Type systems for programs respecting dimensions

Conor McBride and Fredrik Nordvall-Forsberg

Computer and Information Sciences, University of Strathclyde

Type systems can be used for tracking dimensional consistency of numerical computations: we present an extension from dimensions of scalar quantities to dimensions of vectors and matrices, making use of dependent types from programming language theory. We show that our types are unique, and most general. We further show that we can give straightforward dimensioned types to many common matrix operations such as addition, multiplication, determinants, traces, and fundamental row operations.

Keywords: Units of Measure; Dimensions; Type systems; Dependent types.

1. Introduction

Measurements of physical quantitites naturally come with *dimensions* such as mass, length and time, and so on. The casual conceptual conflation of physical quantities with mere numbers is a standard computational practice resulting in lamentable errors, notoriously such as colliding with Mars [1]. A common approach to addressing this problem is to bundle numbers with units in compound data structures, then check dynamically whether arithmetic operations make sense before performing them. However, that approach pushes to run time what is much better managed at the time of a program's construction. It is far better to render dimensional nonsense unthinkable than merely to promise to complain if it happens.

In the field of programming languages, *type systems* are used to classify programs by the type of data they manipulate, such as integers, booleans, functions from integers to integers, and so on. Type information is used to guarantee once and for ever that certain classes of errors will not happen when running the program, such as feeding an integer to a function expecting a boolean as an input. Types not only police mistakes, but also mentor development of programs, by structuring the space of meaningful subprograms that can be used to compute a particular intermediate result.

For numerical programs, both dimension checking and type checking are important for the overall trust in computed results. Type systems are

natural tools for mechanising the concept of dimensional analysis, yielding a lightweight approach to data integrity for metrology. Indeed, there is already work in this direction, with dimension types implemented in mainstream languages using a plethora of techniques, ranging from static types to dynamic run-time checks (see Bennich-Björkman and McKeever [2] for a comprehensive survey of libraries for dimension and unit checking):

- Microsoft's F# [3] has units of measure built in to static type checking;
- C++'s Boost Units library [4] uses templates to check units statically;
- Java has a proposed API adding classes for dimensioned quantities [5], but run-time casts are inevitable;
- Haskell's type system can now encode basic units of measure as a library [6];
- Python libraries such as Pint [7] cannot do static checking of dimensional correctness, but implement run-time checks instead.

None of the above support static units for compound data structures, such as vectors and matrices, except in overly uniform ways (e.g. matrices of quantities all in the same dimension). Here, we develop a type system for the dimensions of those matrices representing linear transformations between "dimensioned" vector spaces. Hart [8] developed the theory of such matrices, to which we gratefully contribute a rendering as a type system. Since dimensions have an equational theory richer than routinely available in mainstream programming languages (Sec. 2), we will need *dependent types* (Sec. 3) to be accurate about dimensioned matrices and operations thereon (Sec. 4). Griffioen has explored this direction [9], but so far without dependent types, yielding a rigid set of primitive matrix operations.

2. Units of Measure, mathematically

When computing with numbers intended as physical quantities, one cheap way to promote virtue over vice is to ascribe *physical dimensions* to those numbers. It is standard (as in the Système Internationale), to fix a set of *fundamental* dimensions such as length, time and mass, then fix standard units (base units) for each: respectively metre, second and kilogram. We may multiply and divide dimensions, so that velocity is length per time and force is mass times distance per time squared. Formally, we define:

Definition 2.1. The **free abelian group** $\mathbb{G}$ on set G has (i) an *identity element*, 1; (ii) *multiplication*, $\_\cdot\_$, satisfying $1 \cdot x = x$, $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ and $x \cdot y = y \cdot x$; (iii) *inversion*, $\_^{-1}$, satisfying $x^{-1} \cdot x = 1$; (iv) *generators*, $G \subset \mathbb{G}$.

We now take the fundamental dimensions as G and obtain the free abelian group $\mathbb{G}$ of all the dimensions. What makes $\mathbb{G}$ *free* is that every homomorphism (structure-preserving interpretation) from $\mathbb{G}$ to any other abelian group is determined exactly by the interpretation of G. We readily obtain a normal form for dimensions by imposing (arbitrarily) a total order on G, e.g. mass M before length L before time T, etc: any dimension may be given as a finite product of distinct fundamental dimensions, in the chosen order, raised to nonzero integral powers. Hence to check equality of dimensions $d \stackrel{?}{=} d'$, we can reduce d and d' to normal forms $d = \mathsf{M}^{n_0} \cdot \mathsf{L}^{n_1} \cdot \mathsf{T}^{n_2}$, $d' = \mathsf{M}^{n'_0} \cdot \mathsf{L}^{n'_1} \cdot \mathsf{T}^{n'_2}$, and then straightforwardly check equality of the exponents $n_i \stackrel{?}{=} n'_i$, rather than applying the group axioms directly.

The crucial step in making dimension checking part of type checking is to allow *abstract* dimensions[10]: addition is not length-specific, but works in one *arbitrary* dimension, which can stand for any dimension in particular. The type system thus does algebra with unknown dimensions in $\mathbb{G}$. In effect, we add variables as generators: ensuring dimensional consistency amounts to computing homomorphisms (if they exist) to instantiate the variables and solve equations. E.g., if we need $x^2 \cdot \mathsf{L} = y \cdot \mathsf{T}^{-1}$, we should take $y = x^2 \cdot \mathsf{L} \cdot \mathsf{T}$, leaving x to vary freely. Recalling that a homomorphism is determined by its action on generators, the above gives a homomorphism from $\mathbb{G}[x, y]$ to $\mathbb{G}[x]$, where brackets show generator extensions. Not all equations have solutions: there is no z such that $z^2 = \mathsf{L}$. If a solution exists, then there is a solution through which all solutions factor — a fact which allows the retention of most general types for a rich class of programs[11].

3. Dependent type systems

Types in programming languages were invented for the benefit of machines, e.g. to support memory layout, but their real payoff is for humans. In typed languages from last century, types are used to detect and avoid basic errors such as dividing an integer by a string. Types are characterised by the *values* they contain, but their job is now to classify *expressions* as meaningful or otherwise. A meaningful expression is expected to compute to a value of its type, when its free variables are substituted by values of their types. For example, matrix multiplication yields a value only when the input sizes (which are numerical values) match appropriately. To do a proper job classifying matrix multiplication as meaningful, our systems must thus let types talk about values. Such type systems are called **dependent type**

systems[12], as types can depend on values. They give us enough language to be honest about meaningfulness as a *relative* notion. If our types of matrices, $\mathsf{Matrix}(n, m)$, specifies their size, then matrix multiplication can be given type

$$\mathsf{Matrix}(n, m) \times \mathsf{Matrix}(m, k) \to \mathsf{Matrix}(n, k)$$

i.e. insisting that the input sizes are compatible, and determining the output size. In Sec. 4, we will generalise the type of matrices from *sizes* to *dimensions*.

This century, types make an active contribution by offering guidance *during* the program construction process, not just criticism afterwards. E.g., only types $\mathsf{Matrix}(n, n)$ admit an identity matrix, and the type already determines which one. Working interactively in a dependently typed language such as Agda[13], we can always see the sizes of the matrix we want, and the components we have available. The space within which we search for programs is correspondingly smaller and better structured.

Notation 3.1. We write $a \in A$ for the assertion that that the expression a is classified as meaningful in type A. We write Set for the type of small types[a]. In general, we write $B \in A \to \mathsf{Set}$ for the assertion that B is a type depending on a value of type A, i.e. given any $a \in A$, $B(a)$ is a type.

With this notation in place, we can now introduce types for dependent functions and dependent pairs — the workhorses of dependently typed programming. Dependent function types give outputs meaning relative to inputs, and dependent pair types reflect the way choices we are offered later can depend on choices we have made earlier.

Definition 3.1. Given a type A and $B \in A \to \mathsf{Set}$, we write $(x \in A) \to B(x)$ for the type of dependent functions, consisting of functions f with domain A, such that $f(a) \in B(a)$ for every $a \in A$. We may just write a standard function type, $A \to B$, for $(x \in A) \to B$ when x does not occur in B. Nested $\to$ groups to the right, i.e. we write $(x \in A) \to (y \in B(x)) \to C(x, y)$ for $(x \in A) \to ((y \in B(x)) \to C(x, y))$.

[a]There cannot be a type of all types, for the same reason that there cannot be a set of all sets: this would lead to a paradox. Most ordinary types, such as the type of natural numbers, the type of lists with elements from a small type, the type of functions between small types, etc, are small, but, crucially, Set itself is not a small type. Worried readers are invited think of small types simply as sets of ordinary values.

Definition 3.2. Given a type A and $B \in A \to \mathsf{Set}$, we write $(x \in A) \times B(x)$ for the type of dependent pairs (a, b), where $a \in A$ and $b \in B(a)$. We may just write a standard Cartesian product, $A \times B$, for $(x \in A) \times B$ when x does not occur in B. $\times$ groups more tightly than $\to$, i.e. we write $A \times B \to C$ for $(A \times B) \to C$.

We write $\mathbb{N}$ for the type of natural numbers, and $\mathbb{1}$ for the trivial type with exactly one value. We will make use of two more so-called inductively defined types: the type of lists, and the type of list index positions. We use their description as an opportunity to give informal examples of how types are introduced in dependent type theory.

Example 3.1. Let $X \in \mathsf{Set}$. Then $\mathsf{List}(X)$ is also a type — of lists with elements drawn from X. We always accept that the empty list ε is a value in $\mathsf{List}(X)$. The type $\mathsf{List}(X)$ also admits values of the form "something, somethings", provided that "something" (the head of the list) is one element of X, and the "somethings" (the tail of the list) constitute a further $\mathsf{List}(X)$. That is, $\mathsf{List}(X)$ admits values as follows:

$$\varepsilon \in \mathsf{List}(X) \quad \Big| \quad \begin{array}{rl} \text{if} & x \in X \text{ and } xs \in \mathsf{List}(X) \\ \text{then} & x, xs \in \mathsf{List}(X) \end{array}$$

By explaining how to make values, we have ipso facto explained how to classify expressions. We may read x and xs in the above as *schematic* variables which may stand for arbitrary program expressions, and thus obtain a procedure for classifying expressions which compute to lists by reading the rules from bottom to top. For example, we may check that $7, (2{+}9), 13, \varepsilon \in \mathsf{List}(\mathbb{N})$ by checking that each of 7, $2{+}9$ and 13 are meaningful as natural numbers.

As an example of a dependent type, consider the following type of *evidence* that a given element occurs in a given list.

Example 3.2. Let $X \in \mathsf{Set}$, $x \in X$, and $xs \in \mathsf{List}(X)$. The type $(x \leftarrow xs)$ admits values as follows:

$$0 \in (x \leftarrow x, xs) \quad \Big| \quad \begin{array}{rl} \text{if} & i \in (x \leftarrow xs) \\ \text{then} & i + 1 \in (x \leftarrow y, xs) \end{array}$$

That is, the type $(x \leftarrow xs)$ is the type of index positions at which x occurs in xs: 0 is the position of the head of a nonempty list; $i + 1$ is the position in a nonempty list corresponding to position i in its tail. We may check that $1 \in (11 \leftarrow 7, (2{+}9), 13, \varepsilon)$ provided we are willing to do enough work to

identify 2+9 with 11. There may be many ways to classify a number (e.g. 1) as meaningful, in accordance with some purpose which we intend. We use $(x \leftarrow xs)$ as a type which gives it additional meaning as an index into a list: we know that this index is bounded by the length of the list, and also know what value sits in the indexed position.

It is not unusual to use numbers as loop indices or as coordinates of matrix entries, and quite often both. By refining our classification of numbers, we can make that connection clear, and begin to characterise the difference between we find at different coordinates.

4. Multidimensional Units of Measure

In this section, we put our expressive types to work by using them to keep track of units of measure for matrices and matrix operations. We start by summarising the necessary algebraic structure.

4.1. *Monoids and semirings*

Our work will rely heavily on algebraic properties of operations such as addition and multiplication, not only for numbers, but also for dimensions, physical quantities and matrices. Our starting point is the structure of a monoid, being a way to crush (possibly empty) sequences of values into a single value without regard to their grouping.

Definition 4.1. A **monoid** $M = (A, \_\cdot\_, 1)$ consists of

$$A \in \mathsf{Set} \quad \text{with} \quad \_\cdot\_ \in A \times A \to A \quad \text{and} \quad 1 \in A.$$

We call A the carrier of M, and often write $|M| = A$. The multiplication $\_\cdot\_$ is required to be associative, and to absorb the identity element 1 on both sides, i.e.

$$x{\cdot}(y{\cdot}z) = (x{\cdot}y){\cdot}z \qquad x{\cdot}1 = x \qquad 1{\cdot}y = y$$

Examples of monoids include $(\mathbb{N}, +, 0)$, $(\mathbb{N}, \times, 1)$, and $(\mathsf{List}(X), +\!\!+, \varepsilon)$, where $+\!\!+$ is list concatenation, and the trivial monoid whose carrier is $\mathbb{1}$. Common refinements of monoids include **commutative monoids**, i.e. where $x{\cdot}y = y{\cdot}x$, and **groups**, where there is also a unary inverse $\_^{-1}$ on the carrier, such that $x^{-1}{\cdot}x = 1$, and **abelian groups**, which combine both.

Definition 4.2. A **monoid homomorphism** between monoids M and N is a function $h \in |M| \to |N|$ which preserves the monoid structure, i.e.

$$h(x \cdot_M y) = h(x) \cdot_N h(y) \qquad h(1_M) = 1_N$$

Examples of monoid homomorphisms include:

- exponentiation k^- from $(\mathbb{N}, +, 0)$ to $(\mathbb{N}, \times, 1)$, for a fixed $k \in \mathbb{N}$;
- summation from $(\mathsf{List}(\mathbb{N}), +\!\!+, \varepsilon)$ to $(\mathbb{N}, +, 0)$;
- iterated product from $(\mathsf{List}(\mathbb{N}), +\!\!+, \varepsilon)$ to $(\mathbb{N}, \times, 1)$;
- remappings $[f(x) \mid x \leftarrow \_]$ from $(\mathsf{List}(X), +\!\!+, \varepsilon)$ to $(\mathsf{List}(Y), +\!\!+, \varepsilon)$, for any $f \in X \to Y$, where $[f(x) \mid x \leftarrow xs]$ means the list of the values $f(x)$ for each x drawn in turn from xs.

Once we have additive and multiplicative monoid structure on the same carrier, and they are compatible, we have the machinery for matrix multiplication. This is the structure of a semiring[b].

Definition 4.3. A **semiring** is a type $R \in \mathsf{Set}$, such that: (i) there is a commutative monoid $(R, +, 0)$; (ii) there is a monoid $(R, \_ \cdot \_, 1)$; (iii) $x \cdot \_$ and $\_ \cdot z$ act on $(R, +, 0)$ as monoid homomorphisms, i.e. multiplication distributes:

$$x \cdot 0 = 0 \quad x \cdot (y + z) = x \cdot y + x \cdot z \qquad 0 \cdot z = z \quad (x + y) \cdot z = x \cdot z + y \cdot z$$

As an example, our previously introduced monoids $(\mathbb{N}, +, 0)$ and $(\mathbb{N}, \times, 1)$ neatly combine to form a semiring. Physical quantities in a given dimension, however, do not: if you add two lengths, you get a length, but if you multiply two lengths, you get not a length but an area. We shall need to refine our notion of semiring to account for the dependency of quantities on dimensions. Before we can tackle dimensioned matrices, we need to treat dimensioned scalars.

4.2. *Dimensioned scalars*

Using dependent types, dimensioned scalars can be described elegantly by the structure of a semiring *graded* over the group of dimensions. Grading is a way of inducing relative notions of structure. Let us have a simple example — length-respecting concatenation of lists.

Example 4.1. It is standard practice in dependently typed programming to refine lists $\mathsf{List}(X) \in \mathsf{Set}$ with a length index: $\mathsf{ListN}(X, \_) \in \mathbb{N} \to \mathsf{Set}$. $\mathsf{ListN}(X, n)$ is the type of lists of length n with elements from X, but it is not a monoid with respect to concatenation: concatenating two lists of length

[b]To obtain the better known notion of a *ring*, upgrade the additive structure of a semiring to an abelian group.

5 makes a list of length 10. However, dependent pairs in $(n \in \mathbb{N}) \times \mathsf{ListN}(X, n)$ recover the monoidal structure of lists; moreover with first projection acting as a monoid homomorphism to $(\mathbb{N}, +, 0)$. For example,

$$(5, (a, b, c, d, e, \varepsilon)) +\!\!+ (5, (f, g, h, i, j, \varepsilon)) = (10, (a, b, c, d, e, f, g, h, i, j, \varepsilon)).$$

In order to construct this monoid on $(n \in \mathbb{N}) \times \mathsf{ListN}(X, n)$, what structure is needed on $\mathsf{ListN}(X, \_)$? We need an "identity element" in $\mathsf{ListN}(X, 0)$, and a way to "multiply" a $\mathsf{ListN}(X, n)$ and a $\mathsf{ListN}(X, m)$ to form a $\mathsf{ListN}(X, n + m)$. Intuitively, ε and $+\!\!+$ fit the bill: i.e., there is a family of concatenators $+\!\!+_{nm} \in \mathsf{ListN}(X, n) \times \mathsf{ListN}(X, m) \to \mathsf{ListN}(X, n + m)$ which respect the ordinary monoid structure of $+$.

Definition 4.4. Let $M = (|M|, \_\cdot\_, 1)$ be a monoid. A **graded monoid over** M is a triple $(L, \_\cdot\_\_\_, 1)$, where

$$L \in |M| \to \mathsf{Set} \quad \text{with} \quad \_\cdot_{ij}\_ \in L(i) \times L(j) \to L(i \cdot j) \quad \text{and} \quad 1 \in L(1).$$

In truth, the multiplication really has a dependent function type

$$(i \in |M|) \to (j \in |M|) \to L(i) \times L(j) \to L(i \cdot j)$$

but in mathematical vernacular, we suppress i and j, and in computational practice, a machine figures them out. The identity and associativity laws are commensurate by the monoid laws of M, and they must hold:

$$x \cdot_{i(jk)} (y \cdot_{jk} z) = (x \cdot_{ij} y) \cdot_{(ij)k} z \qquad x \cdot_{i1} 1 = x \qquad 1 \cdot_{1j} y = y$$

This is how we make the multiplication structure of physical quantities respect the underlying monoid structure of dimensions. Let us now deploy the same grading technique to the multiplicative structure of semirings: for the additive structure, it is enough to have an ordinary commutative monoid at every individual index.

Definition 4.5. Let $M = (|M|, \_\cdot\_, 1)$ be a monoid. A **semiring graded over** M is a dependent type $R \in |M| \to \mathsf{Set}$, such that: (i) for every $i \in |M|$, there is a commutative monoid, $(R(i), +_i, 0_i)$; (ii) there is a graded monoid $(R, \_\cdot\_\_\_, 1)$; (iii) $x \cdot_{ij} \_ \in R(j) \to R(i \cdot j)$ and $\_ \cdot_{ij} x \in R(i) \to R(i \cdot j)$ act as monoid homomorphisms.

Note that if M is the trivial monoid $M = \mathbb{1}$, this reduces to the ordinary notion of a non-graded semiring. We think of $R(d)$ as the commutative monoid of physical quantities with dimension d: we can add physical quantities of the same dimension — if q, $q' \in R(d)$, then $q +_d q'$ is well typed,

and again $q +_d q' \in R(d)$ — and multiplying quantities also multiplies their dimensions: if $q \in R(d)$ and $q' \in R(d')$ then $q \cdot_{dd'} q' \in R(d \cdot d')$. Critically, we do not think of $R(d)$ as a set of *numbers*; both "six foot six" and "two metres" inhabit $R(\mathsf{L})$. Numerical rendering of physical quantities necessitates agreement of *units*!

Definition 4.6. A **unit system** for the graded semiring $R \in |M| \to \mathsf{Set}$ over M is a dependent function $\mathsf{u} \in (d \in |M|) \to R(d)$, satisfying (i) $\mathsf{u}(1) = 1$; (ii) $\mathsf{u}(i \cdot j) = \mathsf{u}(i) \cdot_{ij} \mathsf{u}(j)$; and (iii) $\mathsf{u}(i) \cdot_{i1} x = x \cdot_{1i} \mathsf{u}(i)$ for every $x \in R(1)$.

If M is a free abelian group $M = \mathbb{G}$, most of such a function u is determined by its action on generators — e.g., the SI gives units for all our dimensions, taking $\mathsf{u}(\mathsf{M}) = \mathsf{kg}$, $\mathsf{u}(\mathsf{L}) = \mathsf{m}$ and $\mathsf{u}(\mathsf{T}) = \mathsf{s}$ — but we must still ensure that these are sent to invertible elements which commute with dimensionless quantities. We observe, in this situation, that even though R does not in general support *division*, we have $\mathsf{u}(d) \cdot_{dd^{-1}} \mathsf{u}(d^{-1}) = \mathsf{u}(d \cdot d^{-1}) = \mathsf{u}(1) = 1$. Moreover, the quantities given as $x \cdot_{1d} \mathsf{u}(d)$ constitute a graded semiring in their own right: requirements (ii) and (iii), above, give $(x \cdot_{1i} \mathsf{u}(i)) \cdot_{ij} (y \cdot_{1j} \mathsf{u}(j)) = (x \cdot_{11} y) \cdot_{1(i \cdot j)} \mathsf{u}(i \cdot j)$. Such a u thus allows dimensionless numbers to *encode* physical quantities.

For the rest of the paper, we now assume that we are given a graded semiring $\mathsf{R} \in \mathbb{G} \to \mathsf{Set}$ over our group of physical dimensions.

4.3. *Dimensioned matrices*

What is an appropriate notion of dimensioned matrix? It is not obvious.

On one end of the scale, we could demand that every entry in the matrix is of the same dimension, such as in Fig. 1(a), where we have elided specific quantities for clarity. This is too restrictive in practice: for instance, many elementary applications in physics collect together distance, speed, acceleration, ..., in one matrix — of systematically different dimensions.

On the other end of the scale, we could imagine to let every entry have its own unrelated dimension, as in Fig. 1(d). However most such matrices

$$\begin{pmatrix} \mathsf{L} & \mathsf{L} \\ \mathsf{L} & \mathsf{L} \\ \mathsf{L} & \mathsf{L} \end{pmatrix} \quad \begin{pmatrix} \mathsf{L} \cdot \mathsf{M}^{-1} & \mathsf{T} \cdot \mathsf{M}^{-1} \\ \mathsf{L} \cdot \mathsf{T}^{-1} & \mathsf{T} \cdot \mathsf{T}^{-1} \\ \mathsf{L} \cdot \mathsf{L} & \mathsf{T} \cdot \mathsf{L} \end{pmatrix} \quad \begin{pmatrix} \mathsf{T} & \mathsf{T} \cdot \mathsf{M} \\ \mathsf{L}^{-1} \cdot \mathsf{T} & \mathsf{L}^{-1} \cdot \mathsf{T} \cdot \mathsf{M} \\ \mathsf{M}^{-1} \cdot \mathsf{T} & \mathsf{M}^{-1} \cdot \mathsf{T} \cdot \mathsf{M} \end{pmatrix} \quad \begin{pmatrix} \mathsf{L} & \mathsf{T} \\ \mathsf{T}^{-1} & \mathsf{L}^3 \\ \mathsf{M} \cdot \mathsf{T} & \mathsf{L}^{-1} \end{pmatrix}$$

(a) Uniform (b) Columns and rows (c) Γ-dimensioned (d) Unrelated

Fig. 1. Possible dimensioned matrices

are without metrological and algebraic meaning — they are mere arrays of dimensioned quantities, rather than representations of linear transformations between dimensioned spaces, as we would expect dimensioned matrices to be. One way to see this is to notice that such dimensioned matrices with unrelated dimensions are not closed under multiplication — see Thm. 4.2 below. Matrix multiplication should correspond to composition of linear transformations, and yield the same closure property.

So instead, we consider more structured representations of such, as in Fig. 1(b), where matrix dimensions are an outer product of dimension vectors, i.e. each entry $M_{i,j}$ has dimension $a_i \cdot b_j$ obtained by multiplying the corresponding entries from a column vector $\vec{a} = a_1, \ldots, a_n$ and a row vector $\vec{b} = b_1, \ldots, b_m$. Be warned that this representation of dimensions is not unique: if M has dimensions given by $\vec{a}$ and $\vec{b}$, it also has dimensions given by $c \cdot \vec{a}$ and $c^{-1} \cdot \vec{b}$ for any group element c, since these perturbations cancel. Such redundancy is bad news if we hope to automate the checking of dimensions, e.g. by reducing it to type checking, since it means that any algorithm must do non-trivial work in order to decide equality.

We remove this redundancy by factoring all dimensions through the dimension b of the top left corner, recording the dimensions of the rest of the first column as and row cs respectively. The dimensions of all other entries (i, j) with $i, j > 1$ are given by multiplying the corresponding entries $a_i \cdot b \cdot c_j$, as in Fig. 1(c). It will yield some handy cancellation if all dimensions in as are inverted — without loss of generality, as $\mathbb{G}$ is a group. This leads to our definition of "Γ-dimensioned matrices".

Definition 4.7. Let $\mathsf{R} \in \mathbb{G} \to \mathsf{Set}$ be a graded semiring, $as, cs \in \mathsf{List}(\mathbb{G})$ lists of elements of $\mathbb{G}$, and $b \in \mathbb{G}$ a group element. We define the type of **Γ-dimensioned matrices** (of the given dimensions) as follows:

$$\mathsf{\Gamma\text{-}Matrix}(as, b, cs) \coloneqq (a \leftarrow 1, as) \times (c \leftarrow 1, cs) \to \mathsf{R}\ (a^{-1} \cdot b \cdot c)$$

This is making use of the type of "positions" $(x \leftarrow xs)$ from Ex. 3.2. Hence a Γ-dimensioned matrix is a function M such that

$$M(x, y) \in R\ (a^{-1} \cdot b \cdot c) \quad \text{whenever} \quad x \in a \leftarrow 1, as \text{ and } y \in c \leftarrow 1, cs.$$

Notation 4.1. Mnemonically, we *draw* the type of matrices showing each parameter in the place it characterises, making a literal Γ, hence the name:

$$\left[\begin{array}{|cc} b & cs \\ \hline as^{-1} & \end{array}\right] \coloneqq \mathsf{\Gamma\text{-}Matrix}(as, b, cs)$$

The $^{-1}$ above as is part of the picture, as a reminder that the as get inverted.

Example 4.2. Assume a unit system based on SI units. The following matrix M of physical quantities can be represented as an array of numbers (by writing the value $M(x, y)$ at numerical coordinates (x, y), as usual), and given the following Γ-dimensioned type:

$$\begin{pmatrix} 5\ \mathsf{m}^2 & 2\ \mathsf{kg}\cdot\mathsf{m}^2 \\ 1\ \mathsf{m}^2/\mathsf{s} & 3\ \mathsf{kg}\cdot\mathsf{m}^2/\mathsf{s} \end{pmatrix} \in \left[\begin{array}{|cc|} \hline \mathsf{L}^2 & \mathsf{M} \\ \mathsf{T}^{-1} & \\ \hline \end{array}\right]$$

If $\mathbb{G}$ is the trivial group $\mathbb{G} = \mathbb{1}$, then Def. 4.7 reduces to the familiar notion of $(n+1) \times (m+1)$-sized matrices for natural numbers n and m, as $\mathsf{List}(\mathbb{1}) \cong \mathbb{N}$.

Let us seek solace for the complexity of Γ-dimensioned matrices introduced by eliminating the redundancy from columns-and-rows. We note that for a list $as \in \mathsf{List}(A)$, a position in $(a \leftarrow as)$ is also a position in the remapping $(f(a) \leftarrow [f(x) \mid x \leftarrow as])$ for any function $f \in A \to B$. We can use remapping to normalise dimensions with respect to the corner:

Lemma 4.1. *A columns-and-rows dimensioned matrix with dimensions represented by non-empty lists a, as and b, bs is the same thing as a Γ-dimensioned matrix*

$$\left[\begin{array}{|cc|} \hline a\cdot b & [b^{-1}\cdot x \mid x \leftarrow bs] \\ [x^{-1}\cdot a \mid x \leftarrow as]^{-1} & \\ \hline \end{array}\right]$$

A Γ-dimensioned matrix

$$\left[\begin{array}{|cc|} \hline b & cs \\ as^{-1} & \\ \hline \end{array}\right]$$

is the same as a columns-and-rows dimensioned matrix with dimensions represented by the lists $[x^{-1} \mid x \leftarrow 1, as]$ and $[b\cdot x \mid x \leftarrow 1, bs]$. □

Hence the concepts of columns-and-rows dimensioned matrices and Γ-dimensioned matrices are interdefinable. Note that if we translate from columns-and-rows to Γ-dimensioned and back again, we may not get back the same lists of dimensions. In contrast, if we translate a Γ-dimensioned matrix forth and back, we preserve dimensions, due to the fusion law $[f(y) \mid y \leftarrow [g(x) \mid x \leftarrow as]] = [f(g(x)) \mid x \leftarrow as]$. This is not a coincidence, or a feat of the translation — Γ-dimensions are unique:

Theorem 4.1. *Assume that quantities have a unique dimension, i.e. if $r \in R(d)$ and $r \in R(d')$ then $d = d'$. Then Γ-dimensioned matrices also have unique Γ-dimensions, i.e. if*

$$M \in \left[\begin{array}{|cc|} \hline b & cs \\ as^{-1} & \\ \hline \end{array}\right] \quad \textit{and} \quad M \in \left[\begin{array}{|cc|} \hline b' & cs' \\ as'^{-1} & \\ \hline \end{array}\right]$$

then $as = as'$, $b = b'$ *and* $cs = cs'$. □

One might worry that the class of Γ-dimensioned matrices is too small to be useful, but as the following theorem shows, it is actually as good a class of dimensioned matrices as we can hope for, if we believe that matrix multiplication is a fundamental operation (as we do).

Theorem 4.2. *The collection of* Γ*-dimensioned matrices is the largest class of dimensioned matrices closed under multiplication.* □

The proof exploits the fact that multiplication computes a matrix of inner products whose summands must have uniform dimension.

4.4. *Dimension-aware matrix algebra*

Many well known operations on matrices can be given Γ-dimensioned types. Addition lifts straightforwardly, since it acts componentwise:

Theorem 4.3. *Matrix addition and the zero matrix have types*

$$+ \in \left[\begin{array}{|cc|}\hline b & cs \\ \hline as^{-1} & \\ \hline\end{array}\right] \times \left[\begin{array}{|cc|}\hline b & cs \\ \hline as^{-1} & \\ \hline\end{array}\right] \to \left[\begin{array}{|cc|}\hline b & cs \\ \hline as^{-1} & \\ \hline\end{array}\right] \qquad \mathbf{0} \in \left[\begin{array}{|cc|}\hline b & cs \\ \hline as^{-1} & \\ \hline\end{array}\right]$$

for any $as, cs \in \mathit{List}(\mathbb{G})$ *and* $b \in \mathbb{G}$. □

In contrast, multiplication, $\bullet$, is defined only when the matrices involved are compatible, in size, as we are used to, but here also in dimensions:

Theorem 4.4. *Matrix multiplication and the identity matrix have types*

$$\bullet \in \left[\begin{array}{|cc|}\hline b & cs \\ \hline as^{-1} & \\ \hline\end{array}\right] \times \left[\begin{array}{|cc|}\hline d & es \\ \hline cs^{-1} & \\ \hline\end{array}\right] \to \left[\begin{array}{|cc|}\hline b \cdot d & es \\ \hline as^{-1} & \\ \hline\end{array}\right] \qquad \mathbf{1} \in \left[\begin{array}{|cc|}\hline 1 & as \\ \hline as^{-1} & \\ \hline\end{array}\right]$$

for any $as, cs, es \in \mathit{List}(\mathbb{G})$ *and* $b, d \in \mathbb{G}$. □

Each element of the output is given for fixed positions in $(a \leftarrow 1, as)$ and $(e \leftarrow 1, es)$, but computed by an inner product ranging over all positions in $(c \leftarrow 1, cs)$. As we intend the notation to suggest, the row cs cancels the column cs^{-1}, yielding uniform dimension for all summands: $(a^{-1} \cdot b \cdot c) \cdot (c^{-1} \cdot d \cdot e) = a^{-1} \cdot (b \cdot (c \cdot c^{-1}) \cdot d) \cdot e = a^{-1} \cdot (b \cdot d) \cdot e$ as required.

Note that for $A \bullet B$ to be well typed, we require that the row of 'column dimensions' for A match the column of 'row dimensions' for B — this generalises the usual requirement that the *number of* columns of A should match the number of rows of B, and indeed, if $\mathbb{G}$ is the trivial group $\mathbb{G} = \mathbb{1}$,

then the former reduces exactly to the latter. Similarly the identity matrix is always a 'square' matrix in the sense that its 'row dimensions' match its 'column dimensions'. This is often the appropriate notion of squareness in a dimensioned setting — many operations on square matrices do not make dimensioned sense on arbitrary matrices of square *size*, but do for square matrices in this stronger sense. A good example is the trace of a square matrix, i.e. the sum of the diagonal elements:

Theorem 4.5. *The trace operator has type*

$$\mathbf{tr} \in \left[\begin{array}{|cc|}\hline b & as \\ \hline as^{-1} & \\ \hline\end{array}\right] \to R(b)$$

for any $as \in \mathit{List}(\mathbb{G})$ *and* $b \in \mathbb{G}$. □

However, interestingly, the determinant of a Γ-dimensioned matrix makes sense also for matrices of square *size*. This can be seen by considering the formula for the determinant as a sum over all permutations

$$\mathbf{det}(M) = \sum_{\pi \in S_n} \mathbf{sgn}(\pi) \cdot \prod_{x<n} M(x, \pi(x))$$

and noticing that each summand will always have the same dimension — that of the product of the main diagonal.

Theorem 4.6. *The determinant operator has type*

$$\mathbf{det} \in \left[\begin{array}{|cc|}\hline b & cs \\ \hline as^{-1} & \\ \hline\end{array}\right] \to R\left(\left(\prod_{a \leftarrow as} a\right)^{-1} \cdot b^{n+1} \cdot \left(\prod_{c \leftarrow cs} c\right)\right)$$

for any $b \in \mathbb{G}$, *and* $as, cs \in \mathit{List}(\mathbb{G})$ *of the same length* $|as| = |cs| = n$. □

4.5. *Elementary row operations*

The important method of Gaussian elimination makes sense for Γ-dimensioned matrices, and so can be used to solve systems of dimensioned equations, compute dimensioned inverses, etc. This can be seen by finding dimensioned types for the elementary row operations used by the method. The proof uses Lem. 4.1 to translate back and forth to the more symmetric columns-and-rows dimensioned matrices. We will explain how the operations use their arguments below:

Theorem 4.7. *The elementary row operations have types thus:*

$$\mathit{swapFirst} \in (p \in d \leftarrow as) \to \left[\begin{array}{|cc|}\hline b & cs \\ \hline as^{-1} & \\ \hline\end{array}\right] \to \left[\begin{array}{|cc|}\hline d^{-1} \cdot b & cs \\ \hline [d^{-1} \cdot x \mid x \leftarrow as[p \mapsto 1]]^{-1} & \\ \hline\end{array}\right]$$

$$\mathit{multFirst} \in R(d) \to \left[\boxed{\begin{matrix} b & cs \\ as^{-1} & \end{matrix}}\right] \to \left[\boxed{\begin{matrix} d \cdot b & cs \\ [d \cdot x \mid x \leftarrow as]^{-1} & \end{matrix}}\right]$$

$$\mathit{addMult} \in (d \leftarrow 1, as) \to (e \leftarrow 1, as) \to R(d^{-1} \cdot e) \to \left[\boxed{\begin{matrix} b & cs \\ as^{-1} & \end{matrix}}\right] \to \left[\boxed{\begin{matrix} b & cs \\ as^{-1} & \end{matrix}}\right]$$

□

The operation swapFirst takes a row to swap with as a position p for dimension d in as, and a matrix, returning a matrix with the dimensions also suitably swapped; the output row dimensions are given by swapping a 1 in place of the d, indicated by $as[p \mapsto 1]$, then renormalising with respect to the $d^{-1} \cdot b$ which is now in the top left corner. The operation multFirst is given a scalar and a matrix, and returns the matrix where the first row has been multiplied by the scalar. Of course, by combining swapFirst and multFirst, we can multiply any row by a scalar. Finally the operation addMult is given a source and a target position as indices of two group elements d and e into the list as, as well as a scalar to multiply by, of dimension $d^{-1} \cdot e$. This is a good example of a refined type being able to help with interactive program construction: the dimension of the multiplier already suggests a specific relationship between the source and target rows.

5. Conclusions and Future Work

Following Hart [8], we have shown how to use dependent types to extend type systems for units of measure from scalars to matrices. The key definition is of Γ-dimensioned matrices, recording the dimensions of matrix entries in a maximally liberal and unique way. We have also shown that many well known operations on matrices can be given Γ-dimension-respecting types. Our present technology, a general-purpose dependently typed language, obliges us to write more explicit proofs (elided here) of the list and group identities we rely upon to make types match. To pay for what we propose to get, we are developing a bespoke type checker with additional domain knowledge [14] about the equational theory of lists and abelian groups. Even to reach 'Base Camp Hart', we type theorists will have to raise our game.

Acknowledgments

We would like to thank Alistair Forbes, Keith Lines, Artem Shinkarov, and Ian Smith for interesting discussions related to this work, and the anonymous referee for comments and suggested improvements to the text. This work is

funded by the UK National Physical Laboratory Measurement Fellowship project *Dependent types for trustworthy tools*.

References

1. A. G. Stephenson, L. S. LaPiana, D. R. Mulville, P. J. Rutledge, F. H. Bauer, D. Folta, G. A. Dukeman, R. Sackheim and P. Norvig, *The Mars Climate Orbiter Mishap Investigation Board Phase I Report*, tech. rep., NASA (1999).
2. O. Bennich-Björkman and S. McKeever, The next 700 unit of measurement checkers, in *SLE '18*, (ACM, 2018).
3. A. Kennedy, Programming languages and dimensions, PhD thesis, University of Cambridge, (United Kingdom, 1995).
4. M. C. Schabel and S. Watanabe, Boost C++ libraries, chapter 43 (Boost.Units 1.1.0) `https://www.boost.org/doc/libs/1_74_0/doc/html/boost_units.html`, (2010).
5. J.-M. Dautelle, W. Keil and O. Santana, JSR 385: Units of measurement `https://unitsofmeasurement.github.io/`, (2019).
6. T. Muranushi and R. Eisenberg, Experience report: Type-checking polymorphic units for astrophysics research in Haskell, in *Haskell '14*, (ACM, 2014).
7. Pint: makes units easy `https://pint.readthedocs.io/`, (2020).
8. G. W. Hart, *Multidimensional Analysis* (Springer, 1995).
9. P. Griffioen, A unit-aware matrix language and its application in control and auditing, PhD thesis, University of Amsterdam, (the Netherlands, 2019).
10. M. Wand and P. O'Keefe, Automatic dimensional inference, in *Computational Logic: Essays in Honor of Alan Robinson*, eds. J.-L. Lassez and G. Plotkin (MIT Press, 1991) pp. 479–486.
11. A. Gundry, A typechecker plugin for units of measure: Domain-specific constraint solving in GHC Haskell, in *Haskell '15*, (ACM, 2015).
12. A. Bove and P. Dybjer, *Dependent Types at Work*, in *LerNet ALFA Summer School 2008 Revised Tutorial Lectures*, eds. A. Bove, L. S. Barbosa, A. Pardo and J. S. Pinto (Springer, 2009), pp. 57–99.
13. U. Norell, Towards a practical programming language based on dependent type theory, PhD thesis, Chalmers University, (Sweden, 2007).
14. G. Allais, C. McBride and P. Boutillier, New equations for neutral terms: a sound and complete decision procedure, formalized, in *DTP@ICFP 2013*, ed. S. Weirich (ACM, 2013).