem-Current (::Current itself) = I
instance by (intro-classes, auto simp add: dim-ty-sem-Current-def, (transfer, simp)+)
end
3.1. QUANTITY DIMENSIONS

instantiation Temperature :: basedim-type
begin
definition [si-eq]: dim-ty-sem-Temperature (-::Temperature itself) = \Theta
instance by (intro-classes, auto simp add: dim-ty-sem-Temperature-def, (transfer, simp)+)
end

instantiation Amount :: basedim-type
begin
definition [si-eq]: dim-ty-sem-Amount (-::Amount itself) = N
instance by (intro-classes, auto simp add: dim-ty-sem-Amount-def, (transfer, simp)+)
end

instantiation Intensity :: basedim-type
begin
definition [si-eq]: dim-ty-sem-Intensity (-::Intensity itself) = J
instance by (intro-classes, auto simp add: dim-ty-sem-Intensity-def, (transfer, simp)+)
end

instantiation NoDimension :: dim-type
begin
definition [si-eq]: dim-ty-sem-NoDimension (-::NoDimension itself) = (1::Dimension)
instance by (intro-classes, auto simp add: dim-ty-sem-NoDimension-def, (transfer, simp)+)
end

lemma base-dimension-types [simp]:
  is-BaseDim QD(Length) is-BaseDim QD(Mass) is-BaseDim QD(Time) is-BaseDim QD(Current)
  is-BaseDim QD(Temperature) is-BaseDim QD(Amount) is-BaseDim QD(Intensity)
  by (simp-all add: is-BaseDim)

Dimension Type Constructors: Inner Product and Inverse

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 ('a::dim-type, 'b::dim-type) DimTimes (infixl · 69) = UNIV :: unit set

setup-lifting type-definition-DimTimes

The type 'a · 'b is parameterised by two types, 'a and 'b 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 :: (′a · ′b) itself ⇒ Dimension where
  [si-eq]: dim-ty-sem-DimTimes x = QD(′a) * QD(′b)
  instance by (intro-classes, simp-all add: dim-ty-sem-DimTimes-def, (transfer, simp)+)
end

Similarly, we define inversion of dimension types and prove that dimension types are closed under this.

typedef ′a DimInv ((−1) [999] 999) = UNIV :: unit set ..
setup-lifting type-definition-DimInv
instantiation DimInv :: (dim-type) dim-type
begin
  definition dim-ty-sem-DimInv :: (′a−1) itself ⇒ Dimension where
  [si-eq]: dim-ty-sem-DimInv x = inverse QD(′a)
  instance by (intro-classes, simp-all add: dim-ty-sem-DimInv-def, (transfer, simp)+)
end

Dimension Type Syntax

A division is expressed, as usual, by multiplication with an inverted dimension.

type-synonym (′a, ′b) DimDiv = ′a · (′b−1) (infixl ‘/’ 69)

A number of further type synonyms allow for more compact notation:

type-synonym ′a DimSquare = ′a · ′a ((−2) [999] 999)
type-synonym ′a DimCube = ′a · ′a · ′a ((−3) [999] 999)
type-synonym ′a DimQuart = ′a · ′a · ′a · ′a ((−4) [999] 999)
type-synonym ′a DimInvSquare = (′a2)−1 ((−2) [999] 999)
type-synonym ′a DimInvCube = (′a3)−1 ((−3) [999] 999)
type-synonym ′a DimInvQuart = (′a4)−1 ((−4) [999] 999)

translations (type) ′a−2 <= (type) (′a2)−1
translations (type) ′a−3 <= (type) (′a3)−1
translations (type) ′a−4 <= (type) (′a4)−1

print-translation :
[@{type-syntax DimTimes},
 fn ctx => fn [a, b] =>
  if (a = b)
3.1. QUANTITY DIMENSIONS

Then Const (@{type-syntax DimSquare}, dummyT) $ a
else case a of
   Const (@{type-syntax DimTimes}, -) $ a1 $ a2 =>
      if (a1 = a2 andalso a2 = b)
         then Const (@{type-syntax DimCube}, dummyT) $ a1
      else Const (@{type-syntax DimTimes}, -) $ a1 $ a2
      if (a11 = a12 andalso a2 = b)
         then Const (@{type-syntax DimQuart}, dummyT) $ a11
      else raise Match | - = => raise Match)
⟩

Derived Dimension Types

type-synonym Area = L^2
type-synonym Volume = L^3
type-synonym Acceleration = L·T^{-1}
type-synonym Frequency = T^{-1}
type-synonym Energy = L^2·M·T^{-2}
type-synonym Power = L^2·M·T^{-3}
type-synonym Force = L·M·T^{-2}
type-synonym Pressure = L^{-1}·M·T^{-2}
type-synonym Charge = I·T
type-synonym PotentialDifference = L^2·M·T^{-3}·I^{-1}

type-synonym Capacitance = L^{-2}·M^{-1}·T^4·I^2

3.1.6 ML Functions

We define ML functions for converting a dimension to an integer vector, and vice-versa. These are useful for normalising dimension types.

ML ⟨
signature DIMENSION-TYPE =
sig
   val dim-to-typ: int list —> typ
   val typ-to-dim: typ —> int list
   val normalise: typ —> typ
end

structure Dimension-Type : DIMENSION-TYPE =
struct
   val dims = [@{typ L}, @{typ M}, @{typ T}, @{typ I}, @{typ Θ}, @{typ N},
   @{typ J}];

   fun typ-to-dim (Type (@{type-name Length}, [])) = [1, 0, 0, 0, 0, 0, 0] |
   typ-to-dim (Type (@{type-name Mass}, [])) = [0, 1, 0, 0, 0, 0, 0] |
   typ-to-dim (Type (@{type-name Time}, [])) = [0, 0, 1, 0, 0, 0, 0] |
   typ-to-dim (Type (@{type-name Current}, [])) = [0, 0, 0, 1, 0, 0, 0] |
   typ-to-dim (Type (@{type-name Energy}, [])) = [0, 0, 0, 0, 0, 0, 0] |
   typ-to-dim (Type (@{type-name Power}, [])) = [0, 0, 0, 0, 0, 0, 0] |
   typ-to-dim (Type (@{type-name Acceleration}, [])) = [0, 0, 0, 0, 0, 0, 0] |
   typ-to-dim (Type (@{type-name Frequency}, [])) = [0, 0, 0, 0, 0, 0, 0] |
   typ-to-dim (Type (@{type-name Area}, [])) = [0, 0, 0, 0, 0, 0, 0] |
   typ-to-dim (Type (@{type-name Volume}, [])) = [0, 0, 0, 0, 0, 0, 0] |
   typ-to-dim (Type (@{type-name Charge}, [])) = [0, 0, 0, 0, 0, 0, 0] |
   typ-to-dim (Type (@{type-name Pressure}, [])) = [0, 0, 0, 0, 0, 0, 0] |
   typ-to-dim (Type (@{type-name PotentialDifference}, [])) = [0, 0, 0, 0, 0, 0, 0] |
   typ-to-dim (Type (@{type-name Capacitance}, [])) = [0, 0, 0, 0, 0, 0, 0] |
end}
3.2 Quantities

theory ISQ-Quantities
  imports ISQ-Dimensions
begin

3.2.1 Quantity Semantic Domain

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 ('a, 'd:enum) Quantity =
  mag :: 'a — Magnitude of the quantity.
3.2. QUANTITIES

\[ \text{dim} :: (\text{int, 'd}) \text{ dimvec} \] — Dimension of the quantity – denotes the kind of quantity.

The quantity type is parametric as we permit the magnitude to be represented using any kind of numeric type, such as \text{int}, \text{rat}, or \text{real}, though we usually minimally expect a field.

\begin{lemma}
\text{Quantity-eq-intro}:
\text{assumes } \text{mag } x = \text{mag } y \quad \text{dim } x = \text{dim } y \quad \text{more } x = \text{more } y \\
\text{shows } x = y \\
\text{by } (\text{simp add: assms eq-unit})
\end{lemma}

We can define several arithmetic operators on quantities. Multiplication takes multiplies both the magnitudes and the dimensions.

\begin{instantiation}
\text{Quantity-ext} :: (\text{times, enum, times}) \text{ times}
\begin{definition}
\text{times-Quantity-ext} :: \\
(\text{\textquotesingle a, \textquotesingle b, \textquotesingle c}) \text{ Quantity-scheme} \Rightarrow (\text{\textquotesingle a, \textquotesingle b, \textquotesingle c}) \text{ Quantity-scheme} \Rightarrow (\text{\textquotesingle a, \textquotesingle b, \textquotesingle c}) \text{ Quantity-scheme}
\text{where} \ [\text{si-def}]: \text{times-Quantity-ext } x \ y = () \text{ mag } = \text{mag } x \cdot \text{mag } y, \text{ dim } = \text{dim } x \cdot \text{dim } y, \\
\ldots \text{ = more } x \cdot \text{more } y \\
\text{instance} ..
\end{definition}
\end{instantiation}

\begin{lemma}
\text{mag-times} [\text{simp}]: \text{mag } (x \cdot y) = \text{mag } x \cdot \text{mag } y \text{ by } (\text{simp add: times-Quantity-ext-def})
\end{lemma}

\begin{lemma}
\text{dim-times} [\text{simp}]: \text{dim } (x \cdot y) = \text{dim } x \cdot \text{dim } y \text{ by } (\text{simp add: times-Quantity-ext-def})
\end{lemma}

\begin{lemma}
\text{more-times} [\text{simp}]: \text{more } (x \cdot y) = \text{more } x \cdot \text{more } y \text{ by } (\text{simp add: times-Quantity-ext-def})
\end{lemma}

The zero and one quantities are both dimensionless quantities with magnitude of 0:\text{\textquotesingle a} and 1:\text{\textquotesingle a}, respectively.

\begin{instantiation}
\text{Quantity-ext} :: (\text{zero, enum, zero}) \text{ zero}
\begin{definition}
\text{zero-Quantity-ext} = () \text{ mag } = 0, \text{ dim } = 1, \ldots = 0 \\
\text{instance} ..
\end{definition}
\end{instantiation}

\begin{lemma}
\text{mag-zero} [\text{simp}]: \text{mag } 0 = 0 \text{ by } (\text{simp add: zero-Quantity-ext-def})
\end{lemma}

\begin{lemma}
\text{dim-zero} [\text{simp}]: \text{dim } 0 = 1 \text{ by } (\text{simp add: zero-Quantity-ext-def})
\end{lemma}

\begin{lemma}
\text{more-zero} [\text{simp}]: \text{more } 0 = 0 \text{ by } (\text{simp add: zero-Quantity-ext-def})
\end{lemma}

\begin{instantiation}
\text{Quantity-ext} :: (\text{one, enum, one}) \text{ one}
\begin{definition}
[\text{si-def}]: \text{one-Quantity-ext} = () \text{ mag } = 1, \text{ dim } = 1, \ldots = 1 \\
\text{instance} ..
\end{definition}
\end{instantiation}

\begin{lemma}
\text{mag-one} [\text{simp}]: \text{mag } 1 = 1 \text{ by } (\text{simp add: one-Quantity-ext-def})
\end{lemma}
Quantity inversion inverts both the magnitude and the dimension. Similarly, division of one quantity by another, divides both the magnitudes and the dimensions.

As for dimensions, quantities form a commutative monoid and an abelian group.

We can also define a partial order on quantities.

As for dimensions, quantities form a commutative monoid and an abelian group.

We can also define a partial order on quantities.

As for dimensions, quantities form a commutative monoid and an abelian group.
3.2. QUANTITIES

begin
  definition less-eq-Quantity-ext :: ('a, 'b, 'c) Quantity-scheme ⇒ ('a, 'b, 'c) Quantity-scheme ⇒ bool
    where less-eq-Quantity-ext x y = (mag x ≤ mag y ∧ dim x = dim y ∧ more x ≤ more y)

definition less-Quantity-ext :: ('a, 'b, 'c) Quantity-scheme ⇒ ('a, 'b, 'c) Quantity-scheme ⇒ bool
  where less-Quantity-ext x y = (x ≤ y ∧ ¬ y ≤ x)

instance ..
end

instance Quantity-ext :: (order, enum, order) order
  by (intro-classes, auto simp add: less-Quantity-ext-def less-eq-Quantity-ext-def eq-unit)

We can define plus and minus as well, but these are partial operators as they are defined only when the quantities have the same dimension.

instantiation Quantity-ext :: (plus, enum, plus) plus
begin
  definition plus-Quantity-ext :: ('a, 'b, 'c) Quantity-scheme ⇒ ('a, 'b, 'c) Quantity-scheme ⇒ ('a, 'b, 'c) Quantity-scheme
    where [si-def]:
      dim x = dim y ⇒ plus-Quantity-ext x y = (| mag = mag x + mag y, dim = dim x, ... = more x + more y |)
  instance ..
end

instantiation Quantity-ext :: (uminus, enum, uminus) uminus
begin
  definition uminus-Quantity-ext :: ('a, 'b, 'c) Quantity-scheme ⇒ ('a, 'b, 'c) Quantity-scheme
    where [si-def]: uminus-Quantity-ext x = (| mag = − mag x , dim = dim x, ... = − more x |
  instance ..
end

instantiation Quantity-ext :: (minus, enum, minus) minus
begin
  definition minus-Quantity-ext :: ('a, 'b, 'c) Quantity-scheme ⇒ ('a, 'b, 'c) Quantity-scheme
    where [si-def]:
      dim x = dim y ⇒ minus-Quantity-ext x y = (| mag = mag x − mag y, dim = dim x, ... = more x − more y |
  instance ..
end
3.2.2 Measurement Systems

class unit-system = unitary

lemma unit-system-intro: (UNIV::'s set) = {a} \Rightarrow OFCLASS('s, unit-system-class)
  by (simp add: unit-system-class-def, rule unitary-intro)

record ('a, 'd::enum,'s::unit-system) Measurement-System = ('a, 'd::enum) Quantity +
  unit-sys :: 's — The system of units being employed

definition mmore = Record.iso-tuple-snd Measurement-System-ext-Tuple-Iso

lemma mmore [simp]: mmore (| unit-sys = x, ... = y |) = y
  by (metis Measurement-System.ext-inject Measurement-System.ext-surjective comp-id mmore-def)

lemma mmore-ext [simp]: (|unit-sys = unit, ... = mmore a|) = a
  apply (case-tac a, rename-tac b, case-tac b)
  apply (simp add: Measurement-System-ext-def mmore-def Measurement-System-ext-Tuple-Iso-def
    Record.iso-tuple-snd-def Record.iso-tuple-cons-def Abs-Measurement-System-ext-inverse)
  apply (rename-tac x y z)
  apply (subgoal-tac unit = y)
  apply (simp)
  apply (simp add: eq-unit)
  done

lemma Measurement-System-eq-intro:
  assumes mag x = mag y dim x = dim y more x = more y
  shows x = y
  by (rule Quantity-eq-intro, simp-all add: assms)
    (metis Measurement-System.ext-surjective Quantity.select-convs(3) assms(3) mmore mmore-ext)

instantiation Measurement-System-ext :: (unit-system, zero) zero
begin
  definition zero-Measurement-System-ext :: ('a, 'b) Measurement-System-ext
    where [si-def]: zero-Measurement-System-ext = (| unit-sys = unit, ... = 0 |)
  instance..
end

instantiation Measurement-System-ext :: (unit-system, one) one
begin
  definition one-Measurement-System-ext :: ('a, 'b) Measurement-System-ext
    where [si-def]: one-Measurement-System-ext = (| unit-sys = unit, ... = 1 |)
  instance..
end

instantiation Measurement-System-ext :: (unit-system, times) times
begin
3.2. QUANTITIES

Definition times-Measurement-System-ext ::
\((a', b')\) Measurement-System-ext \Rightarrow (a', b')\) Measurement-System-ext
where [si-def]: times-Measurement-System-ext x y = \([\text{unit-sys = unit}, \ldots = \text{mmore } x \cdot \text{mmore } y]\)

Instance ..
end

Instantiation Measurement-System-ext :: (unit-system, inverse) inverse
begin
Definition inverse-Measurement-System-ext :: (a', b') Measurement-System-ext \Rightarrow (a', b')\) Measurement-System-ext
where [si-def]: inverse-Measurement-System-ext x = \([\text{unit-sys = unit}, \ldots = \text{inverse } \text{mmore } x]\)

Definition divide-Measurement-System-ext ::
\((a', b')\) Measurement-System-ext \Rightarrow (a', b')\) Measurement-System-ext
where [si-def]: divide-Measurement-System-ext x y = \([\text{unit-sys = unit}, \ldots = \text{mmore } x / \text{mmore } y]\)

Instance ..
end

Instance Measurement-System-ext :: (unit-system, comm-monoid-mult) comm-monoid-mult
by (intro-classes, simp-all add: eq-unit one-Measurement-System-ext-def times-Measurement-System-ext-def
mult.assoc, simp add: mult.commute)

Instance Measurement-System-ext :: (unit-system, ab-group-mult) ab-group-mult
by (intro-classes, simp-all add: si-def)

Instantiation Measurement-System-ext :: (unit-system, ord) ord
begin
Definition less-eq-Measurement-System-ext :: (a', b') Measurement-System-ext \Rightarrow (a', b')\) Measurement-System-ext \Rightarrow bool
where less-eq-Measurement-System-ext x y = (mmore x \leq \text{mmore } y)

Definition less-Measurement-System-ext :: (a', b') Measurement-System-ext \Rightarrow (a', b')\) Measurement-System-ext \Rightarrow bool
where less-Measurement-System-ext x y = (x \leq y \land \neg y \leq x)

Instance ..
end

Instance Measurement-System-ext :: (unit-system, order) order
by (intro-classes, simp-all add: less-eq-Measurement-System-ext-def less-Measurement-System-ext-def
, metis mmore-ext)

Instantiation Measurement-System-ext :: (unit-system, plus) plus
begin
Definition plus-Measurement-System-ext ::
\((a', b')\) Measurement-System-ext \Rightarrow (a', b')\) Measurement-System-ext \Rightarrow (a', b')\) Measurement-System-ext

CHAPTER 3. INTERNATIONAL SYSTEM OF QUANTITIES

Measurement-System-ext

where [si-def]:

plus-Measurement-System-ext x y = [] unit-sys = unit, ... = mmore x + mmore y

instance ..

end

instantiation Measurement-System-ext :: (unit-system, uminus) uminus

begin
definition uminus-Measurement-System-ext :: ('a, 'b) Measurement-System-ext ⇒ ('a, 'b) Measurement-System-ext where
[si-def]: uminus-Measurement-System-ext x = [] unit-sys = unit, ... = - mmore x

instance ..

end

instantiation Measurement-System-ext :: (unit-system, minus) minus

begin
definition minus-Measurement-System-ext :: ('a, 'b) Measurement-System-ext ⇒ ('a, 'b) Measurement-System-ext ⇒ ('a, 'b) Measurement-System-ext where
[si-def]: minus-Measurement-System-ext x y = [] unit-sys = unit, ... = mmore x - mmore y

instance ..

end

3.2.3 Dimension Typed Quantities

We can now define the type of quantities with parametrised dimension types.

typedef (overloaded) ('n, 'd::dim-type, 's::unit-system) QuantT ([-, -] [999,0,0] 999)

= {x :: ('n, sdim, 's) Measurement-System. dim x = QD('d)}

morphisms fromQ toQ by (rule-tac x=[] mag = undefined, dim = QD('d), unit-sys = unit []) in exI, simp

setup-lifting type-definition-QuantT

A dimension typed quantity is parameterised by two types: 'a, the numeric type for the magnitude, and 'd for the dimension expression, which is an element of dim-type. The type 'n['d, 's] is to ('n, 'd, 's) Measurement-System as dimension types are to Dimension. Specifically, an element of 'n['d, 's] is a quantity whose dimension is 'd.

Intuitively, the formula x can be read as “x is a quantity of 'd”, for example it might be a quantity of length, or a quantity of mass.

Since quantities can have dimension type expressions that are distinct, but denote the same dimension, it is necessary to define the following function
3.2. QUANTITIES

for coercion between two dimension expressions. This requires that the underlying dimensions are the same.

\[ \text{coerceQuantT} :: 'd_2 \text{ itself} \Rightarrow 'a[('d_1::dim-type, 's::unit-system) \Rightarrow 'a[('d_2::dim-type, 's)]} \]

\[ [\text{si-def}: QD('d_1) = QD('d_2) \Rightarrow \text{coerceQuantT t x = (toQ (fromQ x))} \]

\text{syntax}

\[ -\text{QCOERCE} :: \text{type} \Rightarrow \text{logic} \Rightarrow \text{logic} (\text{QCOERCE}[\cdot]) \]

\text{translations}

\[ \text{QCOERCE['t]} = \text{CONST coerceQuantT TYPE('t)} \]

3.2.4 Predicates on Typed Quantities

The standard HOL order (\(\leq\)) and equality (=) have the homogeneous type 'a \(\Rightarrow\) 'a \(\Rightarrow\) bool and so they cannot compare values of different types. Consequently, we define a heterogeneous order and equivalence on typed quantities.

\[ \text{lift-definition qless-eq :: 'n::order[('a::dim-type, 's::unit-system) \Rightarrow 'n['b::dim-type, 's] \Rightarrow bool (infix \ \lesssim} \]

\[ \text{is (\(\leq\))} \]

\[ \text{lift-definition qequiv :: 'n['a::dim-type, 's::unit-system] \Rightarrow 'n['b::dim-type, 's] \Rightarrow bool (infix \ \sim}\]

\[ \text{is (=)} \]

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

\[ \text{lemma qequiv-refl [simp]: a \ \sim \ a} \]

\[ \text{by (simp add: qequiv-def)} \]

\[ \text{lemma qequiv-sym: a \ \sim \ b \ \Rightarrow \ b \ \sim \ a} \]

\[ \text{by (simp add: qequiv-def)} \]

\[ \text{lemma qequiv-trans: [a \ \sim \ b; b \ \sim \ c] \ \Rightarrow \ a \ \sim \ c} \]

\[ \text{by (simp add: qequiv-def)} \]

\[ \text{theorem qeq-iff-same-dim:} \]

\[ \text{fixes x y :: 'a['d::dim-type, 's::unit-system]} \]

\[ \text{shows x \ \sim \ y \ \iff \ x = y} \]

\[ \text{by (transfer, simp)} \]

\[ \text{lemma coerceQuant-eq-iff:} \]

\[ \text{fixes x :: 'a['d_1::dim-type, 's::unit-system]} \]

\[ \text{assumes QD('d_1) = QD('d_2::dim-type)} \]
CHAPTER 3. INTERNATIONAL SYSTEM OF QUANTITIES

shows (coerceQuantT TYPE ('d2) x) \equiv_Q x
by (metis qequiv_rep_eq assms coerceQuantT_def toQ-cases toQ-inverse)

lemma coerceQuant-eq-iff2:
fixes x :: 'a['d1::dim-type, 's::unit-system]
assumes QD('d1) = QD('d2::dim-type) and y = (coerceQuantT TYPE ('d2) x)
shows x \equiv_Q y
using qequiv_sym assms coerceQuantT_def toQ-equivalence by blast

lemma updown-eq-iff:
fixes x :: 'a['d1::dim-type, 's::unit-system] fixes y :: 'a['d2::dim-type, 's]
assumes QD('d1) = QD('d2::dim-type) and y = (toQ (fromQ x))
shows x \equiv_Q y
by (simp add: assms(1) assms(2) coerceQuant-eq-iff2 coerceQuantT_def)

This is more general that y = x \implies x \equiv_Q y, since x and y may have different
type.

lemma qeq:
fixes x :: 'a['d1::dim-type, 's::unit-system] fixes y :: 'a['d2::dim-type, 's]
assumes x \equiv_Q y
shows QD('d1) = QD('d2)
by (metis (full-types) qequiv_rep_eq assms fromQ mem_Collect_eq)

3.2.5 Operators on 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 :: ('n::comm-ring-1)['a::dim-type, 's::unit-system] \Rightarrow 'n['b::dim-type, 's]
\Rightarrow 'n['a 'b, 's] (infixed \cdot 69)
is (\cdot) by (simp add: dim-ty-sem-DimTimes-def times-Quantity-ext-def)

lift-definition
qinverse :: ('n::field)['a::dim-type, 's::unit-system] \Rightarrow 'n['a^{-1}, 's] ((\cdot)^{-1} [999] 999)
is inverse by (simp add: inverse-Quantity-ext-def dim-ty-sem-DimInv-def)

abbreviation (input)
qdivide :: ('n::field)['a::dim-type, 's::unit-system] \Rightarrow 'n['b::dim-type, 's] \Rightarrow 'n['a/'b, 's] (infixed \div 70) where
qdivide x y \equiv x \cdot y^{-1}

We also provide some helpful notations for expressing heterogeneous powers.

abbreviation qsq \equiv ((\cdot)^2 [999] 999) where u^2 = w \cdot u
abbreviation qcube \equiv ((\cdot)^3 [999] 999) where u^3 = w \cdot u \cdot u
abbreviation qquart \equiv ((\cdot)^4 [999] 999) where u^4 = w \cdot w \cdot w \cdot u
3.2. QUANTITIES

abbreviation \( qneq-sq \) \(( (-2 \cdot 999)^3 999) \) where \( u^{-2} \equiv (u^2)^{-1} \)

abbreviation \( qneq-cube \) \(( (-3 \cdot 999)^3 999) \) where \( u^{-3} \equiv (u^3)^{-1} \)

abbreviation \( qneq-quart \) \(( (-4 \cdot 999)^3 999) \) where \( u^{-4} \equiv (u^4)^{-1} \)

Analogous to the \(( \star R \)) operator for vectors, we define the following scalar multiplication that scales an existing quantity by a numeric value. This operator is especially important for the representation of quantity values, which consist of a numeric value and a unit.

\[
\text{lift-definition } \text{scaleQ} :: \quad'\alpha \Rightarrow '\alpha :: \text{comm-ring-1}[d::\text{dim-type}, 's::\text{unit-system}] \Rightarrow '\alpha[d', 's] \quad(\text{infixr } \ast_Q 63) \\
\text{is } \lambda x. (| \text{mag} = r \ast \text{mag } x, \text{dim} = QD('d), \text{unit-sys} = \text{unit} \}) \text{ by simp}
\]

Finally, we instantiate the arithmetic types classes where possible. We do not instantiate \textit{times} because this results in a nonsensical homogeneous product on quantities.

\[
\text{instantiation } \text{QuantT} :: (zero, \text{dim-type}, \text{unit-system}) \text{ zero begin} \\
\text{lift-definition } \text{zero-QuantT} :: ('a, 'b, 'c) \text{ QuantT is } (| \text{mag} = 0, \text{dim} = QD('b), \text{unit-sys} = \text{unit} \}) \text{ by simp} \\
\text{instance .. end}
\]

\[
\text{instantiation } \text{QuantT} :: (one, \text{dim-type}, \text{unit-system}) \text{ one begin} \\
\text{lift-definition } \text{one-QuantT} :: ('a, 'b, 'c) \text{ QuantT is } (| \text{mag} = 1, \text{dim} = QD('b), \text{unit-sys} = \text{unit} \}) \text{ by simp} \\
\text{instance .. end}
\]

The following specialised one element has both magnitude and dimension 1: it is a dimensionless quantity.

abbreviation \( qone \) :: 'n::\text{one}[1, 's::\text{unit-system}] (1) where \( qone \equiv 1 \)

Unlike for semantic quantities, the plus operator on typed quantities is total, since the type system ensures that the dimensions (and the dimension types) must be the same.

\[
\text{instantiation } \text{QuantT} :: (plus, \text{dim-type}, \text{unit-system}) \text{ plus begin} \\
\text{lift-definition } \text{plus-QuantT} :: 'a[b, 'c] \Rightarrow 'a[b, 'c] \Rightarrow 'a[b, 'c] \\
\text{is } \lambda x y. (| \text{mag} = \text{mag } x + \text{mag } y, \text{dim} = QD('b), \text{unit-sys} = \text{unit} \}) \text{ by (simp)} \\
\text{instance .. end}
\]
We can also show that typed quantities are commutative additive monoids. Indeed, addition is a much easier operator to deal with in typed quantities, unlike product.

\[\text{instance } QuantT :: (semigroup-add, dim-type, unit-system) \text{ semigroup-add by } (\text{intro-classes, transfer, simp add: addassoc})\]

\[\text{instance } QuantT :: (ab-semigroup-add, dim-type, unit-system) \text{ ab-semigroup-add by } (\text{intro-classes, transfer, simp add: addcommute})\]

\[\text{instance } QuantT :: (monoid-add, dim-type, unit-system) \text{ monoid-add by } (\text{intro-classes, transfer, simp: eq-unit})\]

\[\text{instance } QuantT :: (comm-monoid-add, dim-type, unit-system) \text{ comm-monoid-add by } (\text{intro-classes, transfer, simp})\]

\[\text{instantiation } QuantT :: (uminus, dim-type, unit-system) \text{ uminus begin}\]
\[\text{lift-definition } \text{uminus-QuantT} :: 'a['b,'c] \Rightarrow 'a['b,'c] \text{ is } \lambda x. (|mag = -mag x, dim = dim x, unit-sys = unit |) \text{ by (simp)}\]
\[\text{instance ..}\]
\[\text{end}\]

\[\text{instantiation } QuantT :: (minus, dim-type, unit-system) \text{ minus begin}\]
\[\text{lift-definition } \text{minus-QuantT} :: 'a['b,'c] \Rightarrow 'a['b,'c] \Rightarrow 'a['b,'c] \text{ is } \lambda x y. (|mag = mag x - mag y, dim = dim x, unit-sys = unit |) \text{ by (simp)}\]
\[\text{instance ..}\]
\[\text{end}\]

\[\text{instance } QuantT :: (numeral, dim-type, unit-system) \text{ numeral ..}\]

Moreover, types quantities also form an additive group.

\[\text{instance } QuantT :: (ab-group-add, dim-type, unit-system) \text{ ab-group-add by } (\text{intro-classes, transfer, simp})+\]

Typed quantities helpfully can be both partially and a linearly ordered.

\[\text{instantiation } QuantT :: (order, dim-type, unit-system) \text{ order begin}\]
\[\text{lift-definition } \text{less-eq-QuantT} :: 'a['b,'c] \Rightarrow 'a['b,'c] \Rightarrow \text{bool is } \lambda x y. \text{mag } x \leq \text{mag } y .\]
\[\text{lift-definition } \text{less-QuantT} :: 'a['b,'c] \Rightarrow 'a['b,'c] \Rightarrow \text{bool is } \lambda x y. \text{mag } x < \text{mag } y .\]
\[\text{instance by (intro-classes, transfer, simp add: unit-eq less-le-not-le Measurement-System-eq-intro)+) end}\]

\[\text{instance } QuantT :: (linorder, dim-type, unit-system) \text{ linorder by } (\text{intro-classes, transfer, auto})\]
3.2. QUANTITIES

instantiation QuantT :: (scaleR,dim-type,unit-system) scaleR
begin
lift-definition scaleR-QuantT :: real ⇒ 'a[′b,′c] ⇒ 'a[′b,′c]
is λ n q. (! mag = n *R mag q, dim = dim q, unit-sys = unit) by (simp)
instance ..
end

instance QuantT :: (real-vector,dim-type,unit-system) real-vector
by (intro-classes, (transfer, simp add: eq-unit scaleR-add-left scaleR-add-right)+)

instantiation QuantT :: (norm,dim-type,unit-system) norm
begin
lift-definition norm-QuantT :: 'a[′b,′c] ⇒ real
is λ x. norm (mag x) .
instance ..
end

instantiation QuantT :: (sgn-div-norm,dim-type,unit-system) sgn-div-norm
begin
definition sgn-QuantT :: 'a[′b,′c] ⇒ 'a[′b,′c] where
sgn-QuantT x = x /R norm x
instance by (intro-classes, simp add: sgn-QuantT-def)
end

instantiation QuantT :: (dist-norm,dim-type,unit-system) dist-norm
begin
definition dist-QuantT :: 'a[′b,′c] ⇒ 'a[′b,′c] ⇒ real where
dist-QuantT x y = norm (x − y)
instance
by (intro-classes, simp add: dist-QuantT-def)
end

instantiation QuantT :: ({uniformity-dist,dist-norm},dim-type,unit-system) uniformity-dist
begin
definition uniformity-QuantT :: ('a[′b,′c] × 'a[′b,′c]) filter where
uniformity-QuantT = (INF e∈{0 <..}. principal {(x, y). dist x y < e})
instance
by (intro-classes, simp add: uniformity-QuantT-def)
end

instantiation QuantT :: ({dist-norm,open-uniformity,uniformity-dist},dim-type,unit-system)
begin
open-uniformity

definition open-QuantT :: ('a[′b,′c]) set ⇒ bool where
open-QuantT U = (∀ x∈U. eventually (λ(x', y). x' = x −→ y ∈ U) uniformity)
instance by (intro-classes, simp add: open-QuantT-def)
Quantities form a real normed vector space.

instance QuantT :: (real-normed-vector, dim-type, unit-system) real-normed-vector
  by (intro-classes; transfer, auto simp add: eq-unit norm-triangle-ineq)

3.3 Proof Support for Quantities

theory ISQ-Proof
  imports ISQ-Quantities
begin

named-theorems si-transfer

definition magQ :: 'a[u::dim-type, s::unit-system] ⇒ 'a (\[\|\]_Q) where
  [si-def]: magQ x = mag (fromQ x)

definition dimQ :: 'a[u::dim-type, s::unit-system] ⇒ Dimension where
  [si-def]: dimQ x = dim (fromQ x)

lemma quant-eq-iff-mag-eq [si-eq]:
  x = y ⇔ \[\[x\]_Q = \[\[y\]_Q\]
  by (auto simp add: magQ-def, transfer, simp add: eq-unit)

lemma quant-eqI [si-transfer]:
  \[\[x\]_Q = \[\[y\]_Q\]
  ⇒ x = y
  by (simp add: quant-eq-iff-mag-eq)

lemma quant-equiv-iff [si-eq]:
  fixes x :: 'a[u_1::dim-type, s::unit-system] and y :: 'a[u_2::dim-type, s::unit-system]
  shows x ∼ \[\[y\]_Q\] ⇔ \[\[x\]_Q\] = \[\[y\]_Q\]
  proof
  have ∀ t ta. (ta::'a[u_2, s]) = t ∨ mag (fromQ ta) ≠ mag (fromQ t)
    by (simp add: magQ-def quant-eq-iff-mag-eq)
  then show ?thesis
    by (metis (full-types) gequiv.rep-eq coerceQuant-eq-iff2 qeq magQ-def)
  qed

lemma quant-equivI [si-transfer]:
  fixes x :: 'a[u_1::dim-type, s::unit-system] and y :: 'a[u_2::dim-type, s::unit-system]
  assumes QD(u_1) = QD(u_2) QD(u_1) = QD(u_2) ⇒ \[\[x\]_Q = \[\[y\]_Q\]
  shows x ∼ y
  using assms quant-equiv-iff by blast

lemma quant-le-iff-magn-le [si-eq]:
  x ≤ y ⇔ \[\[x\]_Q ≤ \[\[y\]_Q\]
  by (auto simp add: magQ-def; (transfer, simp))
3.3. PROOF SUPPORT FOR QUANTITIES

lemma quant-leI [si-transfer]:
\([x]_Q \leq [y]_Q \implies z \leq y\)
by (simp add: quant-le-iff-magn-le)

lemma quant-less-iff-magn-less [si-eq]:
\(x < y \iff [x]_Q < [y]_Q\)
by (auto simp add: magQ-def; (transfer, simp))

lemma quant-lessI [si-transfer]:
\([x]_Q < [y]_Q \implies x < y\)
by (simp add: quant-less-iff-magn-less)

lemma magQ-zero [si-eq]: \([0]_Q = 0\)
by (simp add: magQ-def, transfer, simp)

lemma magQ-one [si-eq]: \([1]_Q = 1\)
by (simp add: magQ-def, transfer, simp)

lemma magQ-plus [si-eq]: \([x + y]_Q = [x]_Q + [y]_Q\)
by (simp add: magQ-def, transfer, simp)

lemma magQ-minus [si-eq]: \([x - y]_Q = [x]_Q - [y]_Q\)
by (simp add: magQ-def, transfer, simp)

lemma magQ-uminus [si-eq]: \([-x]_Q = -[x]_Q\)
by (simp add: magQ-def, transfer, simp)

lemma magQ-scaleQ [si-eq]: \([x * y]_Q = x * [y]_Q\)
by (simp add: magQ-def, transfer, simp)

lemma magQ-qtimes [si-eq]: \([x \cdot y]_Q = [x]_Q \cdot [y]_Q\)
by (simp add: magQ-def, transfer, simp)

lemma magQ-qinverse [si-eq]: \([x^{-1}]_Q = \text{inverse } [x]_Q\)
by (simp add: magQ-def, transfer, simp)

lemma magQ-qdivivide [si-eq]: \([(x::'a::field)-]'y\}/y\}Q = [x]Q/ [y]Q\)
by (simp add: magQ-def, transfer, simp add: field-class.field-divide-inverse)

lemma magQ-numeral [si-eq]: \([\text{numeral } n]_Q = \text{numeral } n\)
apply (induct n, simp-all add: si-def)
apply (metis magQ-def magQ-one)
apply (metis magQ-def magQ-plus numeral-code(2))
apply (metis magQ-def magQ-one magQ-plus numeral-code(3))
done

lemma magQ-coerce [si-eq]:
fixed q :: 'a::'d1::dim-type, 's::unit-system and t :: 'd2::dim-type itself
assumes \( QD'(d_1) = QD'(d_2) \)
shows \([\text{coerceQuant} T \ i \ q]_Q = [i]_Q\)
by (simp add: \text{coerceQuant} T-def \text{mag}Q-def \text{assms}, metis \text{assms} \text{qequiv} \text{rep-eq} \text{updown-eq-iff})

\( \text{lemma dimQ [simp]}: \dim Q(x :: 'a[d::dim-type, 's::unit-system]) = QD'(d) \)
by (simp add: \text{dimQ-def}, transfer, simp)

The following tactic breaks an SI conjecture down to numeric and unit properties
\( \text{method si-simp uses add =} \)
\( \quad \text{(rule-tac \text{si-transfer}; simp add: add \text{si-eq field-simps})} \)

The next tactic additionally compiles the semantics of the underlying units
\( \text{method si-calc uses add =} \)
\( \quad \text{(si-simp add: add; simp add: si-def add)} \)

\( \text{lemma QD}(N \cdot \Theta \cdot N) = QD(\Theta \cdot N^2) \text{ by (simp add: \text{si-eq si-def})} \)
end

3.4 Algebraic Laws

theory ISQ-Algebra
  imports ISQ-Proof
begin

3.4.1 Quantity Scale

\( \text{lemma scaleQ-add-right: a *Q x + y = (a *Q x) + (a *Q y)} \)
by (si-simp add: distrib-left)

\( \text{lemma scaleQ-add-left: a + b *Q x = (a *Q x) + (b *Q x)} \)
by (si-simp add: distrib-right)

\( \text{lemma scaleQ-scaleQ [simp]: a *Q b *Q x = a \cdot b *Q x} \)
by si-simp

\( \text{lemma scaleQ-one [simp]: 1 *Q x = x} \)
by si-simp

\( \text{lemma scaleQ-zero [simp]: 0 *Q x = 0} \)
by si-simp

\( \text{lemma scaleQ-inv: } -a *Q x = a *Q -x \)
by si-calc

\( \text{lemma scaleQ-as-qprod: a *Q x \equiv_Q (a *Q 1) \cdot x} \)
3.4. ALGEBRAIC LAWS

by si-simp

lemma mult-scaleQ-left [simp]: \((a \ast_Q x) \cdot y = a \ast_Q x \cdot y\)
by si-simp

lemma mult-scaleQ-right [simp]: \(x \cdot (a \ast_Q y) = a \ast_Q x \cdot y\)
by si-simp

3.4.2 Field Laws

lemma qtimes-commute: \(x \cdot y \cong_Q y \cdot x\)
by si-calc

lemma qtimes-assoc: \((x \cdot y) \cdot z \cong_Q x \cdot (y \cdot z)\)
by (si-calc)

lemma qtimes-left-unit: \(1 \cdot x \cong_Q x\)
by (si-calc)

lemma qtimes-right-unit: \(x \cdot 1 \cong_Q x\)
by (si-calc)

The following weak congruences will allow for replacing equivalences in contexts built from product and inverse.

lemma qtimes-weak-cong-left:
assumes \(x \cong_Q y\)
shows \(xz \cong_Q yz\)
using assms by si-simp

lemma qtimes-weak-cong-right:
assumes \(x \cong_Q y\)
shows \(zx \cong_Q yz\)
using assms by si-calc

lemma qinverse-weak-cong:
assumes \(x \cong_Q y\)
shows \(x^{-1} \cong_Q y^{-1}\)
using assms by si-calc

lemma scaleQ-cong:
assumes \(y \cong_Q z\)
shows \(x \ast_Q y \cong_Q x \ast_Q z\)
using assms by si-calc

lemma qinverse-qinverse: \(x^{-1}^{-1} \cong_Q x\)
by si-calc

lemma qinverse-nonzero-iff-nonzero: \(x^{-1} = 0 \iff x = 0\)
by (auto, si-calc+)
CHAPTER 3. INTERNATIONAL SYSTEM OF QUANTITIES

Lemma qinverse-times: \((x \cdot y)^{-1} \cong_{Q} x^{-1} \cdot y^{-1}\)
by (si-simp add: inverse-distrib)

Lemma qinverse-qdivide: \((x / y)^{-1} \cong_{Q} y / x\)
by si-simp

Lemma qtimes-cancel: \(x \neq 0 \implies x / x \cong_{Q} 1\)
by si-calc

end

3.5 Units

theory ISQ-Units
  imports ISQ-Proof
begin

Parallel to the base quantities, there are base units. In the implementation of the SI unit system, we fix these to be precisely those quantities that have a base dimension and a magnitude of 1::'a. Consequently, a base unit corresponds to a unit in the algebraic sense.

lift-definition is-base-unit :: 'a::one['d::dim-type, 's::unit-system] ⇒ bool
  is λ x. mag x = 1 ∧ is-BaseDim (dim x).

definition mk-base-unit :: 'a itself ⇒ 's itself ⇒ (a::basedim-type, 's::unit-system]
  where mk-base-unit t s = 1

syntax -mk-base-unit :: type ⇒ type ⇒ logic (BUNIT('a', 's'))
translations BUNIT('a', 's') == CONST mk-base-unit TYPE('a) TYPE('s)

lemma mk-base-unit: is-base-unit (mk-base-unit a s)
  by (simp add: mk-base-unit-def si-eq, transfer, simp add: is-BaseDim)

lemma magQ-mk [si-eq]: [BUNIT('u::basedim-type, 's::unit-system)]Q = 1
  by (simp add: mk-base-unit-def magQ-def si-eq, transfer, simp)

end

3.6 Conversion Between Unit Systems

theory ISQ-Conversion
  imports ISQ-Units
begin
3.6. CONVERSION BETWEEN UNIT SYSTEMS

3.6.1 Conversion Schemas

A conversion schema provides factors for each of the base units for converting between two systems of units. We currently only support conversion between systems that can meaningfully characterise a subset of the seven SI dimensions.

```
record ConvSchema =
cLengthF :: rat
cMassF :: rat
cTimeF :: rat
cCurrentF :: rat
cTemperatureF :: rat
cAmountF :: rat
cIntensityF :: rat
```

We require that all the factors of greater than zero.

```
typedef (′s1::unit-system, ′s2::unit-system) Conversion (([-/ ⇒U ·] [1, 0]) 0) =
{ c :: ConvSchema. cLengthF c > 0 ∧ cMassF c > 0 ∧ cTimeF c > 0 ∧ cCurrentF c > 0 ∧ cTemperatureF c > 0 ∧ cAmountF c > 0 ∧ cIntensityF c > 0 }
by (rule-tac x≡(). cLengthF = 1, cMassF = 1, cTimeF = 1, cCurrentF = 1 , cTemperatureF = 1, cAmountF = 1, cIntensityF = 1 ) in exI, simp
```

```
setup-lifting type-definition-Conversion

lift-definition LengthF :: (′s1::unit-system ⇒U ′s2::unit-system) ⇒ rat is cLengthF.
lift-definition MassF :: (′s1::unit-system ⇒U ′s2::unit-system) ⇒ rat is cMassF.
lift-definition TimeF :: (′s1::unit-system ⇒U ′s2::unit-system) ⇒ rat is cTimeF.
lift-definition CurrentF :: (′s1::unit-system ⇒U ′s2::unit-system) ⇒ rat is cCurrentF.
lift-definition TemperatureF :: (′s1::unit-system ⇒U ′s2::unit-system) ⇒ rat is cTemperatureF.
lift-definition AmountF :: (′s1::unit-system ⇒U ′s2::unit-system) ⇒ rat is cAmountF.
lift-definition IntensityF :: (′s1::unit-system ⇒U ′s2::unit-system) ⇒ rat is cIntensityF.
```

```
lemma Conversion-props [simp]: LengthF c > 0 MassF c > 0 TimeF c > 0 CurrentF c > 0 TemperatureF c > 0 AmountF c > 0 IntensityF c > 0
by (transfer, simp)+
```

3.6.2 Conversion Algebra

```
lift-definition convid :: ′s::unit-system ⇒U ′s (id_C)
```
CHAPTER 3. INTERNATIONAL SYSTEM OF QUANTITIES

is

\( \lambda \) cLengthF = 1

\( \lambda \) cMassF = 1

\( \lambda \) cTimeF = 1

\( \lambda \) cCurrentF = 1

\( \lambda \) cTemperatureF = 1

\( \lambda \) cAmountF = 1

\( \lambda \) cIntensityF = 1

by simp

lift-definition convec ::

\( (s_2 \Rightarrow_U s_3::\text{unit-system}) \Rightarrow (s_1::\text{unit-system} \Rightarrow_U s_2::\text{unit-system}) \Rightarrow (s_1 \Rightarrow_U s_3) \) (infixl \( \circ_C \) 55)

\( \lambda \) c1 c2. \( \lambda \) cLengthF = cLengthF c1 * cLengthF c2, cMassF = cMassF c1 * cMassF c2

\( \lambda \) cTimeF = cTimeF c1 * cTimeF c2, cCurrentF = cCurrentF c1 * cCurrentF c2

\( \lambda \) cTemperatureF = cTemperatureF c1 * cTemperatureF c2

\( \lambda \) cAmountF = cAmountF c1 * cAmountF c2, cIntensityF = cIntensityF c1 * cIntensityF c2

by simp

lift-definition convinv ::

\( (s_1::\text{unit-system} \Rightarrow_U s_2::\text{unit-system}) \Rightarrow (s_2 \Rightarrow_U s_1) \) (invC)

\( \lambda \) c. \( \lambda \) cLengthF = inverse (cLengthF c), cMassF = inverse (cMassF c), cTimeF = inverse (cTimeF c)

\( \lambda \) cCurrentF = inverse (cCurrentF c), cTemperatureF = inverse (cTemperatureF c)

\( \lambda \) cAmountF = inverse (cAmountF c), cIntensityF = inverse (cIntensityF c)

by simp

lemma convinv-inverse [simp]; coninv (coninv c) = c

by (transfer, simp)

lemma convec-inv [simp]; c \( \circ_C \) invC c = idC

by (transfer, simp)

lemma inv-convec [simp]; invC c \( \circ_C \) c = idC

by (transfer, simp)

lemma Conversion-invss [simp]; LengthF (invC x) = inverse (LengthF x) MassF (invC x) = inverse (MassF x)

TimeF (invC x) = inverse (TimeF x) CurrentF (invC x) = inverse (CurrentF x)

TemperatureF (invC x) = inverse (TemperatureF x) AmountF (invC x) = inverse (AmountF x)

IntensityF (invC x) = inverse (IntensityF x)

by (transfer, simp)

lemma Conversion-comps [simp]; LengthF (c1 \( \circ_C \) c2) = LengthF c1 * LengthF c2
3.6. CONVERSION BETWEEN UNIT SYSTEMS

\[ \text{Mass}_{F_1} (c_1 \circ C) = \text{Mass}_{F_2} c_1 \ast \text{Mass}_{F_2} c_2 \]
\[ \text{Time}_{F_1} (c_1 \circ C) = \text{Time}_{F_2} c_1 \ast \text{Time}_{F_2} c_2 \]
\[ \text{Current}_{F_1} (c_1 \circ C) = \text{Current}_{F_2} c_1 \ast \text{Current}_{F_2} c_2 \]
\[ \text{Temperature}_{F_1} (c_1 \circ C) = \text{Temperature}_{F_2} c_1 \ast \text{Temperature}_{F_2} c_2 \]
\[ \text{Amount}_{F_1} (c_1 \circ C) = \text{Amount}_{F_2} c_1 \ast \text{Amount}_{F_2} c_2 \]
\[ \text{Intensity}_{F_1} (c_1 \circ C) = \text{Intensity}_{F_2} c_1 \ast \text{Intensity}_{F_2} c_2 \]

by (transfer, simp)+

3.6.3 Conversion Functions

definition dconvfactor :: ('s_1::unit-system ⇒ 'U' s_2::unit-system) ⇒ Dimension ⇒
rat where
\[ \text{dconvfactor } c \ d = \]
\[ \text{Length}_{F_1} c \circ Z \ dim-nth d \text{ Length} \]
\[ \ast \ \text{Mass}_{F_1} c \circ Z \ dim-nth d \text{ Mass} \]
\[ \ast \ \text{Time}_{F_1} c \circ Z \ dim-nth d \text{ Time} \]
\[ \ast \ \text{Current}_{F_1} c \circ Z \ dim-nth d \text{ Current} \]
\[ \ast \ \text{Temperature}_{F_1} c \circ Z \ dim-nth d \text{ Temperature} \]
\[ \ast \ \text{Amount}_{F_1} c \circ Z \ dim-nth d \text{ Amount} \]
\[ \ast \ \text{Intensity}_{F_1} c \circ Z \ dim-nth d \text{ Intensity} \]

lemma dconvfactor-pos [simp]: dconvfactor c d > 0
by (simp add: dconvfactor-def)

lemma dconvfactor-nz [simp]: dconvfactor c d ≠ 0
by (metis dconvfactor-pos less-numeral-extra (3))

lemma dconvfactor-compose: dconvfactor (convinv c) d = inverse (dconvfactor c d)
by (simp add: dconvfactor-def intpow-inverse[THEN sym])

lemma dconvfactor-id [simp]: dconvfactor id_C d = 1
by (simp add: dconvfactor-def, transfer, simp)

lemma dconvfactor-compose:
\[ \text{dconvfactor } (c_1 \circ C) c_2 \ d = \text{dconvfactor } c_1 \ d \ast \text{dconvfactor } c_2 \ d \]
by (simp add: dconvfactor-def, transfer, simp add: mult-ac intpow-mult-distrib)

lemma dconvfactor-inverse:
\[ \text{dconvfactor } c \ (\text{inverse } d) = \text{inverse } (\text{dconvfactor } c \ d) \]
by (simp add: dconvfactor-def inverse-dimvec-def intpow-uminus)

lemma dconvfactor-times:
\[ \text{dconvfactor } c \ (x \cdot y) = \text{dconvfactor } c x \cdot \text{dconvfactor } c y \]
by (auto simp add: dconvfactor-def mult-ac intpow-mult-combine times-dimvec-def)

lift-definition qconv :: ('s_1, 's_2) Conversion ⇒ ('a::field-char-0)'d::dim-type, 's_1::unit-system) ⇒ 'a'[d, 's_2::unit-system]
is λ c q. l [mag = of-rat (dconvfactor c (dim q)) * mag q, dim = dim q, unit-sys]
= unit | by simp

lemma magQ-qconv: \[ qconv c \ q \] \( Q \) = of-rat (dconvfactor c (dimQ q)) * \( q \) \( Q \)
by (simp add: si-def, transfer, simp)

lemma qconv-id [simp]: qconv id\( C \) \( x \) = \( x \)
by (transfer', simp add: Measurement-System-eq-intro)

lemma qconv-comp: qconv \( c_1 \circ c_2 \) \( x \) = qconv \( c_1 \) (qconv \( c_2 \) \( x \))
by (transfer, simp add: dconvfactor-compose of-rat-mult)

lemma qconv-convinv [simp]: qconv (convinv c) (qconv c \( x \)) = \( x \)
by (transfer, simp add: dconvfactor-convinv mult assoc THEN sym of-rat-mult THEN sym Measurement-System-eq-intro)

lemma qconv-scaleQ [simp]: qconv c \( d \ast Q \) \( x \) = \( d \ast Q \) qconv c \( x \)
by (transfer, simp)

lemma qconv-plus [simp]: qconv c \( x + y \) = qconv c \( x \) + qconv c \( y \)
by (transfer, auto simp add: plus-Quantity-ext-def mult.commute ring-class.ring-distrib)

lemma qconv-minus [simp]: qconv c \( x - y \) = qconv c \( x \) - qconv c \( y \)
by (transfer, auto simp add: plus-Quantity-ext-def mult.commute ring-class.ring-distrib)

lemma qconv-qmult [simp]: qconv c \( x \cdot y \) = qconv c \( x \) \cdot qconv c \( y \)
by (transfer, simp add: times-Quantity-ext-def times-Measurement-System-ext-def dconvfactor-times of-rat-mult)

lemma qconv-qinverse [simp]: qconv c \( x^{-1} \) = (qconv c \( x \))^{-1}

lemma qconv-Length [simp]: qconv c BUNIT\( L \), -) = LengthF c *Q BUNIT\( L \), -)
by (simp add: dconvfactor-def magQ-qconv si-eq mk-BaseDim-def one-dimvec-def)

lemma qconv-Mass [simp]: qconv c BUNIT\( M \), -) = MassF c *Q BUNIT\( M \), -)
by (simp add: dconvfactor-def magQ-qconv si-eq mk-BaseDim-def one-dimvec-def)

lemma qconv-Time [simp]: qconv c BUNIT\( T \), -) = TimeF c *Q BUNIT\( T \), -)
by (simp add: dconvfactor-def magQ-qconv si-eq mk-BaseDim-def one-dimvec-def)

lemma qconv-Current [simp]: qconv c BUNIT\( I \), -) = CurrentF c *Q BUNIT\( I \), -)
by (simp add: dconvfactor-def magQ-qconv si-eq mk-BaseDim-def one-dimvec-def)

lemma qconv-Temperature [simp]: qconv c BUNIT\( \Theta \), -) = TemperatureF c *Q BUNIT\( \Theta \), -)
by (simp add: dconvfactor-def magQ-qconv si-eq mk-BaseDim-def one-dimvec-def)
3.7. META-THEORY FOR ISQ

lemma qconv-Amount [simp]: \( qconv \ c \ BUNIT(N, -) = AmountF \ c \ *Q \ BUNIT(N, -) \)
  by (simp add: dconvfactor-def magQ-qconv si-eq mk-BaseDim-def one-dimvec-def)

lemma qconv-Intensity [simp]: \( qconv \ c \ BUNIT(J, -) = IntensityF \ c \ *Q \ BUNIT(J, -) \)
  by (simp add: dconvfactor-def magQ-qconv si-eq mk-BaseDim-def one-dimvec-def)

end

3.7 Meta-Theory for ISQ

theory ISQ
  imports ISQ-Dimensions ISQ-Quantities ISQ-Proof ISQ-Algebra ISQ-Units ISQ-Conversion
begin end
Chapter 4

International System of Units

4.1 SI Units Semantics

typedef $SI = UNIV :: \text{unit set by simp}$

instance $SI :: \text{unit-system}$
  by (rule unit-system-intro[of Abs-SI()], metis (full-types) Abs-SI-cases UNIV.eq-I
  insert-iff old.unit.exhaust)

abbreviation $SI \equiv \text{unit :: SI}$

type-synonym ('n, 'd) SIUnitT = ('n, 'd, SI) QuantT (-[-] [999,0] 999)

We now define the seven base units. Effectively, these definitions axiomatise
given names for the $I::'a$ elements of the base quantities.

definitions $\text{metre} \equiv BUNIT(L, SI)$
 definitions $\text{kilogram} \equiv BUNIT(M, SI)$
 definitions $\text{ampere} \equiv BUNIT(I, SI)$
 definitions $\text{kelvin} \equiv BUNIT(\Theta, SI)$
 definitions $\text{mole} \equiv BUNIT(N, SI)$
 definitions $\text{candela} \equiv BUNIT(I, SI)$

The second is commonly used in unit systems other than SI. Consequently,
we define it polymorphically, and require that the system type instantiate a
type class to use it.

class $\text{time-second} = \text{unit-system}$
instance SI :: time-second ..

abbreviation second ≡ BUNIT(T, 'a::time-second)

Note that as a consequence of our construction, the term metre is a SI Unit constant of SI-type 'a[L, SI], so a unit of dimension L with the magnitude of type 'a. A magnitude instantiation can be, e.g., an integer, a rational number, a real number, or a vector of type real^3. Note than when considering vectors, dimensions refer to the norm of the vector, not to its components.

lemma BaseUnits:
  is-base-unit metre is-base-unit second is-base-unit kilogram is-base-unit ampere
  is-base-unit kelvin is-base-unit mole is-base-unit candela
  by (simp-all add: mk-base-unit)

The effect of the above encoding is that we can use the SI base units as synonyms for their corresponding dimensions at the type level.

type-synonym 'a metre = 'a[Length, SI]
type-synonym 'a second = 'a[Time, SI]
type-synonym 'a kilogram = 'a[Mass, SI]
type-synonym 'a ampere = 'a[Current, SI]
type-synonym 'a kelvin = 'a[Temperature, SI]
type-synonym 'a mole = 'a[Amount, SI]
type-synonym 'a candela = 'a[Intensity, SI]

We can therefore construct a quantity such as 5, which unambiguously identifies that the unit of 5 is metres using the type system. This works because each base unit it the one element.

4.1.1 Example Unit Equations

lemma (metre ∙ second^-1) ∙ second ≃Q metre
  by (si-calc)

4.1.2 Metrification

class metrifiable = unit-system +
  fixes convschema :: 'a itself ⇒ ('a, SI) Conversion (schemaC)

instantiation SI :: metrifiable
begin
lift-definition convschema-SI :: SI itself ⇒ (SI, SI) Conversion is λ s.
  | cLengthF = 1,
  | cMassF = 1,
  | cTimeF = 1,
  | cCurrentF = 1,
  | cTemperatureF = 1,
  | cAmountF = 1
4.2 Centimetre-Gram-Second System

theory CGS
  imports SI-Units
begin

4.2.1 Preliminaries

typedef CGS = UNIV :: unit set ..
instance CGS :: unit-system
  by (rule unit-system-intro[of Abs-CGS ()], metis (full-types)
      Abs-CGS-cases UNIV-eq-I insert-iff old.unit.exhaust)
instance CGS :: time-second ..
abbreviation CGS ≡ unit :: CGS

4.2.2 Base Units

abbreviation centimetre ≡ BUNIT(L, CGS)
abbreviation gram ≡ BUNIT(M, CGS)

4.2.3 Conversion to SI

instantiation CGS :: metrifiable
begin

lift-definition convschema-CGS :: CGS itself ⇒ (CGS, SI) Conversion is
\( \lambda x. (| cLengthF = 0.01, cMassF = 0.001, cTimeF = 1, cCurrentF = 1, cTemperatureF = 1, cAmountF = 1, cIntensityF = 1 |) \)
by simp

instance ..
end

lemma CGS-SI-simps [simp]: LengthF (convschema (a::CGS itself)) = 0.01 MassF (convschema a) = 0.001
TimeF (convschema a) = 1 CurrentF (convschema a) = 1 TemperatureF (convschema a) = 1
by (transfer, simp)+

4.2.4 Conversion Examples

lemma metrify ((100::rat) *Q centimetre) = 1 *Q metre
by (si-simp)

end

4.3 Physical Constants

theory SI-Constants
imports SI-Units
begin

4.3.1 Core Derived Units

abbreviation (input) hertz ≡ second\(^{-1}\)

abbreviation radian ≡ metre · metre\(^{-1}\)

abbreviation steradian ≡ metre\(^2\) · metre\(^{-2}\)

abbreviation joule ≡ kilogram · metre\(^2\) · second\(^{-2}\)

type-synonym 'a joule = 'a[M · L\(^2\) · T\(^{-2}\), SI]

abbreviation watt ≡ kilogram · metre\(^2\) · second\(^{-3}\)

type-synonym 'a watt = 'a[M · L\(^2\) · T\(^{-3}\), SI]

abbreviation coulomb ≡ ampere · second

type-synonym 'a coulomb = 'a[I · T, SI]
4.3. PHYSICAL CONSTANTS

**abbreviation** lumen ≡ candela · steradian

**type-synonym** 'a lumen = 'a[J · (L² · L⁻²), SI]

### 4.3.2 Constants

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

**default-sort** field-char-0

Hyperfine transition frequency of frequency of Cs

**abbreviation** caesium-frequency :: 'a[T⁻¹,SI] (ΔvCs) where caesium-frequency ≡ 9192631770 *Q hertz

Speed of light in vacuum

**abbreviation** speed-of-light :: 'a[L · T⁻¹,SI] (c) where speed-of-light ≡ 299792458 *Q (metre-second⁻¹)

Planck constant

**abbreviation** Planck :: 'a[M · L² · T⁻² · Θ⁻¹,SI] (h) where Planck ≡ (6.62607015 · 1/(10⁻³⁴)) *Q (joule-second)

Elementary charge

**abbreviation** elementary-charge :: 'a[I · T,SI] (e) where elementary-charge ≡ (1.602176634 · 1/(10⁻¹⁹)) *Q coulomb

The Boltzmann constant

**abbreviation** Boltzmann :: 'a[M · L² · T⁻² · Θ⁻¹,SI] (k) where Boltzmann ≡ (1.380649·1/(10⁻²³)) *Q (joule / kelvin)

The Avogadro number

**abbreviation** Avogadro :: 'a[N⁻¹,SI] (NA) where Avogadro ≡ 6.02214076·(10⁻²³) *Q (mole⁻¹)

**abbreviation** max-luminous-frequency :: 'a[T⁻¹,SI] where max-luminous-frequency ≡ (540·10¹²) *Q hertz

The luminous efficacy of monochromatic radiation of frequency max-luminous-frequency.

**abbreviation** luminous-efficacy :: 'a[J · (L² · L⁻²) · (M · L² · T⁻³)⁻¹,SI] (Kcd)

where luminous-efficacy ≡ 683 *Q (lumen/watt)

### 4.3.3 Checking Foundational Equations of the SI System

**theorem** second-definition:

1 *Q second ≡ₚ(9192631770 *Q 1) / ΔvCs

by si-calc
theorem metre-definition:
\begin{align*}
I \cdot Q \text{ metre} &\cong Q \left( \frac{c}{(299792458 \cdot Q \text{ 1})} \right) \cdot \text{second} \\
I \cdot Q \text{ metre} &\cong Q \left( \frac{9192631770}{299792458} \cdot \frac{c}{\Delta v_{Cs}} \right)
\end{align*}
by si-calc+

theorem kilogram-definition:
\begin{align*}
(1 \cdot Q \text{ kilogram}) \cdot \text{a kilogram} &\cong Q \left( \frac{h}{6.62607015 \cdot 1/(10^{-34}) \cdot Q \text{ 1}} \right) \cdot \text{metre}^2 \cdot \text{second} \\
&\text{by si-calc}
\end{align*}

abbreviation approx-ice-point $\equiv 273.15 \cdot Q \text{ kelvin}$

default-sort type

end

4.4 SI Prefixes

theory SI-Prefix
  imports SI-Constants
begin

4.4.1 Definitions

Prefixes are simply numbers that can be composed with units using the scalar multiplication operator ($\cdot Q$).

default-sort ring-char-0

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 giga :: 'a where [si-eq]: giga = $10^9$
definition tera :: 'a where [si-eq]: tera = $10^{12}$
definition peta :: 'a where [si-eq]: peta = $10^{15}$
definition exa :: 'a where [si-eq]: exa = $10^{18}$
definition zetta :: 'a where [si-eq]: zetta = $10^{21}$
4.4. SI PREFIXES

**definition** gotta :: 'a where [si-eq]: gotta = 10^24

**default-sort** field-char-0

**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

**definition** nano :: 'a where [si-eq]: nano = 1/10^9

**definition** pico :: 'a where [si-eq]: pico = 1/10^12

**definition** femto :: 'a where [si-eq]: femto = 1/10^15

**definition** atto :: 'a where [si-eq]: atto = 1/10^18

**definition** zepto :: 'a where [si-eq]: zepto = 1/10^21

**definition** yocto :: 'a where [si-eq]: yocto = 1/10^24

### 4.4.2 Examples

**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)

### 4.4.3 Binary Prefixes

Although not in general applicable to physical quantities, we include these prefixes for completeness.

**default-sort** ring-char-0

**definition** kibi :: 'a where [si-eq]: kibi = 2^10

**definition** mebi :: 'a where [si-eq]: mebi = 2^20

**definition** gibi :: 'a where [si-eq]: gibi = 2^30

**definition** tebi :: 'a where [si-eq]: tebi = 2^40

**definition** pebi :: 'a where [si-eq]: pebi = 2^50

**definition** exbi :: 'a where [si-eq]: exbi = 2^60
CHAPTER 4. INTERNATIONAL SYSTEM OF UNITS

4.5 Derived SI-Units

theory SI-Derived
  imports SI-Prefix
begin

4.5.1 Definitions

abbreviation newton ≡ kilogram · metre · second$^{-2}$

abbreviation pascal ≡ kilogram · metre$^{-1}$ · second$^{-2}$

abbreviation volt ≡ kilogram · metre$^2$ · second$^{-3}$ · ampere$^{-1}$

abbreviation farad ≡ kilogram$^{-1}$ · metre$^{-2}$ · second$^4$ · ampere$^2$

abbreviation ohm ≡ kilogram · metre$^2$ · second$^{-3}$ · ampere$^{-2}$

abbreviation siemens ≡ kilogram$^{-1}$ · metre$^{-2}$ · second$^3$ · ampere$^2$

abbreviation weber ≡ kilogram · metre$^2$ · second$^{-2}$ · ampere$^{-1}$

abbreviation tesla ≡ kilogram · second$^{-2}$ · ampere$^{-1}$

abbreviation henry ≡ kilogram · metre$^2$ · second$^{-2}$ · ampere$^{-2}$

abbreviation lux ≡ candela · steradian · metre$^{-2}$

abbreviation (input) becquerel ≡ second$^{-1}$
4.5. DERIVED SI-UNITS

abbreviation gray ≡ metre\(^2\) · second\(^{-2}\)

abbreviation sievert ≡ metre\(^2\) · second\(^{-2}\)

abbreviation katal ≡ mole · second\(^{-1}\)

definition degrees-celsius :: 'a::field-char-θ ⇒ 'a[Θ] (-XXXC [999] 999)
where [si-eq]: degrees-celsius \(x = (x \ast_Q \text{kelvin}) + \text{approx-ice-point}\)

definition [si-eq]: gram = milli \(\ast_Q\) kilogram

4.5.2 Equivalences

lemma joule-alt-def: joule \(\cong_Q\) newton · metre
   by si-calc

lemma watt-alt-def: watt \(\cong_Q\) joule / second
   by si-calc

lemma volt-alt-def: volt = watt / ampere
   by simp

lemma farad-alt-def: farad \(\cong_Q\) coulomb / volt
   by si-calc

lemma ohm-alt-def: ohm \(\cong_Q\) volt / ampere
   by si-calc

lemma siemens-alt-def: siemens \(\cong_Q\) ampere / volt
   by si-calc

lemma weber-alt-def: weber \(\cong_Q\) volt · second
   by si-calc

lemma tesla-alt-def: tesla \(\cong_Q\) weber / metre\(^2\)
   by si-calc

lemma henry-alt-def: henry \(\cong_Q\) weber / ampere
   by si-calc

lemma lux-alt-def: lux = lumen / metre\(^2\)
   by simp

lemma gray-alt-def: gray \(\cong_Q\) joule / kilogram
   by si-calc

lemma sievert-alt-def: sievert \(\cong_Q\) joule / kilogram
   by si-calc
4.5.3 Properties

**lemma** kilogram: kilo *Q gram = kilogram
  by (si-simp)

**lemma** celsius-to-kelvin: $T_{XXC} = (T *Q \text{ kelvin}) + (273.15 *Q \text{ kelvin})$
  by (si-simp)

end

4.6 Non-SI Units Accepted for SI use

**theory** SI-Accepted
  **imports** SI-Derived
  begin

  **definition** [si-def, si-eq]: minute = 60 *Q second

  **definition** [si-def, si-eq]: hour = 60 *Q minute

  **definition** [si-def, si-eq]: day = 24 *Q hour

  **definition** [si-def, si-eq]: astronomical-unit = $149597870700 *Q \text{ metre}$

  **definition** degree :: 'a::real-field[X/Y] where [si-def, si-eq]: degree = $(2 \cdot \text{(of-real pi)} / 180) *Q \text{ radian}$

  **abbreviation** degrees (-XXX [999] 999) where nXXX ≡ n *Q degree

  **definition** [si-def, si-eq]: litre = $1/1000 *Q \text{ metre}^3$

  **definition** [si-def, si-eq]: tonne = $10^3 *Q \text{ kilogram}$

  **definition** [si-def, si-eq]: dalton = $1.6605390666 * (1 / 10^{-27}) *Q \text{ kilogram}$

4.6.1 Example Unit Equations

**lemma** 1 *Q hour = 3600 *Q second
  by (si-simp)

**lemma** watt · hour ≡Q 3600 *Q joule  by (si-calc)

**lemma** 25 *Q metre / second = 90 *Q (kilo *Q metre) / hour
  by (si-calc)

end
4.7 Imperial Units via SI Units

4.7.1 Units of Length

default-sort field-char-0

The units of length are defined in terms of the international yard, as standardised in 1959.

definition yard :: 'a[L] where
[si-eq]: yard = 0.9144 *Q metre

definition foot :: 'a[L] where
[si-eq]: foot = 1/3 *Q yard

lemma foot-alt-def: foot = 0.3048 *Q metre
  by (si-simp)

definition inch :: 'a[L] where
[si-eq]: inch = (1 / 36) *Q yard

lemma inch-alt-def: inch = 25.4 *Q milli *Q metre
  by (si-simp)

definition mile :: 'a[L] where
[si-eq]: mile = 1760 *Q yard

lemma mile-alt-def: mile = 1609.344 *Q metre
  by (si-simp)

definition nautical-mile :: 'a[L] where
[si-eq]: nautical-mile = 1852 *Q metre

4.7.2 Units of Mass

The units of mass are defined in terms of the international yard, as standardised in 1959.

definition pound :: 'a[M] where
[si-eq]: pound = 0.45359237 *Q kilogram

definition ounce :: 'a[M] where
[si-eq]: ounce = 1/16 *Q pound

definition stone :: 'a[M] where
[si-eq]: stone = 14 *Q pound
4.7.3 Other Units

**definition knot :: \( 'a[L \cdot T^{-1}] \)** where

[si-eq]: knot = 1 *Q (nautical-mile / hour)

**definition pint :: \( 'a[Volume] \)** where

[si-eq]: pint = 0.56826125 *Q litre

**definition gallon :: \( 'a[Volume] \)** where

[si-eq]: gallon = 8 *Q pint

**definition degrees-farenheit :: \( 'a[\Theta] \)** where

[si-eq]: degrees-farenheit \( x = (x + 459.67) \cdot 5/9 \) *Q kelvin

4.7.4 Unit Equations

**lemma miles-to-feet: mile = 5280 *Q foot**

by si-simp

**lemma mph-to-kmh: 1 *Q (mile / hour) = 1.609344 *Q ((kilo *Q metre) / hour)**

by si-simp

**lemma farenheit-to-celcius: TXXXF = ((T - 32) \cdot 5/9).XXXC**

by si-simp

end

4.8 Meta-Theory for SI Units

theory SI

imports SI-Units SI-Constants SI-Prefix SI-Derived SI-Accepted SI-Imperial

begin end

4.9 Astronomical Constants

theory SI-Astronomical

imports SI HOL-Decision-Procs.Approximation

begin

We create a number of astronomical constants and prove relationships between some of them. For this, we use the approximation method that can compute bounds on transcendental functions.

**definition julian-year :: \( 'a::field[T] \)** where

[si-eq]: julian-year = 365.25 *Q day

**definition light-year :: \( 'a::field-char-0[L] \)** where
4.10. PARSING AND PRETTY PRINTING OF SI UNITS

light-year = QCOERCE[L] (c ∙ julian-year)

We need to apply a coercion in the definition of light year to convert the
dimension type from \( L \cdot T^{-1} \cdot T \) to \( L \). The correctness of this coercion is
confirmed by the following equivalence theorem.

**lemma** light-year: light-year \( \cong \) c ∙ julian-year

**unfolding** light-year-def by (si-calc)

**lemma** light-year-eq [si-eq]: [light-year]_Q = [c ∙ julian-year]_Q

**using** light-year quant-equiv-iff by blast

HOL can characterise \( \pi \) exactly and so we also give an exact value for the
parsec.

**definition** parsec :: real[L] where

[si-eq]: parsec = 648000 / \( \pi \) *astronomical-unit

We calculate some conservative bounds on the parsec: it is somewhere be-
tween 3.26 and 3.27 light-years.

**lemma** parsec-lb: 3.26 *Q light-year < parsec

**by** (si-simp, approximation 12)

**lemma** parsec-ub: parsec < 3.27 *Q light-year

**by** (si-simp, approximation 12)

The full beauty of the approach is perhaps revealed here, with the type of
a classical three-dimensional gravitation field:

**type-synonym** gravitation-field = real^3[L] ⇒ (real^3[L ∙ T^{-2}])

end

4.10 Parsing and Pretty Printing of SI Units

theory SI-Pretty

imports SI

begin

4.10.1 Syntactic SI Units

The following syntactic representation can apply at both the type and value
level.

**nonterminal** si

**syntax**

-si-metre :: si (m)

-si-kilogram :: si (kg)

-si-second :: si (s)
-si-ampere :: si (A)
-si-kelvin :: si (K)
-si-mole :: si (mol)
-si-candela :: si (cd)

-si-square :: si ⇒ si ((−)² [999] 999)
-si-cube :: si ⇒ si ((−)³ [999] 999)
-si-quart :: si ⇒ si ((−)⁴ [999] 999)
-si-inverse :: si ⇒ si ((−)⁻¹ [999] 999)
-si-invsquare :: si ⇒ si ((−)⁻² [999] 999)
-si-invcube :: si ⇒ si ((−)⁻³ [999] 999)
-si-invquart :: si ⇒ si ((−)⁻⁴ [999] 999)
-si-times :: si ⇒ si ⇒ si (infixl · 70)

4.10.2 Type Notation

Pretty notation for SI units at the type level.

no-type-notation SIUnitT (-[-] [999,0] 999)

syntax
-si-unit :: type ⇒ si ⇒ type (-[-] [999,0] 999)
-si-print :: type ⇒ si (SI PRINT(‘-‘))

translations
(type) ‘a[SI PRINT(‘d’)] == (type) ‘a[‘d, SI]
(si) SI PRINT(‘d’)^2 == (si) SI PRINT(‘d^2’)
(si) SI PRINT(‘d’)^3 == (si) SI PRINT(‘d^3’)
(si) SI PRINT(‘d’)^4 == (si) SI PRINT(‘d^4’)
(si) SI PRINT(‘d’)^⁻¹ == (si) SI PRINT(‘d⁻¹’)
(si) SI PRINT(‘d’)^⁻² == (si) SI PRINT(‘d⁻²’)
(si) SI PRINT(‘d’)^⁻³ == (si) SI PRINT(‘d⁻³’)
(si) SI PRINT(‘d’)^⁻⁴ == (si) SI PRINT(‘d⁻⁴’)
(si) SI PRINT(‘d_1’) · SI PRINT(‘d_2’) == (si) SI PRINT(‘d_1 · d_2’)
(si) m == (si) SI PRINT(M)
(si) kg == (si) SI PRINT(T)
(si) A == (si) SI PRINT(T)
(si) K == (si) SI PRINT(Θ)
(si) mol == (si) SI PRINT(N)
(si) cd == (si) SI PRINT(J)

-si-invquart x <= -si-inverse -si-square x
-si-invcube x <= -si-inverse -si-cube x
-si-invquart x <= -si-inverse -si-quart x

-si-invquart x <= -si-square -si-inverse x
-si-invcube x <= -si-cube -si-inverse x
4.10. Parsing and Pretty Printing of SI Units

\[-si\text{-invquart } x \leq -si\text{-quart } (-si\text{-inverse } x)\]

\textbf{typ} \[real[m\cdot s^{-2}]\]
\textbf{typ} \[real[m\cdot s^{-2}\cdot A^2]\]
\textbf{term} \[5 \times_5 Q \text{ joule}\]

4.10.3 Value Notations

Pretty notation for SI units at the type level. Currently, it is not possible to support prefixes, as this would require a more sophisticated cartouche parser.

\textbf{definition} \[SIQ \ n \ u = n \times_5 Q \ u\]

\textbf{syntax}
\begin{align*}
-si\text{-term} & :: \text{si} \Rightarrow \text{logic} (SI'(\cdot)) \\
-siq\text{-term} & :: \text{logic} \Rightarrow \text{si} \Rightarrow \text{logic} (SI[\cdot, \cdot]) \\
-siq\text{-print} & :: \text{logic} \Rightarrow \text{si}
\end{align*}

\textbf{translations}
\begin{align*}
-siq\text{-term } n \ u & => \text{CONST } SIQ \ n \ (-si\text{-term } u) \\
-siq\text{-term } n \ (-si\text{-print } u) & <= \text{CONST } SIQ \ n \ u \\
-si\text{-term } (-si\text{-times } x \ y) & == (-si\text{-term } x) \cdot (-si\text{-term } y) \\
-si\text{-term } (-si\text{-inverse } x) & == (-si\text{-term } x)^{-1} \\
-si\text{-term } (-si\text{-square } x) & == (-si\text{-term } x)^2 \\
-si\text{-term } (-si\text{-cube } x) & == (-si\text{-term } x)^3 \\
SI(m) & => \text{CONST } \text{metre} \\
SI(kg) & => \text{CONST } \text{kilogram} \\
SI(s) & => \text{CONST } \text{second} \\
SI(A) & => \text{CONST } \text{ampere} \\
SI(K) & => \text{CONST } \text{kelvin} \\
SI(mol) & => \text{CONST } \text{mole} \\
SI(cd) & => \text{CONST } \text{candela}
\end{align*}

\begin{align*}
-si\text{-inverse } (-si\text{-print } x) & <= -si\text{-print } (x^{-1}) \\
-si\text{-invsquare } (-si\text{-print } x) & <= -si\text{-print } (x^{-2}) \\
-si\text{-invcube } (-si\text{-print } x) & <= -si\text{-print } (x^{-3}) \\
-si\text{-invquart } (-si\text{-print } x) & <= -si\text{-print } (x^{-4}) \\
-si\text{-square } (-si\text{-print } x) & <= -si\text{-print } (x^2) \\
-si\text{-cube } (-si\text{-print } x) & <= -si\text{-print } (x^3) \\
-si\text{-quart } (-si\text{-print } x) & <= -si\text{-print } (x^4) \\
-si\text{-times } (-si\text{-print } x) \ (-si\text{-print } y) & <= -si\text{-print } (x \cdot y)
\end{align*}

\begin{align*}
-si\text{-metre} & <= -si\text{-print } (\text{CONST } \text{metre}) \\
-si\text{-kilogram} & <= -si\text{-print } (\text{CONST } \text{kilogram}) \\
-si\text{-second} & <= -si\text{-print } (\text{CONST } \text{second}) \\
-si\text{-ampere} & <= -si\text{-print } (\text{CONST } \text{ampere}) \\
-si\text{-kelvin} & <= -si\text{-print } (\text{CONST } \text{kelvin}) \\
-si\text{-mole} & <= -si\text{-print } (\text{CONST } \text{mole})
\end{align*}
CHAPTER 4. INTERNATIONAL SYSTEM OF UNITS

\[-si\text{-candela} \leq -si\text{-print} (\text{CONST candela})\]

term $SI[5, m^2]$
term $SI[22, m s^{-1}]$

end

4.11 British Imperial System (1824/1897)

thory $BIS$
imports $ISQ$ $SI$-Units $CGS$
begin

The values in the British Imperial System (BIS) are derived from the UK Weights and Measures Act 1824.

4.11.1 Preliminaries
typedef $BIS = \text{UNIV :: unit set }$
instance $BIS :: \text{unit-system}$
  by (rule unit-system-intro[of $Abs-BIS ()$],
      metis (full-types) $Abs-BIS$-cases $UNIV$-eq-I $\text{insert-iff old}$ $\text{unit}$ $\text{exhaust}$)
instance $BIS :: \text{time-second }$
abbreviation $BIS \equiv \text{unit :: BIS}$

4.11.2 Base Units
abbreviation $yard \equiv BUNIT(L, BIS)$
abbreviation $pound \equiv BUNIT(M, BIS)$
abbreviation $rankine \equiv BUNIT(\Theta, BIS)$

We chose Rankine rather than Farenheit as this is more compatible with the SI system and avoids the need for having an offset in conversion functions.

4.11.3 Derived Units
definition [si-eq]: $foot = 1/3 \ast_Q yard$
definition [si-eq]: $inch = 1/12 \ast_Q foot$
definition [si-eq]: $furlong = 220 \ast_Q yard$
definition [si-eq]: $mile = 1760 \ast_Q yard$
definition [si-eq]: $acre = 4840 \ast_Q yard^2$
definition [si-eq]: $ounce = 1/12 \ast_Q pound$
4.11. BRITISH IMPERIAL SYSTEM (1824/1897)

definition [si-eq]: gallon = 277.421 \*Q inch³

definition [si-eq]: quart = 1/4 \*Q gallon

definition [si-eq]: pint = 1/8 \*Q gallon

definition [si-eq]: peck = 2 \*Q gallon

definition [si-eq]: bushel = 8 \*Q gallon

definition [si-eq]: minute = 60 \*Q second

definition [si-eq]: hour = 60 \*Q minute

4.11.4 Conversion to SI

instantiation BIS :: metrifiable

begin

lift-definition convschema-BIS :: BIS itself ⇒ (BIS, SI) Conversion is

\[ \lambda x. (\begin{aligned}
\text{cLengthF} &= 0.9143993, \\
\text{cMassF} &= 0.453592338, \\
\text{cTimeF} &= 1, \\
\text{cCurrentF} &= 1, \\
\text{cTemperatureF} &= 5/9, \\
\text{cAmountF} &= 1, \\
\text{cIntensityF} &= 1
\end{aligned}) \]

by simp

instance ..

end

lemma BIS-SI-simps [simp]:

\[ \begin{aligned}
\text{LengthF} (\text{convschema} (a::\text{BIS itself})) &= 0.9143993 \\
\text{MassF} (\text{convschema} a) &= 0.453592338 \\
\text{TimeF} (\text{convschema} a) &= 1 \\
\text{CurrentF} (\text{convschema} a) &= 1 \\
\text{TemperatureF} (\text{convschema} a) &= 5/9
\end{aligned} \]

by (transfer, simp)+

4.11.5 Conversion Examples

lemma metrify (foot :: rat[L, BIS]) = 0.9143993 / 3 \*Q metre

by (simp add: foot-def)

lemma metrify ((70::rat) \*Q mile / hour) = (704087461 / 22500000) \*Q (metre / second)

by (si-simp)

lemma QMC(CGS → BIS) ((1::rat) \*Q centimetre) = 100000 / 9143993 \*Q yard

by simp

end
Bibliography


