

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

Simon Foster Burkhart Wolff

May 26, 2024

Abstract

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

Contents

1	ISQ and SI: An Introduction	7
2	Preliminaries	11
2.1	Integer Powers	11
2.2	Enumeration Extras	12
2.2.1	First Index Function	12
2.2.2	Enumeration Indices	13
2.3	Multiplication Groups	14
3	International System of Quantities	17
3.1	Quantity Dimensions	17
3.1.1	Preliminaries	17
3.1.2	Dimension Vectors	18
3.1.3	Code Generation	19
3.1.4	Dimension Semantic Domain	21
3.1.5	Dimension Type Expressions	23
3.1.6	ML Functions	27
3.2	Quantities	28
3.2.1	Quantity Semantic Domain	28
3.2.2	Measurement Systems	31
3.2.3	Dimension Typed Quantities	34
3.2.4	Predicates on Typed Quantities	35
3.2.5	Operators on Typed Quantities	36
3.3	Proof Support for Quantities	40
3.4	Algebraic Laws	42
3.4.1	Quantity Scale	42
3.4.2	Field Laws	43
3.5	Units	44
3.6	Conversion Between Unit Systems	44
3.6.1	Conversion Schemas	44
3.6.2	Conversion Algebra	45
3.6.3	Conversion Functions	47
3.7	Meta-Theory for ISQ	49

4	International System of Units	51
4.1	SI Units Semantics	51
4.1.1	Example Unit Equations	52
4.1.2	Metrification	52
4.2	Centimetre-Gram-Second System	53
4.2.1	Preliminaries	53
4.2.2	Base Units	53
4.2.3	Conversion to SI	53
4.2.4	Conversion Examples	54
4.3	Physical Constants	54
4.3.1	Core Derived Units	54
4.3.2	Constants	55
4.3.3	Checking Foundational Equations of the SI System	55
4.4	SI Prefixes	56
4.4.1	Definitions	56
4.4.2	Examples	57
4.4.3	Binary Prefixes	57
4.5	Derived SI-Units	58
4.5.1	Definitions	58
4.5.2	Equivalences	59
4.5.3	Properties	60
4.6	Non-SI Units Accepted for SI use	60
4.6.1	Example Unit Equations	60
4.7	Imperial Units via SI Units	61
4.7.1	Units of Length	61
4.7.2	Units of Mass	61
4.7.3	Other Units	62
4.7.4	Unit Equations	62
4.8	Meta-Theory for SI Units	62
4.9	Astronomical Constants	62
4.10	Parsing and Pretty Printing of SI Units	63
4.10.1	Syntactic SI Units	63
4.10.2	Type Notation	64
4.10.3	Value Notations	65
4.11	British Imperial System (1824/1897)	66
4.11.1	Preliminaries	66
4.11.2	Base Units	66
4.11.3	Derived Units	66
4.11.4	Conversion to SI	67
4.11.5	Conversion Examples	67

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 Métrologie* (VIM) [2]. The VIM actually defines two systems: the *International System of Quantities* (ISQ) and the *International System of Units* (SI, abbreviated from the French *Système international (dunités)*). The latter is also documented in the *SI Brochure* [3], a standard that is updated periodically, most recently in 2019. Finally, the VIM defines concrete reference measurement procedures as well as a terminology for measurement errors.

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

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

tities and units are represented in a way that they have a *quantity type* as well as a *unit type* based on its base vectors and their magnitudes. Since the algebra of quantities induces congruences on quantity and SI types, specific tactic support has been developed to capture these. Our construction is validated by a test-set of known equivalences between both quantities and SI units. Moreover, the presented theory can be used for type-safe conversions between the SI system and others, like the British Imperial System (BIS).

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

1. *dimensions* represented by a type $(int, 'd::enum) dimvec$, 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 such as *metre*, *kilogram*, *second*, *ampere*, *kelvin*, *mole* and *candela* (together with procedures how to measure a metre, for example, which are defined in accompanying standards);
2. a standardised set of symbols for units such as *m*, *kg*, *s*, *A*, *K*, *mol*, and *cd*;
3. a standardised set of symbols of SI prefixes for multiples of SI units, such as *giga* ($= 10^9$), *kilo* ($= 10^3$), *milli* ($= 10^{-3}$), etc.; and a set of
4. *unit equations* and conversion equations such as $J = kg\,m^2/s^2$ 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:*

$$1 *_Q \text{ metre} \cong_Q (\mathbf{c} / (299792458 *_Q \mathbf{1})) \cdot \text{second}$$

by *si-calc*

theorem *kilogram-definition:*

$$1 *_Q \text{ kilogram} \cong_Q (\mathbf{h} / (6.62607015 \cdot 1 / (10^{34}) *_Q \mathbf{1})) \cdot \text{metre}^{-2} \cdot \text{second}$$

by *si-calc*

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

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

Chapter 2

Preliminaries

2.1 Integer Powers

```
theory Power-int
  imports HOL.Real
begin
```

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

```
definition intpow :: 'a::{linordered-field} => int => 'a (infixr ^_Z 80) where
  intpow x n = (if (n < 0) then inverse (x ^ nat (-n)) else (x ^ nat n))
```

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

```
lemma intpow-spos [simp]: x > 0 ==> x ^_Z n > 0
  by (simp add: intpow-def)
```

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

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

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

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

```
lemma intpow-pos [simp]: n >= 0 ==> x ^_Z n = x ^ nat n
```

by (simp add: intpow-def)

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

lemma *intpow-uminus-nat*: $n \geq 0 \implies x \hat{=}_{\mathbb{Z}} -n = \text{inverse } (x \hat{=}_{\mathbb{Z}} \text{nat } n)$
 by (simp add: intpow-def)

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

lemma *intpow-mult-distrib*: $(x * y) \hat{=}_{\mathbb{Z}} m = x \hat{=}_{\mathbb{Z}} m * y \hat{=}_{\mathbb{Z}} m$
 by (simp add: intpow-def power-mult-distrib)

end

2.2 Enumeration Extras

theory *Enum-extra*
imports *HOL-Library.Code-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*:
 $x \in \text{set}(xs) \implies \text{first-ind } xs \ x \ i < \text{length}(xs) + i$
 by (induct xs arbitrary: i, auto, metis add-Suc-right)

lemma *nth-first-ind*:
 $\llbracket \text{distinct } xs; x \in \text{set}(xs) \rrbracket \implies xs ! (\text{first-ind } xs \ x \ i - i) = x$
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*:
 $\llbracket \text{distinct } xs; i < \text{length } xs \rrbracket \implies \text{first-ind } xs \ (xs ! i) \ j = i + j$
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* \Rightarrow *logic* (*ENUM*'(-'))

translations

ENUM('a') \Rightarrow *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 \Rightarrow *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 ($\forall i < \text{length } \text{ENUM}('a). \forall j < \text{length } \text{ENUM}('a). i \neq j \longrightarrow \text{ENUM}('a) ! i \neq \text{ENUM}('a) ! j$)

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) \wedge *Enum.enum* ! *i* = *x*)

proof (*rule sym, rule the-equality, safe*)

```

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

```

```

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

```

```

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

```

end

2.3 Multiplication Groups

```

theory Groups-mult
  imports Main
begin

```

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

```

notation times (infixl  $\cdot$  70)

```

```

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

```

```

lemma div-conv-mult-inverse:  $a / b = a \cdot (\text{inverse } b)$ 
  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}  $\Rightarrow$  nat  $\Rightarrow$  'a ((-) [1000,999]
999)
  where npower  $x\ n \equiv inverse (x \wedge n)$ 

end

```


Chapter 3

International System of Quantities

3.1 Quantity Dimensions

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

3.1.1 Preliminaries

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

```
definition unit = (undefined::'a)
```

```
lemma UNIV-unitary: UNIV = {a::'a}
```

```
proof -
```

```
  have card(UNIV :: 'a set) = 1
```

```
    by (simp add: local.unitary-unit-pres)
```

```
  thus ?thesis
```

```
    by (metis (full-types) UNIV-I card-1-singletonE empty-iff insert-iff)
```

```
qed
```

```
lemma eq-unit: (a::'a) = b
```

```
  by (metis (full-types) UNIV-unitary iso-tuple-UNIV-I singletonD)
```

```
end
```

```
lemma unitary-intro: (UNIV::'s set) = {a}  $\implies$  OFCLASS('s, unitary-class)
```

```
  apply (intro-classes, auto)
```

```
  using finite.simps apply blast
```

```

using card-1-singleton-iff apply blast
done

named-theorems si-def and si-eq

instantiation unit :: comm-monoid-add
begin
  definition zero-unit = ()
  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* ($\lambda i. 0$)
instance ..
end

instantiation *dimvec* :: (*plus, enum*) *times*
begin
definition *times-dimvec* :: ('a, 'b) dimvec \Rightarrow ('a, 'b) dimvec \Rightarrow ('a, 'b) dimvec
where
times-dimvec $x\ y = \text{dim-lambda } (\lambda i. \text{dim-nth } x\ i + \text{dim-nth } y\ i)$
instance ..
end

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

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

instantiation *dimvec* :: (*{plus, uminus}*, *enum*) *inverse*
begin
definition *inverse-dimvec* :: ('a, 'b) dimvec \Rightarrow ('a, 'b) dimvec **where**
inverse-dimvec $x = \text{dim-lambda } (\lambda i. - \text{dim-nth } x\ i)$
definition *divide-dimvec* :: ('a, 'b) dimvec \Rightarrow ('a, 'b) dimvec \Rightarrow ('a, 'b) dimvec
where
[*code-unfold*]: *divide-dimvec* $x\ y = x * (\text{inverse } y)$
instance ..
end

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

3.1.3 Code Generation

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

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

code-datatype *mk-dimvec*

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

proof (*rule inj-onI, safe*)

fix $x\ y :: 'n\ \text{list}$

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

then show $x = y$
by ($metis\ a(2)\ a(3)\ nth\text{-equality}I$)
qed

lemma $mk\text{-dimvec}\text{-eq}\text{-iff}$ [$simp$]:
assumes $length\ x = CARD('d)$ $length\ y = CARD('d)$
shows ($(mk\text{-dimvec } x :: ('n::ring\text{-}1, 'd::enum) \text{ dimvec}$) = $mk\text{-dimvec } y$) \longleftrightarrow ($x = y$)
by ($rule\ inj\text{-on}\text{-eq}\text{-iff}[OF\ mk\text{-dimvec}\text{-inj}],\ simp\text{-all}\ add: assms$)

lemma $one\text{-mk}\text{-dimvec}$ [$code, si\text{-def}$]: ($1 :: ('n::ring\text{-}1, 'a::enum) \text{ dimvec}$) = $mk\text{-dimvec}$
 $(replicate\ CARD('a)\ 0)$
by ($auto\ simp\ add: mk\text{-dimvec}\text{-def}\ one\text{-dimvec}\text{-def}$)

lemma $times\text{-mk}\text{-dimvec}$ [$code, si\text{-def}$]:
 $(mk\text{-dimvec } xs * mk\text{-dimvec } ys :: ('n::ring\text{-}1, 'a::enum) \text{ dimvec}) =$
 $(if\ (length\ xs = CARD('a) \wedge length\ ys = CARD('a))$
 $\quad then\ mk\text{-dimvec } (map\ (\lambda (x, y). x + y) (zip\ xs\ ys))$
 $\quad else\ if\ length\ xs = CARD('a)\ then\ mk\text{-dimvec } xs\ else\ mk\text{-dimvec } ys)$
by ($auto\ simp\ add: times\text{-dimvec}\text{-def}\ mk\text{-dimvec}\text{-def}\ fun\text{-eq}\text{-iff}\ one\text{-dimvec}\text{-def}$)

lemma $power\text{-mk}\text{-dimvec}$ [$si\text{-def}$]:
 $(power\ (mk\text{-dimvec } xs)\ n :: ('n::ring\text{-}1, 'a::enum) \text{ dimvec}) =$
 $(if\ (length\ xs = CARD('a))\ then\ mk\text{-dimvec } (map\ ((*)\ (of\text{-nat } n))\ xs)\ else$
 $mk\text{-dimvec } xs)$
by ($induct\ n,\ simp\ add: one\text{-dimvec}\text{-def}\ mk\text{-dimvec}\text{-def}$)
 $(auto\ simp\ add: times\text{-mk}\text{-dimvec}\ zip\text{-map}\text{-map}[where\ f=id,\ simplified]\ comp\text{-def}$
 $split\text{-beta}'\ zip\text{-same}\text{-conv}\text{-map}\ distrib\text{-right}\ mult.\text{commute})$

lemma $inverse\text{-mk}\text{-dimvec}$ [$code, si\text{-def}$]:
 $(inverse\ (mk\text{-dimvec } xs) :: ('n::ring\text{-}1, 'a::enum) \text{ dimvec}) =$
 $(if\ (length\ xs = CARD('a))\ then\ mk\text{-dimvec } (map\ uminus\ xs)\ else\ 1)$
by ($auto\ simp\ add: inverse\text{-dimvec}\text{-def}\ one\text{-dimvec}\text{-def}\ mk\text{-dimvec}\text{-def}\ fun\text{-eq}\text{-iff}$)

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

```

    (if (length xs = CARD('a) ∧ length ys = CARD('a))
        then mk-dimvec (map (λ (x, y). x - y) (zip xs ys))
        else if length ys = CARD('a) then mk-dimvec (map uminus ys) else mk-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-BaseDim d = dim-lambda (λ i. if (i = d) then 1 else 0)

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

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

definition *is-BaseDim* :: (int, 'd::enum) dimvec ⇒ bool
where *is-BaseDim* x ≡ (∃ i. x = dim-lambda ((λ x. 0)(i := 1)))

lemma *is-BaseDim-mk* [simp]: $\text{is-BaseDim } (\text{mk-BaseDim } x)$
 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*: $(\text{UNIV} :: \text{sdim set}) = \{\text{Length}, \text{Mass}, \text{Time}, \text{Current}, \text{Temperature}, \text{Amount}, \text{Intensity}\}$
 using *sdim.exhaust* by blast

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

instantiation *sdim* :: enum

begin

definition *enum-sdim* = [Length, Mass, Time, Current, Temperature, Amount, Intensity]

definition *enum-all-sdim* P ↔ P Length ∧ P Mass ∧ P Time ∧ P Current ∧ P Temperature ∧ P Amount ∧ P Intensity

definition *enum-ex-sdim* P ↔ P Length ∨ P Mass ∨ P Time ∨ P Current ∨ P Temperature ∨ P Amount ∨ 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 **L·M·T⁻²**

term **M·L⁻³**

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

lemma $L \cdot M \cdot T^{-2} = mk\text{-dimvec } [1, 1, -2, 0, 0, 0, 0]$
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* :: $'a \text{ itself} \Rightarrow \text{Dimension}$

syntax
 $-QD :: \text{type} \Rightarrow \text{logic } (QD('a))$

translations
 $QD('a) == \text{CONST } \text{dim-ty-sem } \text{TYPE}('a)$

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  $\Theta$  = 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)  $\Theta$  <= (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) =  $\Theta$ 
instance by (intro-classes, auto simp add: dim-ty-sem-Temperature-def, (transfer, simp)+)
end

instantiation Amount :: basedim-type
begin
definition [si-eq]: dim-ty-sem-Amount (-::Amount itself) =  $\mathbf{N}$ 
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) =  $\mathbf{J}$ 
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  $\cdot$  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 = ('a2)-1 ((-)-2 [999] 999)
type-synonym 'a DimInvCube = ('a3)-1 ((-)-3 [999] 999)
type-synonym 'a DimInvQuart = ('a4)-1 ((-)-4 [999] 999)

```

```

translations (type) 'a-2 <= (type) ('a2)-1

```

```

translations (type) 'a-3 <= (type) ('a3)-1

```

```

translations (type) 'a-4 <= (type) ('a4)-1

```

```

print-translation <

```

```

  [(@{type-syntax DimTimes},

```

```

    fn ctx => fn [a, b] =>

```

```

      if (a = b)

```

```

        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 = L2
type-synonym Volume = L3
type-synonym Acceleration = L·T-1
type-synonym Frequency = T-1
type-synonym Energy = L2·M·T-2
type-synonym Power = L2·M·T-3
type-synonym Force = L·M·T-2
type-synonym Pressure = L-1·M·T-2
type-synonym Charge = I·T
type-synonym PotentialDifference = L2·M·T-3·I-1
type-synonym Capacitance = L-2·M-1·T4·I2

```

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-ty: int list -> typ
  val ty-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 ty-to-dim (Type (@{type-name Length}, [])) = [1, 0, 0, 0, 0, 0, 0] |
    ty-to-dim (Type (@{type-name Mass}, [])) = [0, 1, 0, 0, 0, 0, 0] |
    ty-to-dim (Type (@{type-name Time}, [])) = [0, 0, 1, 0, 0, 0, 0] |
    ty-to-dim (Type (@{type-name Current}, [])) = [0, 0, 0, 1, 0, 0, 0] |
    ty-to-dim (Type (@{type-name Temperature}, [])) = [0, 0, 0, 0, 1, 0, 0] |
    ty-to-dim (Type (@{type-name Amount}, [])) = [0, 0, 0, 0, 0, 1, 0] |
    ty-to-dim (Type (@{type-name Intensity}, [])) = [0, 0, 0, 0, 0, 0, 1] |
    ty-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* $x\ y =$ ($mag = mag\ x \cdot mag\ y$, $dim = dim\ x \cdot dim\ y$,

$\dots = more\ x \cdot more\ y$)

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* = ($mag = 0$, $dim = 1$, $\dots = 0$)

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* = ($mag = 1$, $dim = 1$, $\dots = 1$)

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* \Rightarrow ('a, 'b, 'c) *Quantity-scheme* **where**
[*si-def*]: *inverse-Quantity-ext* $x = (\mid \text{mag} = \text{inverse} (\text{mag } x), \text{dim} = \text{inverse} (\text{dim } x), \dots = \text{inverse} (\text{more } x) \mid)$
definition *divide-Quantity-ext* :: ('a, 'b, 'c) *Quantity-scheme* \Rightarrow ('a, 'b, 'c) *Quantity-scheme* **where**
[*si-def*]: *divide-Quantity-ext* $x \ y = (\mid \text{mag} = \text{mag } x / \text{mag } y, \text{dim} = \text{dim } x / \text{dim } y, \dots = \text{more } x / \text{more } y \mid)$
instance ..
end

lemma *mag-inverse* [*simp*]: $\text{mag} (\text{inverse } x) = \text{inverse} (\text{mag } x)$
by (*simp* *add*: *inverse-Quantity-ext-def*)

lemma *dim-inverse* [*simp*]: $\text{dim} (\text{inverse } x) = \text{inverse} (\text{dim } x)$
by (*simp* *add*: *inverse-Quantity-ext-def*)

lemma *more-inverse* [*simp*]: $\text{more} (\text{inverse } x) = \text{inverse} (\text{more } x)$
by (*simp* *add*: *inverse-Quantity-ext-def*)

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

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

lemma *more-divide* [*simp*]: $\text{more} (x / y) = \text{more } x / \text{more } y$
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* \Rightarrow ('a, 'b, 'c) *Quantity-scheme* \Rightarrow *bool*
where *less-eq-Quantity-ext* $x \ y = (\text{mag } x \leq \text{mag } y \wedge \text{dim } x = \text{dim } y \wedge \text{more } x \leq \text{more } y)$
end

definition *less-Quantity-ext* :: ('a, 'b, 'c) *Quantity-scheme* \Rightarrow ('a, 'b, 'c) *Quantity-scheme* \Rightarrow bool

where *less-Quantity-ext* $x\ y = (x \leq y \wedge \neg y \leq x)$

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* \Rightarrow ('a, 'b, 'c) *Quantity-scheme* \Rightarrow ('a, 'b, 'c) *Quantity-scheme*

where [si-def]:

$dim\ x = dim\ y \Longrightarrow$

$plus-Quantity-ext\ x\ y = (\mid mag = mag\ x + mag\ y, dim = dim\ x, \dots = more\ x + more\ y \mid)$

instance ..

end

instantiation *Quantity-ext* :: (uminus, enum, uminus) uminus

begin

definition *uminus-Quantity-ext* :: ('a, 'b, 'c) *Quantity-scheme* \Rightarrow ('a, 'b, 'c) *Quantity-scheme* **where**

[si-def]: $uminus-Quantity-ext\ x = (\mid mag = -\ mag\ x, dim = dim\ x, \dots = -\ more\ x \mid)$

instance ..

end

instantiation *Quantity-ext* :: (minus, enum, minus) minus

begin

definition *minus-Quantity-ext* :: ('a, 'b, 'c) *Quantity-scheme* \Rightarrow ('a, 'b, 'c) *Quantity-scheme* \Rightarrow ('a, 'b, 'c) *Quantity-scheme* **where**

[si-def]:

$dim\ x = dim\ y \Longrightarrow$

$minus-Quantity-ext\ x\ y = (\mid mag = mag\ x - mag\ y, dim = dim\ x, \dots = more\ x - more\ y \mid)$

instance ..

end

3.2.2 Measurement Systems

class *unit-system* = unitary

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

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

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

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

lemma *mmore-ext* [*simp*]: $(\ ()\ unit-sys = unit, \dots = 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, \dots = 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, \dots = 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* $x\ y = (\mid$ *unit-sys* = *unit*, ... = *mmore* $x \cdot$ *mmore* $y \mid)$

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* $x = (\mid$ *unit-sys* = *unit*, ... = *inverse* (*mmore* $x \mid)$

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* $x\ y = (\mid$ *unit-sys* = *unit*, ... = *mmore* $x /$ *mmore* $y \mid)$

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* $x\ y = (\text{mmore } x \leq \text{mmore } y)$

definition *less-Measurement-System-ext* :: ($'a$, $'b$) *Measurement-System-ext* \Rightarrow ($'a$, $'b$) *Measurement-System-ext* \Rightarrow *bool*

where *less-Measurement-System-ext* $x\ y = (x \leq y \wedge \neg y \leq x)$

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 $x\ y = (\mid$ *unit-sys* = *unit*, ... = *mmore* $x +$ *mmore*

```

y )
instance ..
end

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

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

```

3.2.3 Dimension Typed Quantities

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

```

typedef (overloaded) ('n, 'd::dim-type, 's::unit-system) QuantT (-[-, -] [999,0,0]
999)
    = {x :: ('n, sdim, 's) Measurement-System. dim x = QD('d)}
  morphisms fromQ toQ by (rule-tac x=(| mag = undefined, dim = QD('d),
unit-sys = unit |) in exI, simp)

```

setup-lifting *type-definition-QuantT*

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

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

Since quantities can have dimension type expressions that are distinct, but denote the same dimension, it is necessary to define the following function for coercion between two dimension expressions. This requires that the underlying dimensions are the same.

definition $coerceQuantT :: 'd_2 \text{ itself} \Rightarrow 'a['d_1::dim\text{-}type, 's::unit\text{-}system] \Rightarrow 'a['d_2::dim\text{-}type, 's]$ **where**

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

syntax

$-QCOERCE :: type \Rightarrow logic \Rightarrow logic \ (QCOERCE[-])$

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

lift-definition $qequiv :: 'n['a::dim\text{-}type, 's::unit\text{-}system] \Rightarrow 'n['b::dim\text{-}type, 's] \Rightarrow bool$ (**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 $qequiv\text{-}refl \ [simp]: a \cong_Q a$
by ($simp \ add: \ qequiv\text{-}def$)

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

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

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

lemma $coerceQuant\text{-}eq\text{-}iff:$
fixes $x :: 'a['d_1::dim\text{-}type, 's::unit\text{-}system]$
assumes $QD('d_1) = QD('d_2::dim\text{-}type)$
shows $(coerceQuantT \ TYPE('d_2) \ x) \cong_Q x$
by ($metis \ qequiv.rep\text{-}eq \ assms \ coerceQuant\text{-}def \ toQ\text{-}cases \ toQ\text{-}inverse$)

lemma *coerceQuant-eq-iff2*:

fixes $x :: 'a['d_1::dim\text{-}type, 's::unit\text{-}system]$
assumes $QD('d_1) = QD('d_2::dim\text{-}type)$ **and** $y = (coerceQuantT\ TYPE('d_2)\ x)$
shows $x \cong_Q y$
using *equiv-sym assms(1) assms(2) coerceQuant-eq-iff by blast*

lemma *updown-eq-iff*:

fixes $x :: 'a['d_1::dim\text{-}type, 's::unit\text{-}system]$ **fixes** $y :: 'a['d_2::dim\text{-}type, 's]$
assumes $QD('d_1) = QD('d_2::dim\text{-}type)$ **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 than $y = x \implies x \cong_Q y$, since x and y may have different type.

lemma *qeq*:

fixes $x :: 'a['d_1::dim\text{-}type, 's::unit\text{-}system]$ **fixes** $y :: 'a['d_2::dim\text{-}type, 's]$
assumes $x \cong_Q y$
shows $QD('d_1) = QD('d_2)$
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 $:: ('n::comm\text{-}ring\text{-}1)['a::dim\text{-}type, 's::unit\text{-}system] \Rightarrow 'n['b::dim\text{-}type, 's] \Rightarrow 'n['a \cdot 'b, 's]$ (**infixl** \cdot 69)
is $(*)$ **by** (*simp add: dim-ty-sem-DimTimes-def times-Quantity-ext-def*)

lift-definition

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

abbreviation (*input*)

qdivide $:: ('n::field)['a::dim\text{-}type, 's::unit\text{-}system] \Rightarrow 'n['b::dim\text{-}type, 's] \Rightarrow 'n['a/'b, 's]$ (**infixl** $'/$ 70) **where**
qdivide $x\ y \equiv x \cdot y^{-1}$

We also provide some helpful notations for expressing heterogeneous powers.

abbreviation *qsq* $((-)^2$ [999] 999) **where** $u^2 \equiv u \cdot u$

abbreviation *qcube* $((-)^3$ [999] 999) **where** $u^3 \equiv u \cdot u \cdot u$

abbreviation *qquart* $((-)^4$ [999] 999) **where** $u^4 \equiv u \cdot u \cdot u \cdot u$

abbreviation *qneg-sq* $((-)^{-2}$ [999] 999) **where** $u^{-2} \equiv (u^2)^{-1}$

abbreviation *qneg-cube* $((-)^{-3}$ [999] 999) **where** $u^{-3} \equiv (u^3)^{-1}$

abbreviation *qneg-quant* $((-)^{-4} [999] 999)$ **where** $u^{-4} \equiv (u^3)^{-1}$

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* :: $'a \Rightarrow 'a :: \text{comm-ring-1} ['d :: \text{dim-type}, 's :: \text{unit-system}] \Rightarrow 'a ['d, 's]$ (**infixr** $*_Q$ 63)
is $\lambda r x. (\text{mag} = r * \text{mag } x, \text{dim} = QD('d), \text{unit-sys} = \text{unit})$ **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* :: $(\text{zero}, \text{dim-type}, \text{unit-system})$ *zero*
begin
lift-definition *zero-QuantT* :: $('a, 'b, 'c)$ *QuantT* **is** $(\text{mag} = 0, \text{dim} = QD('b), \text{unit-sys} = \text{unit})$
by *simp*
instance ..
end

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

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

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

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

instantiation *QuantT* :: $(\text{plus}, \text{dim-type}, \text{unit-system})$ *plus*
begin
lift-definition *plus-QuantT* :: $'a ['b, 'c] \Rightarrow 'a ['b, 'c] \Rightarrow 'a ['b, 'c]$
is $\lambda x y. (\text{mag} = \text{mag } x + \text{mag } y, \text{dim} = QD('b), \text{unit-sys} = \text{unit})$
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  $\lambda x.$  (| 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  $\lambda x y.$  (| 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  $\lambda x y.$  mag x  $\leq$ 
  mag y .
  lift-definition less-QuantT :: 'a['b,'c]  $\Rightarrow$  'a['b,'c]  $\Rightarrow$  bool is  $\lambda x y.$  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 $\lambda n q. (\mid \text{mag} = n *_R \text{mag } q, \text{dim} = \text{dim } q, \text{unit-sys} = \text{unit} \mid)$ **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 $\lambda x. \text{norm } (\text{mag } x)$.
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] \times 'a['b,'c]*) *filter* **where**
uniformity-QuantT = (*INF e \in {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* = ($\forall x \in U. \text{eventually } (\lambda(x', y). x' = x \longrightarrow y \in U)$ *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] \Rightarrow 'a (\llbracket - \rrbracket_Q) **where**
 [*si-def*]: *magQ* *x* = *mag* (*fromQ* *x*)

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

lemma *quant-eq-iff-mag-eq* [*si-eq*]:
 $x = y \longleftrightarrow \llbracket x \rrbracket_Q = \llbracket y \rrbracket_Q$
 by (auto simp add: *magQ-def*, transfer, simp add: eq-unit)

lemma *quant-eqI* [*si-transfer*]:
 $\llbracket x \rrbracket_Q = \llbracket y \rrbracket_Q \Longrightarrow x = y$
 by (simp add: *quant-eq-iff-mag-eq*)

lemma *quant-equiv-iff* [*si-eq*]:
 fixes *x* :: 'a['u₁::dim-type, 's::unit-system] **and** *y* :: 'a['u₂::dim-type, 's::unit-system]
 shows $x \cong_Q y \longleftrightarrow \llbracket x \rrbracket_Q = \llbracket y \rrbracket_Q \wedge QD('u_1) = QD('u_2)$

proof –

have $\forall t$ ta. (ta::'a['u₂, '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['u₁::dim-type, 's::unit-system] **and** *y* :: 'a['u₂::dim-type, 's::unit-system]
 assumes $QD('u_1) = QD('u_2)$ $QD('u_1) = QD('u_2) \Longrightarrow \llbracket x \rrbracket_Q = \llbracket y \rrbracket_Q$
 shows $x \cong_Q y$
 using *assms* *quant-equiv-iff* **by** *blast*

lemma *quant-le-iff-magn-le* [*si-eq*]:
 $x \leq y \longleftrightarrow \llbracket x \rrbracket_Q \leq \llbracket y \rrbracket_Q$
 by (auto simp add: *magQ-def*; (transfer, simp))

lemma *quant-leI* [*si-transfer*]:
 $\llbracket x \rrbracket_Q \leq \llbracket y \rrbracket_Q \Longrightarrow x \leq y$

by (*simp add: quant-le-iff-magn-le*)

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

$x < y \longleftrightarrow \llbracket x \rrbracket_Q < \llbracket y \rrbracket_Q$

by (*auto simp add: magQ-def; (transfer, simp)*)

lemma *quant-lessI* [*si-transfer*]:

$\llbracket x \rrbracket_Q < \llbracket y \rrbracket_Q \implies x < y$

by (*simp add: quant-less-iff-magn-less*)

lemma *magQ-zero* [*si-eq*]: $\llbracket 0 \rrbracket_Q = 0$

by (*simp add: magQ-def, transfer, simp*)

lemma *magQ-one* [*si-eq*]: $\llbracket 1 \rrbracket_Q = 1$

by (*simp add: magQ-def, transfer, simp*)

lemma *magQ-plus* [*si-eq*]: $\llbracket x + y \rrbracket_Q = \llbracket x \rrbracket_Q + \llbracket y \rrbracket_Q$

by (*simp add: magQ-def, transfer, simp*)

lemma *magQ-minus* [*si-eq*]: $\llbracket x - y \rrbracket_Q = \llbracket x \rrbracket_Q - \llbracket y \rrbracket_Q$

by (*simp add: magQ-def, transfer, simp*)

lemma *magQ-uminus* [*si-eq*]: $\llbracket -x \rrbracket_Q = -\llbracket x \rrbracket_Q$

by (*simp add: magQ-def, transfer, simp*)

lemma *magQ-scaleQ* [*si-eq*]: $\llbracket x *_Q y \rrbracket_Q = x * \llbracket y \rrbracket_Q$

by (*simp add: magQ-def, transfer, simp*)

lemma *magQ-qtimes* [*si-eq*]: $\llbracket x \cdot y \rrbracket_Q = \llbracket x \rrbracket_Q \cdot \llbracket y \rrbracket_Q$

by (*simp add: magQ-def, transfer, simp*)

lemma *magQ-qinverse* [*si-eq*]: $\llbracket x^{-1} \rrbracket_Q = \text{inverse } \llbracket x \rrbracket_Q$

by (*simp add: magQ-def, transfer, simp*)

lemma *magQ-qdivide* [*si-eq*]: $\llbracket (x::('a::field)[-,-]) / y \rrbracket_Q = \llbracket x \rrbracket_Q / \llbracket y \rrbracket_Q$

by (*simp add: magQ-def, transfer, simp add: field-class.field-divide-inverse*)

lemma *magQ-numeral* [*si-eq*]: $\llbracket \text{numeral } n \rrbracket_Q = \text{numeral } n$

apply (*induct n, simp-all add: si-def*)

apply (*metis magQ-def magQ-one*)

apply (*metis magQ-def magQ-plus numeral-code(2)*)

apply (*metis magQ-def magQ-one magQ-plus numeral-code(3)*)

done

lemma *magQ-coerce* [*si-eq*]:

fixes $q :: 'a['d_1::\text{dim-type}, 's::\text{unit-system}]$ **and** $t :: 'd_2::\text{dim-type}$ *itself*

assumes $QD('d_1) = QD('d_2)$

shows $\llbracket \text{coerceQuant } t \ q \rrbracket_Q = \llbracket q \rrbracket_Q$

by (*simp add: coerceQuant-def magQ-def assms, metis assms qequiv.rep-eq up-*)

down-eq-iff)

lemma *dimQ [simp]*: $\text{dimQ}(x :: 'a['d::\text{dim-type}, 's::\text{unit-system}]) = \text{QD}('d)$
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 $\text{QD}(N \cdot \Theta \cdot N) = \text{QD}(\Theta \cdot N^2)$ **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 $x^{-1} \cong_Q y^{-1}$
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*: $x^{-1-1} \cong_Q x$
by *si-calc*

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

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

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

lemma *qtimes-cancel*: $x \neq 0 \implies x / x \cong_Q \mathbf{1}$
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* :: $'a::one['d::dim-type, 's::unit-system] \Rightarrow bool$
is $\lambda x. mag\ x = 1 \wedge is-BaseDim\ (dim\ x)$.

definition *mk-base-unit* :: $'u\ itself \Rightarrow 's\ itself \Rightarrow ('a::one)['u::basedim-type, 's::unit-system]$

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

syntax *-mk-base-unit* :: $type \Rightarrow type \Rightarrow logic\ (BUNIT'(-, -'))$
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*]: $\llbracket BUNIT('u::basedim-type, 's::unit-system) \rrbracket_Q = 1$
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 ('s1::unit-system, 's2::unit-system) Conversion ((-/ ⇒U -) [1, 0] 0) =
  {c :: ConvSchema. cLengthF c > 0 ∧ cMassF c > 0 ∧ cTimeF c > 0 ∧ cCurrentF
  c > 0
  ∧ cTemperatureF c > 0 ∧ cAmountF c > 0 ∧ cIntensityF c > 0}
  by (rule-tac x=(| cLengthF = 1, cMassF = 1, cTimeF = 1, cCurrentF = 1
  , cTemperatureF = 1, cAmountF = 1, cIntensityF = 1 |) in exI,
  simp)
```

setup-lifting type-definition-Conversion

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

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

3.6.2 Conversion Algebra

```
lift-definition convId :: 's::unit-system ⇒U 's (idC)
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**
 $\lambda c_1 c_2. (\mid cLengthF = cLengthF c_1 * cLengthF c_2, cMassF = cMassF c_1 * cMassF c_2$
 c_2
 $, cTimeF = cTimeF c_1 * cTimeF c_2, cCurrentF = cCurrentF c_1 * cCurrentF c_2$
 $, cTemperatureF = cTemperatureF c_1 * cTemperatureF c_2$
 $, cAmountF = cAmountF c_1 * cAmountF c_2, cIntensityF = cIntensityF c_1$
 $* cIntensityF c_2 \mid)$
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**

$\lambda c. (\mid cLengthF = inverse (cLengthF c), cMassF = inverse (cMassF c), cTimeF = inverse (cTimeF c)$
 $, cCurrentF = inverse (cCurrentF c), cTemperatureF = inverse (cTemperatureF c)$
 $, cAmountF = inverse (cAmountF c), cIntensityF = inverse (cIntensityF c)$
 $\mid)$ **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**

$dconvfactor c d =$
 $LengthF c \hat{\ }_Z dim\text{-}nth d Length$
 $* MassF c \hat{\ }_Z dim\text{-}nth d Mass$
 $* TimeF c \hat{\ }_Z dim\text{-}nth d Time$
 $* CurrentF c \hat{\ }_Z dim\text{-}nth d Current$
 $* TemperatureF c \hat{\ }_Z dim\text{-}nth d Temperature$
 $* AmountF c \hat{\ }_Z dim\text{-}nth d Amount$
 $* IntensityF c \hat{\ }_Z dim\text{-}nth d Intensity$

lemma $dconvfactor\text{-}pos$ [simp]: $dconvfactor c d > 0$
by (simp add: dconvfactor-def)

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

lemma $dconvfactor\text{-}convinv$: $dconvfactor (convinv c) d = inverse (dconvfactor c d)$
by (simp add: dconvfactor-def intpow-inverse[THEN sym])

lemma $dconvfactor\text{-}id$ [simp]: $dconvfactor id_C d = 1$
by (simp add: dconvfactor-def, transfer, simp)

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

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

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

lift-definition $qconv :: ('s_1, 's_2) Conversion \Rightarrow ('a::field\text{-}char\text{-}0)['d::dim\text{-}type, 's_1::unit\text{-}system] \Rightarrow 'a['d, 's_2::unit\text{-}system]$
is $\lambda c q. (\ \ mag = of\text{-}rat (dconvfactor c (dim q)) * mag q, dim = dim q, unit\text{-}sys = unit \)$ **by** simp

lemma $magQ\text{-}qconv$: $\llbracket qconv c q \rrbracket_Q = of\text{-}rat (dconvfactor c (dim Q q)) * \llbracket q \rrbracket_Q$
by (simp add: si-def, transfer, simp)

lemma *qconv-id* [*simp*]: $qconv\ id_C\ x = x$

by (*transfer'*, *simp add: Measurement-System-eq-intro*)

lemma *qconv-comp*: $qconv\ (c_1\ \circ_C\ c_2)\ x = qconv\ c_1\ (qconv\ c_2\ x)$

by (*transfer*, *simp add: dconvfactor-compose of-rat-mult*)

lemma *qconv-convinv* [*simp*]: $qconv\ (convinv\ c)\ (qconv\ c\ x) = x$

by (*transfer*, *simp add: dconvfactor-convinv mult.assoc[THEN sym] of-rat-mult[THEN sym] Measurement-System-eq-intro*)

lemma *qconv-scaleQ* [*simp*]: $qconv\ c\ (d\ *_Q\ x) = d\ *_Q\ qconv\ c\ x$

by (*transfer*, *simp*)

lemma *qconv-plus* [*simp*]: $qconv\ c\ (x + y) = qconv\ c\ x + qconv\ c\ y$

by (*transfer*, *auto simp add: plus-Quantity-ext-def mult.commute ring-class.ring-distrib*)

lemma *qconv-minus* [*simp*]: $qconv\ c\ (x - y) = qconv\ c\ x - qconv\ c\ y$

by (*transfer*, *auto simp add: plus-Quantity-ext-def mult.commute ring-class.ring-distrib*)

lemma *qconv-qlmult* [*simp*]: $qconv\ c\ (x \cdot y) = qconv\ c\ x \cdot qconv\ c\ y$

by (*transfer*, *simp add: times-Quantity-ext-def times-Measurement-System-ext-def dconvfactor-times of-rat-mult*)

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

by (*transfer*, *simp add: inverse-Quantity-ext-def inverse-Measurement-System-ext-def dconvfactor-inverse of-rat-inverse*)

lemma *qconv-Length* [*simp*]: $qconv\ c\ BUNIT(L, -) = LengthF\ c\ *_Q\ BUNIT(L, -)$

by (*simp add: dconvfactor-def magQ-qconv si-eq mk-BaseDim-def one-dimvec-def*)

lemma *qconv-Mass* [*simp*]: $qconv\ c\ BUNIT(M, -) = MassF\ c\ *_Q\ BUNIT(M, -)$

by (*simp add: dconvfactor-def magQ-qconv si-eq mk-BaseDim-def one-dimvec-def*)

lemma *qconv-Time* [*simp*]: $qconv\ c\ BUNIT(T, -) = TimeF\ c\ *_Q\ BUNIT(T, -)$

by (*simp add: dconvfactor-def magQ-qconv si-eq mk-BaseDim-def one-dimvec-def*)

lemma *qconv-Current* [*simp*]: $qconv\ c\ BUNIT(I, -) = CurrentF\ c\ *_Q\ BUNIT(I, -)$

by (*simp add: dconvfactor-def magQ-qconv si-eq mk-BaseDim-def one-dimvec-def*)

lemma *qconv-Temperature* [*simp*]: $qconv\ c\ BUNIT(\Theta, -) = TemperatureF\ c\ *_Q\ BUNIT(\Theta, -)$

by (*simp add: dconvfactor-def magQ-qconv si-eq mk-BaseDim-def one-dimvec-def*)

lemma *qconv-Amount* [*simp*]: $qconv\ c\ BUNIT(N, -) = AmountF\ c\ *_Q\ BUNIT(N, -)$

by (*simp add: dconvfactor-def magQ-qconv si-eq mk-BaseDim-def one-dimvec-def*)


```

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

```

3.7 Meta-Theory for ISQ

```

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

```


Chapter 4

International System of Units

4.1 SI Units Semantics

```
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  $\equiv$  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  $\equiv$  BUNIT(L, SI)
abbreviation kilogram  $\equiv$  BUNIT(M, SI)
abbreviation ampere  $\equiv$  BUNIT(I, SI)
abbreviation kelvin  $\equiv$  BUNIT( $\Theta$ , SI)
abbreviation mole  $\equiv$  BUNIT(N, SI)
abbreviation candela  $\equiv$  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 is the one element.

4.1.1 Example Unit Equations

lemma (*metre* · *second*⁻¹) · *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]  $\Rightarrow$  'a['d::dim-type,
SI] where
metrify  $\equiv$  qconv (convschema (TYPE('s)))

```

Conversion via SI units

```

abbreviation qmconv ::
's1 itself  $\Rightarrow$  's2 itself
 $\Rightarrow$  ('a::field-char-0)['d::dim-type, 's1::metrifiable]
 $\Rightarrow$  'a['d::dim-type, 's2::metrifiable] where
qmconv s1 s2 x  $\equiv$  qconv (invC (schemaC s2)  $\circ$ C schemaC s1) x

```

syntax

```
-qmconv :: type  $\Rightarrow$  type  $\Rightarrow$  logic (QMC'(-  $\rightarrow$  -'))
```

translations

```
QMC('s1  $\rightarrow$  's2) == CONST qmconv TYPE('s1) TYPE('s2)
```

```

lemma qmconv-self: QMC('s::metrifiable  $\rightarrow$  '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  $\equiv$  unit :: CGS

```

4.2.2 Base Units

```

abbreviation centimetre  $\equiv$  BUNIT(L, CGS)
abbreviation gram  $\equiv$  BUNIT(M, CGS)

```

4.2.3 Conversion to SI

```

instantiation CGS :: metrifiable

```

begin

lift-definition *convschema-CGS* :: *CGS itself* \Rightarrow (*CGS*, *SI*) *Conversion is*
 $\lambda x. (\mid cLengthF = 0.01, cMassF = 0.001, cTimeF = 1$
 $\quad, cCurrentF = 1, cTemperatureF = 1, cAmountF = 1, cIntensityF = 1 \mid)$ **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* \equiv *second*⁻¹

abbreviation *radian* \equiv *metre* · *metre*⁻¹

abbreviation *steradian* \equiv *metre*² · *metre*⁻²

abbreviation *joule* \equiv *kilogram* · *metre*² · *second*⁻²

type-synonym 'a *joule* = 'a[*M* · *L*² · *T*⁻², *SI*]

abbreviation *watt* \equiv *kilogram* · *metre*² · *second*⁻³

type-synonym 'a *watt* = 'a[*M* · *L*² · *T*⁻³, *SI*]

abbreviation *coulomb* \equiv *ampere* · *second*

type-synonym 'a *coulomb* = 'a[*I* · *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 $9192631770 *_{\mathcal{Q}}$ *hertz*

Speed of light in vacuum

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

Planck constant

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

Elementary charge

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

The Boltzmann constant

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

The Avogadro number

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

abbreviation *max-luminous-frequency* :: *'a*[T^{-1} ,*SI*] **where**
max-luminous-frequency \equiv ($540 \cdot 10^{12}$) $*_{\mathcal{Q}}$ *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 $683 *_{\mathcal{Q}}$ (*lumen/watt*)

4.3.3 Checking Foundational Equations of the SI System

theorem *second-definition*:

$$1 *_{\mathcal{Q}} \textit{second} \cong_{\mathcal{Q}} (9192631770 *_{\mathcal{Q}} \mathbf{1}) / \Delta v_{Cs}$$

by *si-calc*

theorem *metre-definition*:

$1 *_{\mathcal{Q}} \text{metre} \cong_{\mathcal{Q}} (\mathbf{c} / (299792458 *_{\mathcal{Q}} \mathbf{1})) \cdot \text{second}$
 $1 *_{\mathcal{Q}} \text{metre} \cong_{\mathcal{Q}} (9192631770 / 299792458) *_{\mathcal{Q}} (\mathbf{c} / \Delta v_{Cs})$
by *si-calc+*

theorem *kilogram-definition*:

$((1 *_{\mathcal{Q}} \text{kilogram}) :: 'a \text{ kilogram}) \cong_{\mathcal{Q}} (\mathbf{h} / (6.62607015 \cdot 1 / (10^{34}) *_{\mathcal{Q}} \mathbf{1})) \cdot \text{metre}^{-2} \cdot \text{second}$
by *si-calc*

abbreviation *approx-ice-point* $\equiv 273.15 *_{\mathcal{Q}} \text{kelvin}$

default-sort *type*

end

4.4 SI Prefixes

theory *SI-Prefix*

imports *SI-Constants*

begin

4.4.1 Definitions

Prefixes are simply numbers that can be composed with units using the scalar multiplication operator ($*_{\mathcal{Q}}$).

default-sort *ring-char-0*

definition *deca* :: 'a **where** [*si-eq*]: *deca* = 10^1

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

definition *kilo* :: 'a **where** [*si-eq*]: *kilo* = 10^3

definition *mega* :: 'a **where** [*si-eq*]: *mega* = 10^6

definition *giga* :: 'a **where** [*si-eq*]: *giga* = 10^9

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

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

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

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

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 $2.3 *_{\mathbb{Q}} (\text{centi} *_{\mathbb{Q}} \text{metre})^3 = 2.3 \cdot 1/10^6 *_{\mathbb{Q}} \text{metre}^3$
by (*si-simp*)

lemma $1 *_{\mathbb{Q}} (\text{centi} *_{\mathbb{Q}} \text{metre})^{-1} = 100 *_{\mathbb{Q}} \text{metre}^{-1}$
by (*si-simp*)

4.4.3 Binary Prefixes

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

default-sort *ring-char-0*

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

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

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

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

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

definition *exbi* :: 'a **where** [*si-eq*]: *exbi* = 2^{60}

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* · *metre* · *second*⁻²

type-synonym 'a *newton* = 'a[*M* · *L* · *T*⁻², *SI*]

abbreviation *pascal* \equiv *kilogram* · *metre*⁻¹ · *second*⁻²

type-synonym 'a *pascal* = 'a[*M* · *L*⁻¹ · *T*⁻², *SI*]

abbreviation *volt* \equiv *kilogram* · *metre*² · *second*⁻³ · *ampere*⁻¹

type-synonym 'a *volt* = 'a[*M* · *L*² · *T*⁻³ · *I*⁻¹, *SI*]

abbreviation *farad* \equiv *kilogram*⁻¹ · *metre*⁻² · *second*⁴ · *ampere*²

type-synonym 'a *farad* = 'a[*M*⁻¹ · *L*⁻² · *T*⁴ · *I*², *SI*]

abbreviation *ohm* \equiv *kilogram* · *metre*² · *second*⁻³ · *ampere*⁻²

type-synonym 'a *ohm* = 'a[*M* · *L*² · *T*⁻³ · *I*⁻², *SI*]

abbreviation *siemens* \equiv *kilogram*⁻¹ · *metre*⁻² · *second*³ · *ampere*²

abbreviation *weber* \equiv *kilogram* · *metre*² · *second*⁻² · *ampere*⁻¹

abbreviation *tesla* \equiv *kilogram* · *second*⁻² · *ampere*⁻¹

abbreviation *henry* \equiv *kilogram* · *metre*² · *second*⁻² · *ampere*⁻²

abbreviation *lux* \equiv *candela* · *steradian* · *metre*⁻²

abbreviation (*input*) *becquerel* \equiv *second*⁻¹

abbreviation *gray* \equiv *metre*² · *second*⁻²

abbreviation *sievert* \equiv *metre*² · *second*⁻²

abbreviation *katal* \equiv *mole* · *second*⁻¹

definition *degrees-celcius* :: 'a::field-char-0 \Rightarrow 'a[Θ] (-XXXC [999] 999)
where [*si-eq*]: *degrees-celcius* $x = (x *_{\mathcal{Q}}$ *kelvin*) + *approx-ice-point*

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

4.5.2 Equivalences

lemma *joule-alt-def*: *joule* $\cong_{\mathcal{Q}}$ *newton* · *metre*
by *si-calc*

lemma *watt-alt-def*: *watt* $\cong_{\mathcal{Q}}$ *joule* / *second*
by *si-calc*

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

lemma *farad-alt-def*: *farad* $\cong_{\mathcal{Q}}$ *coulomb* / *volt*
by *si-calc*

lemma *ohm-alt-def*: *ohm* $\cong_{\mathcal{Q}}$ *volt* / *ampere*
by *si-calc*

lemma *siemens-alt-def*: *siemens* $\cong_{\mathcal{Q}}$ *ampere* / *volt*
by *si-calc*

lemma *weber-alt-def*: *weber* $\cong_{\mathcal{Q}}$ *volt* · *second*
by *si-calc*

lemma *tesla-alt-def*: *tesla* $\cong_{\mathcal{Q}}$ *weber* / *metre*²
by *si-calc*

lemma *henry-alt-def*: *henry* $\cong_{\mathcal{Q}}$ *weber* / *ampere*
by *si-calc*

lemma *lux-alt-def*: *lux* = *lumen* / *metre*²
by *simp*

lemma *gray-alt-def*: *gray* $\cong_{\mathcal{Q}}$ *joule* / *kilogram*
by *si-calc*

lemma *sievert-alt-def*: *sievert* $\cong_{\mathcal{Q}}$ *joule* / *kilogram*
by *si-calc*

4.5.3 Properties

lemma *kilogram*: $\text{kilo} *_{\mathcal{Q}} \text{gram} = \text{kilogram}$
by (*si-simp*)

lemma *celcius-to-kelvin*: $TXXXC = (T *_{\mathcal{Q}} \text{kelvin}) + (273.15 *_{\mathcal{Q}} \text{kelvin})$
by (*si-simp*)

end

4.6 Non-SI Units Accepted for SI use

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

definition [*si-def, si-eq*]: $\text{minute} = 60 *_{\mathcal{Q}} \text{second}$

definition [*si-def, si-eq*]: $\text{hour} = 60 *_{\mathcal{Q}} \text{minute}$

definition [*si-def, si-eq*]: $\text{day} = 24 *_{\mathcal{Q}} \text{hour}$

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

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

abbreviation *degrees* (-XXX [999] 999) **where** $nXXX \equiv n *_{\mathcal{Q}} \text{degree}$

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

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

definition [*si-def, si-eq*]: $\text{dalton} = 1.66053906660 * (1 / 10^{27}) *_{\mathcal{Q}} \text{kilogram}$

4.6.1 Example Unit Equations

lemma $1 *_{\mathcal{Q}} \text{hour} = 3600 *_{\mathcal{Q}} \text{second}$
by (*si-simp*)

lemma $\text{watt} \cdot \text{hour} \cong_{\mathcal{Q}} 3600 *_{\mathcal{Q}} \text{joule}$ **by** (*si-calc*)

lemma $25 *_{\mathcal{Q}} \text{metre} / \text{second} = 90 *_{\mathcal{Q}} (\text{kilo} *_{\mathcal{Q}} \text{metre}) / \text{hour}$
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-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 · 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*]: $\llbracket \textit{light-year} \rrbracket_Q = \llbracket \textit{c} \cdot \textit{julian-year} \rrbracket_Q$
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*] \Rightarrow (*real*³[*L* · *T*⁻²])

end

4.10 Parsing and Pretty Printing of SI Units

theory *SI-Pretty*
imports *SI*
begin

4.10.1 Syntactic SI Units

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

nonterminal *si*

syntax

-*si-metre* :: *si* (*m*)
-*si-kilogram* :: *si* (*kg*)
-*si-second* :: *si* (*s*)

```

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

-si-square  :: si => si ((-)2 [999] 999)
-si-cube    :: si => si ((-)3 [999] 999)
-si-quart   :: si => si ((-)4 [999] 999)

-si-inverse :: si => si ((-)-1 [999] 999)
-si-invsquare :: si => si ((-)-2 [999] 999)
-si-invcube  :: si => si ((-)-3 [999] 999)
-si-invquart :: si => si ((-)-4 [999] 999)

-si-times   :: si => si => si (infixl · 70)

```

4.10.2 Type Notation

Pretty notation for SI units at the type level.

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

syntax

```

-si-unit      :: type => si => type (-[-] [999,0] 999)
-si-print     :: type => si (SIPRINT'(-))

```

translations

```

(type) 'a[SIPRINT('d)] == (type) 'a['d, SI]
(si) SIPRINT('d)2 == (si) SIPRINT('d2)
(si) SIPRINT('d)3 == (si) SIPRINT('d3)
(si) SIPRINT('d)4 == (si) SIPRINT('d4)
(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('d1) · SIPRINT('d2) == (si) SIPRINT('d1 · 'd2)
(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 $x \leq -si\text{-quart } (-si\text{-inverse } x)$

typ $real[m \cdot s^{-2}]$
typ $real[m \cdot s^{-2} \cdot A^2]$
term $5 *_{\mathcal{Q}} \text{joule}$

4.10.3 Value Notations

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

definition $SIQ\ n\ u = n *_{\mathcal{Q}}\ u$

syntax

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

translations

-siq-term $n\ u \Rightarrow CONST\ SIQ\ n\ (-si\text{-term } u)$
-siq-term $n\ (-siq\text{-print } u) \leq CONST\ SIQ\ n\ u$
-si-term $(-si\text{-times } x\ y) == (-si\text{-term } x) \cdot (-si\text{-term } y)$
-si-term $(-si\text{-inverse } x) == (-si\text{-term } x)^{-1}$
-si-term $(-si\text{-square } x) == (-si\text{-term } x)^2$
-si-term $(-si\text{-cube } x) == (-si\text{-term } x)^3$
 $SI(m) \Rightarrow CONST\ \text{metre}$
 $SI(kg) \Rightarrow CONST\ \text{kilogram}$
 $SI(s) \Rightarrow CONST\ \text{second}$
 $SI(A) \Rightarrow CONST\ \text{ampere}$
 $SI(K) \Rightarrow CONST\ \text{kelvin}$
 $SI(mol) \Rightarrow CONST\ \text{mole}$
 $SI(cd) \Rightarrow CONST\ \text{candela}$

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

-si-square $(-siq\text{-print } x) \leq -siq\text{-print } (x^2)$
-si-cube $(-siq\text{-print } x) \leq -siq\text{-print } (x^3)$
-si-quart $(-siq\text{-print } x) \leq -siq\text{-print } (x^4)$
-si-times $(-siq\text{-print } x)\ (-siq\text{-print } y) \leq -siq\text{-print } (x \cdot y)$

-si-metre $\leq -siq\text{-print } (CONST\ \text{metre})$
-si-kilogram $\leq -siq\text{-print } (CONST\ \text{kilogram})$
-si-second $\leq -siq\text{-print } (CONST\ \text{second})$
-si-ampere $\leq -siq\text{-print } (CONST\ \text{ampere})$
-si-kelvin $\leq -siq\text{-print } (CONST\ \text{kelvin})$
-si-mole $\leq -siq\text{-print } (CONST\ \text{mole})$

```

-si-candela <= -sig-print (CONST candela)

term SI[5, m2]
term SI[22, m·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 Fahrenheit 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 yard3
definition [si-eq]: ounce = 1/12 *Q pound

```

definition [*si-eq*]: $\text{gallon} = 277.421 *_{\mathcal{Q}} \text{inch}^3$

definition [*si-eq*]: $\text{quart} = 1/4 *_{\mathcal{Q}} \text{gallon}$

definition [*si-eq*]: $\text{pint} = 1/8 *_{\mathcal{Q}} \text{gallon}$

definition [*si-eq*]: $\text{peck} = 2 *_{\mathcal{Q}} \text{gallon}$

definition [*si-eq*]: $\text{bushel} = 8 *_{\mathcal{Q}} \text{gallon}$

definition [*si-eq*]: $\text{minute} = 60 *_{\mathcal{Q}} \text{second}$

definition [*si-eq*]: $\text{hour} = 60 *_{\mathcal{Q}} \text{minute}$

4.11.4 Conversion to SI

instantiation *BIS* :: *metrifiable*

begin

lift-definition *convschema-BIS* :: *BIS* itself \Rightarrow (*BIS*, *SI*) *Conversion is*

$\lambda x. (| \text{cLengthF} = 0.9143993, \text{cMassF} = 0.453592338, \text{cTimeF} = 1$

$, \text{cCurrentF} = 1, \text{cTemperatureF} = 5/9, \text{cAmountF} = 1, \text{cIntensityF} = 1 |)$

by *simp*

instance ..

end

lemma *BIS-SI-simps* [*simp*]: $\text{LengthF} (\text{convschema } (a::\text{BIS } \textit{itself})) = 0.9143993$

$\text{MassF} (\text{convschema } a) = 0.453592338$

$\text{TimeF} (\text{convschema } a) = 1$

$\text{CurrentF} (\text{convschema } a) = 1$

$\text{TemperatureF} (\text{convschema } a) = 5/9$

by (*transfer*, *simp*)⁺

4.11.5 Conversion Examples

lemma *metrify* (*foot* :: *rat*[*L*, *BIS*]) = $0.9143993 / 3 *_{\mathcal{Q}} \text{metre}$

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

lemma *metrify* ((*70*::*rat*) *_Q *mile / hour*) = $(704087461 / 22500000) *_{\mathcal{Q}} (\text{metre} / \text{second})$

by (*si-simp*)

lemma *QMC*(*CGS* \rightarrow *BIS*) ((*1*::*rat*) *_Q *centimetre*) = $100000 / 9143993 *_{\mathcal{Q}} \text{yard}$

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.