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

Simon Foster Burkhart Wolff

February 23, 2021

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

Chapter 1

ISQ and SI: An Introduction

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

The relevant international standard for quantities and measurements is distributed by the *Bureau International des Poids et des Mesures* (BIPM), which also provides the *Vocabulaire International de Mètrologie* (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]. Quantities and units are represented in a way that they have a *quantity type* as well as a *unit type* based on its base vectors and their magnitudes. Since the algebra of quantities induces congruences on quantity and SI types, specific tactic support has been developed to capture these. Our construction is validated by a test-set of known equivalences between both quantities and SI units. Moreover, the presented theory can be used for type-safe conversions between the SI system and others, like the British Imperial System (BIS).

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 *(int, 'd::enum) dimvec* , i.e. a *'d*-indexed 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::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, Θ, N, J corresponding to the above mentioned base vectors.

6. *(Abstract) Measurement Systems* represented by type *('a, 'd::enum, 's::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 sich 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 $J = kg\, m^2/s^2$ 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 $\cong_Q$ in semantic terms. This means that we can relate quantities with syntactically different dimension types, yet with same dimension semantics. This paves the way for derived rules that do computations of terms, which represent type computations indirectly. This principle is the basis for the tactic support, which allows for the dimensional type checking of key definitions of the SI system. Some examples are given below.

theorem *metre-definition*:
$1 *_Q \mathit{metre} \cong_Q (\mathbf{c} \ / \ (299792458 *_Q \mathbf{1}))\cdot \mathit{second}$
by *si-calc*

theorem *kilogram-definition*:
$1 *_Q \mathit{kilogram} \cong_Q (\mathbf{h} \ / \ (6.62607015 \cdot 1/(10\hat{\ }34) *_Q \mathbf{1}))\cdot \mathit{metre}^{-\mathbf{2}}\cdot \mathit{second}$
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* :: $'a::\{linordered\text{-}field\} \Rightarrow int \Rightarrow 'a$ (**infixr** $\hat{}_Z$ *80*) **where**
$intpow\ x\ n = (if\ (n < 0)\ then\ inverse\ (x\ \hat{}\ nat\ (-n))\ else\ (x\ \hat{}\ nat\ n))$

lemma *intpow-zero* [*simp*]: $x\ \hat{}_Z\ 0 = 1$
by (*simp add*: *intpow-def*)

lemma *intpow-spos* [*simp*]: $x > 0 \Longrightarrow x\ \hat{}_Z\ n > 0$
by (*simp add*: *intpow-def*)

lemma *intpow-one* [*simp*]: $x\ \hat{}_Z\ 1 = x$
by (*simp add*: *intpow-def*)

lemma *one-intpow* [*simp*]: $1\ \hat{}_Z\ n = 1$
by (*simp add*: *intpow-def*)

lemma *intpow-plus*: $x > 0 \Longrightarrow x\ \hat{}_Z\ (m + n) = x\ \hat{}_Z\ m * x\ \hat{}_Z\ n$
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*: $x > 0 \Longrightarrow x\ \hat{}_Z\ m * (x\ \hat{}_Z\ n * y) = x\ \hat{}_Z\ (m + n) * y$
by (*simp add*: *intpow-plus*)

lemma *intpow-pos* [*simp*]: $n \geq 0 \Longrightarrow x\ \hat{}_Z\ n = x\ \hat{}\ nat\ n$

by (*simp add*: *intpow-def*)

lemma *intpow-uminus*: $x\ \hat{}_Z\ -n = inverse\ (x\ \hat{}_Z\ n)$
by (*simp add*: *intpow-def*)

lemma *intpow-uminus-nat*: $n \geq 0 \Longrightarrow x\ \hat{}_Z\ -n = inverse\ (x\ \hat{}\ nat\ n)$
by (*simp add*: *intpow-def*)

lemma *intpow-inverse*: $inverse\ a\ \hat{}_Z\ n = inverse\ (a\ \hat{}_Z\ n)$
by (*simp add*: *intpow-def power-inverse*)

lemma *intpow-mult-distrib*: $(x * y)\ \hat{}_Z\ m = x\ \hat{}_Z\ m * y\ \hat{}_Z\ m$
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* :: $'a\ list \Rightarrow\ 'a \Rightarrow nat \Rightarrow nat$ **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*:
$x \in set(xs) \Longrightarrow$ *first-ind xs x i* $< length(xs) + i$
by (*induct xs arbitrary*: *i*, *auto*, *metis add-Suc-right*)

lemma *nth-first-ind*:
$[\![$ *distinct xs*; $x \in set(xs)$ $]\!] \Longrightarrow$ *xs* ! (*first-ind xs x 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*:
$[\![$ *distinct xs*; $i < length\ xs$ $]\!] \Longrightarrow$ *first-ind xs* (*xs* ! *i*) *j* = *i* + *j*
apply (*induct xs arbitrary*: *i j*)
apply (*auto*)
apply (*metis less-Suc-eq-le nth-equal-first-eq*)

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* **where**
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 *enum-ind* $x < CARD('a)$
    by (*simp add*: *enum-ind-less-CARD*[*of* x])
  show *enum-class.enum* ! *enum-ind* $x = x$
    by *simp*
  show $\bigwedge i.\ i < CARD('a) \Longrightarrow x = ENUM('a)\ !\ i \Longrightarrow i =$ *enum-ind* $(ENUM('a)$
$!\ i)$
    by (*simp add*: *enum-ind-nth*)
qed

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

lemma *enum-ind-neq* [*simp*]: $x \neq y \Longrightarrow$ *enum-ind* $x \neq$ *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 *times* (**infixl** $\cdot$ *70*)

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

lemma *div-conv-mult-inverse*: $a\ /\ b = a \cdot$ (*inverse* b)
  by *simp*

sublocale *mult*: *group times 1 inverse*
  by *standard* (*simp-all add*: *left-inverse*)

lemma *diff-self* [*simp*]: $a\ /\ a = 1$
  using *mult.right-inverse* **by** *auto*

lemma *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 *ab-group-mult* = *comm-monoid-mult* + *group-mult*
begin

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* ((-⁻ˉ) [*1000*,*999*] *999*)
where *npower x n* ≡ *inverse* $(x \ \hat{} \ n)$

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*} ⟹ *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* = ()
definition *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* $\Rightarrow$ $'n$) *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

definition *one-dimvec* :: ('*a*, '*b*) *dimvec* **where** *one-dimvec* = *dim-lambda* (λ *i*. *0*)
instance ..
end

instantiation *dimvec* :: (*plus*, *enum*) *times*
begin
definition *times-dimvec* :: ('*a*, '*b*) *dimvec* ⇒ ('*a*, '*b*) *dimvec* ⇒ ('*a*, '*b*) *dimvec*
where
times-dimvec x y = *dim-lambda* (λ *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* ⇒ ('*a*, '*b*) *dimvec* **where**
inverse-dimvec x = *dim-lambda* (λ *i*. − *dim-nth x i*)

definition *divide-dimvec* :: ('*a*, '*b*) *dimvec* ⇒ ('*a*, '*b*) *dimvec* ⇒ ('*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* ⇒ ('*n*::*ring-1*, '*d*::*enum*) *dimvec*
where *mk-dimvec ds* = (*if* (*length ds* = *CARD*('*d*)) *then dim-lambda* (λ *d*. *ds* ! *enum-ind d*) *else 1*)

code-datatype *mk-dimvec*

lemma *mk-dimvec-inj*: *inj-on* (*mk-dimvec* :: '*n list* ⇒ ('*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 $\bigwedge i.\ i <$ *length x* $\Longrightarrow$ *x* ! *i* = *y* ! *i*
proof –
fix *i*
assume *i* < *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*) $\longleftrightarrow$ (*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*($'a$) $\wedge$ *length ys* = *CARD*($'a$))
then mk-dimvec (*map* (λ (*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*]:
(*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*=*id*, *simplified*] *comp-def split-beta*$'$ *zip-same-conv-map distrib-right mult.commute*)

lemma *inverse-mk-dimvec* [*code*, *si-def*]:
(*inverse* (*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 mk-dimvec-def fun-eq-iff*)

lemma *divide-mk-dimvec* [*code*, *si-def*]:
(*mk-dimvec xs* / *mk-dimvec ys* :: ($'n$::*ring-1*, $'a$::*enum*) *dimvec*) =

$(if\ (length\ xs = CARD('a) \wedge length\ ys = CARD('a))$
$then\ mk\text{-}dimvec\ (map\ (\lambda\ (x,\ y).\ x - y)\ (zip\ xs\ ys))$
$else\ if\ length\ ys = CARD('a)\ then\ mk\text{-}dimvec\ (map\ uminus\ ys)\ else\ mk\text{-}dimvec\ xs)$
by (*auto simp add*: *divide-dimvec-def inverse-mk-dimvec times-mk-dimvec zip-map-map*[**where** *f=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.

definition *mk-BaseDim* :: *'d::enum* ⇒ (*int*, *'d*) *dimvec* **where**
$mk\text{-}BaseDim\ d = dim\text{-}lambda\ (\lambda\ i.\ if\ (i = d)\ then\ 1\ else\ 0)$

lemma *mk-BaseDim-neq* [*simp*]: $x \neq y \Longrightarrow mk\text{-}BaseDim\ x \neq mk\text{-}BaseDim\ y$
by (*auto simp add*: *mk-BaseDim-def fun-eq-iff*)

lemma *mk-BaseDim-code* [*code*]: $mk\text{-}BaseDim\ (d::'d::enum) = mk\text{-}dimvec\ (list\text{-}update\ (replicate\ CARD('d)\ 0)\ (enum\text{-}ind\ d)\ 1)$
by (*auto simp add*: *mk-BaseDim-def mk-dimvec-def fun-eq-iff*)

definition *is-BaseDim* :: (*int*, *'d::enum*) *dimvec* ⇒ *bool*
where $is\text{-}BaseDim\ x \equiv (\exists\ i.\ x = dim\text{-}lambda\ ((\lambda\ x.\ 0)(i := 1)))$

lemma *is-BaseDim-mk* [*simp*]: *is-BaseDim* (*mk-BaseDim x*)
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.

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

lemma *sdim-UNIV*: (*UNIV* :: *sdim set*) = {*Length*, *Mass*, *Time*, *Current*, *Temperature*, *Amount*, *Intensity*}
using *sdim.exhaust* **by** *blast*

lemma *CARD-sdim* [*simp*]: $CARD(sdim) = 7$
by (*simp add*: *sdim-UNIV*)

instantiation *sdim* :: *enum*
begin
definition *enum-sdim* = [*Length*, *Mass*, *Time*, *Current*, *Temperature*, *Amount*, *Intensity*]
definition $enum\text{-}all\text{-}sdim\ P \longleftrightarrow P\ Length \wedge P\ Mass \wedge P\ Time \wedge P\ Current \wedge P\ Temperature \wedge P\ Amount \wedge P\ Intensity$
definition $enum\text{-}ex\text{-}sdim\ P \longleftrightarrow P\ Length \vee P\ Mass \vee P\ Time \vee P\ Current \vee P\ Temperature \vee P\ Amount \vee P\ Intensity$
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 $\mathbf{L}{\cdot}\mathbf{M}{\cdot}\mathbf{T}^{-2}$
term $\mathbf{M}{\cdot}\mathbf{L}^{-3}$

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

lemma $\mathbf{L{\cdot}M{\cdot}T^{-2}} = \textit{mk-dimvec}\ [1,\ 1,\ -\ 2,\ 0,\ 0,\ 0,\ 0]$
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.

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:

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.

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

instantiation *Temperature* :: *basedim-type*

begin
definition [*si-eq*]: *dim-ty-sem-Temperature* (-::*Temperature itself*) = **Θ**
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 $'a \cdot 'b$ 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* = ('*a*^{2})$^{-1}$ ((-)$^{-2}$ [*999*] *999*)
type-synonym '*a DimInvCube* = ('*a*^{3})$^{-1}$ ((-)$^{-3}$ [*999*] *999*)
type-synonym '*a DimInvQuart* = ('*a*^{4})$^{-1}$ ((-)$^{-4}$ [*999*] *999*)

translations (*type*) '*a*$^{-2}$ <= (*type*) ('*a*^{2})$^{-1}$
translations (*type*) '*a*$^{-3}$ <= (*type*) ('*a*^{3})$^{-1}$
translations (*type*) '*a*$^{-4}$ <= (*type*) ('*a*^{4})$^{-1}$

print-translation ‹
 [(@{*type-syntax DimTimes*},
  fn ctx => *fn* [*a*, *b*] =>
   if (*a* = *b*)
    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 case a1 of
            Const (@{type-syntax DimTimes}, -) $ a11 $ a12 =>
              if (a11 = a12 andalso a12 = a2 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* = $L{\cdot}T^{-1}$
type-synonym *Frequency* = T^{-1}
type-synonym *Energy* = $L^2{\cdot}M{\cdot}T^{-2}$
type-synonym *Power* = $L^2{\cdot}M{\cdot}T^{-3}$
type-synonym *Force* = $L{\cdot}M{\cdot}T^{-2}$
type-synonym *Pressure* = $L^{-1}{\cdot}M{\cdot}T^{-2}$
type-synonym *Charge* = $I{\cdot}T$
type-synonym *PotentialDifference* = $L^2{\cdot}M{\cdot}T^{-3}{\cdot}I^{-1}$
type-synonym *Capacitance* = $L^{-2}{\cdot}M^{-1}{\cdot}T^4{\cdot}I^2$

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 Temperature}, []))   = [0, 0, 0, 0, 1, 0, 0] |
      typ-to-dim (Type (@{type-name Amount}, []))   = [0, 0, 0, 0, 0, 1, 0] |
```

```
    typ-to-dim (Type (@{type-name Intensity}, []))   = [0, 0, 0, 0, 0, 0, 1] |
    typ-to-dim (Type (@{type-name NoDimension}, []))  = [0, 0, 0, 0, 0, 0, 0] |
    typ-to-dim (Type (@{type-name DimInv}, [x])) = map (fn x => 0 − x) (typ-to-dim x) |
    typ-to-dim (Type (@{type-name DimTimes}, [x, y]))
      = map (fn (x, y) => x + y) (ListPair.zip (typ-to-dim x, typ-to-dim y)) |
    typ-to-dim - = raise Match;

  fun DimPow 0 - = Type (@{type-name NoDimension}, []) |
    DimPow 1 t = t |
    DimPow n t = (if (n > 0) then Type (@{type-name DimTimes}, [DimPow (n − 1) t, t])
                              else Type (@{type-name DimInv}, [DimPow (0 − n) t]));

  fun dim-to-typ ds =
    let val dts = map (fn (n, d) => DimPow n d) (filter (fn (n, -) => n <> 0) (ListPair.zip (ds, dims)))
    in if (dts = []) then @{typ NoDimension} else
         foldl1 (fn (x, y) => Type (@{type-name DimTimes}, [x, y])) dts
    end;

  val normalise = dim-to-typ o typ-to-dim;

end;

Dimension-Type.typ-to-dim @{typ L⁻²·M⁻¹·T⁴·I²·M};
Dimension-Type.normalise @{typ L⁻²·M⁻¹·T⁴·I²·M};
⟩
```

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.
dim :: (*int*, *'d*) *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 *int*, *rat*, or *real*, though we usually minimally expect a field.

lemma *Quantity-eq-intro*:
  assumes *mag x = mag y dim x = dim y more x = more y*
  shows *x = y*
  by (*simp add*: *assms eq-unit*)

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

instantiation *Quantity-ext* :: (*times*, *enum*, *times*) *times*
begin
definition *times-Quantity-ext* ::
  $('a, 'b, 'c)$ *Quantity-scheme* $\Rightarrow$ $('a, 'b, 'c)$ *Quantity-scheme* $\Rightarrow$ $('a, 'b, 'c)$ *Quantity-scheme*
  where [*si-def*]: *times-Quantity-ext x y* = $(\!|$ *mag* = *mag x* $\cdot$ *mag y*, *dim* = *dim x* $\cdot$ *dim y*,
  $\ldots$ = *more x* $\cdot$ *more y* $|\!)$
instance ..
end

lemma *mag-times* [*simp*]: *mag* $(x \cdot y)$ = *mag x* $\cdot$ *mag y* **by** (*simp add*: *times-Quantity-ext-def*)
lemma *dim-times* [*simp*]: *dim* $(x \cdot y)$ = *dim x* $\cdot$ *dim y* **by** (*simp add*: *times-Quantity-ext-def*)
lemma *more-times* [*simp*]: *more* $(x \cdot y)$ = *more x* $\cdot$ *more y* **by** (*simp add*: *times-Quantity-ext-def*)

The zero and one quantities are both dimensionless quantities with magnitude of $0::'a$ and $1::'a$, respectively.

instantiation *Quantity-ext* :: (*zero*, *enum*, *zero*) *zero*
begin
  definition *zero-Quantity-ext* = $(\!|$ *mag* = *0*, *dim* = *1*, $\ldots$ = *0* $|\!)$
instance ..
end

lemma *mag-zero* [*simp*]: *mag 0 = 0* **by** (*simp add*: *zero-Quantity-ext-def*)
lemma *dim-zero* [*simp*]: *dim 0 = 1* **by** (*simp add*: *zero-Quantity-ext-def*)
lemma *more-zero* [*simp*]: *more 0 = 0* **by** (*simp add*: *zero-Quantity-ext-def*)

instantiation *Quantity-ext* :: (*one*, *enum*, *one*) *one*
begin
  definition [*si-def*]: *one-Quantity-ext* = $(\!|$ *mag* = *1*, *dim* = *1*, $\ldots$ = *1* $|\!)$
instance ..
end

lemma *mag-one* [*simp*]: *mag 1 = 1* **by** (*simp add*: *one-Quantity-ext-def*)
lemma *dim-one* [*simp*]: *dim 1 = 1* **by** (*simp add*: *one-Quantity-ext-def*)
lemma *more-one* [*simp*]: *more 1 = 1* **by** (*simp add*: *one-Quantity-ext-def*)

Quantity inversion inverts both the magnitude and the dimension. Similarly, division of one quantity by another, divides both the magnitudes and the dimensions.

instantiation *Quantity-ext* :: (*inverse*, *enum*, *inverse*) *inverse*
begin
definition *inverse-Quantity-ext* :: ('*a*, '*b*, '*c*) *Quantity-scheme* ⇒ ('*a*, '*b*, '*c*) *Quantity-scheme* **where**
[*si-def*]: *inverse-Quantity-ext x* = ⦇ *mag* = *inverse* (*mag x*), *dim* = *inverse* (*dim x*), ... = *inverse* (*more x*) ⦈
definition *divide-Quantity-ext* :: ('*a*, '*b*, '*c*) *Quantity-scheme* ⇒ ('*a*, '*b*, '*c*) *Quantity-scheme* ⇒ ('*a*, '*b*, '*c*) *Quantity-scheme* **where**
[*si-def*]: *divide-Quantity-ext x y* = ⦇ *mag* = *mag x* / *mag y*, *dim* = *dim x* / *dim y*, ... = *more x* / *more y* ⦈
instance ..
end

lemma *mag-inverse* [*simp*]: *mag* (*inverse x*) = *inverse* (*mag x*)
by (*simp add*: *inverse-Quantity-ext-def*)

lemma *dim-inverse* [*simp*]: *dim* (*inverse x*) = *inverse* (*dim x*)
by (*simp add*: *inverse-Quantity-ext-def*)

lemma *more-inverse* [*simp*]: *more* (*inverse x*) = *inverse* (*more x*)
by (*simp add*: *inverse-Quantity-ext-def*)

lemma *mag-divide* [*simp*]: *mag* (*x* / *y*) = *mag x* / *mag y*
by (*simp add*: *divide-Quantity-ext-def*)

lemma *dim-divide* [*simp*]: *dim* (*x* / *y*) = *dim x* / *dim y*
by (*simp add*: *divide-Quantity-ext-def*)

lemma *more-divide* [*simp*]: *more* (*x* / *y*) = *more x* / *more y*
by (*simp add*: *divide-Quantity-ext-def*)

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

instance *Quantity-ext* :: (*comm-monoid-mult*, *enum*, *comm-monoid-mult*) *comm-monoid-mult*
by (*intro-classes*, *simp-all add*: *eq-unit one-Quantity-ext-def times-Quantity-ext-def mult.assoc*
,*simp add*: *mult.commute*)

instance *Quantity-ext* :: (*ab-group-mult*, *enum*, *ab-group-mult*) *ab-group-mult*
by (*intro-classes*, *rule Quantity-eq-intro*, *simp-all add*: *eq-unit*)

We can also define a partial order on quantities.

instantiation *Quantity-ext* :: (*ord*, *enum*, *ord*) *ord*
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* ⇒ (*'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*: $(\mathit{UNIV}::'s\ \mathit{set}) = \{a\} \Longrightarrow \mathit{OFCLASS}('s, \mathit{unit\text{-}system\text{-}class})$
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.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

definition *times-Measurement-System-ext* ::
$('a, 'b)$ *Measurement-System-ext* $\Rightarrow$ $('a, 'b)$ *Measurement-System-ext* $\Rightarrow$ $('a, 'b)$ *Measurement-System-ext*
where [*si-def*]: *times-Measurement-System-ext x y* = (| *unit-sys* = *unit*, ... = *mmore x* · *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* = (| *unit-sys* = *unit*, ... = *inverse* (*mmore x*) |)
definition *divide-Measurement-System-ext* ::
$('a, 'b)$ *Measurement-System-ext* $\Rightarrow$ $('a, 'b)$ *Measurement-System-ext* $\Rightarrow$ $('a, 'b)$ *Measurement-System-ext*
where [*si-def*]: *divide-Measurement-System-ext x y* = (| *unit-sys* = *unit*, ... = *mmore x* / *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* $\leq$ *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* = $(x \leq y \wedge \neg\, y \leq x)$
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
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 magntitude, 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 for coercion between two dimension expressions. This requires that the underlying dimensions are the same.

definition *coerceQuantT* :: $'d_2$ *itself* ⇒ $'a['d_1$::*dim-type*, $'s$::*unit-system*] ⇒ $'a['d_2$::*dim-type*, $'s$] **where**
[*si-def*]: $QD('d_1) = QD('d_2) \Longrightarrow$ *coerceQuantT t x* = (*toQ* (*fromQ x*))

syntax
-*QCOERCE* :: *type* ⇒ *logic* ⇒ *logic* (*QCOERCE*[-])

translations
QCOERCE[$'t$] == *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.

lift-definition *qless-eq* :: $'n$::*order*[$'a$::*dim-type*, $'s$::*unit-system*] ⇒ $'n['b$::*dim-type*, $'s$] ⇒ *bool* (**infix** $\lesssim_Q$ *50*)
is ($\leq$) .

lift-definition *qequiv* :: $'n['a$::*dim-type*, $'s$::*unit-system*] ⇒ $'n['b$::*dim-type*, $'s$] ⇒ *bool* (**infix** $\cong_Q$ *50*)
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.

lemma *qequiv-refl* [*simp*]: $a \cong_Q a$
by (*simp add*: *qequiv-def*)

lemma *qequiv-sym*: $a \cong_Q b \Longrightarrow b \cong_Q a$
by (*simp add*: *qequiv-def*)

lemma *qequiv-trans*: $[\![a \cong_Q b; b \cong_Q c]\!] \Longrightarrow a \cong_Q c$
by (*simp add*: *qequiv-def*)

theorem *qeq-iff-same-dim*:
fixes *x y* :: $'a['d$::*dim-type*, $'s$::*unit-system*]
shows $x \cong_Q y \longleftrightarrow x = y$
by (*transfer*, *simp*)

lemma *coerceQuant-eq-iff*:
fixes *x* :: $'a['d_1$::*dim-type*, $'s$::*unit-system*]
assumes $QD('d_1) = QD('d_2$::*dim-type*)

shows (*coerceQuantT TYPE('d₂) x*) $\cong_Q$ *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* $\cong_Q$ *y*
using *qequiv-sym assms(1) assms(2) coerceQuant-eq-iff* **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* $\cong_Q$ *y*
by (*simp add*: *assms(1) assms(2) coerceQuant-eq-iff2 coerceQuantT-def*)

This is more general that $y = x \Longrightarrow x \cong_Q y$, 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* $\cong_Q$ *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]* ⇒ *'n['b::dim-type, 's]* ⇒ *'n['a ·'b, 's]* (**infixl** · *69*)
is (∗) **by** (*simp add*: *dim-ty-sem-DimTimes-def times-Quantity-ext-def*)

lift-definition
qinverse :: (*'n::field*)*['a::dim-type, 's::unit-system]* ⇒ *'n['a*$^{-1}$*, 's]* ((-$^{-\mathbf{1}}$) [*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]* ⇒ *'n['b::dim-type, 's]* ⇒ *'n['a/'b, 's]* (**infixl** '/ *70*) **where**
qdivide x y ≡ *x* · *y*$^{-\mathbf{1}}$

We also provide some helpful notations for expressing heterogeneous powers.

abbreviation *qsq* ((-)$^{\mathbf{2}}$ [*999*] *999*) **where** $u^{\mathbf{2}} \equiv u{\cdot}u$
abbreviation *qcube* ((-)$^{\mathbf{3}}$ [*999*] *999*) **where** $u^{\mathbf{3}} \equiv u{\cdot}u{\cdot}u$
abbreviation *qquart* ((-)$^{\mathbf{4}}$ [*999*] *999*) **where** $u^{\mathbf{4}} \equiv u{\cdot}u{\cdot}u{\cdot}u$

abbreviation *qneq-sq* $((\text{-})^{-\mathbf{2}}$ [*999*] *999*) **where** $u^{-\mathbf{2}} \equiv (u^{\mathbf{2}})^{-\mathbf{1}}$
abbreviation *qneq-cube* $((\text{-})^{-\mathbf{3}}$ [*999*] *999*) **where** $u^{-\mathbf{3}} \equiv (u^{\mathbf{3}})^{-\mathbf{1}}$
abbreviation *qneq-quart* $((\text{-})^{-\mathbf{4}}$ [*999*] *999*) **where** $u^{-\mathbf{4}} \equiv (u^{\mathbf{3}})^{-\mathbf{1}}$

Analogous to the $(*_R)$ 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.

lift-definition *scaleQ* :: $'a \Rightarrow 'a$::*comm-ring-1*$['d$::*dim-type*, $'s$::*unit-system*$] \Rightarrow 'a['d, 's]$ (**infixr** $*_Q$ *63*)
is λ *r x*. (| *mag* = *r* $*$ *mag x*, *dim* = *QD*('*d*), *unit-sys* = *unit* |) **by** *simp*

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

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

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

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

abbreviation *qone* :: '*n*::*one*[**1**, '*s*::*unit-system*] (**1**) **where** *qone* ≡ *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.

instantiation *QuantT* :: (*plus*, *dim-type*, *unit-system*) *plus*
begin
lift-definition *plus-QuantT* :: $'a['b, 'c] \Rightarrow 'a['b, 'c] \Rightarrow 'a['b, 'c]$
is λ *x y*. (| *mag* = *mag x* + *mag y*, *dim* = *QD*('*b*), *unit-sys* = *unit* |)
by (*simp*)
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.

instance *QuantT* :: (*semigroup-add*,*dim-type*,*unit-system*) *semigroup-add*
  by (*intro-classes*, *transfer*, *simp add*: *add.assoc*)

instance *QuantT* :: (*ab-semigroup-add*,*dim-type*,*unit-system*) *ab-semigroup-add*
  by (*intro-classes*, *transfer*, *simp add*: *add.commute*)

instance *QuantT* :: (*monoid-add*,*dim-type*,*unit-system*) *monoid-add*
  by (*intro-classes*; (*transfer*, *simp add*: *eq-unit*))

instance *QuantT* :: (*comm-monoid-add*,*dim-type*,*unit-system*) *comm-monoid-add*
  by (*intro-classes*; *transfer*, *simp*)

instantiation *QuantT* :: (*uminus*,*dim-type*,*unit-system*) *uminus*
begin
lift-definition *uminus-QuantT* :: $'a['b,'c] \Rightarrow 'a['b,'c]$
  is λ *x*. ⦇ *mag* = − *mag x*, *dim* = *dim x*, *unit-sys* = *unit* ⦈ **by** (*simp*)
instance ..
end

instantiation *QuantT* :: (*minus*,*dim-type*,*unit-system*) *minus*
begin
lift-definition *minus-QuantT* :: $'a['b,'c] \Rightarrow 'a['b,'c] \Rightarrow 'a['b,'c]$
  is λ *x y*. ⦇ *mag* = *mag x* − *mag y*, *dim* = *dim x*, *unit-sys* = *unit* ⦈ **by** (*simp*)

instance ..
end

instance *QuantT* :: (*numeral*,*dim-type*,*unit-system*) *numeral* **..**

Moreover, types quantities also form an additive group.

instance *QuantT* :: (*ab-group-add*,*dim-type*,*unit-system*) *ab-group-add*
  by (*intro-classes*, (*transfer*, *simp*)+)

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

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

instance *QuantT* :: (*linorder*,*dim-type*,*unit-system*) *linorder*

by (*intro-classes*, *transfer*, *auto*)

instantiation *QuantT* :: (*scaleR*,*dim-type*,*unit-system*) *scaleR*
begin
lift-definition *scaleR-QuantT* :: *real* $\Rightarrow$ $'a['b,'c]$ $\Rightarrow$ $'a['b,'c]$
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* :: $'a['b,'c]$ $\Rightarrow$ *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* :: $'a['b,'c]$ $\Rightarrow$ $'a['b,'c]$ **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* :: $'a['b,'c]$ $\Rightarrow$ $'a['b,'c]$ $\Rightarrow$ *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*)
open-uniformity
begin

definition *open-QuantT* :: ($'a['b,'c]$) *set* $\Rightarrow$ *bool* **where**

open-QuantT U = ($\forall$ *x*$\in$*U*. *eventually* (λ(*x*′, *y*). *x*′ = *x* $\longrightarrow$ *y* $\in$ *U*) *uniformity*)
instance by (*intro-classes*, *simp add*: *open-QuantT-def*)
end

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*)

end

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* $\longleftrightarrow$ $[\![x]\!]_Q = [\![y]\!]_Q$
by (*auto simp add*: *magQ-def*, *transfer*, *simp add*: *eq-unit*)

lemma *quant-eqI* [*si-transfer*]:
$[\![x]\!]_Q = [\![y]\!]_Q \Longrightarrow x = y$
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 $x \cong_Q y \longleftrightarrow [\![x]\!]_Q = [\![y]\!]_Q \wedge QD('u_1) = QD('u_2)$
proof −
have $\forall$ *t ta*. (*ta*::′*a*[′u_2, ′*s*]) = *t* $\vee$ *mag* (*fromQ ta*) $\neq$ *mag* (*fromQ t*)
by (*simp add*: *magQ-def quant-eq-iff-mag-eq*)
then show *?thesis*
by (*metis* (*full-types*) *qequiv.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 $QD('u_1) = QD('u_2)$ $QD('u_1) = QD('u_2) \Longrightarrow [\![x]\!]_Q = [\![y]\!]_Q$
shows $x \cong_Q y$
using *assms quant-equiv-iff* **by** *blast*

lemma *quant-le-iff-magn-le* [*si-eq*]:

$x \leq y \longleftrightarrow [\![x]\!]_Q \leq [\![y]\!]_Q$
by (*auto simp add*: *magQ-def*; (*transfer*, *simp*))

lemma *quant-leI* [*si-transfer*]:
$[\![x]\!]_Q \leq [\![y]\!]_Q \Longrightarrow x \leq y$
by (*simp add*: *quant-le-iff-magn-le*)

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

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

lemma *magQ-zero* [*si-eq*]: $[\![\mathit{0}]\!]_Q = \mathit{0}$
by (*simp add*: *magQ-def*, *transfer*, *simp*)

lemma *magQ-one* [*si-eq*]: $[\![\mathit{1}]\!]_Q = \mathit{1}$
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*]: $[\![x *_Q y]\!]_Q = x * [\![y]\!]_Q$
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*]: $[\![x^{-\mathbf{1}}]\!]_Q = \mathit{inverse}\ [\![x]\!]_Q$
by (*simp add*: *magQ-def*, *transfer*, *simp*)

lemma *magQ-qdivivide* [*si-eq*]: $[\![(x{::}('a{::}\mathit{field})[\text{-},\text{-}])\ /\ y]\!]_Q = [\![x]\!]_Q\ /\ [\![y]\!]_Q$
by (*simp add*: *magQ-def*, *transfer*, *simp add*: *field-class.field-divide-inverse*)

lemma *magQ-numeral* [*si-eq*]: $[\![\mathit{numeral}\ n]\!]_Q = \mathit{numeral}\ n$
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*]:
fixes q :: $'a['d_1$::*dim-type*, $'s$::*unit-system*] **and** t :: $'d_2$::*dim-type itself*
assumes $QD('d_1) = QD('d_2)$
shows $[\![coerceQuantT\ t\ q]\!]_Q = [\![q]\!]_Q$
by (*simp add*: *coerceQuantT-def magQ-def assms*, *metis assms qequiv.rep-eq updown-eq-iff*)

lemma *dimQ* [*simp*]: $dimQ(x$ :: $'a['d$::*dim-type*, $'s$::*unit-system*]) $= QD('d)$
by (*simp add*: *dimQ-def*, *transfer*, *simp*)

The following tactic breaks an SI conjecture down to numeric and unit properties

method *si-simp* **uses** *add* =
(*rule-tac si-transfer*; *simp add*: *add si-eq field-simps*)

The next tactic additionally compiles the semantics of the underlying units

method *si-calc* **uses** *add* =
(*si-simp add*: *add*; *simp add*: *si-def add*)

lemma $QD(N \cdot \Theta \cdot N) = QD(\Theta \cdot N^2)$ **by** (*simp add*: *si-eq si-def*)

end

3.4 Algebraic Laws

theory *ISQ-Algebra*
imports *ISQ-Proof*
begin

3.4.1 Quantity Scale

lemma *scaleQ-add-right*: $a *_Q x + y = (a *_Q x) + (a *_Q y)$
by (*si-simp add*: *distrib-left*)

lemma *scaleQ-add-left*: $a + b *_Q x = (a *_Q x) + (b *_Q x)$
by (*si-simp add*: *distrib-right*)

lemma *scaleQ-scaleQ* [*simp*]: $a *_Q b *_Q x = a \cdot b *_Q x$
by *si-simp*

lemma *scaleQ-one* [*simp*]: $1 *_Q x = x$
by *si-simp*

lemma *scaleQ-zero* [*simp*]: $0 *_Q x = 0$
by *si-simp*

lemma *scaleQ-inv*: $-a *_Q x = a *_Q -x$
by *si-calc*

lemma *scaleQ-as-qprod*: $a *_Q x \cong_Q (a *_Q \mathbf{1}) \cdot x$
by *si-simp*

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

lemma *mult-scaleQ-right* [*simp*]: $x \cdot (a *_Q y) = a *_Q x \cdot y$
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*: $\mathbf{1} \cdot x \cong_Q x$
by (*si-calc*)

lemma *qtimes-right-unit*: $x \cdot \mathbf{1} \cong_Q x$
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 $x \cdot z \cong_Q y \cdot z$
using *assms* **by** *si-simp*

lemma *qtimes-weak-cong-right*:
assumes $x \cong_Q y$
shows $z \cdot x \cong_Q z \cdot y$
using *assms* **by** *si-calc*

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

lemma *scaleQ-cong*:
assumes $y \cong_Q z$
shows $x *_Q y \cong_Q x *_Q z$
using *assms* **by** *si-calc*

lemma *qinverse-qinverse*: $x^{-\mathbf{1}-\mathbf{1}} \cong_Q x$
by *si-calc*

lemma *qinverse-nonzero-iff-nonzero*: $x^{-\mathbf{1}} = 0 \longleftrightarrow x = 0$
  by (*auto*, *si-calc+*)

lemma *qinverse-qtimes*: $(x \cdot y)^{-\mathbf{1}} \cong_Q x^{-\mathbf{1}} \cdot y^{-\mathbf{1}}$
  by (*si-simp add*: *inverse-distrib*)

lemma *qinverse-qdivide*: $(x \mathbin{/} y)^{-\mathbf{1}} \cong_Q y \mathbin{/} x$
  by *si-simp*

lemma *qtimes-cancel*: $x \neq 0 \Longrightarrow x \mathbin{/} x \cong_Q \mathbf{1}$
  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* :: *'u itself* ⇒ *'s itself* ⇒ (*'a::one*)[*'u::basedim-type*, *'s::unit-system*]

  where *mk-base-unit t s* = *1*

syntax *-mk-base-unit* :: *type* ⇒ *type* ⇒ *logic* (*BUNIT'*(-, -'))
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

3.6.1 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 ($'s_1$::*unit-system*, $'s_2$::*unit-system*) *Conversion* ((-/ $\Rightarrow_U$ -) [*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* :: ($'s_1$::*unit-system* $\Rightarrow_U$ $'s_2$::*unit-system*) ⇒ *rat* **is** *cLengthF* **.**
lift-definition *MassF* :: ($'s_1$::*unit-system* $\Rightarrow_U$ $'s_2$::*unit-system*) ⇒ *rat* **is** *cMassF* **.**
lift-definition *TimeF* :: ($'s_1$::*unit-system* $\Rightarrow_U$ $'s_2$::*unit-system*) ⇒ *rat* **is** *cTimeF* **.**
lift-definition *CurrentF* :: ($'s_1$::*unit-system* $\Rightarrow_U$ $'s_2$::*unit-system*) ⇒ *rat* **is** *cCurrentF* **.**
lift-definition *TemperatureF* :: ($'s_1$::*unit-system* $\Rightarrow_U$ $'s_2$::*unit-system*) ⇒ *rat* **is** *cTemperatureF* **.**
lift-definition *AmountF* :: ($'s_1$::*unit-system* $\Rightarrow_U$ $'s_2$::*unit-system*) ⇒ *rat* **is** *cAmountF* **.**
lift-definition *IntensityF* :: ($'s_1$::*unit-system* $\Rightarrow_U$ $'s_2$::*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* $\Rightarrow_U$ $'s$ (id_C)

is
(| *cLengthF* = *1*
, *cMassF* = *1*
, *cTimeF* = *1*
, *cCurrentF* = *1*
, *cTemperatureF* = *1*
, *cAmountF* = *1*
, *cIntensityF* = *1* |) **by** *simp*

lift-definition *convcomp* ::
$('s_2 \Rightarrow_U 's_3$::*unit-system*$) \Rightarrow ('s_1$::*unit-system* $\Rightarrow_U 's_2$::*unit-system*$) \Rightarrow ('s_1 \Rightarrow_U 's_3)$ (**infixl** $\circ_C$ *55*) **is**
λ c_1 c_2. (| *cLengthF* = *cLengthF* c_1 ∗ *cLengthF* c_2, *cMassF* = *cMassF* c_1 ∗ *cMassF* c_2
, *cTimeF* = *cTimeF* c_1 ∗ *cTimeF* c_2, *cCurrentF* = *cCurrentF* c_1 ∗ *cCurrentF* c_2
, *cTemperatureF* = *cTemperatureF* c_1 ∗ *cTemperatureF* c_2
, *cAmountF* = *cAmountF* c_1 ∗ *cAmountF* c_2, *cIntensityF* = *cIntensityF* c_1 ∗ *cIntensityF* c_2 |)
by *simp*

lift-definition *convinv* :: $('s_1$::*unit-system* $\Rightarrow_U 's_2$::*unit-system*$) \Rightarrow ('s_2 \Rightarrow_U 's_1)$ (inv_C) **is**
λ *c*. (| *cLengthF* = *inverse* (*cLengthF c*), *cMassF* = *inverse* (*cMassF c*), *cTimeF* = *inverse* (*cTimeF c*)
, *cCurrentF* = *inverse* (*cCurrentF c*), *cTemperatureF* = *inverse* (*cTemperatureF c*)
, *cAmountF* = *inverse* (*cAmountF c*), *cIntensityF* = *inverse* (*cIntensityF c*) |) **by** *simp*

lemma *convinv-inverse* [*simp*]: *convinv* (*convinv c*) = *c*
by (*transfer*, *simp*)

lemma *convcomp-inv* [*simp*]: $c \circ_C inv_C\ c = id_C$
by (*transfer*, *simp*)

lemma *inv-convcomp* [*simp*]: $inv_C\ c \circ_C c = id_C$
by (*transfer*, *simp*)

lemma *Conversion-invs* [*simp*]: *LengthF* (inv_C *x*) = *inverse* (*LengthF x*) *MassF* (inv_C *x*) = *inverse* (*MassF x*)
TimeF (inv_C *x*) = *inverse* (*TimeF x*) *CurrentF* (inv_C *x*) = *inverse* (*CurrentF x*)
TemperatureF (inv_C *x*) = *inverse* (*TemperatureF x*) *AmountF* (inv_C *x*) = *inverse* (*AmountF x*)
IntensityF (inv_C *x*) = *inverse* (*IntensityF x*)
by (*transfer*, *simp*)+

lemma *Conversion-comps* [*simp*]: *LengthF* ($c_1 \circ_C c_2$) = *LengthF* c_1 ∗ *LengthF* c_2

$MassF\ (c_1 \circ_C c_2) = MassF\ c_1 * MassF\ c_2$
$TimeF\ (c_1 \circ_C c_2) = TimeF\ c_1 * TimeF\ c_2$
$CurrentF\ (c_1 \circ_C c_2) = CurrentF\ c_1 * CurrentF\ c_2$
$TemperatureF\ (c_1 \circ_C c_2) = TemperatureF\ c_1 * TemperatureF\ c_2$
$AmountF\ (c_1 \circ_C c_2) = AmountF\ c_1 * AmountF\ c_2$
$IntensityF\ (c_1 \circ_C c_2) = IntensityF\ c_1 * IntensityF\ c_2$
by (*transfer*, *simp*)+

3.6.3 Conversion Functions

definition *dconvfactor* :: ($'s_1$::*unit-system* $\Rightarrow_U$ $'s_2$::*unit-system*) $\Rightarrow$ *Dimension* $\Rightarrow$ *rat* **where**
dconvfactor c d =
LengthF c $\hat{}_Z$ *dim-nth d Length*
$*$ *MassF c* $\hat{}_Z$ *dim-nth d Mass*
$*$ *TimeF c* $\hat{}_Z$ *dim-nth d Time*
$*$ *CurrentF c* $\hat{}_Z$ *dim-nth d Current*
$*$ *TemperatureF c* $\hat{}_Z$ *dim-nth d Temperature*
$*$ *AmountF c* $\hat{}_Z$ *dim-nth d Amount*
$*$ *IntensityF c* $\hat{}_Z$ *dim-nth d Intensity*

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

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

lemma *dconvfactor-convinv*: *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*:
dconvfactor ($c_1 \circ_C c_2$) *d* = *dconvfactor* c_1 *d* $*$ *dconvfactor* c_2 *d*
by (*simp add*: *dconvfactor-def*, *transfer*, *simp add*: *mult-ac intpow-mult-distrib*)

lemma *dconvfactor-inverse*:
dconvfactor c (*inverse d*) = *inverse* (*dconvfactor c d*)
by (*simp add*: *dconvfactor-def inverse-dimvec-def intpow-uminus*)

lemma *dconvfactor-times*:
dconvfactor c ($x \cdot y$) = *dconvfactor c x* $\cdot$ *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* $\Rightarrow$ ($'a$::*field-char-0*)[$'d$::*dim-type*, $'s_1$::*unit-system*] $\Rightarrow$ $'a$[$'d$, $'s_2$::*unit-system*]
is λ *c q*. (| *mag* = *of-rat* (*dconvfactor c* (*dim q*)) $*$ *mag q*, *dim* = *dim q*, *unit-sys*

$= \mathit{unit}$ $\rparen$ **by** *simp*

lemma *magQ-qconv*: $[\![\mathit{qconv}\ c\ q]\!]_Q = \mathit{of\text{-}rat}\ (\mathit{dconvfactor}\ c\ (\mathit{dimQ}\ q)) * [\![q]\!]_Q$
by (*simp add*: *si-def*, *transfer*, *simp*)

lemma *qconv-id* [*simp*]: $\mathit{qconv}\ \mathit{id}_C\ x = x$
by (*transfer'*, *simp add*: *Measurement-System-eq-intro*)

lemma *qconv-comp*: $\mathit{qconv}\ (c_1 \circ_C c_2)\ x = \mathit{qconv}\ c_1\ (\mathit{qconv}\ c_2\ x)$
by (*transfer*, *simp add*: *dconvfactor-compose of-rat-mult*)

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

lemma *qconv-scaleQ* [*simp*]: $\mathit{qconv}\ c\ (d *_Q x) = d *_Q \mathit{qconv}\ c\ x$
by (*transfer*, *simp*)

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

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

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

lemma *qconv-qinverse* [*simp*]: $\mathit{qconv}\ c\ (x^{-\mathbf{1}}) = (\mathit{qconv}\ c\ x)^{-\mathbf{1}}$
by (*transfer*, *simp add*: *inverse-Quantity-ext-def inverse-Measurement-System-ext-def dconvfactor-inverse of-rat-inverse*)

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

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

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

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

lemma *qconv-Temperature* [*simp*]: $\mathit{qconv}\ c\ \mathit{BUNIT}(\Theta,\ \text{-}) = \mathit{TemperatureF}\ c *_Q \mathit{BUNIT}(\Theta,\ \text{-})$
by (*simp add*: *dconvfactor-def magQ-qconv si-eq mk-BaseDim-def one-dimvec-def*)

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

lemma *qconv-Intensity* [*simp*]: *qconv c* $BUNIT(J, \text{-})$ = *IntensityF c* $*_Q$ $BUNIT(J, \text{-})$
 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

theory *SI-Units*
 imports *ISQ*
begin

An SI unit is simply a particular kind of quantity with an SI tag.

typedef *SI* = *UNIV* :: *unit set* **by** *simp*

instance *SI* :: *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* ≡ *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 *1*::$'a$ elements of the base quantities.

abbreviation *metre* ≡ *BUNIT*(*L*, *SI*)
abbreviation *kilogram* ≡ *BUNIT*(*M*, *SI*)
abbreviation *ampere* ≡ *BUNIT*(*I*, *SI*)
abbreviation *kelvin* ≡ *BUNIT*(Θ, *SI*)
abbreviation *mole* ≡ *BUNIT*(*N*, *SI*)
abbreviation *candela* ≡ *BUNIT*(*J*, *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 *time-second* = *unit-system*

instance *SI* :: *time-second* **..**

abbreviation *second* $\equiv$ *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 $(metre \cdot second^{-\mathbf{1}}) \cdot second \cong_Q metre$
by (*si-calc*)

4.1.2 Metrification

class *metrifiable* = *unit-system* +
fixes *convschema* :: *'a itself* $\Rightarrow$ (*'a*, *SI*) *Conversion* ($schema_C$)

instantiation *SI* :: *metrifiable*
begin
lift-definition *convschema-SI* :: *SI itself* $\Rightarrow$ (*SI*, *SI*) *Conversion*
is λ *s*.
(| *cLengthF* = *1*
, *cMassF* = *1*
, *cTimeF* = *1*
, *cCurrentF* = *1*
, *cTemperatureF* = *1*
, *cAmountF* = *1*

, *cIntensityF* = *1* ⦈ **by** *simp*
instance ..
end

abbreviation *metrify* :: ($'a$::*field-char-0*)[$'d$::*dim-type*, $'s$::*metrifiable*] ⇒ $'a$[$'d$::*dim-type*, *SI*] **where**
metrify ≡ *qconv* (*convschema* (*TYPE*($'s$)))

Conversion via SI units

abbreviation *qmconv* ::
$'s_1$ *itself* ⇒ $'s_2$ *itself*
⇒ ($'a$::*field-char-0*)[$'d$::*dim-type*, $'s_1$::*metrifiable*]
⇒ $'a$[$'d$::*dim-type*, $'s_2$::*metrifiable*] **where**
qmconv s_1 s_2 x ≡ *qconv* (inv_C ($schema_C$ s_2) $\circ_C$ $schema_C$ s_1) x

syntax
-*qmconv* :: *type* ⇒ *type* ⇒ *logic* (*QMC*'(- → -'))

translations
QMC($'s_1$ → $'s_2$) == *CONST qmconv TYPE*($'s_1$) *TYPE*($'s_2$)

lemma *qmconv-self*: *QMC*($'s$::*metrifiable* → $'s$) = *id*
by (*simp add*: *fun-eq-iff*)

end

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**
λ *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* ≡ $second^{-\mathbf{1}}$

abbreviation *radian* ≡ *metre* · $metre^{-\mathbf{1}}$

abbreviation *steradian* ≡ $metre^{\mathbf{2}}$ · $metre^{-\mathbf{2}}$

abbreviation *joule* ≡ *kilogram* · $metre^{\mathbf{2}}$ · $second^{-\mathbf{2}}$

type-synonym *'a joule* = *'a*[$M \cdot L^2 \cdot T^{-2}$, *SI*]

abbreviation *watt* ≡ *kilogram* · $metre^{\mathbf{2}}$ · $second^{-\mathbf{3}}$

type-synonym *'a watt* = *'a*[$M \cdot L^2 \cdot T^{-3}$, *SI*]

abbreviation *coulomb* ≡ *ampere* · *second*

type-synonym *'a coulomb* = *'a*[$I \cdot T$, *SI*]

abbreviation *lumen* $\equiv$ *candela* $\cdot$ *steradian*

type-synonym $'a$ *lumen* $= 'a[J \cdot (L^2 \cdot L^{-2}),\ SI]$

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*:: $'a[T^{-1},SI]$ (Δv_{Cs}) **where**
caesium-frequency $\equiv$ *9192631770* $*_Q$ *hertz*

Speed of light in vacuum

abbreviation *speed-of-light* :: $'a[L \cdot T^{-1},SI]$ (**c**) **where**
speed-of-light $\equiv$ *299792458* $*_Q$ (*metre*$\cdot$*second*$^{-\mathbf{1}}$)

Planck constant

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

Elementary charge

abbreviation *elementary-charge* :: $'a[I \cdot T,SI]$ (**e**) **where**
elementary-charge $\equiv$ (*1.602176634* $\cdot$ *1/(10^19)*) $*_Q$ *coulomb*

The Boltzmann constant

abbreviation *Boltzmann* :: $'a[M \cdot L^2 \cdot T^{-2} \cdot \Theta^{-1},SI]$ (**k**) **where**
Boltzmann $\equiv$ (*1.380649*$\cdot$*1/(10^23)*) $*_Q$ (*joule / kelvin*)

The Avogadro number

abbreviation *Avogadro* :: $'a[N^{-1},SI]$ (N_A) **where**
Avogadro $\equiv$ *6.02214076*$\cdot$*(10^23)* $*_Q$ (*mole*$^{-\mathbf{1}}$)

abbreviation *max-luminous-frequency* :: $'a[T^{-1},SI]$ **where**
max-luminous-frequency $\equiv$ (*540*$\cdot$*10^12*) $*_Q$ *hertz*

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

abbreviation *luminous-efficacy* :: $'a[J \cdot (L^2 \cdot L^{-2}) \cdot (M \cdot L^2 \cdot T^{-3})^{-1},SI]$ (K_{cd})
where
luminous-efficacy $\equiv$ *683* $*_Q$ (*lumen/watt*)

4.3.3 Checking Foundational Equations of the SI System

theorem *second-definition*:
1 $*_Q$ *second* $\cong_Q$ (*9192631770* $*_Q$ **1**) / Δv_{Cs}
by *si-calc*

theorem *metre-definition*:
  $1 *_Q metre \cong_Q (\mathbf{c} \mathbin{/} (299792458 *_Q \mathbf{1})) \cdot second$
  $1 *_Q metre \cong_Q (9192631770 \mathbin{/} 299792458) *_Q (\mathbf{c} \mathbin{/} \Delta v_{Cs})$
  by *si-calc+*

theorem *kilogram-definition*:
  $((1 *_Q kilogram)::'a\ kilogram) \cong_Q (\mathbf{h} \mathbin{/} (6.62607015 \cdot 1/(10\hat{\ }34) *_Q \mathbf{1})) \cdot metre^{-\mathbf{2}} \cdot second$

  by *si-calc*

abbreviation *approx-ice-point* $\equiv 273.15 *_Q kelvin$

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 ($*_Q$).

default-sort *ring-char-0*

definition *deca* :: *'a* **where** [*si-eq*]: $deca = 10\hat{\ }1$

definition *hecto* :: *'a* **where** [*si-eq*]: $hecto = 10\hat{\ }2$

definition *kilo* :: *'a* **where** [*si-eq*]: $kilo = 10\hat{\ }3$

definition *mega* :: *'a* **where** [*si-eq*]: $mega = 10\hat{\ }6$

definition *giga* :: *'a* **where** [*si-eq*]: $giga = 10\hat{\ }9$

definition *tera* :: *'a* **where** [*si-eq*]: $tera = 10\hat{\ }12$

definition *peta* :: *'a* **where** [*si-eq*]: $peta = 10\hat{\ }15$

definition *exa* :: *'a* **where** [*si-eq*]: $exa = 10\hat{\ }18$

definition *zetta* :: *'a* **where** [*si-eq*]: $zetta = 10\hat{\ }21$

definition *yotta* :: *'a* **where** [*si-eq*]: *yotta* = *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 $2.3 *_Q (centi *_Q metre)^{\mathbf{3}} = 2.3 \cdot 1/10\hat{}6 *_Q metre^{\mathbf{3}}$
by (*si-simp*)

lemma $1 *_Q (centi *_Q metre)^{-\mathbf{1}} = 100 *_Q metre^{-\mathbf{1}}$
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*

definition *zebi* :: *'a* **where** [*si-eq*]: *zebi* = *2^70*

definition *yobi* :: *'a* **where** [*si-eq*]: *yobi* = *2^80*

default-sort *type*

end

4.5 Derived SI-Units

theory *SI-Derived*
imports *SI-Prefix*
begin

4.5.1 Definitions

abbreviation *newton* $\equiv$ *kilogram* $\cdot$ *metre* $\cdot$ *second*$^{-\mathbf{2}}$

type-synonym *'a newton* = *'a*$[M \cdot L \cdot T^{-2}, SI]$

abbreviation *pascal* $\equiv$ *kilogram* $\cdot$ *metre*$^{-\mathbf{1}}$ $\cdot$ *second*$^{-\mathbf{2}}$

type-synonym *'a pascal* = *'a*$[M \cdot L^{-1} \cdot T^{-2}, SI]$

abbreviation *volt* $\equiv$ *kilogram* $\cdot$ *metre*$^{\mathbf{2}}$ $\cdot$ *second*$^{-\mathbf{3}}$ $\cdot$ *ampere*$^{-\mathbf{1}}$

type-synonym *'a volt* = *'a*$[M \cdot L^2 \cdot T^{-3} \cdot I^{-1}, SI]$

abbreviation *farad* $\equiv$ *kilogram*$^{-\mathbf{1}}$ $\cdot$ *metre*$^{-\mathbf{2}}$ $\cdot$ *second*$^{\mathbf{4}}$ $\cdot$ *ampere*$^{\mathbf{2}}$

type-synonym *'a farad* = *'a*$[M^{-1} \cdot L^{-2} \cdot T^4 \cdot I^2, SI]$

abbreviation *ohm* $\equiv$ *kilogram* $\cdot$ *metre*$^{\mathbf{2}}$ $\cdot$ *second*$^{-\mathbf{3}}$ $\cdot$ *ampere*$^{-\mathbf{2}}$

type-synonym *'a ohm* = *'a*$[M \cdot L^2 \cdot T^{-3} \cdot I^{-2}, SI]$

abbreviation *siemens* $\equiv$ *kilogram*$^{-\mathbf{1}}$ $\cdot$ *metre*$^{-\mathbf{2}}$ $\cdot$ *second*$^{\mathbf{3}}$ $\cdot$ *ampere*$^{\mathbf{2}}$

abbreviation *weber* $\equiv$ *kilogram* $\cdot$ *metre*$^{\mathbf{2}}$ $\cdot$ *second*$^{-\mathbf{2}}$ $\cdot$ *ampere*$^{-\mathbf{1}}$

abbreviation *tesla* $\equiv$ *kilogram* $\cdot$ *second*$^{-\mathbf{2}}$ $\cdot$ *ampere*$^{-\mathbf{1}}$

abbreviation *henry* $\equiv$ *kilogram* $\cdot$ *metre*$^{\mathbf{2}}$ $\cdot$ *second*$^{-\mathbf{2}}$ $\cdot$ *ampere*$^{-\mathbf{2}}$

abbreviation *lux* $\equiv$ *candela* $\cdot$ *steradian* $\cdot$ *metre*$^{-\mathbf{2}}$

abbreviation (*input*) *becquerel* $\equiv$ *second*$^{-\mathbf{1}}$

abbreviation *gray* $\equiv$ *metre*$^{\mathbf{2}}$ $\cdot$ *second*$^{-\mathbf{2}}$

abbreviation *sievert* $\equiv$ *metre*$^{\mathbf{2}}$ $\cdot$ *second*$^{-\mathbf{2}}$

abbreviation *katal* $\equiv$ *mole* $\cdot$ *second*$^{-\mathbf{1}}$

definition *degrees-celcius* :: $'a$::*field-char-0* $\Rightarrow$ $'a[\Theta]$ (*-XXXC* [*999*] *999*)
where [*si-eq*]: *degrees-celcius* x = ($x *_Q$ *kelvin*) + *approx-ice-point*

definition [*si-eq*]: *gram* = *milli* $*_Q$ *kilogram*

4.5.2 Equivalences

lemma *joule-alt-def*: *joule* $\cong_Q$ *newton* $\cdot$ *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* $\cdot$ *second*
by *si-calc*

lemma *tesla-alt-def*: *tesla* $\cong_Q$ *weber* / *metre*$^{\mathbf{2}}$
by *si-calc*

lemma *henry-alt-def*: *henry* $\cong_Q$ *weber* / *ampere*
by *si-calc*

lemma *lux-alt-def*: *lux* = *lumen* / *metre*$^{\mathbf{2}}$
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 *celcius-to-kelvin*: $TXXXC = (T *_Q kelvin) + (273.15 *_Q kelvin)$
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*]: $astronomical\text{-}unit = 149597870700 *_Q metre$

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

abbreviation *degrees* (-XXX [999] 999) **where** $nXXX \equiv n *_Q degree$

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

definition [*si-def*, *si-eq*]: $tonne = 10\hat{}3 *_Q kilogram$

definition [*si-def*, *si-eq*]: $dalton = 1.66053906660 * (1 / 10\hat{}27) *_Q kilogram$

4.6.1 Example Unit Equations

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

lemma $watt \cdot hour \cong_Q 3600 *_Q joule$ **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

theory *SI-Imperial*
imports *SI-Accepted*
begin

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*]: $nautical\text{-}mile = 1852 *_Q metre$

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* ⇒ *'a*[Θ] (*-XXXF* [*999*] *999*)
where [*si-eq*]: *degrees-farenheit x* = (*x* + *459.67*)·*5*/*9* $*_Q$ *kelvin*

default-sort *type*

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*) · *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**

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* $\cong_Q$ **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* $*_Q$ *astronomical-unit*

We calculate some conservative bounds on the parsec: it is somewhere between 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}[$L \cdot T^{-2}$])

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* ($(\text{-})^2$ [*999*] *999*)
-*si-cube* :: *si* ⇒ *si* ($(\text{-})^3$ [*999*] *999*)
-*si-quart* :: *si* ⇒ *si* ($(\text{-})^4$ [*999*] *999*)

-*si-inverse* :: *si* ⇒ *si* ((-^{-1}) [*999*] *999*)
-*si-invsquare* :: *si* ⇒ *si* ($(\text{-})^{-2}$ [*999*] *999*)
-*si-invcube* :: *si* ⇒ *si* ($(\text{-})^{-3}$ [*999*] *999*)
-*si-invquart* :: *si* ⇒ *si* ($(\text{-})^{-4}$ [*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* (*SIPRINT*′(-′))

translations
(*type*) ′*a*[*SIPRINT*(′*d*)] == (*type*) ′*a*[′*d*, *SI*]
(*si*) $SIPRINT('d)^2$ == (*si*) $SIPRINT('d^2)$
(*si*) $SIPRINT('d)^3$ == (*si*) $SIPRINT('d^3)$
(*si*) $SIPRINT('d)^4$ == (*si*) $SIPRINT('d^4)$
(*si*) $SIPRINT('d)^{-1}$ == (*si*) $SIPRINT('d^{-1})$
(*si*) $SIPRINT('d)^{-2}$ == (*si*) $SIPRINT('d^{-2})$
(*si*) $SIPRINT('d)^{-3}$ == (*si*) $SIPRINT('d^{-3})$
(*si*) $SIPRINT('d)^{-4}$ == (*si*) $SIPRINT('d^{-4})$
(*si*) $SIPRINT('d_1) \cdot SIPRINT('d_2)$ == (*si*) $SIPRINT('d_1 \cdot 'd_2)$
(*si*) *m* == (*si*) *SIPRINT*(*L*)
(*si*) *kg* == (*si*) *SIPRINT*(*M*)
(*si*) *s* == (*si*) *SIPRINT*(*T*)
(*si*) *A* == (*si*) *SIPRINT*(*I*)
(*si*) *K* == (*si*) *SIPRINT*(Θ)
(*si*) *mol* == (*si*) *SIPRINT*(*N*)
(*si*) *cd* == (*si*) *SIPRINT*(*J*)

-*si-invsquare* *x* <= -*si-inverse* (-*si-square* *x*)
-*si-invcube* *x* <= -*si-inverse* (-*si-cube* *x*)
-*si-invquart* *x* <= -*si-inverse* (-*si-quart* *x*)

-*si-invsquare* *x* <= -*si-square* (-*si-inverse* *x*)
-*si-invcube* *x* <= -*si-cube* (-*si-inverse* *x*)

-si-invquart x <= -si-quart (-si-inverse x)

typ $real[m{\cdot}s^{-2}]$
typ $real[m{\cdot}s^{-2}{\cdot}A^2]$
term $5 *_Q joule$

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.

definition $SIQ\ n\ u = n *_Q u$

syntax
-si-term :: $si \Rightarrow logic$ (SI'('-'))
-siq-term :: $logic \Rightarrow si \Rightarrow logic$ (SI[-, -])
-siq-print :: $logic \Rightarrow si$

translations
-siq-term n u => CONST SIQ n (-si-term u)
-siq-term n (-siq-print u) <= CONST SIQ n u
-si-term (-si-times x y) == (-si-term x) · (-si-term y)
-si-term (-si-inverse x) == (-si-term x)$^{\mathbf{-1}}$
-si-term (-si-square x) == (-si-term x)$^{\mathbf{2}}$
-si-term (-si-cube x) == (-si-term x)$^{\mathbf{2}}$
SI(m) => CONST metre
SI(kg) => CONST kilogram
SI(s) => CONST second
SI(A) => CONST ampere
SI(K) => CONST kelvin
SI(mol) => CONST mole
SI(cd) => CONST candela

-si-inverse (-siq-print x) <= -siq-print ($x^{\mathbf{-1}}$)
-si-invsquare (-siq-print x) <= -siq-print ($x^{\mathbf{-2}}$)
-si-invcube (-siq-print x) <= -siq-print ($x^{\mathbf{-3}}$)
-si-invquart (-siq-print x) <= -siq-print ($x^{\mathbf{-4}}$)

-si-square (-siq-print x) <= -siq-print ($x^{\mathbf{2}}$)
-si-cube (-siq-print x) <= -siq-print ($x^{\mathbf{3}}$)
-si-quart (-siq-print x) <= -siq-print ($x^{\mathbf{4}}$)
-si-times (-siq-print x) (-siq-print y) <= -siq-print (x · y)

-si-metre <= -siq-print (CONST metre)
-si-kilogram <= -siq-print (CONST kilogram)
-si-second <= -siq-print (CONST second)
-si-ampere <= -siq-print (CONST ampere)
-si-kelvin <= -siq-print (CONST kelvin)
-si-mole <= -siq-print (CONST mole)

-si-candela <= *-siq-print* (*CONST candela*)

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

end

4.11 British Imperial System (1824/1897)

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

4.11.2 Base Units

abbreviation *yard* ≡ *BUNIT*(*L*, *BIS*)
abbreviation *pound* ≡ *BUNIT*(*M*, *BIS*)
abbreviation *rankine* ≡ *BUNIT*(Θ, *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* $*_Q$ *yard*

definition [*si-eq*]: *inch* = *1/12* $*_Q$ *foot*

definition [*si-eq*]: *furlong* = *220* $*_Q$ *yard*

definition [*si-eq*]: *mile* = *1760* $*_Q$ *yard*

definition [*si-eq*]: *acre* = *4840* $*_Q$ *yard*$^{\mathbf{3}}$

definition [*si-eq*]: *ounce* = *1/12* $*_Q$ *pound*

definition [*si-eq*]: $gallon = 277.421 *_Q inch^{\mathbf{3}}$

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**
λ *x*. ⦇ *cLengthF* = *0.9143993*, *cMassF* = *0.453592338*, *cTimeF* = *1*
, *cCurrentF* = *1*, *cTemperatureF* = *5/9*, *cAmountF* = *1*, *cIntensityF* = *1* ⦈
by *simp*

instance ..
end

lemma *BIS-SI-simps* [*simp*]: *LengthF* (*convschema* (*a*::*BIS itself*)) = *0.9143993*
MassF (*convschema a*) = *0.453592338*
TimeF (*convschema a*) = *1*
CurrentF (*convschema a*) = *1*
TemperatureF (*convschema a*) = *5/9*
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

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

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

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

[4] I. J. Hayes and B. P. Mahony. Using units of measurement in formal specifications. *Formal Aspects of Computing*, 7(3):329–347, 1995.

[5] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283. 2002.