

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

Simon Foster Burkhart Wolff

February 6, 2026

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	12
2.3	Multiplication Groups	13
3	International System of Quantities	15
3.1	Quantity Dimensions	15
3.1.1	Preliminaries	15
3.1.2	Dimension Vectors	16
3.1.3	Code Generation	17
3.1.4	Dimension Semantic Domain	18
3.1.5	Dimension Type Expressions	20
3.1.6	ML Functions	24
3.2	Quantities	24
3.2.1	Quantity Semantic Domain	24
3.2.2	Measurement Systems	27
3.2.3	Dimension Typed Quantities	30
3.2.4	Predicates on Typed Quantities	31
3.2.5	Operators on Typed Quantities	32
3.3	Proof Support for Quantities	35
3.4	Algebraic Laws	38
3.4.1	Quantity Scale	38
3.4.2	Field Laws	38
3.5	Units	39
3.6	Conversion Between Unit Systems	40
3.6.1	Conversion Schemas	40
3.6.2	Conversion Algebra	41
3.6.3	Conversion Functions	42
3.7	Meta-Theory for ISQ	44

4	International System of Units	45
4.1	SI Units Semantics	45
4.1.1	Example Unit Equations	46
4.1.2	Metrification	46
4.2	Centimetre-Gram-Second System	47
4.2.1	Preliminaries	47
4.2.2	Base Units	47
4.2.3	Conversion to SI	48
4.2.4	Conversion Examples	48
4.3	Physical Constants	48
4.3.1	Core Derived Units	48
4.3.2	Constants	49
4.3.3	Checking Foundational Equations of the SI System	50
4.4	SI Prefixes	50
4.4.1	Definitions	50
4.4.2	Examples	51
4.4.3	Binary Prefixes	51
4.5	Derived SI-Units	52
4.5.1	Definitions	52
4.5.2	Equivalences	53
4.5.3	Properties	54
4.6	Non-SI Units Accepted for SI use	54
4.6.1	Example Unit Equations	54
4.7	Imperial Units via SI Units	55
4.7.1	Units of Length	55
4.7.2	Units of Mass	55
4.7.3	Other Units	56
4.7.4	Unit Equations	56
4.8	Meta-Theory for SI Units	56
4.9	Astronomical Constants	56
4.10	Parsing and Pretty Printing of SI Units	57
4.10.1	Syntactic SI Units	57
4.10.2	Type Notation	58
4.10.3	Value Notations	59
4.11	British Imperial System (1824/1897)	60
4.11.1	Preliminaries	60
4.11.2	Base Units	60
4.11.3	Derived Units	60
4.11.4	Conversion to SI	61
4.11.5	Conversion Examples	61

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}* \Rightarrow *int* \Rightarrow 'a (**infix** $\langle \hat{_Z} \rangle$ 80) **where**
intpow *x* *n* = (if (*n* < 0) then *inverse* (*x* $\hat{_Z}$ *nat* (-*n*)) else (*x* $\hat{_Z}$ *nat* *n*))

lemma *intpow-zero* [*simp*]: $x \hat{_Z} 0 = 1$
<proof>

lemma *intpow-spos* [*simp*]: $x > 0 \Longrightarrow x \hat{_Z} n > 0$
<proof>

lemma *intpow-one* [*simp*]: $x \hat{_Z} 1 = x$
<proof>

lemma *one-intpow* [*simp*]: $1 \hat{_Z} n = 1$
<proof>

lemma *intpow-plus*: $x > 0 \Longrightarrow x \hat{_Z} (m + n) = x \hat{_Z} m * x \hat{_Z} n$
<proof>

lemma *intpow-mult-combine*: $x > 0 \Longrightarrow x \hat{_Z} m * (x \hat{_Z} n * y) = x \hat{_Z} (m + n) * y$
<proof>

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

lemma *intpow-uminus*: $x \hat{_Z} -n = \text{inverse } (x \hat{_Z} n)$

<proof>

lemma *intpow-uminus-nat*: $n \geq 0 \implies x \hat{=} -n = \text{inverse } (x \hat{=} \text{nat } n)$
<proof>

lemma *intpow-inverse*: $\text{inverse } a \hat{=} n = \text{inverse } (a \hat{=} n)$
<proof>

lemma *intpow-mult-distrib*: $(x * y) \hat{=} m = x \hat{=} m * y \hat{=} m$
<proof>

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*:
 $x \in \text{set}(xs) \implies \text{first-ind } xs \ x \ i < \text{length}(xs) + i$
<proof>

lemma *nth-first-ind*:
 $\llbracket \text{distinct } xs; x \in \text{set}(xs) \rrbracket \implies xs ! (\text{first-ind } xs \ x \ i - i) = x$
<proof>

lemma *first-ind-nth*:
 $\llbracket \text{distinct } xs; i < \text{length } xs \rrbracket \implies \text{first-ind } xs \ (xs ! i) \ j = i + j$
<proof>

2.2.2 Enumeration Indices

syntax
-ENUM :: type \Rightarrow logic ($\langle \text{ENUM}'(-)' \rangle$)

syntax-consts
-ENUM == *Enum.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 $length-enum-CARD$: $length ENUM('a) = CARD('a)$
 $\langle proof \rangle$

lemma $CARD-length-enum$: $CARD('a) = length ENUM('a)$
 $\langle proof \rangle$

lemma $enum-ind-less-CARD$ [simp]: $enum-ind (x :: 'a::enum) < CARD('a)$
 $\langle proof \rangle$

lemma $enum-nth-ind$ [simp]: $Enum.enum ! (enum-ind x) = x$
 $\langle proof \rangle$

lemma $enum-distinct-conv-nth$:
assumes $i < CARD('a) j < CARD('a) ENUM('a) ! i = ENUM('a) ! j$
shows $i = j$
 $\langle proof \rangle$

lemma $enum-ind-nth$ [simp]:
assumes $i < CARD('a::enum)$
shows $enum-ind (ENUM('a) ! i) = i$
 $\langle proof \rangle$

lemma $enum-ind-spec$:
 $enum-ind (x :: 'a::enum) = (THE i. i < CARD('a) \wedge Enum.enum ! i = x)$
 $\langle proof \rangle$

lemma $enum-ind-inj$: $inj (enum-ind :: 'a::enum \Rightarrow nat)$
 $\langle proof \rangle$

lemma $enum-ind-neq$ [simp]: $x \neq y \Longrightarrow enum-ind x \neq enum-ind y$
 $\langle proof \rangle$

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** $\langle \cdot \rangle$ 70)

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

lemma *div-conv-mult-inverse*: $a / b = a \cdot (inverse\ b)$
 $\langle proof \rangle$

sublocale *mult*: *group times 1 inverse*
 $\langle proof \rangle$

lemma *diff-self* [*simp*]: $a / a = 1$
 $\langle proof \rangle$

lemma *mult-distrib-inverse* [*simp*]: $(a * b) / b = a$
 $\langle proof \rangle$

end

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

lemma *mult-distrib-inverse'* [*simp*]: $(a * b) / a = b$
 $\langle proof \rangle$

lemma *inverse-distrib*: $inverse\ (a * b) = (inverse\ a) * (inverse\ b)$
 $\langle proof \rangle$

lemma *inverse-divide* [*simp*]: $inverse\ (a / b) = b / a$
 $\langle proof \rangle$

end

abbreviation (*input*) *npower* :: 'a::{*power, inverse*} \Rightarrow *nat* \Rightarrow 'a ($\langle (-^{-}) \rangle$) [*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:  $\text{card } (UNIV::'a \text{ set}) = 1$ 
begin
```

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

```
lemma UNIV-unitary:  $UNIV = \{a::'a\}$ 
<proof>
```

```
lemma eq-unit:  $(a::'a) = b$ 
<proof>
```

```
end
```

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

```
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⟩
end

```

```

instantiation unit :: comm-monoid-mult
begin
  definition one-unit = ()
  definition times-unit (x::unit) (y::unit) = ()
  instance ⟨proof⟩
end

```

```

instantiation unit :: inverse
begin
  definition inverse-unit (x::unit) = ()
  definition divide-unit (x::unit) (y::unit) = ()
  instance ⟨proof⟩
end

```

```

instance unit :: ab-group-mult
  ⟨proof⟩

```

3.1.2 Dimension Vectors

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

```

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

```

```

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

```

```

instantiation dimvec :: (zero, enum) one
begin
  definition one-dimvec :: ('a, 'b) dimvec where one-dimvec = dim-lambda (λ i. 0)
  instance ⟨proof⟩
end

```

```

instantiation dimvec :: (plus, enum) times
begin
  definition times-dimvec :: ('a, 'b) dimvec ⇒ ('a, 'b) dimvec ⇒ ('a, 'b) dimvec
  where

```

```

times-dimvec x y = dim-lambda (λ i. dim-nth x i + dim-nth y i)
instance ⟨proof⟩
end

```

```

instance dimvec :: (comm-monoid-add, enum) comm-monoid-mult
  ⟨proof⟩

```

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

```

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

```

```

definition inverse-dimvec :: ('a, 'b) dimvec ⇒ ('a, 'b) dimvec where
inverse-dimvec x = dim-lambda (λ i. - dim-nth x i)

```

```

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

```

```

  instance ⟨proof⟩
end

```

```

instance dimvec :: (ab-group-add, enum) ab-group-mult
  ⟨proof⟩

```

3.1.3 Code Generation

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

```

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

```

```

code-datatype mk-dimvec

```

```

lemma mk-dimvec-inj: inj-on (mk-dimvec :: 'n list ⇒ ('n::ring-1, 'd::enum) dimvec)
  {xs. length xs = CARD('d)}
  ⟨proof⟩

```

```

lemma mk-dimvec-eq-iff [simp]:
  assumes length x = CARD('d) length y = CARD('d)
  shows ((mk-dimvec x :: ('n::ring-1, 'd::enum) dimvec) = mk-dimvec y) ⟷ (x
= y)
  ⟨proof⟩

```

```

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

```

lemma *times-mk-dimvec* [*code*, *si-def*]:

```
(mk-dimvec xs * mk-dimvec ys :: ('n::ring-1, 'a::enum) dimvec) =
  (if (length xs = CARD('a) ^ length ys = CARD('a))
    then mk-dimvec (map (\ (x, y). x + y) (zip xs ys))
    else if length xs = CARD('a) then mk-dimvec xs else mk-dimvec ys)
  <proof>
```

lemma *power-mk-dimvec* [*si-def*]:

```
(power (mk-dimvec xs) n :: ('n::ring-1, 'a::enum) dimvec) =
  (if (length xs = CARD('a)) then mk-dimvec (map ((* (of-nat n)) xs) else
mk-dimvec xs)
  <proof>
```

lemma *inverse-mk-dimvec* [*code*, *si-def*]:

```
(inverse (mk-dimvec xs) :: ('n::ring-1, 'a::enum) dimvec) =
  (if (length xs = CARD('a)) then mk-dimvec (map uminus xs) else 1)
  <proof>
```

lemma *divide-mk-dimvec* [*code*, *si-def*]:

```
(mk-dimvec xs / mk-dimvec ys :: ('n::ring-1, 'a::enum) dimvec) =
  (if (length xs = CARD('a) ^ length ys = CARD('a))
    then mk-dimvec (map (\ (x, y). x - y) (zip xs ys))
    else if length ys = CARD('a) then mk-dimvec (map uminus ys) else mk-dimvec
xs)
  <proof>
```

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* (\ i. if (i = d) then 1 else 0)

lemma *mk-BaseDim-neq* [*simp*]: $x \neq y \implies \text{mk-BaseDim } x \neq \text{mk-BaseDim } y$
 <proof>

lemma *mk-BaseDim-code* [*code*]: *mk-BaseDim* (d::'d::enum) = *mk-dimvec* (*list-update*
 (*replicate* CARD('d) 0) (*enum-ind* d) 1)
 <proof>

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

lemma *is-BaseDim-mk* [*simp*]: *is-BaseDim* (*mk-BaseDim* x)
 <proof>

3.1.4 Dimension Semantic Domain

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

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

lemma *sdim-UNIV*: (*UNIV* :: *sdim* set) = {*Length*, *Mass*, *Time*, *Current*, *Temperature*, *Amount*, *Intensity*}
 ⟨*proof*⟩

lemma *CARD-sdim* [*simp*]: *CARD*(*sdim*) = 7
 ⟨*proof*⟩

instantiation *sdim* :: *enum*

begin

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

definition *enum-all-sdim* *P* \longleftrightarrow *P* *Length* \wedge *P* *Mass* \wedge *P* *Time* \wedge *P* *Current* \wedge *P* *Temperature* \wedge *P* *Amount* \wedge *P* *Intensity*

definition *enum-ex-sdim* *P* \longleftrightarrow *P* *Length* \vee *P* *Mass* \vee *P* *Time* \vee *P* *Current* \vee *P* *Temperature* \vee *P* *Amount* \vee *P* *Intensity*

instance

⟨*proof*⟩

end

instantiation *sdim* :: *card-UNIV*

begin

definition *finite-UNIV* = *Phantom*(*sdim*) *True*

definition *card-UNIV* = *Phantom*(*sdim*) 7

instance ⟨*proof*⟩

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

⟨*proof*⟩

type-synonym *Dimension* = (*int*, *sdim*) *dimvec*

abbreviation *LengthBD* (⟨**L**⟩) **where** **L** \equiv *mk-BaseDim* *Length*

abbreviation *MassBD* (⟨**M**⟩) **where** **M** \equiv *mk-BaseDim* *Mass*

abbreviation *TimeBD* (⟨**T**⟩) **where** **T** \equiv *mk-BaseDim* *Time*

abbreviation *CurrentBD* (⟨**I**⟩) **where** **I** \equiv *mk-BaseDim* *Current*

abbreviation *TemperatureBD* (⟨**Θ**⟩) **where** **Θ** \equiv *mk-BaseDim* *Temperature*

abbreviation *AmountBD* (⟨**N**⟩) **where** **N** \equiv *mk-BaseDim* *Amount*

abbreviation *IntensityBD* (⟨**J**⟩) **where** **J** \equiv *mk-BaseDim* *Intensity*

abbreviation *BaseDimensions* \equiv {**L**, **M**, **T**, **I**, **Θ**, **N**, **J**}

lemma *BD-mk-dimvec* [*si-def*]:

L = *mk-dimvec* [1, 0, 0, 0, 0, 0, 0]

$\mathbf{M} = mk\text{-dimvec } [0, 1, 0, 0, 0, 0, 0]$
 $\mathbf{T} = mk\text{-dimvec } [0, 0, 1, 0, 0, 0, 0]$
 $\mathbf{I} = mk\text{-dimvec } [0, 0, 0, 1, 0, 0, 0]$
 $\mathbf{\Theta} = mk\text{-dimvec } [0, 0, 0, 0, 1, 0, 0]$
 $\mathbf{N} = mk\text{-dimvec } [0, 0, 0, 0, 0, 1, 0]$
 $\mathbf{J} = mk\text{-dimvec } [0, 0, 0, 0, 0, 0, 1]$
<proof>

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

lemma *seven-BaseDimensions: card BaseDimensions = 7*
<proof>

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

term $\mathbf{L} \cdot \mathbf{M} \cdot \mathbf{T}^{-2}$
term $\mathbf{M} \cdot \mathbf{L}^{-3}$

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

lemma $\mathbf{L} \cdot \mathbf{M} \cdot \mathbf{T}^{-2} = mk\text{-dimvec } [1, 1, -2, 0, 0, 0, 0]$
<proof>

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*, since a dimension type is used only to mark a quantity.

class *dim-type = unitary +*
fixes *dim-ty-sem :: 'a itself \Rightarrow Dimension*

syntax
 $-QD :: type \Rightarrow logic (\langle QD'(-) \rangle)$

syntax-consts
 $-QD == dim\text{-ty}\text{-sem}$

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

The notation $QD('a)$ allows to obtain the dimension of a dimension type $'a$. The subset of basic dimension types can be characterized by the following type class:

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

Base Dimension Type Expressions

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

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

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

translations

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

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

```
instantiation Length :: basedim-type
begin
definition [si-eq]: dim-ty-sem-Length (::Length itself) = L
instance ⟨proof⟩
end
```

instantiation *Mass* :: *basedim-type*

begin

definition [*si-eq*]: *dim-ty-sem-Mass* (-::*Mass* *itself*) = **M**

instance ⟨*proof*⟩

end

instantiation *Time* :: *basedim-type*

begin

definition [*si-eq*]: *dim-ty-sem-Time* (-::*Time* *itself*) = **T**

instance ⟨*proof*⟩

end

instantiation *Current* :: *basedim-type*

begin

definition [*si-eq*]: *dim-ty-sem-Current* (-::*Current* *itself*) = **I**

instance ⟨*proof*⟩

end

instantiation *Temperature* :: *basedim-type*

begin

definition [*si-eq*]: *dim-ty-sem-Temperature* (-::*Temperature* *itself*) = **Θ**

instance ⟨*proof*⟩

end

instantiation *Amount* :: *basedim-type*

begin

definition [*si-eq*]: *dim-ty-sem-Amount* (-::*Amount* *itself*) = **N**

instance ⟨*proof*⟩

end

instantiation *Intensity* :: *basedim-type*

begin

definition [*si-eq*]: *dim-ty-sem-Intensity* (-::*Intensity* *itself*) = **J**

instance ⟨*proof*⟩

end

instantiation *NoDimension* :: *dim-type*

begin

definition [*si-eq*]: *dim-ty-sem-NoDimension* (-::*NoDimension* *itself*) = (*1*::*Dimension*)

instance ⟨*proof*⟩

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)*

⟨*proof*⟩

Dimension Type Constructors: Inner Product and Inverse

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

```
typedef ('a::dim-type, 'b::dim-type) DimTimes (infixl <·> 69) = UNIV :: unit set
<proof>
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 <proof>
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 <proof>
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 <proof>
end
```

Dimension Type Syntax

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

```
type-synonym ('a, 'b) DimDiv = 'a · ('b-1) (infixl </> 69)
```

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

```
type-synonym 'a DimSquare = 'a · 'a (<(-)2> [999] 999)
type-synonym 'a DimCube = 'a · 'a · 'a (<(-)3> [999] 999)
type-synonym 'a DimQuart = 'a · 'a · 'a · 'a (<(-)4> [999] 999)
```

type-synonym *'a DimInvSquare* = (*'a*²)⁻¹ (*'(-)*⁻², [999] 999)

type-synonym *'a DimInvCube* = (*'a*³)⁻¹ (*'(-)*⁻³, [999] 999)

type-synonym *'a DimInvQuart* = (*'a*⁴)⁻¹ (*'(-)*⁻⁴, [999] 999)

translations (*type*) *'a*⁻² <= (*type*) (*'a*²)⁻¹

translations (*type*) *'a*⁻³ <= (*type*) (*'a*³)⁻¹

translations (*type*) *'a*⁻⁴ <= (*type*) (*'a*⁴)⁻¹

⟨ML⟩

Derived Dimension Types

type-synonym *Area* = *L*²

type-synonym *Volume* = *L*³

type-synonym *Acceleration* = *L*·*T*⁻¹

type-synonym *Frequency* = *T*⁻¹

type-synonym *Energy* = *L*²·*M*·*T*⁻²

type-synonym *Power* = *L*²·*M*·*T*⁻³

type-synonym *Force* = *L*·*M*·*T*⁻²

type-synonym *Pressure* = *L*⁻¹·*M*·*T*⁻²

type-synonym *Charge* = *I*·*T*

type-synonym *PotentialDifference* = *L*²·*M*·*T*⁻³·*I*⁻¹

type-synonym *Capacitance* = *L*⁻²·*M*⁻¹·*T*⁴·*I*²

3.1.6 ML Functions

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

⟨ML⟩

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 $x = y$

$\langle proof \rangle$

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 $\langle proof \rangle$

end

lemma *mag-times* $[simp]: mag\ (x \cdot y) = mag\ x \cdot mag\ y\ \langle proof \rangle$

lemma *dim-times* $[simp]: dim\ (x \cdot y) = dim\ x \cdot dim\ y\ \langle proof \rangle$

lemma *more-times* $[simp]: more\ (x \cdot y) = more\ x \cdot more\ y\ \langle proof \rangle$

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

instantiation *Quantity-ext* :: $(zero, enum, zero)\ zero$

begin

definition *zero-Quantity-ext* = $(mag = 0, dim = 1, \dots = 0)$

instance $\langle proof \rangle$

end

lemma *mag-zero* $[simp]: mag\ 0 = 0\ \langle proof \rangle$

lemma *dim-zero* $[simp]: dim\ 0 = 1\ \langle proof \rangle$

lemma *more-zero* $[simp]: more\ 0 = 0\ \langle proof \rangle$

instantiation *Quantity-ext* :: $(one, enum, one)\ one$

begin

definition $[si-def]: one-Quantity-ext = (mag = 1, dim = 1, \dots = 1)$

instance $\langle proof \rangle$

end

lemma *mag-one* $[simp]: mag\ 1 = 1\ \langle proof \rangle$

lemma *dim-one* $[simp]: dim\ 1 = 1\ \langle proof \rangle$

lemma *more-one* [*simp*]: *more* 1 = 1 \langle *proof* \rangle

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 = (\lfloor *mag* = *inverse* (*mag* x), *dim* = *inverse* (*dim* x), ... = *inverse* (*more* x) \rfloor)

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

[*si-def*]: *divide-Quantity-ext* x y = (\lfloor *mag* = *mag* x / *mag* y, *dim* = *dim* x / *dim* y, ... = *more* x / *more* y \rfloor)

instance \langle *proof* \rangle

end

lemma *mag-inverse* [*simp*]: *mag* (*inverse* x) = *inverse* (*mag* x) \langle *proof* \rangle

lemma *dim-inverse* [*simp*]: *dim* (*inverse* x) = *inverse* (*dim* x) \langle *proof* \rangle

lemma *more-inverse* [*simp*]: *more* (*inverse* x) = *inverse* (*more* x) \langle *proof* \rangle

lemma *mag-divide* [*simp*]: *mag* (x / y) = *mag* x / *mag* y \langle *proof* \rangle

lemma *dim-divide* [*simp*]: *dim* (x / y) = *dim* x / *dim* y \langle *proof* \rangle

lemma *more-divide* [*simp*]: *more* (x / y) = *more* x / *more* y \langle *proof* \rangle

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* \langle *proof* \rangle

instance *Quantity-ext* :: (*ab-group-mult*, *enum*, *ab-group-mult*) *ab-group-mult* \langle *proof* \rangle

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* $x\ y = (\text{mag } x \leq \text{mag } y \wedge \text{dim } x = \text{dim } y \wedge \text{more } x \leq \text{more } y)$

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

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

instance $\langle \text{proof} \rangle$

end

instance *Quantity-ext* $:: (\text{order}, \text{enum}, \text{order})\ \text{order}$
 $\langle \text{proof} \rangle$

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* $:: (\text{plus}, \text{enum}, \text{plus})\ \text{plus}$

begin

definition *plus-Quantity-ext* $:: ('a, 'b, 'c)\ \text{Quantity-scheme} \Rightarrow ('a, 'b, 'c)\ \text{Quantity-scheme} \Rightarrow ('a, 'b, 'c)\ \text{Quantity-scheme}$

where $[si-def]:$

$\text{dim } x = \text{dim } y \Longrightarrow$

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

instance $\langle \text{proof} \rangle$

end

instantiation *Quantity-ext* $:: (\text{uminus}, \text{enum}, \text{uminus})\ \text{uminus}$

begin

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

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

instance $\langle \text{proof} \rangle$

end

instantiation *Quantity-ext* $:: (\text{minus}, \text{enum}, \text{minus})\ \text{minus}$

begin

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

$[si-def]:$

$\text{dim } x = \text{dim } y \Longrightarrow$

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

instance $\langle \text{proof} \rangle$

end

3.2.2 Measurement Systems

class *unit-system* = *unitary*

lemma *unit-system-intro*: $(UNIV::'s\ set) = \{a\} \implies OFCLASS('s, unit-system-class)$
 $\langle proof \rangle$

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*]: $mmore (\ unit-sys = x, \dots = y) = y$
 $\langle proof \rangle$

lemma *mmore-ext* [*simp*]: $(\ unit-sys = unit, \dots = mmore\ a) = a$
 $\langle proof \rangle$

lemma *Measurement-System-eq-intro*:
assumes $mag\ x = mag\ y\ dim\ x = dim\ y\ more\ x = more\ y$
shows $x = y$
 $\langle proof \rangle$

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 $\langle proof \rangle$

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 $\langle proof \rangle$

end

instantiation *Measurement-System-ext* :: $(unit-system, times) times$

begin

definition *times-Measurement-System-ext* ::

$('a, 'b) Measurement-System-ext \Rightarrow ('a, 'b) Measurement-System-ext \Rightarrow ('a, 'b) Measurement-System-ext$

where [*si-def*]: $times-Measurement-System-ext\ x\ y = (\ unit-sys = unit, \dots = mmore\ x \cdot mmore\ y)$

instance $\langle proof \rangle$

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* $x = (\mid \text{unit-sys} = \text{unit}, \dots = \text{inverse} \text{ mmore } x \mid)$

definition *divide-Measurement-System-ext* ::

$('a, 'b) \text{ Measurement-System-ext} \Rightarrow ('a, 'b) \text{ Measurement-System-ext} \Rightarrow ('a, 'b) \text{ Measurement-System-ext}$

where [*si-def*]: *divide-Measurement-System-ext* $x y = (\mid \text{unit-sys} = \text{unit}, \dots = \text{mmore } x / \text{mmore } y \mid)$

instance $\langle \text{proof} \rangle$

end

instance *Measurement-System-ext* :: $(\text{unit-system}, \text{comm-monoid-mult}) \text{ comm-monoid-mult}$
 $\langle \text{proof} \rangle$

instance *Measurement-System-ext* :: $(\text{unit-system}, \text{ab-group-mult}) \text{ ab-group-mult}$
 $\langle \text{proof} \rangle$

instantiation *Measurement-System-ext* :: $(\text{unit-system}, \text{ord}) \text{ ord}$

begin

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

where *less-eq-Measurement-System-ext* $x y = (\text{mmore } x \leq \text{mmore } y)$

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

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

instance $\langle \text{proof} \rangle$

end

instance *Measurement-System-ext* :: $(\text{unit-system}, \text{order}) \text{ order}$
 $\langle \text{proof} \rangle$

instantiation *Measurement-System-ext* :: $(\text{unit-system}, \text{plus}) \text{ plus}$

begin

definition *plus-Measurement-System-ext* ::

$('a, 'b) \text{ Measurement-System-ext} \Rightarrow ('a, 'b) \text{ Measurement-System-ext} \Rightarrow ('a, 'b) \text{ Measurement-System-ext}$

where [*si-def*]:

plus-Measurement-System-ext $x y = (\mid \text{unit-sys} = \text{unit}, \dots = \text{mmore } x + \text{mmore } y \mid)$

instance $\langle \text{proof} \rangle$

end

instantiation *Measurement-System-ext* :: $(\text{unit-system}, \text{uminus}) \text{ uminus}$

begin

definition *uminus-Measurement-System-ext* :: $('a, 'b) \text{ Measurement-System-ext} \Rightarrow ('a, 'b) \text{ Measurement-System-ext}$ **where**

[*si-def*]: *uminus-Measurement-System-ext* $x = (\mid \text{unit-sys} = \text{unit}, \dots = - \text{mmore } x \mid)$

instance $\langle \text{proof} \rangle$

end

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

begin

definition *minus-Measurement-System-ext* ::

(*'a*, *'b*) *Measurement-System-ext* \Rightarrow (*'a*, *'b*) *Measurement-System-ext* \Rightarrow (*'a*, *'b*)

Measurement-System-ext **where**

[*si-def*]:

minus-Measurement-System-ext *x y* = (*unit-sys* = *unit*, ... = *mmore* *x* –
mmore *y*)

instance *<proof>*

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* (\langle -[, -] \rangle [*999*, *0*, *0*]
999)

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

morphisms *fromQ* *toQ* *<proof>*

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 *itself* \Rightarrow *'a*[*'d*₁::*dim-type*, *'s*::*unit-system*] \Rightarrow *'a*[*'d*₂::*dim-type*,
's] **where**

[*si-def*]: *QD*(*'d*₁) = *QD*(*'d*₂) \implies *coerceQuantT* *t x* = (*toQ* (*fromQ* *x*))

syntax

-*QCOERCE* :: *type* \Rightarrow *logic* \Rightarrow *logic* (\langle *QCOERCE*[*-*] \rangle)

syntax-consts

-*QCOERCE* == *coerceQuantT*

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** $\langle \lesssim_Q \rangle$ 50)
is (\leq) $\langle proof \rangle$

lift-definition $qequiv :: 'n['a::dim\text{-}type, 's::unit\text{-}system] \Rightarrow 'n['b::dim\text{-}type, 's] \Rightarrow bool$ (**infix** $\langle \cong_Q \rangle$ 50)
is ($=$) $\langle proof \rangle$

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 $qequiv\text{-}refl$ [$simp$]: $a \cong_Q a$
 $\langle proof \rangle$

lemma $qequiv\text{-}sym$: $a \cong_Q b \implies b \cong_Q a$
 $\langle proof \rangle$

lemma $qequiv\text{-}trans$: $\llbracket a \cong_Q b; b \cong_Q c \rrbracket \implies a \cong_Q c$
 $\langle proof \rangle$

theorem $qeq\text{-}iff\text{-}same\text{-}dim$:
fixes $x\ y :: 'a['d::dim\text{-}type, 's::unit\text{-}system]$
shows $x \cong_Q y \longleftrightarrow x = y$
 $\langle proof \rangle$

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 $(coerceQuantT\ TYPE('d_2)\ x) \cong_Q x$
 $\langle proof \rangle$

lemma $coerceQuant\text{-}eq\text{-}iff2$:
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 $x \cong_Q y$
 $\langle proof \rangle$

lemma $updown\text{-}eq\text{-}iff$:
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 $x \cong_Q y$

<proof>

This is more general than $y = x \implies x \cong_Q y$, since x and y may have different type.

lemma *qeq*:

fixes $x :: 'a['d_1::dim-type, 's::unit-system]$ **fixes** $y :: 'a['d_2::dim-type, 's]$
assumes $x \cong_Q y$
shows $QD('d_1) = QD('d_2)$
<proof>

3.2.5 Operators on Typed Quantities

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

lift-definition

qtimes :: $('n::comm-ring-1)['a::dim-type, 's::unit-system] \Rightarrow 'n['b::dim-type, 's] \Rightarrow 'n['a \cdot 'b, 's]$ (**infixl** $\langle \cdot \rangle$ 69)
is $(*)$ *<proof>*

lift-definition

qinverse :: $('n::field)['a::dim-type, 's::unit-system] \Rightarrow 'n['a^{-1}, 's]$ ($\langle (-^{-1}) \rangle$ [999]
999)
is *inverse* *<proof>*

abbreviation (*input*)

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

We also provide some helpful notations for expressing heterogeneous powers.

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

abbreviation *qneg-sq* $\langle (-)^{-2} \rangle$ [999] 999 **where** $u^{-2} \equiv (u^2)^{-1}$
abbreviation *qneg-cube* $\langle (-)^{-3} \rangle$ [999] 999 **where** $u^{-3} \equiv (u^3)^{-1}$
abbreviation *qneg-quart* $\langle (-)^{-4} \rangle$ [999] 999 **where** $u^{-4} \equiv (u^4)^{-1}$

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

lift-definition *scaleQ* :: $'a \Rightarrow 'a::comm-ring-1['d::dim-type, 's::unit-system] \Rightarrow 'a['d, 's]$ (**infixr** $\langle *_Q \rangle$ 63)
is $\lambda r x. (mag = r * mag x, dim = QD('d), unit-sys = unit)$ *<proof>*

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

```
instantiation QuantT :: (zero, dim-type, unit-system) zero
begin
lift-definition zero-QuantT :: ('a, 'b, 'c) QuantT is (| mag = 0, dim = QD('b),
unit-sys = unit |)
  <proof>
instance <proof>
end
```

```
instantiation QuantT :: (one, dim-type, unit-system) one
begin
lift-definition one-QuantT :: ('a, 'b, 'c) QuantT is (| mag = 1, dim = QD('b),
unit-sys = unit |)
  <proof>
instance <proof>
end
```

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

```
abbreviation gone :: 'n::one[1, 's::unit-system] (<1>) where gone ≡ 1
```

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

```
instantiation QuantT :: (plus, dim-type, unit-system) plus
begin
lift-definition plus-QuantT :: 'a['b, 'c] ⇒ 'a['b, 'c] ⇒ 'a['b, 'c]
  is λ x y. (| mag = mag x + mag y, dim = QD('b), unit-sys = unit |)
  <proof>
instance <proof>
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
  <proof>
```

```
instance QuantT :: (ab-semigroup-add, dim-type, unit-system) ab-semigroup-add
  <proof>
```

```
instance QuantT :: (monoid-add, dim-type, unit-system) monoid-add
  <proof>
```

```
instance QuantT :: (comm-monoid-add, dim-type, unit-system) comm-monoid-add
```

<proof>

instantiation *QuantT* :: (*uminus,dim-type,unit-system*) *uminus*

begin

lift-definition *uminus-QuantT* :: 'a['b,'c] ⇒ 'a['b,'c]

is λ x. (| *mag* = − *mag* x, *dim* = *dim* x, *unit-sys* = *unit* |) *<proof>*

instance *<proof>*

end

instantiation *QuantT* :: (*minus,dim-type,unit-system*) *minus*

begin

lift-definition *minus-QuantT* :: 'a['b,'c] ⇒ 'a['b,'c] ⇒ 'a['b,'c]

is λ x y. (| *mag* = *mag* x − *mag* y, *dim* = *dim* x, *unit-sys* = *unit* |) *<proof>*

instance *<proof>*

end

instance *QuantT* :: (*numeral,dim-type,unit-system*) *numeral* *<proof>*

Moreover, types quantities also form an additive group.

instance *QuantT* :: (*ab-group-add,dim-type,unit-system*) *ab-group-add*

<proof>

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] ⇒ 'a['b,'c] ⇒ bool **is** λ x y. *mag* x ≤ *mag* y *<proof>*

lift-definition *less-QuantT* :: 'a['b,'c] ⇒ 'a['b,'c] ⇒ bool **is** λ x y. *mag* x < *mag* y *<proof>*

instance *<proof>*

end

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

<proof>

instantiation *QuantT* :: (*scaleR,dim-type,unit-system*) *scaleR*

begin

lift-definition *scaleR-QuantT* :: real ⇒ 'a['b,'c] ⇒ 'a['b,'c]

is λ n q. (| *mag* = n *_R *mag* q, *dim* = *dim* q, *unit-sys* = *unit* |) *<proof>*

instance *<proof>*

end

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

<proof>

instantiation *QuantT* :: (*norm,dim-type,unit-system*) *norm*

begin

lift-definition *norm-QuantT* :: 'a['b,'c] ⇒ real

```

is  $\lambda x. \text{norm } (\text{mag } x) \langle \text{proof} \rangle$ 
instance  $\langle \text{proof} \rangle$ 
end

```

```

instantiation  $\text{QuantT} :: (\text{sgn-div-norm}, \text{dim-type}, \text{unit-system}) \text{sgn-div-norm}$ 
begin
definition  $\text{sgn-QuantT} :: 'a['b, 'c] \Rightarrow 'a['b, 'c] \text{ where}$ 
 $\text{sgn-QuantT } x = x /_R \text{norm } x$ 
instance  $\langle \text{proof} \rangle$ 
end

```

```

instantiation  $\text{QuantT} :: (\text{dist-norm}, \text{dim-type}, \text{unit-system}) \text{dist-norm}$ 
begin
definition  $\text{dist-QuantT} :: 'a['b, 'c] \Rightarrow 'a['b, 'c] \Rightarrow \text{real} \text{ where}$ 
 $\text{dist-QuantT } x \ y = \text{norm } (x - y)$ 
instance
 $\langle \text{proof} \rangle$ 
end

```

```

instantiation  $\text{QuantT} :: (\{\text{uniformity-dist}, \text{dist-norm}\}, \text{dim-type}, \text{unit-system}) \text{uni-}$ 
 $\text{formity-dist}$ 
begin
definition  $\text{uniformity-QuantT} :: ('a['b, 'c] \times 'a['b, 'c]) \text{ filter} \text{ where}$ 
 $\text{uniformity-QuantT} = (\text{INF } e \in \{0 <.. \}. \text{principal } \{(x, y). \text{dist } x \ y < e\})$ 
instance
 $\langle \text{proof} \rangle$ 
end

```

```

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

```

open-uniformity

```

begin

```

```

definition  $\text{open-QuantT} :: ('a['b, 'c]) \text{ set} \Rightarrow \text{bool} \text{ where}$ 
 $\text{open-QuantT } U = (\forall x \in U. \text{eventually } (\lambda(x', y). x' = x \longrightarrow y \in U) \text{uniformity})$ 
instance  $\langle \text{proof} \rangle$ 
end

```

Quantities form a real normed vector space.

```

instance  $\text{QuantT} :: (\text{real-normed-vector}, \text{dim-type}, \text{unit-system}) \text{real-normed-vector}$ 
 $\langle \text{proof} \rangle$ 

```

```

end

```

3.3 Proof Support for Quantities

```

theory ISQ-Proof
  imports ISQ-Quantities
begin

```

named-theorems *si-transfer*

definition $\text{mag}Q :: 'a['u::\text{dim-type}, 's::\text{unit-system}] \Rightarrow 'a (\langle \llbracket - \rrbracket_Q \rangle)$ **where**
[si-def]: $\text{mag}Q x = \text{mag} (\text{from}Q x)$

definition $\text{dim}Q :: 'a['u::\text{dim-type}, 's::\text{unit-system}] \Rightarrow \text{Dimension}$ **where**
[si-def]: $\text{dim}Q x = \text{dim} (\text{from}Q x)$

lemma *quant-eq-iff-mag-eq* [*si-eq*]:
 $x = y \longleftrightarrow \llbracket x \rrbracket_Q = \llbracket y \rrbracket_Q$
<proof>

lemma *quant-eqI* [*si-transfer*]:
 $\llbracket x \rrbracket_Q = \llbracket y \rrbracket_Q \Longrightarrow x = y$
<proof>

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

lemma *quant-equivI* [*si-transfer*]:
fixes $x :: 'a['u_1::\text{dim-type}, 's::\text{unit-system}]$ **and** $y :: 'a['u_2::\text{dim-type}, 's::\text{unit-system}]$
assumes $QD('u_1) = QD('u_2)$ $QD('u_1) = QD('u_2) \Longrightarrow \llbracket x \rrbracket_Q = \llbracket y \rrbracket_Q$
shows $x \cong_Q y$
<proof>

lemma *quant-le-iff-magn-le* [*si-eq*]:
 $x \leq y \longleftrightarrow \llbracket x \rrbracket_Q \leq \llbracket y \rrbracket_Q$
<proof>

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

lemma *quant-less-iff-magn-less* [*si-eq*]:
 $x < y \longleftrightarrow \llbracket x \rrbracket_Q < \llbracket y \rrbracket_Q$
<proof>

lemma *quant-lessI* [*si-transfer*]:
 $\llbracket x \rrbracket_Q < \llbracket y \rrbracket_Q \Longrightarrow x < y$
<proof>

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

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

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

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

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

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

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

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

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

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

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 $\llbracket \text{coerceQuantT } t \ q \rrbracket_Q = \llbracket q \rrbracket_Q$
 ⟨*proof*⟩

lemma *dimQ* [*simp*]: $\text{dimQ}(x :: 'a['d::\text{dim-type}, 's::\text{unit-system}]) = QD('d)$
 ⟨*proof*⟩

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

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

The next tactic additionally compiles the semantics of the underlying units

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

lemma $QD(N \cdot \Theta \cdot N) = QD(\Theta \cdot N^2)$ ⟨*proof*⟩

end

3.4 Algebraic Laws

```
theory ISQ-Algebra
  imports ISQ-Proof
begin
```

3.4.1 Quantity Scale

lemma *scaleQ-add-right*: $a *_Q x + y = (a *_Q x) + (a *_Q y)$
 ⟨proof⟩

lemma *scaleQ-add-left*: $a + b *_Q x = (a *_Q x) + (b *_Q x)$
 ⟨proof⟩

lemma *scaleQ-scaleQ* [simp]: $a *_Q b *_Q x = a \cdot b *_Q x$
 ⟨proof⟩

lemma *scaleQ-one* [simp]: $1 *_Q x = x$
 ⟨proof⟩

lemma *scaleQ-zero* [simp]: $0 *_Q x = 0$
 ⟨proof⟩

lemma *scaleQ-inv*: $-a *_Q x = a *_Q -x$
 ⟨proof⟩

lemma *scaleQ-as-qprod*: $a *_Q x \cong_Q (a *_Q \mathbf{1}) \cdot x$
 ⟨proof⟩

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

lemma *mult-scaleQ-right* [simp]: $x \cdot (a *_Q y) = a *_Q x \cdot y$
 ⟨proof⟩

3.4.2 Field Laws

lemma *qtimes-commute*: $x \cdot y \cong_Q y \cdot x$
 ⟨proof⟩

lemma *qtimes-assoc*: $(x \cdot y) \cdot z \cong_Q x \cdot (y \cdot z)$
 ⟨proof⟩

lemma *qtimes-left-unit*: $\mathbf{1} \cdot x \cong_Q x$
 ⟨proof⟩

lemma *qtimes-right-unit*: $x \cdot \mathbf{1} \cong_Q x$
 ⟨proof⟩

The following weak congruences will allow for replacing equivalences in con-

texts built from product and inverse.

lemma *qtimes-weak-cong-left*:

assumes $x \cong_Q y$
shows $x \cdot z \cong_Q y \cdot z$
<proof>

lemma *qtimes-weak-cong-right*:

assumes $x \cong_Q y$
shows $z \cdot x \cong_Q z \cdot y$
<proof>

lemma *qinverse-weak-cong*:

assumes $x \cong_Q y$
shows $x^{-1} \cong_Q y^{-1}$
<proof>

lemma *scaleQ-cong*:

assumes $y \cong_Q z$
shows $x *_Q y \cong_Q x *_Q z$
<proof>

lemma *qinverse-qinverse*: $x^{-1-1} \cong_Q x$

<proof>

lemma *qinverse-nonzero-iff-nonzero*: $x^{-1} = 0 \longleftrightarrow x = 0$

<proof>

lemma *qinverse-qtimes*: $(x \cdot y)^{-1} \cong_Q x^{-1} \cdot y^{-1}$

<proof>

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

<proof>

lemma *qtimes-cancel*: $x \neq 0 \implies x / x \cong_Q \mathbf{1}$

<proof>

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. 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 $\lambda x. \text{mag } x = 1 \wedge \text{is-BaseDim } (\text{dim } x) \langle \text{proof} \rangle$

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 ($\langle \text{BUNIT}'(-, -)' \rangle$)

syntax-consts *-mk-base-unit* == *mk-base-unit*

translations *BUNIT*('a, 's) == *CONST* *mk-base-unit* *TYPE*('a) *TYPE*('s)

lemma *mk-base-unit: is-base-unit* (*mk-base-unit* a s)
 $\langle \text{proof} \rangle$

lemma *magQ-mk [si-eq]*: $\llbracket \text{BUNIT}'(\text{'u}::\text{basedim-type}, \text{'s}::\text{unit-system}) \rrbracket_Q = 1$
 $\langle \text{proof} \rangle$

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* ($\langle (-/ \Rightarrow_U -) \rangle [1, 0] 0$) =
 $\{c :: \text{ConvSchema}. \text{cLengthF } c > 0 \wedge \text{cMassF } c > 0 \wedge \text{cTimeF } c > 0 \wedge \text{cCurrentF } c > 0$
 $\wedge \text{cTemperatureF } c > 0 \wedge \text{cAmountF } c > 0 \wedge \text{cIntensityF } c > 0\}$
 $\langle \text{proof} \rangle$

setup-lifting *type-definition-Conversion*

lift-definition $LengthF$ $:: ('s_1::unit-system \Rightarrow_U 's_2::unit-system) \Rightarrow rat$ is
 $cLengthF$ $\langle proof \rangle$

lift-definition $MassF$ $:: ('s_1::unit-system \Rightarrow_U 's_2::unit-system) \Rightarrow rat$ is
 $cMassF$ $\langle proof \rangle$

lift-definition $TimeF$ $:: ('s_1::unit-system \Rightarrow_U 's_2::unit-system) \Rightarrow rat$ is
 $cTimeF$ $\langle proof \rangle$

lift-definition $CurrentF$ $:: ('s_1::unit-system \Rightarrow_U 's_2::unit-system) \Rightarrow rat$ is
 $cCurrentF$ $\langle proof \rangle$

lift-definition $TemperatureF$ $:: ('s_1::unit-system \Rightarrow_U 's_2::unit-system) \Rightarrow rat$ is
 $cTemperatureF$ $\langle proof \rangle$

lift-definition $AmountF$ $:: ('s_1::unit-system \Rightarrow_U 's_2::unit-system) \Rightarrow rat$ is
 $cAmountF$ $\langle proof \rangle$

lift-definition $IntensityF$ $:: ('s_1::unit-system \Rightarrow_U 's_2::unit-system) \Rightarrow rat$ is
 $cIntensityF$ $\langle proof \rangle$

lemma *Conversion-props* $[simp]: LengthF\ c > 0\ MassF\ c > 0\ TimeF\ c > 0\ CurrentF\ c > 0$
 $TemperatureF\ c > 0\ AmountF\ c > 0\ IntensityF\ c > 0$
 $\langle proof \rangle$

3.6.2 Conversion Algebra

lift-definition $convid :: 's::unit-system \Rightarrow_U 's (\langle id_C \rangle)$
is

$(\langle cLengthF = 1$
 $, cMassF = 1$
 $, cTimeF = 1$
 $, cCurrentF = 1$
 $, cTemperatureF = 1$
 $, cAmountF = 1$
 $, cIntensityF = 1 \rangle \langle proof \rangle$

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** $\langle \circ_C \rangle$ 55) **is**
 $\lambda\ c_1\ c_2. (\langle cLengthF = cLengthF\ c_1 * cLengthF\ c_2, cMassF = cMassF\ c_1 * cMassF\ c_2$
 $, cTimeF = cTimeF\ c_1 * cTimeF\ c_2, cCurrentF = cCurrentF\ c_1 * cCurrentF\ c_2$
 $, cTemperatureF = cTemperatureF\ c_1 * cTemperatureF\ c_2$
 $, cAmountF = cAmountF\ c_1 * cAmountF\ c_2, cIntensityF = cIntensityF\ c_1$
 $* cIntensityF\ c_2 \rangle$
 $\langle proof \rangle$

lift-definition $convinv :: ('s_1::unit-system \Rightarrow_U 's_2::unit-system) \Rightarrow ('s_2 \Rightarrow_U 's_1)$
 $(\langle inv_C \rangle)$ **is**
 $\lambda\ c. (\langle cLengthF = inverse\ (cLengthF\ c), cMassF = inverse\ (cMassF\ c), cTimeF = inverse\ (cTimeF\ c)$

, $cCurrentF = inverse (cCurrentF c)$, $cTemperatureF = inverse (cTemperatureF c)$
 $cAmountF = inverse (cAmountF c)$, $cIntensityF = inverse (cIntensityF c)$
) *<proof>*

lemma *convinv-inverse* [simp]: $convinv (convinv c) = c$
<proof>

lemma *convcomp-inv* [simp]: $c \circ_C inv_C c = id_C$
<proof>

lemma *inv-convcomp* [simp]: $inv_C c \circ_C c = id_C$
<proof>

lemma *Conversion-invs* [simp]: $LengthF (inv_C x) = inverse (LengthF x) MassF (inv_C x) = inverse (MassF x)$
 $TimeF (inv_C x) = inverse (TimeF x) CurrentF (inv_C x) = inverse (CurrentF x)$
 $TemperatureF (inv_C x) = inverse (TemperatureF x) AmountF (inv_C x) = inverse (AmountF x)$
 $IntensityF (inv_C x) = inverse (IntensityF x)$
<proof>

lemma *Conversion-comps* [simp]: $LengthF (c_1 \circ_C c_2) = LengthF c_1 * LengthF c_2$
 $MassF (c_1 \circ_C c_2) = MassF c_1 * MassF c_2$
 $TimeF (c_1 \circ_C c_2) = TimeF c_1 * TimeF c_2$
 $CurrentF (c_1 \circ_C c_2) = CurrentF c_1 * CurrentF c_2$
 $TemperatureF (c_1 \circ_C c_2) = TemperatureF c_1 * TemperatureF c_2$
 $AmountF (c_1 \circ_C c_2) = AmountF c_1 * AmountF c_2$
 $IntensityF (c_1 \circ_C c_2) = IntensityF c_1 * IntensityF c_2$
<proof>

3.6.3 Conversion Functions

definition *dconvfactor* :: $(s_1::unit-system \Rightarrow_U s_2::unit-system) \Rightarrow Dimension \Rightarrow rat$ where

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

lemma *dconvfactor-pos* [simp]: $dconvfactor c d > 0$
<proof>

lemma *dconvfactor-nz* [simp]: $dconvfactor c d \neq 0$

<proof>

lemma *dconvfactor-convinv*: $dconvfactor (convinv\ c)\ d = inverse (dconvfactor\ c\ d)$

<proof>

lemma *dconvfactor-id [simp]*: $dconvfactor\ id_C\ d = 1$

<proof>

lemma *dconvfactor-compose*:

$dconvfactor (c_1 \circ_C c_2)\ d = dconvfactor\ c_1\ d * dconvfactor\ c_2\ d$

<proof>

lemma *dconvfactor-inverse*:

$dconvfactor\ c (inverse\ d) = inverse (dconvfactor\ c\ d)$

<proof>

lemma *dconvfactor-times*:

$dconvfactor\ c (x \cdot y) = dconvfactor\ c\ x \cdot dconvfactor\ c\ y$

<proof>

lift-definition *qconv* :: $(s_1, s_2)\ Conversion \Rightarrow (a::field-char-0)[d::dim-type, s_1::unit-system] \Rightarrow a[d, s_2::unit-system]$

is $\lambda\ c\ q. (\downarrow\ mag = of-rat (dconvfactor\ c (dim\ q)) * mag\ q, dim = dim\ q, unit-sys = unit) \downarrow$ *<proof>*

lemma *magQ-qconv*: $\llbracket qconv\ c\ q \rrbracket_Q = of-rat (dconvfactor\ c (dimQ\ q)) * \llbracket q \rrbracket_Q$

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *qconv-qinverse* [*simp*]: $qconv\ c\ (x^{-1}) = (qconv\ c\ x)^{-1}$
 ⟨*proof*⟩

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

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

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

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

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

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

lemma *qconv-Intensity* [*simp*]: $qconv\ c\ BUNIT(J, -) = IntensityF\ c\ *_Q\ BUNIT(J, -)$
 ⟨*proof*⟩

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 <proof>
```

```
instance SI :: unit-system
  <proof>
```

```
abbreviation SI ≡ unit :: SI
```

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

We now define the seven base units. Effectively, these definitions axiomatise given names for the 1 elements of the base quantities.

```
abbreviation metre ≡ BUNIT(L, SI)
abbreviation kilogram ≡ BUNIT(M, SI)
abbreviation ampere ≡ BUNIT(I, SI)
abbreviation kelvin ≡ BUNIT(Θ, SI)
abbreviation mole ≡ BUNIT(N, SI)
abbreviation candela ≡ BUNIT(J, SI)
```

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

```
class time-second = unit-system
```

```
instance SI :: time-second <proof>
```

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
 ⟨*proof*⟩

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* $\cong_{\mathcal{Q}}$ *metre*
 ⟨*proof*⟩

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 |) ⟨*proof*⟩

instance $\langle proof \rangle$
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 $qmconv ::$
 $'s_1 \text{ itself} \Rightarrow 's_2 \text{ itself}$
 $\Rightarrow ('a::field-char-0)['d::dim-type, 's_1::metrifiable]$
 $\Rightarrow 'a['d::dim-type, 's_2::metrifiable]$ **where**
 $qmconv s_1 s_2 x \equiv qconv (inv_C (schema_C s_2) \circ_C schema_C s_1) x$

syntax
 $-qmconv :: type \Rightarrow type \Rightarrow logic (\langle QMC'(- \rightarrow -) \rangle)$

syntax-consts
 $-qmconv == qmconv$

translations
 $QMC('s_1 \rightarrow 's_2) == CONST qmconv TYPE('s_1) TYPE('s_2)$

lemma $qmconv-self: QMC('s::metrifiable \rightarrow 's) = id$
 $\langle proof \rangle$

end

4.2 Centimetre-Gram-Second System

theory CGS
imports $SI-Units$
begin

4.2.1 Preliminaries

typedef $CGS = UNIV :: unit \text{ set} \langle proof \rangle$
instance $CGS :: unit\text{-system}$
 $\langle proof \rangle$
instance $CGS :: time\text{-second} \langle proof \rangle$
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*

$\lambda x. (\ cLengthF = 0.01, cMassF = 0.001, cTimeF = 1$

$\ , cCurrentF = 1, cTemperatureF = 1, cAmountF = 1, cIntensityF = 1 \)$

$\langle proof \rangle$

instance $\langle proof \rangle$

end

lemma *CGS-SI-simps* [*simp*]: $LengthF (convschema (a::CGS\ itself)) = 0.01\ MassF$

$(convschema\ a) = 0.001$

$TimeF (convschema\ a) = 1\ CurrentF (convschema\ a) = 1\ TemperatureF (convschema$

$a) = 1$

$\langle proof \rangle$

4.2.4 Conversion Examples

lemma *metrify* ($(100::rat) *Q\ centimetre$) = $1 *Q\ metre$

$\langle proof \rangle$

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* $\equiv metre \cdot metre^{-1}$

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

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

type-synonym 'a *joule* = 'a[M · L² · T⁻², SI]

abbreviation *watt* $\equiv kilogram \cdot metre^2 \cdot second^{-3}$

type-synonym 'a *watt* = 'a[M · L² · T⁻³, SI]

abbreviation *coulomb* $\equiv ampere \cdot second$

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

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

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

4.3.2 Constants

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

default-sort *field-char-0*

Hyperfine transition frequency of frequency of Cs

abbreviation *caesium-frequency*:: *'a*[T^{-1} ,*SI*] ($\langle \Delta v_{Cs} \rangle$) **where**
caesium-frequency \equiv $9192631770 *_{\mathcal{Q}}$ *hertz*

Speed of light in vacuum

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

Elementary charge

abbreviation *elementary-charge* :: *'a*[$I \cdot T$,*SI*] ($\langle e \rangle$) **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*] ($\langle k \rangle$) **where**
Boltzmann \equiv ($1.380649 \cdot 1/(10^{23})$) $*_{\mathcal{Q}}$ (*joule / kelvin*)

The Avogadro number

abbreviation *Avogadro* :: *'a*[N^{-1} ,*SI*] ($\langle N_A \rangle$) **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*] ($\langle K_{cd} \rangle$)
where
luminous-efficacy \equiv $683 *_{\mathcal{Q}}$ (*lumen/watt*)

4.3.3 Checking Foundational Equations of the SI System

theorem *second-definition:*

$$1 *_Q \text{ second} \cong_Q (9192631770 *_Q \mathbf{1}) / \Delta v_{Cs}$$

<proof>

theorem *metre-definition:*

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

$$1 *_Q \text{ metre} \cong_Q (9192631770 / 299792458) *_Q (\mathbf{c} / \Delta v_{Cs})$$

<proof>

theorem *kilogram-definition:*

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

<proof>

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

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 ($*_Q$).

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 $2.3 *_Q (\text{centi} *_Q \text{metre})^3 = 2.3 \cdot 1/10^6 *_Q \text{metre}^3$
 ⟨proof⟩

lemma $1 *_Q (\text{centi} *_Q \text{metre})^{-1} = 100 *_Q \text{metre}^{-1}$
 ⟨proof⟩

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

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

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

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

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

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

definition [*si-eq*]: $gram = milli *_{\mathbb{Q}}\ kilogram$

4.5.2 Equivalences

lemma $joule-alt-def: joule \cong_{\mathbb{Q}}\ newton \cdot metre$
<proof>

lemma $watt-alt-def: watt \cong_{\mathbb{Q}}\ joule / second$
<proof>

lemma $volt-alt-def: volt = watt / ampere$
<proof>

lemma $farad-alt-def: farad \cong_{\mathbb{Q}}\ coulomb / volt$
<proof>

lemma $ohm-alt-def: ohm \cong_{\mathbb{Q}}\ volt / ampere$
<proof>

lemma $siemens-alt-def: siemens \cong_{\mathbb{Q}}\ ampere / volt$
<proof>

lemma $weber-alt-def: weber \cong_{\mathbb{Q}}\ volt \cdot second$
<proof>

lemma $tesla-alt-def: tesla \cong_{\mathbb{Q}}\ weber / metre^2$
<proof>

lemma $henry-alt-def: henry \cong_{\mathbb{Q}}\ weber / ampere$
<proof>

lemma $lux-alt-def: lux = lumen / metre^2$
<proof>

lemma $gray-alt-def: gray \cong_{\mathbb{Q}}\ joule / kilogram$
<proof>

lemma *sievert-alt-def*: $\text{sievert} \cong_Q \text{joule} / \text{kilogram}$
 ⟨*proof*⟩

4.5.3 Properties

lemma *kilogram*: $\text{kilo} *_Q \text{gram} = \text{kilogram}$
 ⟨*proof*⟩

lemma *celcius-to-kelvin*: $T_{\text{XXXC}} = (T *_Q \text{kelvin}) + (273.15 *_Q \text{kelvin})$
 ⟨*proof*⟩

end

4.6 Non-SI Units Accepted for SI use

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

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

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

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

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

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

abbreviation *degrees* (‹-XXX› [999] 999) **where** $n_{\text{XXX}} \equiv n *_Q \text{degree}$

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

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

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

4.6.1 Example Unit Equations

lemma $1 *_Q \text{hour} = 3600 *_Q \text{second}$
 ⟨*proof*⟩

lemma $\text{watt} \cdot \text{hour} \cong_Q 3600 *_Q \text{joule}$ ⟨*proof*⟩

lemma $25 *_Q \text{metre} / \text{second} = 90 *_Q (\text{kilo} *_Q \text{metre}) / \text{hour}$
 ⟨*proof*⟩

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*: $foot = 0.3048 *_Q metre$
 ⟨*proof*⟩

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

lemma *inch-alt-def*: $inch = 25.4 *_Q milli *_Q metre$
 ⟨*proof*⟩

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

lemma *mile-alt-def*: $mile = 1609.344 *_Q metre$
 ⟨*proof*⟩

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] (⟨-XXXXF⟩ [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*

⟨*proof*⟩

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

⟨*proof*⟩

lemma *farenheit-to-celcius*: TXXXXF = ((T - 32) · 5/9)XXXXC

⟨*proof*⟩

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*]: $julian-year = 365.25 *_{\mathcal{Q}} day$

definition $light-year :: 'a::field-char-0[L]$ **where**
 $light-year = QCOERCE[L] (\mathbf{c} \cdot julian-year)$

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 $light-year$: $light-year \cong_{\mathcal{Q}} \mathbf{c} \cdot julian-year$
 ⟨*proof*⟩

lemma $light-year-eq$ [*si-eq*]: $\llbracket light-year \rrbracket_{\mathcal{Q}} = \llbracket \mathbf{c} \cdot julian-year \rrbracket_{\mathcal{Q}}$
 ⟨*proof*⟩

HOL can characterise π exactly and so we also give an exact value for the parsec.

definition $parsec :: real[L]$ **where**
 [*si-eq*]: $parsec = 648000 / \pi *_{\mathcal{Q}} astronomical-unit$

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

lemma $parsec-lb$: $3.26 *_{\mathcal{Q}} light-year < parsec$
 ⟨*proof*⟩

lemma $parsec-ub$: $parsec < 3.27 *_{\mathcal{Q}} light-year$
 ⟨*proof*⟩

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

type-synonym $gravitation-field = real^3[L] \Rightarrow (real^3[L \cdot T^{-2}])$

end

4.10 Parsing and Pretty Printing of SI Units

theory *SI-Pretty*
imports *SI*
begin

4.10.1 Syntactic SI Units

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

nonterminal *si*

syntax

```

-si-metre    :: si (⟨m⟩)
-si-kilogram :: si (⟨kg⟩)
-si-second   :: si (⟨s⟩)
-si-ampere   :: si (⟨A⟩)
-si-kelvin   :: si (⟨K⟩)
-si-mole     :: si (⟨mol⟩)
-si-candela  :: si (⟨cd⟩)

-si-square   :: si ⇒ si (⟨(-)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)

```

$-si-invsquare\ x \leq -si-square\ (-si-inverse\ x)$
 $-si-invcube\ x \leq -si-cube\ (-si-inverse\ x)$
 $-si-invquart\ x \leq -si-quart\ (-si-inverse\ x)$

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

4.10.3 Value Notations

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

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

syntax

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

translations

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

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

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

$-si-metre \leq -siq-print\ (CONST\ metre)$
 $-si-kilogram \leq -siq-print\ (CONST\ kilogram)$
 $-si-second \leq -siq-print\ (CONST\ second)$

```

-si-ampere <= -siq-print (CONST ampere)
-si-kelvin <= -siq-print (CONST kelvin)
-si-mole <= -siq-print (CONST mole)
-si-candela <= -siq-print (CONST candela)

```

```

term SI[5, 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 <proof>
instance BIS :: unit-system
  <proof>
instance BIS :: time-second <proof>
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*]: $gallon = 277.421 *_Q inch^3$

definition [*si-eq*]: $quart = 1/4 *_Q gallon$

definition [*si-eq*]: $pint = 1/8 *_Q gallon$

definition [*si-eq*]: $peck = 2 *_Q gallon$

definition [*si-eq*]: $bushel = 8 *_Q gallon$

definition [*si-eq*]: $minute = 60 *_Q second$

definition [*si-eq*]: $hour = 60 *_Q minute$

4.11.4 Conversion to SI

instantiation $BIS :: metrifiable$
begin

lift-definition $convschema-BIS :: BIS\ itself \Rightarrow (BIS, SI)\ Conversion\ is$
 $\lambda x. (\mid cLengthF = 0.9143993, cMassF = 0.453592338, cTimeF = 1$
 $\quad, cCurrentF = 1, cTemperatureF = 5/9, cAmountF = 1, cIntensityF = 1 \mid)$
 $\langle proof \rangle$

instance $\langle proof \rangle$
end

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

4.11.5 Conversion Examples

lemma $metrify (foot :: rat[L, BIS]) = 0.9143993 / 3 *_Q metre$
 $\langle proof \rangle$

lemma $metrify ((70::rat) *_Q mile / hour) = (704087461 / 22500000) *_Q (metre / second)$
 $\langle proof \rangle$

lemma $QMC(CGS \rightarrow BIS) ((1::rat) *_Q centimetre) = 100000 / 9143993 *_Q yard$
 $\langle proof \rangle$

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.