Type Inference for Array Programming with Dimensioned Vector Spaces

P.R. Griffioen
CWI
Amsterdam, the Netherlands
p.r.griffioen@cwi.nl

ABSTRACT

Linear algebra operations are a typical application for array programming, but arrays are a more general data structure and not just used for numbers. This generality gives a lot of flexibility, but when numerical data structures like vectors and matrices are build from arrays, it is hard to infer properties like the shape statically, requiring costly out of bound checks to be done at runtime. We extend arrays with units of measurement, and Hindley-Milner typing with a matrix type based on the algebraic structure of units of measurement in matrices that allows type inference up to dimensioned vector spaces. The inference is sound and complete and gives the most general type of any linear algebra expression. Experiments show that the explicit support for linear algebra increases type safety, and that it leads to a more functional and index-free style of programming. It refines the types for linear algebra operators significantly, while the use of arrays as general containers has to be replaced by other data structures. As validation the matrix type system is implemented in the functional matrix language Pacioli.

CCS Concepts

•Theory of computation → Type structures; •Software and its engineering → Functional languages; Data types and structures;

Keywords

Matrix language; linear algebra; type inference; unit of measurement

1. INTRODUCTION

The combination of parametric polymorphism and type inference in functional programming has enhanced the compile time safety of programs enormously, but support for numbers is still limited. Vectors and matrices are common tools in numerical software, but while specific support for linear algebra operators has the potential to increase safety, it is not typically part of type systems. A parametric type system usually provides some built in primitive type for single numbers. Compound numerical data can then be built with the usual containers like tuples, lists, records, or any form of data abstraction. The usual checks on these types provide many guarantees for the correct usage, but are not readily capable of detecting properties like the correct shape of vectors and matrices. Such dimension checks are too specific for the general type system, and have to be performed at runtime, at the expense of safety and performance.

To improve the support for numerical data we developed dimensioned arrays and a corresponding matrix type that infers types up to dimensioned vector spaces. The term dimensioned is in this case not about the size(s) of an array, but about units of measurement. A dimensioned vector space is a vector space that couples a unit of measurement with each vector entry. Since linear algebra operations are index free, which means they work on vectors and matrices as a whole without referencing individual entries, dimensioned vector spaces are well-suited to type them. The underlying idea behind this index-free typing is to split a vector into an element-wise product of a vector of numbers and a vector of units of measurement. For example

$$\text{replace} \begin{bmatrix} 10\,\text{kg} \\ 25\,\text{box} \\ 50\,\text{bag} \end{bmatrix} \text{ by } \begin{bmatrix} 10 \\ 25 \\ 50 \end{bmatrix} \begin{bmatrix} \text{kg} \\ \text{box} \\ \text{bag} \end{bmatrix}$$

The unit of measurement vector (also called uom vector from now on) cannot only describe the units in one dimensional arrays (vectors), but can also be combined to describe the units in higher dimensional arrays (matrices and tensors). In Section 2 an overview is given how array shapes are extended with uom vectors to give dimensioned arrays. After describing the base case for vectors, the the structure of units in matrices is discussed. Dimensioned matrices are based on Hart's [8] work on dimensioned vector spaces and are the core of the extension. Tensors are a generalization of the matrix case. The overview ends with an informal description of the matrix type and its relation with dimensioned arrays.

The mathematical details of units of measurement and dimensioned vector spaces on which everything is based are discussed next in Section 3. Hart's theorem about the structure of units in matrices is the basis for the evaluation rules for dimensioned arrays. In Section 4 rules are given for common linear algebra operations and for operations that are specific to units of measurement. Since the rules are based on dimensioned vector spaces they catch errors that normal array checks miss. For example forgetting to transpose or invert a square matrix cannot be detected by customary size checks, but can by this paper's matrix type.

The matrix type is based on a naming scheme for dimensioned vector spaces and somewhat stricter than the runtime dimensioned arrays. Section 5 gives type rules corresponding to the evaluation rules from the previous section. The type is an algebraic expression

IFL '15, September 14-16, 2015, Koblenz, Germany
© 2015 ACM. ISBN 978-1-4503-4273-5/15/09…$15.00
DOI: http://dx.doi.org/10.1145/2897336.2897341

with the names of dimensioned vector spaces among the terminals. The type expressions are an extension of Kennedy's dimension expressions[12, 13], and a unification algorithm is given for the matrix type, based on Kennedy's unification for units.

To validate the matrix type we implemented a prototype for the statically typed matrix language Pacioli[1]. A consequence of the matrix type is that cases where matrices are used as general containers instead of linear transformations cannot be typed and have to be changed to a different data structure. An example is the case of the Fourier Motzkin algorithm. The original algorithm starts with gluing two matrices together, and proceeds by growing and shrinking this combined matrix. These operations cannot be typed with dimensioned vector spaces. The Pacioli version uses a list of vectors instead of a growing and shrinking matrix. The remaining vector and matrix operations are all operations from linear algebra and can be typed correctly. The Fourier Motzkin algorithm is a typical case where shape analysis is complicated because matrices are used as general containers. The type system favors a more index-free and functional programming style. For the rewritten version Pacioli can infer the types of the algorithm without any annotation. Pacioli's implementation and the Fourier-Motzkin example are discussed in Section 6.

2. AN OVERVIEW

A dimensioned number is the product of a number and a unit of measurement. The number is typically real or complex. The unit is build from base units by multiplication and division. Examples of base units are the well known SI units like kilogram (kg), metre (m), second (s), etc., and an example unit made from these bases is $\text{kg} \times \text{m} \,/\, \text{s}^2$. A unit measure a quantity in a certain dimension. For example a kilogram is a unit for dimension mass and metre for dimension length. Different units may exists for the same dimension, for example second and hour for the time dimension.

Simply adding units of measurement to an array's numbers is too expensive and not suitable for typing. A unit of measurement object is much more expensive than a primitive number, so coupling a unit with each number would give an enormous overhead. An alternative would be to associate one unit with a vector and use a parametric type like `Vector(a)`. This would work well with Kennedy's inference for units, but then each element has the same unit `a`. Parametric polymorphism would limit arrays to numbers with homogeneous units. This is a general problem for container data types of varying size at run time and is similar to a database with numbers where it is insufficient to associate a unit with a column if the units can vary per row. Parametric types cannot describe compound numerical data if the elements can have different units of measurement.

The limitations of parametric polymorphism and the high overhead of units lead to the investigation of a more efficient alternative that adds units of measurement to an array's shape. The concept is based on Hart's dimensioned vector spaces. Adding units to an array's shape is done by allowing uom vectors in unit expressions. Operations on uom vectors in these expressions are element-wise. Let $b_1^{x_1} \cdots b_n^{x_n}$ be some unit with each base b a uom vector, then

$$(b_1^{x_1} \cdots b_n^{x_n})[i] = b_1[i]^{x_1} \cdots b_n[i]^{x_n} \tag{1}$$

For instance, using units hour (hr) and metre (m):

$$\begin{bmatrix}\text{hr}\\ \text{m}^2\end{bmatrix}^{-1} = \begin{bmatrix}\text{hr}^{-1}\\ \text{m}^{-2}\end{bmatrix} \text{ and } \begin{bmatrix}\text{hr}\\ \text{m}^2\end{bmatrix}^{2} = \begin{bmatrix}\text{hr}^2\\ \text{m}^4\end{bmatrix}$$

[1]Pacioli can be found at http://pgriffel.github.io/pacioli/

Dimensioned arrays use unit expressions with vector bases like this instead of individual units per number. Lifting the units to vectors like this allows them to be added to array shapes.

The shape of arrays as used in array programming is typically an array $[s_1, \ldots, s_k]$ of natural numbers, where s_i describes the length of the i-th dimension. The number of dimensions k is called the array's rank. For example $[10]$ denotes a 1-dimensional array of size 10 and $[3,4]$ denotes a 3 by 4 matrix. The empty array $[]$ denotes a scalar number. The shape array is called an array descriptor. To ease notation we let s^k stand for $s_1, \ldots, s_k$ so array descriptors can be written as $[s^k]$. An index is an array $[i^k]$ with $1 \leq i_k \leq s_k$.

Dimensioned arrays require not only the extension of array shapes with unit of measurements, but also the distinction between covariant and contravariant dimensions as in tensor calculus. Covariance and contravariance in tensor calculus describe how units change with a change of basis and should not be confused with covariance and contravariance in types. The extended array shape is a tuple

$$\langle a, [m^k], u, [n^l], v \rangle$$

where the single array descriptor $[s^k]$ is replaced by a contravariant descriptor $[m^k]$, and a covariant descriptor $[n^l]$. Number m_i is the length of the i-th contravariant dimension, and n_j is the length of the j-th covariant dimension. The pair $\langle k, l \rangle$ is called the array's order, and now the array's rank is the sum $k+l$. A scalar has order $\langle 0,0 \rangle$, a column vector has order $\langle 1,0 \rangle$, a row vector has order $\langle 0,1 \rangle$, and a matrix $\langle 1,1 \rangle$. Arrays with higher rank are called tensors. Unit a is a scalar unit object, unit u a unit object that contains the uom vectors for the k contravariant dimensions, and unit v a unit object that contains the uom vectors for the l covariant dimensions. With these three unit objects the units of measurement in any array can be described.

The units objects in the extended array shape use special bases to make it work. The bases in unit objects u and v are pairs $\langle x, i \rangle$, with x a uom vector and i an integer that indicates the dimension to which x applies. The size of uom vector x must be equal to the size of the i-th dimension. Index i is only relevant for tensors; for vectors and matrices i is always 1 since there is at most one dimension. For scalars there are no bases because u and v are empty.

The distinction between covariant and contravariant dimensions is not commonly found in array implementations, but necessary for the unit support introduced in this paper. The overall unit is the unit for the contravariant dimensions divided by the units for the covariant dimensions. So, for indices $[i^k], [j^l]$ the unit at these coordinates is

$$a \cdot \frac{u[i^k]}{v[j^l]} \tag{2}$$

The equation shows that units are multiplied for contravariant dimensions and divided for covariant dimensions.

2.1 Vectors

To see how the extended shapes are used, consider the following data that we want to store in arrays. The data might come from a database, or some other source.

Product	prod_unit	pur_price	sales_price	sales
butter	gram	0.40	-	-
flour	gram	0.25	-	-
apples	gram	0.09	0.20	2.00
sugar	gram	0.08	-	-
ice water	millilitre	0.00	-	-
pastry	kilogram	-	-	-
pie	-	-	499.00	25.00
piece	-	-	49.0	100.00

The data contains numerical data for the purchase price, the sales price, and the sales amount for various products. The numbers are meaningless without units of measurement, and that's what the `prod_unit` column is for. In this case there is a single uom column, but in other cases it might for example be convenient to have different units like `trade_unit` and `ingredient_unit`. The units apply to the numbers in the following way:

$$\begin{aligned}\texttt{pur\_price}\colon & \langle \texttt{cent}, [8], \langle \texttt{prod\_unit}, 1\rangle^{-1}, [], 1\rangle \\ \texttt{sales\_price}\colon & \langle \texttt{cent}, [8], \langle \texttt{prod\_unit}, 1\rangle^{-1}, [], 1\rangle \\ \texttt{sales}\colon & \langle 1, [8], \langle \texttt{prod\_unit}, 1\rangle, [], 1\rangle\end{aligned}$$

The shapes contain unit expressions build from the base unit `cent` and the base uom vector `prod_unit`. From Equation (2) it follows that for any product p the unit for the `pur_price` and for the `sales_price` quantity is `cent/prod_unit`$[p]$. For example the purchase price for butter is 0.40 ¢/g. For an array with the `sales` data the unit is simply `prod_unit`$[p]$.

The dimensioned arrays are much more efficient than arrays with dimensioned numbers. The base vector is actually just a reference to a uom vector, which makes it not more expensive than an ordinary unit, but since there are now just three unit objects, instead of one for each array entry this scales well. Furthermore the uom vectors have to be stored in memory only once, which makes it also more memory efficient.

2.2 Matrices

Units of measurement in matrices can be described by combining uom vector bases, without the need for any additional primitives. According to Hart's theorem that will be explained in detail in Section 3, units of measurement in a matrix can be completely described by just two uom vectors. Just as for vectors this makes support for units much more efficient and gives a convenient way to state evaluation rules.

In general a matrix has shape $\langle a, [m], u, [n], v\rangle$ where the base vectors in unit expression u have size m and the base vectors in unit expression v have size n. For example, shape $\langle 1, [2], \langle \mathbf{u}, 1\rangle, [3], \langle \mathbf{v}, 1\rangle\rangle$ with $\mathbf{u} = \begin{bmatrix}\text{hr}\\ \text{m}^2\end{bmatrix}$ and $\mathbf{v} = \begin{bmatrix}\text{kg}\\ \text{box}\\ \text{bag}\end{bmatrix}$ describes a 2×3 matrix. From Equation (2) follows that the units are:

$$\begin{bmatrix}\text{hr/kg} & \text{hr/box} & \text{hr/bag}\\ \text{m}^2\text{/kg} & \text{m}^2\text{/box} & \text{m}^2\text{/bag}\end{bmatrix}$$

The entry for i, j is $\mathbf{u}[i]/\mathbf{v}[j]$. The units in a matrix follow from the units in the row and in the column dimension.

The form of the units in a matrix follow from the linear transformation that the matrix performs and is the basis for evaluation and units checking. Notice that multiplying the example matrix with vector $\mathbf{v}$ gives vector $\mathbf{u}$. In general a matrix with shape $\langle a, [m], u, [n], v\rangle$ transforms vectors with units v into vectors with units u. Now if a matrix transforms from w to v, and another matrix from v to u, then their product transforms from w to u. This can be stated as an evaluation rule for the matrix product between dimensioned matrices:

$$\frac{x \Rightarrow \langle a, [m], u, [k], v\rangle, \quad y \Rightarrow \langle a', [k], v, [n], w\rangle}{x \cdot y \Rightarrow \langle a \times a', [m], u, [n], w\rangle} \tag{3}$$

This rule is just an illustration for the units in matrices only, but in Section 4 this will be generalized to any shape and including numbers. Note the similarity to the well-known matrix product rule $\mathbb{R}^{m\times k} \times \mathbb{R}^{k\times n} \to \mathbb{R}^{\text{m}\times\text{n}}$.

An example that illustrates units of measurement in matrices well is the "explosion" and "netting" problem in requirements planning from operations research. It involves computations with a bill of material matrix where units of measurement are vital. A miniature example of a bill of material is the following ingredient list from a recipe for apple pie.

Product	Product	BoM
butter	pastry	360.00 g/kg
flour	pastry	550.00 g/kg
ice water	pastry	100.00 ml/kg
pastry	pie	0.40 kg
apples	pie	700.00 g
sugar	pie	225.00 g
butter	pie	115.00 g
pie	piece	0.13

Pastry is an intermediate product, made from butter, flour and ice water. The pastry is combined with apples, sugar and butter into a pie. Finally a pie is cut into 8 pieces, which means that a piece of pie 'consists' of 0.13 pie. The computations for the explosion and netting problem in requirements planning are done with a bill of material matrix. This square matrix is indexed by products in the row and the column dimension and tells how much of a product is contained in an other product. In this case it would be an 8×8 matrix with the shape as follows:

$$\texttt{BoM}\colon \langle 1, [8], \langle \texttt{prod\_unit}, 1\rangle, [8], \langle \texttt{prod\_unit}, 1\rangle\rangle$$

The units are `prod_unit`$[i]$/`prod_unit`$[j]$. In contrast to the other table in this case the units have been added for to the printed representation for readability.

The improved error detection capabilities can be illustrated with matrix multiplication. The bill of material matrix can be multiplied with the sales amounts to compute the amounts for parts. It can also be multiplied with the the purchase price to compute part prices, but this requires the transpose.

$$\begin{aligned}\texttt{BoM}\cdot\texttt{sales}\colon & \langle 1, [8], \langle \texttt{prod\_unit}, 1\rangle, [], 1\rangle \\ \texttt{BoM}^T \cdot \texttt{pur\_price}\colon & \langle \texttt{cent}, [8], \texttt{prod\_unit}^{-1}, [], 1\rangle\end{aligned}$$

The rule for the transpose follows from the mathematical properties of units and is as follows

$$\frac{x \Rightarrow \langle a, [m], u, [n], v\rangle}{x^T \Rightarrow \langle a, [n], v^{-1}, [m], u^{-1}\rangle} \tag{4}$$

The rule swaps the row and column dimensions and also takes the reciprocal of the units of measurement. It is an easy mistake to mix up the direction or forget a transpose. The matrix type catches these errors because `prod_unit` does not equal `prod_unit`$^{-1}$. Shape information without units cannot detect these errors, because the bill of material matrix is square and therefore its shape and the shape of its transpose would be the same.

2.3 Tensors

Multi-linear algebra generalizes a matrix to a tensor. A matrix is a map $P \times Q \to \mathbb{R}$ with exactly one contravariant and one covariant index, whereas a tensor is a map $P_1 \times \ldots \times P_k \times Q_1 \times \ldots \times Q_l \to \mathbb{R}$ with any number of contravariant and covariant indices. Pair $\langle k, l\rangle$ is the tensor's order.

The units of measurement in a tensor can be described by associating unit vectors with the various dimensions, and this is where index i in the unit bases $\langle x, i\rangle$ comes into play. For vectors and matrices this index is always 1, but for tensors it tells to which dimension unit vector x belongs.

The columns `sales` and `amount` from the following table are examples of multi-dimensional data. The sales column has homogeneous unit dollar. Oil amounts are in barrels and gold amounts in ounces.

Region	Year	Commodity	sales	amount
North	1990	Oil	5,000.00 $	155.00 bbl
South	1990	Oil	87,000.00 $	400.00 bbl
West	1990	Gold	64,000.00 $	150.00 oz
South	1990	Gold	99,000.00 $	235.00 oz
North	1991	Gold	8,000.00 $	18.00 oz
East	1991	Gold	7,000.00 $	16.00 oz

Let's say there are four regions, two years of data, and just the two commodities oil and gold. If it is further given that for the commodities' units

$$\texttt{comm\_unit} = \begin{bmatrix} \text{bbl} \\ \text{oz} \end{bmatrix}$$

then the shapes for the data is:

$$\begin{aligned} \texttt{sales:}\ & \langle \texttt{dollar}, [4,2,2], 1, [], 1 \rangle \\ \texttt{amount:}\ & \langle 1, [4,2,2], \langle \texttt{comm\_unit, 3} \rangle, [], 1 \rangle \end{aligned}$$

The `sales` and `amount` columns are examples of $\langle 3,0 \rangle$ tensors. For any entry $[r,y,c]$, the unit for the amount is $\texttt{comm\_unit}[c]$.

2.4 The Matrix Type Notation

The matrix type that will be developed in Section 5 is based on a special naming scheme for uom vectors. This naming scheme works with generalized index sets instead of natural numbers indices, which means mathematically that matrices are of the form $\mathbb{R}^{P \times Q}$ with index sets P and Q, instead of the from $\mathbb{R}^{m \times n}$. A base uom vector is identified by the combination of an index set name and some identifier, separated by an exclamation mark. The notation acts as a kind of constraint or quantification on a uom vector identifier that states that the vector's indices are from some given index set. In the following matrix product using the 2×3 matrix from the beginning of Section 2.2, the types are shown above the arguments. It computes the required resources given some desired output of goods.

$$\overbrace{\begin{bmatrix} 400\,\text{hr} \\ 185\,\text{m}^2 \end{bmatrix}}^{\texttt{Resource!unit}} = \overbrace{\begin{bmatrix} 5\,\text{hr/kg} & 4\,\text{hr/box} & 5\,\text{hr/bag} \\ 1\,\text{m}^2\text{/kg} & 3\,\text{m}^2\text{/box} & 2\,\text{m}^2\text{/bag} \end{bmatrix}}^{\texttt{Resource!unit per Good!unit}} \cdot \overbrace{\begin{bmatrix} 10\,\text{kg} \\ 25\,\text{box} \\ 50\,\text{bag} \end{bmatrix}}^{\texttt{Good!unit}}$$

The types use the names `Good!unit` and `Resource!unit` to indicate the unit vectors **u** and **v**. The names `Resource` and `Good` indicate index sets, with for example `Resource`={labor, storage} and `Good`={potatoes, beans, rice}. For both index sets the name `unit` is an identifier for a specific unit vector of the given index set. This creates a simple two-level hierarchical name space; the same unit name can be used with different index sets but it is unique in combination with an index set. The matrix type is an algebraic expression in which names like `Good!unit` and `Resource!unit` are base elements. The type `Resource!unit per Good!unit` denotes a matrix that transforms vectors from the `Resource!unit` space to the `Good!unit` space. The `per` operator is defined by

$$(\mathbf{u}\ \texttt{per}\ \mathbf{v})[i,j] = \frac{\mathbf{u}[i]}{\mathbf{v}[j]}$$

This notation is in line with Equation (2) and is based on Hart's theory about the structure of units of measurement in matrices.

The naming scheme effectively allows the identification of dimensioned vector spaces. Dimensioned vector spaces will be defined in Section 3, but informally two dimensioned vector spaces $P!u$ and $Q!v$ are guaranteed to have the same size if and only if index set P equals index set Q, and they are guaranteed to have the same units as well if and only if additionally u equals v. For each index set P there exists one unique dimensionless vector space which is written as $P!$, for example `Good!` denotes a dimensionless good vector.

The matrix type is more restrictive than dimensioned array shapes, which excludes some runtime behavior but also can improve safety. Normal array shapes see vectors and matrices of the same size as compatible, while they conceptually may have different index sets and not be compatible at all. Consequently, the matrix type can be mapped to a dimensioned array shape, but not the other way around. The algebraic matrix type has a normal form and the most general matrix type is

$$a \cdot P!u\ \texttt{per}\ Q!v$$

It matches any matrix type. Or put differently, any other matrix type can be obtained from it by a proper substitution. If $P!u$ denotes unit vector **u**, $Q!v$ denotes unit vector **v**, m is **u**'s size, and n is **v**'s size, then $\langle a, [m], \langle \mathbf{u}, 1 \rangle, [n], \langle \mathbf{v}, 1 \rangle \rangle$ is the corresponding array shape.

The type system infers the type of any linear algebra expression. An example is the following Pacioli function to compute the area of a triangle in three dimensional geometric space. The index set `Space` is defined as the set {x, y, z}.

```
define triangle_area(x, y, z) =
    norm(cross(y - x, z - x)) / 2;
↪ triangle_area :: for_unit a:
        (a∗Space!, a∗Space!, a∗Space!) → a^2
```

The type `a∗Space!` is scalar unit variable `a` multiplied by the dimensionless space vector which gives a homogeneous vector of unit `a`. The function expects three points in space given as space vectors and computes the area of the triangle enclosed by these points. Symbol ↪ indicates that the type of a definition is inferred, similar to an interaction with an implementation in a REPL. The result is correctly derived as the square of the unit of measurement of the geometric space. So, if for example `a` is the unit metre, the result would have unit m^2.

3. DIMENSIONED VECTOR SPACES

Formally units of measurement form an abelian group $(\mathbb{U}, 1, \times, ^{-1})$ freely generated from a set of base units $\mathscr{B}$. As usual division is multiplication by the reciprocal, and exponents are repeated multiplication. The group's unit `1` denotes the dimensionless numbers.

A dimensioned number is a product ru from $\mathbb{R} \times \mathbb{U}$ or $\mathbb{C} \times \mathbb{U}$. It pairs magnitude r with a unit u. In this paper we focus on the units $\mathbb{U}$ and ignore the different number types.

Units of measurement are commonly implemented as a map from the base units to the integers. Any unit can be written as a power product $\Pi b : b^{f(b)}$ where b is a base unit from $\mathscr{B}$ and $f(b)$ is its power. For example the unit from the previous section can be written as kg m s^{-2}. The unit is completely described by function f, and therefore any unit can be implemented as a map of type $\mathscr{B} \to \mathbb{Z}$. The set $\{b \in \mathscr{B} \mid f(b) \neq 0\}$ is called the dimension's support. For efficiency reasons only bases from the support need to be stored in the map.

The dimensioned array shapes are based on Hart's theory of dimensioned matrices. A dimensioned matrix can be written as an element-wise product **RU** with **R** a magnitude matrix from $\mathbb{R}^{m \times n}$

and $\mathbf{U}$ a unit matrix from $\mathbb{U}^{m\times n}$. A dimensioned vector is an $n \times 1$ dimensioned matrix. The base units $\mathscr{B}$ in the abelian group of units are replaced by uom vectors $\{\mathbf{u},\mathbf{v},\mathbf{w},\ldots\}$. The operations on uom vectors are element-wise.

$$(\mathbf{v}\times\mathbf{w})_i = \mathbf{v}_i \times \mathbf{w}_i$$
$$(\mathbf{v} \,/\, \mathbf{w})_i = \mathbf{v}_i \,/\, \mathbf{w}_i$$
$$(\mathbf{v}^n)_i = (\mathbf{v}_i)^n$$

This is equivalent to the formulation in Equation (1).

Hart defines a natural equivalence relation on dimensional matrices using the unit parts. Since we use units instead of dimensions these equivalences are stricter in our case, but from a technical point of view this makes no difference. Let $\dim(\mathbf{RU}) = \mathbf{U}$.

Definition 1. For all matrices $\mathbf{A}$ and $\mathbf{B}$

$$\mathbf{A} \equiv_d \mathbf{B} \;\Leftrightarrow\; \dim(\mathbf{A}) = \dim(\mathbf{B}) \qquad \text{dimensional similarity}$$

The relation equates two matrices when they have the same units in corresponding components.

The equivalence relation is used to define the vector space to which a given vector belongs.

Definition 2. Let $\mathbf{v}$ be a dimensioned vector. The *complete dimensioned vector space* of type $\mathbf{v}$ is the set of all $\mathbf{w}$ such that $\mathbf{v} \equiv_d \mathbf{w}$.

Definition 3. A *dimensioned vector space* of type $\mathbf{v}$ is a subset of a complete dimensioned vector space which is closed under addition and scalar multiplication by dimensionless scalars. A dimensioned vector space of type $\mathbf{v}$ is called a $\mathbf{v}$-space.

In dimensioned vector spaces the adjoint isn't the transpose, but the *dimensional inverse*. Hart uses it to show that the units in a dimensioned matrix are a rank-one matrix[2].

Definition 4. The *dimensional inverse* $\mathbf{A}^{\sim}$ of a matrix is defined by

$$\mathbf{A}^{\sim}_{ij} \cdot \mathbf{A}_{ji} \;=\; 1$$

The dimensional inverse combines the transpose and the reciprocal; from the definition follows $\mathbf{A}^{\sim} = (\mathbf{A}^T)^{-1}$ directly.

THEOREM 1 (HART). *Any matrix of a linear transformation from a* $\mathbf{v}$*-space to a* $\mathbf{u}$*-space has the form* $\mathbf{u}\cdot\mathbf{v}^{\sim}$

This result is the basis for the units u and v in the array shape introduced in Section 2.

Hart's theorem can be generalized to the multi-linear case with the outer product $\otimes$. Let $\mathbf{x}\otimes\mathbf{y}$ stands for the outer product $\mathbf{x}\cdot\mathbf{y}^T$. Uom vectors like $\mathbf{u}$ generalize to unit vector products $\mathbf{u_1}\otimes\ldots\otimes\mathbf{u_k}$, and Hart's result about the structure of units in $\langle 1,1\rangle$ tensors generalizes to the following structure for $\langle k,l\rangle$ tensors:

$$(\mathbf{u_1}\otimes\ldots\otimes\mathbf{u_k})\cdot(\mathbf{v_1}\otimes\ldots\otimes\mathbf{v_l})^{\sim}$$

The tensor consists of k contravariant and l covariant unit vectors.

The unit at coordinates $i_1,\ldots i_k$ in a uom vector product is the product of each dimension's unit:

$$(\mathbf{u_1}\otimes\ldots\otimes\mathbf{u_k})[i_1,\ldots,i_k] = \mathbf{u_1}[i_1]\times\ldots\times\mathbf{u_k}[i_k]$$

It follows that it is sufficient to store one unit vector for each dimension.

[2]Rank here means the rank as found in linear algebra, not the rank from array programming used earlier to indicate the array dimensions.

Uom vector products like $\mathbf{u_1}\otimes\ldots\otimes\mathbf{u_k}$ are an abelian group in which bases are base tensors $\langle\mathbf{u},i\rangle$ from $\mathscr{B}\times\mathbb{N}$. Pair $\langle\mathbf{u},i\rangle$ means a product $1\otimes\ldots\otimes 1\otimes\mathbf{u}\otimes 1\otimes\ldots\otimes 1$ with $\mathbf{u}$ at the i-th position and 1s everywhere else. Bases have these form because the product of abelian groups with bases X resp. Y is a abelian group with base $(X\times\{1\})\cup(\{1\}\times Y)$. For example product $\mathbf{u}^{-1}\otimes\mathbf{v}/\mathbf{w}^2\otimes\mathbf{x}^3$ is normalized into unit

$$\langle\mathbf{u},1\rangle^{-1}\times\langle\mathbf{v},2\rangle\times\langle\mathbf{w},2\rangle^{-2}\times\langle\mathbf{x},3\rangle^{3}$$

The replacement of base vectors $\mathbf{u}$ by base tensors $\langle\mathbf{u},i\rangle$ generalizes matrices to tensors.

Putting everything together we get that units in a $\langle k,l\rangle$ tensor are a triple $\langle a,u,v\rangle$, with

- a unit expression with scalar bases b
- u unit expression with tensor bases $\langle\mathbf{u},k\rangle$
- v unit expression with tensor bases $\langle\mathbf{v},k\rangle$

Let $u = \Pi_{\langle\mathbf{u},k\rangle} : \langle\mathbf{u},k\rangle^{f\langle\mathbf{u},k\rangle}$ and let $v = \Pi_{\langle\mathbf{v},l\rangle} : \langle\mathbf{v},l\rangle^{g\langle\mathbf{v},l\rangle}$, then the unit at coordinates $i_1,\ldots,i_k,\ j_1,\ldots,j_l$ is

$$a\times\frac{\Pi_{\langle\mathbf{u},k\rangle} : \langle\mathbf{u},k\rangle[i_k]^{f(\mathbf{u},k)}}{\Pi_{\langle\mathbf{v},l\rangle} : \langle\mathbf{v},l\rangle[j_l]^{g(\mathbf{v},l)}} \tag{5}$$

This details the shape that was announced in Equation (2).

4. DIMENSIONED ARRAYS

A dimensioned array combines the extended array shape introduced in Section 2 with the actual numbers. An array value has the form $\langle[c^{p\cdot q}],a,[m^k],u,[n^l],v\rangle$, with $p=\Pi_i : m_i$ and $q=\Pi_i : n_i$. How the numbers $[c^{p\cdot q}]$ in the array are exactly stored is implementation dependent. The array may for example be sparse and not store all numbers, but for the shape this makes no difference.

Figure 1 gives evaluation rules for common linear algebra operations. For all operators it is assumed that the operation on the numbers is understood. The sum operation adds two numerical arrays. It requires that the argument arrays have the same sizes and also the same units. The element-wise product requires that the argument arrays have the same sizes, but the units can be different. The unit of the product is the product of the units. The element-wise exponent expects a single number as second argument and raises the numbers and the units to that power. The matrix product generalizes Equation (3) and adds numbers to it. As usual the matrix product is overloaded with scaling. The first argument is a scalar and multiplied with every element from the second argument. Scaling from the right could be added similarly. The rule for the transpose is in line with Equation (4). While swapping the indices and the units, it also takes the reciprocal of the units. The rule for the reciprocal is given next. Its notation should not be confused with the inverse. The final rule is the dimensional inverse. It combines the transpose and the reciprocal as explained in Section 3. All these operators except for the last one, are dimensioned variants of the regular linear algebra operations.

Dimensioned arrays give rise to various operators that are specific for the extension with units of measurement. Figure 2 gives evaluation rules for various operations on array shapes and units. Just like for the operators from Figure 1 it is assumed that unboxed operators for the data part exist. Operator `getnum` retrieves an individual number from an array. Operator `getunit` retrieves an individual unit from an array. Functions `row` and `column` get a single row or column from an array. They get the numbers and the right units. Functions `unitfactor`, `rowunit` and `columnunit` retrieve

$$\frac{\begin{array}{c} x \Rightarrow \langle [c^p], a, [m^k], u, [n^l], v \rangle, \\ y \Rightarrow \langle [c'^p], a, [m^k], u, [n^l], v \rangle \end{array}}{x + y \Rightarrow \langle [c^p] + [c'^p], a, [m^k], u, [n^l], v \rangle}$$

$$\frac{\begin{array}{c} x \Rightarrow \langle [c'^p], a, [m^k], u, [n^l], v \rangle, \\ y \Rightarrow \langle [c'^p], a', [m^k], u', [n^l], v' \rangle \end{array}}{x \times y \Rightarrow \langle [c^p] \times [c'^p], a \times a', [m^k], u \times u', [n^l], v \times v' \rangle}$$

$$\frac{\begin{array}{c} x \Rightarrow \langle [c^p], a, [m^k], u, [n^l], v \rangle, \\ y \Rightarrow \langle [i], 1, [1], 1, [], 1 \rangle \end{array}}{x \mathbin{\hat{}} y \Rightarrow \langle [c^p]^i, a^i, [m^k], u^i, [n^l], v^i \rangle}$$

$$\frac{\begin{array}{c} x \Rightarrow \langle [c^p], a, [m^k], u, [s^d], v \rangle, \\ y \Rightarrow \langle [c'^{p'}], a', [s^d], v, [n^l], w \rangle \end{array}}{x \cdot y \Rightarrow \langle [c^p] \cdot [c'^{p'}], a \times a', [m^k], u, [n^l], w \rangle}$$

$$\frac{\begin{array}{c} x \Rightarrow \langle [c], a, [], 1, [], 1 \rangle \ , \\ y \Rightarrow \langle [c'^p], a', [m^k], u, [n^l], v \rangle \end{array}}{x \cdot y \Rightarrow \langle [c] \cdot [c'^p], a \times a', [m^k], u, [n^l], v \rangle}$$

$$\frac{x \Rightarrow \langle [c^p], a, [m^k], u, [n^l], v \rangle}{x^T \Rightarrow \langle [c^p]^T, a, [n^l], v^{-1}, [m^k], u^{-1} \rangle}$$

$$\frac{x \Rightarrow \langle [c^p], a, [m^k], u, [n^l], v \rangle}{x^{-1} \Rightarrow \langle [c^p]^{-1}, a^{-1}, [m^k], u^{-1}, [n^l], v^{-1} \rangle}$$

$$\frac{x \Rightarrow \langle [c^p], a, [m^k], u, [n^l], v \rangle}{x^{\sim} \Rightarrow \langle [c^p]^{\sim}, a^{-1}, [n^l], v, [m^k], u \rangle}$$

Figure 1: Evaluation of numerical array operations. From top to bottom they are the sum, the element-wise product, the element-wise exponent, the matrix product, scaling, transpose, reciprocal and the dimensional inverse.

$$\frac{\begin{array}{c} x \Rightarrow \langle [c^p], a, [m^k], u, [n^l], v \rangle, \\ i \Rightarrow \langle [i^k], 1, [k], 1, [], 1 \rangle, \\ j \Rightarrow \langle [j^l], 1, [l], 1, [], 1 \rangle \end{array}}{\text{getnum}\ x\ i\ j \Rightarrow \langle \text{getnum}\ [c^p]\ [i^k]\ [j^l], 1, [], 1, [], 1 \rangle}$$

$$\frac{\begin{array}{c} x \Rightarrow \langle [c^p], a, [m^k], u, [n^l], v \rangle \\ i \Rightarrow \langle [i^k], 1, [k], 1, [], 1 \rangle \\ j \Rightarrow \langle [j^l], 1, [l], 1, [], 1 \rangle \end{array}}{\text{getunit}\ x\ i\ j \Rightarrow \langle 1, a \times u[i^k]/v[j^l], [], 1, [], 1 \rangle}$$

$$\frac{\begin{array}{c} x \Rightarrow \langle [c^p], a, [m^k], u, [n^l], v \rangle \\ i \Rightarrow \langle [i^k], 1, [k], 1, [], 1 \rangle \end{array}}{\text{row}\ x\ i \Rightarrow \langle \text{row}\ [c^p]\ [i^k], a \times u[i^k], [], 1, [n^l], v \rangle}$$

$$\frac{\begin{array}{c} x \Rightarrow \langle [c^p], a, [m^k], u, [n^l], v \rangle \\ j \Rightarrow \langle [j^l], 1, [l], 1, [], 1 \rangle \end{array}}{\text{column}\ j\ a \Rightarrow \langle \text{column}\ [c^p]\ [j^l], a/v[j^l], [m^k], u, [], 1 \rangle}$$

$$\frac{x \Rightarrow \langle [c^p], a, [m^k], u, [n^l], v \rangle}{\text{unitfactor}\ x \Rightarrow \langle [1], a, [], 1, [], 1 \rangle}$$

$$\frac{x \Rightarrow \langle [c^p], a, [m^k], u, [n^l], v \rangle}{\text{rowunit}\ x \Rightarrow \langle [1^p], 1, [m^k], u, [], 1 \rangle}$$

$$\frac{x \Rightarrow \langle [c^p], a, [m^k], u, [n^l], v \rangle}{\text{columnunit}\ x \rightarrow \langle [1^p], 1, [n^l], v, [], 1 \rangle}$$

$$\frac{x \Rightarrow \langle [c^p], a, [m^k], u, [n^l], v \rangle}{\text{magnitude}\ x \rightarrow \langle [c^p], 1, [m^k], 1, [n^l], 1 \rangle}$$

Figure 2: Evaluation of array operations concerning shape and units.

the individual unit parts from the shape. They can be combined to create the complete unit array as follows:

$$\text{unit}\, x = \text{unitfactor}\, x \ \cdot \ \text{rowunit}\, x \ \cdot \ (\text{columnunit}\, x)^{\sim}$$

Finally function `magnitude` gets the numbers only. Any dimensioned array x can be factored into product `magnitude`$x \times$ `unit` x.

Some special facilities for constructing, manipulating and converting arrays are given in Figure 3. The first two rules are for special syntax `|...|` to write literal units. For example `|kilogram|` for the unit `kg`. The second form is for unit arrays. In that case an identifier for a unit array should be given between the pipes, for example `|prod_unit|`. In both case the magnitude is one.

The outer product $\otimes$ combines two arrays into a higher ranked one, whereas the projection operator $\downarrow$ reduces an array's rank. In an outer product, the array descriptors are concatenated and the numbers are combined into a tensor. The $\otimes_n$ operator does the required operation on units by shifting the bases in the second argument n positions:

$$u \ \otimes_n \Pi\langle b,k\rangle^{f\langle b,k\rangle} = u \times \Pi\langle b,k{+}n\rangle^{f\langle b,k\rangle}$$

This is in line with $\otimes$'s definitions in Section 3. The projection operator $\downarrow$ reduces an array's rank in the contravariant dimension, by summarizing the array values. The summary can be the usual operations like summing, averaging, or counting. It is a requirement that the projected columns have appropriate units for the summarizing operation. For example if some dimensions are summed, the units of all the summed entries must be the same.

The final operation is the construction of a conversion matrix. Given two dimensioned arrays of the same size but possibly different units, it creates a matrix that converts arrays from one unit to the other, or throws an error if the units are not compatible. For any $[i^k]$ the unit $a'/u[i^k]$ should be dimensionally equivalent to unit $u'[i^k]/a$. Expression `conv` $[m^k]$ a a' u u' creates a diagonal matrix with the appropriate conversion factors. Multiplying a au matrix with the conversion matrix gives an a'u' matrix.

5. THE MATRIX TYPE

The matrix type is an expression of the form δ per δ, where each dimension expression δ is an algebraic expression with names of units and dimensioned vector spaces as base elements. The syntax of dimension expressions δ is given in the grammar from Figure 4. It extends Kennedy's scalar dimension expressions with dimensioned vectors and the Kronecker product. The first dimension

$$\frac{}{|\,a\,| \Rightarrow \langle [1], a, [], 1, [], 1 \rangle}$$

$$\frac{}{|\,u\,| \Rightarrow \langle [1^m], 1, [m], u, [], 1 \rangle}$$

$$\frac{x \Rightarrow \langle [c^p], a, [m^k], u, [n^l], v \rangle \qquad y \Rightarrow \langle [c'^{p'}], a', [m'^{k'}], u', [n'^{l'}], v' \rangle}{x \otimes y \Rightarrow \langle [c^p] \otimes [c'^{p'}], a \times a', [m^k, m'^{k'}], u \otimes_k u', [n^l, n'^{l'}], v \otimes_l v' \rangle}$$

$$\frac{x \Rightarrow \langle [c^p], a, [m^k], u, [n^l], v \rangle \qquad y \Rightarrow \langle [r^d], 1, [d], 1, [], 1 \rangle}{x \downarrow y \Rightarrow \langle [c^p] \downarrow [r^d], a, [m^k] \downarrow [r^d], u \downarrow [r^d], [n^l], v \rangle}$$

$$\frac{x \Rightarrow \langle [1^p], a, [m^k], u, [], 1 \rangle \qquad y \Rightarrow \langle [1^p], a', [m^k], u', [], 1 \rangle}{\texttt{conv}\ x\ y \Rightarrow \langle \text{conv}\ [m^k]\ a\ a'\ u\ u', a'/a, [m^k], u', [m^k], u \rangle}$$

Figure 3: Special primitives for tensors and conversions.

expression in the matrix type describes a matrix' row dimension and the second one a matrix' column dimension. Together they describe the shape of a matrix, just as notations like $\mathbb{R}^{m \times n}$ or the more general $\mathbb{R}^{\mathrm{P} \times \mathrm{Q}}$ from mathematics, but refines it to dimensioned vector spaces.

With type expressions, a large variety of matrix types can be defined. Figure 5 lists some typical examples. Remember that $P!u$ is read as some uom vector or vector space with index set P. As usual a column vector is a $n \times 1$ matrix, a row vector is a $1 \times n$ matrix, and a scalar is a 1×1 matrix. In the table, and throughout the paper, the convention is used to omit the `per 1` part when the column dimension is 1. This makes the type more readable and a proper extension of the scalar case. For the vector case it is useful to distinguish dimensionless, homogeneous and heterogeneous vectors. Dimensionless and heterogeneous vectors have their own notation. A homogeneous vector type is a product of a scalar unit with a dimensionless vector. It is a special subset of all the possible matrix types that plays a role when typing indexing operations.

The Kronecker product adds support for multi-dimensional numerical data via the matricization of tensors. A $\langle k, l \rangle$ tensor has type $X_1 \otimes \cdots \otimes X_k$ per $Y_1 \otimes \cdots \otimes Y_l$ where expressions X_i type the row dimensions and expressions Y_i type the column dimensions. In type expressions the Kronecker product separates the different matrix dimensions. For example the $\langle 3, 0 \rangle$ tensor from Section 2.3 is typed as `Region!` $\otimes$ `Year!` $\otimes$ `Commodity!unit`. With matricization, tensors are turned into matrices with compound indices. This hides the extension from the matrix type and gives only the multi-dimensional data structure of tensors, not the tensor calculus. This is achieved by disallowing variables in Kronecker products. So an expression like `Region!` $\otimes P \otimes$ `Commodity!unit` with P a variable is not allowed. Unification variables can only match the whole row and column dimensions, and not parts of them. The matrix type is unsuitable for the varying number of indices in tensor calculus, but via matricization multi-dimensional data can be handled.

5.1 Unification

Unification for units of measurement requires the notion of dimension equivalence and a normal form, because it is a semantic unification problem. Kennedy defines equivalence relation $=_D$ as

δ	$::= 1$	unit dimension
	$\mid d$	dimension variables
	$\mid B$	base dimensions
	$\mid \xi!\mu$	dimensioned vector
	$\mid \delta^{-1}$	reciprocal
	$\mid \delta \times \delta$	element-wise product
	$\mid \delta \otimes \delta$	Kronecker product (δ without variables)
ξ	$::= I$	index set
	$\mid p$	index variable
μ	$::= 1$	unit dimension
	$\mid d$	dimension variable
	$\mid U$	base vector

Figure 4: Syntax of Kennedy's dimension expressions, extended with the $\xi!\mu$ notation for dimensioned vector spaces. The ξ part denotes an index set and the μ part a particular space.

Type	Short	Description
1 per 1	1	dimensionless scalar
a per 1	a	dimensioned scalar
$P!$ per 1	$P!$	dimensionless vector
$P!u$ per 1	$P!u$	dimensioned vector
$a \cdot P!$ per 1	$a \cdot P!$	homogeneous dimensioned vector
1 per $Q!v$		dimensioned row vector
$P!$ per $Q!$		dimensionless rectangular matrix
$P!u$ per $P!v$		square matrix
$P!u$ per $P!u$		even more square matrix
...	...	...

Figure 5: Some examples of matrices that can be typed with the matrix type. The type can indicate a scalar, a vector or a matrix. The shorthand allows the omission of an empty column dimension.

the congruence on dimension expressions generated by the following equations:

$$\begin{array}{rcll}
\delta_1 \times \delta_2 & =_D & \delta_2 \times \delta_1 & \text{commutativity} \\
(\delta_1 \times \delta_2) \times \delta_3 & =_D & \delta_1 \times (\delta_2 \times \delta_3) & \text{associativity} \\
1 \times \delta & =_D & \delta & \text{identity} \\
\delta \times \delta^{-1} & =_D & 1 & \text{inverses}
\end{array}$$

This equivalence gives the abelian group required for units of measurement. A dimension expression is normalized if it is of the form $d_1^{x_1} \cdots d_m^{x_m} \cdot B_1^{y_1} \cdots B_n^{y_n}$. This assumes some ordering on the unit variables d and base units B. The normal form of some dimension expression μ is written as $nf(\mu)$.

Kennedy's unification algorithm uses the normal form in the computation of the most general unifier for two units. The algorithm is based on Knuth's adaptation of Euclid's greatest common divisor algorithm. For two elements from the free abelian group of units it computes the most general unifier, or it fails if no such unifier exists.

ALGORITHM 1 (KENNEDY'S UNIT UNIFICATION).

```
$UnifyUnits(\delta_0, \delta_1) = UnitUnify(\delta_0 \cdot \delta_1^{-1})$

$UnitUnify(\delta) =$
  let $nf(\delta) = d_1^{x_1} \cdots d_m^{x_m} \cdot B_1^{y_1} \cdots B_n^{y_n}$
  where $|x_1| \leq \cdots \leq |x_m|$
  in
    if $m = 0$ and $n = 0$ then $\{\}$
    if $m = 0$ and $n \neq 0$ then fail
    else if $m = 1$ and $x_1 | y_i$ for all i then
      $\{d_1 \mapsto B_1^{-y_1/x_1} \cdots B_n^{-y_n/x_1}\}$
    else if $m = 1$ then fail
    else $S \circ U$ where
      $U = \{d_1 \mapsto d_1 \cdot d_2^{-\lfloor x_2/x_1 \rfloor} \cdots d_m^{-\lfloor x_m/x_1 \rfloor} \cdot$
              $\cdot B_1^{-\lfloor y_1/x_1 \rfloor} \cdots B_n^{-\lfloor y_n/x_1 \rfloor}\}$
      $S = UnitUnify(U(\delta))$
```

The algorithm is sound and complete.

The unification of the matrix also requires a normal form, and additionally requires breaking down the compound identifiers in the $\xi!\mu$ notation. The main operators in the syntax are the normal element-wise product and the Kronecker product. The Kronecker product forms the dimensions while the element-wise product is used within each dimensions. Therefore the row and the column dimension of the matrix type are arranged into Kronecker products of element-wise products. If furthermore bi-linearity is used to move all constants to the front then any matrix type can be written as

$$a_1^{x_1} \cdots a_s^{x_s} \cdot \bigotimes_{1 \leq i \leq k} \prod_{1 \leq j \leq m_i} P_i!u_{ij}^{y_{ij}} \quad \text{per} \quad \bigotimes_{1 \leq i \leq l} \prod_{1 \leq j \leq n_i} Q_i!v_{ij}^{z_{ij}}$$

with m_i the number of bases in the units of the i-th row dimension and with n_i the number of bases in the units of the i-th column dimension. Essentially this form is the $\mathrm{Hom}(V,W)$ space of linear maps $V \to W$. This form is still not suitable for unification because the combination of two identifiers in the $\xi!\mu$ notation is not allowed in the abelian group's base. Instead the following isomorphic tuple form is chosen:

$$\langle a_1^{x_1} \cdots a_s^{x_s},\ [P_1, \ldots, P_k],\ u,\ [Q_1, \ldots, Q_l],\ v\rangle$$

with

$$u = \prod_{1 \leq i \leq k} \prod_{1 \leq j \leq m_i} \langle u_{ij}, i\rangle^{y_{ij}} \text{ and } v \quad = \prod_{1 \leq i \leq l} \prod_{1 \leq j \leq n_i} \langle v_{ij}, i\rangle^{z_{ij}}$$

Units u and v contain the unit vectors paired with an integer that indicates to which dimension it belongs. Since the index sets P_i and Q_i must be the same within each dimension i because it is an element-wise product, it follows that the pairing of the units vectors is sufficient to reconstruct the matrix type from the tuple. In this form the index sets are separated from the vector space identifiers. The remaining expressions in the first, the third and the fifth position are dimension expressions in the Kennedy's original form and suitable for unit unification.

The tuple form of the matrix type is almost the form of dimensioned arrays. If the index sets are replaced by their sizes then the forms are the same. This is the mapping from matrix types to dimensioned arrays that was discussed in Section 2.4.

Unification for the matrix type is defined component-wise on the tuple form. The three cases with dimension expressions are unified with the unification algorithm for units, the other two with Hindley-Damas-Milner style unification,

ALGORITHM 2 (MATRIX TYPE UNIFICATION).

```
$UnifyMatrices(\tau, \tau') = S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1$

  with
  $nf(\tau) = \langle a, P, u, Q, v\rangle$
  $nf(\tau') = \langle a', P', u', Q', v'\rangle$
  and
    $S_1 = UnifyUnits(a, a')$
    $S_2 = Unify(P, P')$
    $S_3 = Unify(S_2(Q), S_2(Q'))$
    $S_4 = UnifyUnits(u, u')$
    $S_5 = UnifyUnits(S_4(v), S_4(v'))$
```

The correctness of this matrix type unification follows directly from the correctness of unit and standard unification.

THEOREM 2 (KENNEDY). *For any δ the algorithm $UnitUnify(\delta)$ terminates with failure or with a substitution S, and:*

- *If $UnifUnify(\delta) = S$ then $S(\delta) =_D 1$*
- *If $S'(\delta) = 1$ then $UnitUnify(\delta) = S$ such that $S' =_D R \circ S$ for some substitution R.*

COROLLARY 1. *For every matrix type τ_0 and τ_1 the unification algorithm $UnifyMatrices(\tau_0, \tau_1)$ terminates with failure or with a substitution S satisfying $S(\tau_0) =_D S(\tau_1)$*

PROOF. *Since the dimension expressions in a normalized matrix type are reduced to Kennedy's original syntax, it follows that unit unification is applicable to these three cases. The other two cases are the index sets. These are just variables or identifiers and can trivially be unified with standard syntactic unification. Since unification is composable the result follows from each of the five unification occurrences in Algorithm 2.* $\square$

5.2 Type Inference

As usual we start with the definition of types and expressions. A type schema introduces quantified variables. Index set variables p are added to the unit variables d and the normal type variables t.

$\sigma ::= \tau$	types
$\mid \forall t.\sigma$	type quantification
$\mid \forall d.\sigma$	dimension quantification
$\mid \forall p.\sigma$	index set quantification

$\tau ::= t$	type variable
$\mid \tau \to \tau$	function
$\mid \delta \text{ per } \delta$	matrix type

A type τ is a variable, a function, or a matrix type. The δ in the matrix type is given in Figure 4. In a practical setting other types could be added.

The syntax for expressions e is exactly the same as Damas' syntax. Lambda abstraction and function application are the core primitives.

$e ::= x$	variable
$\mid \lambda x.e$	abstraction
$\mid e\ e$	application
$\mid \text{let } x = e \text{ in}$	let binding

$$\frac{\Gamma \vdash x\colon a \cdot P!u \text{ per } Q!v, \quad \Gamma \vdash y\colon a \cdot P!u \text{ per } Q!v}{\Gamma \vdash x + y\colon a \cdot P!u \text{ per } Q!v}$$

$$\frac{\Gamma \vdash x\colon a \cdot P!u \text{ per } Q!v, \quad \Gamma \vdash y\colon b \cdot P!w \text{ per } Q!z}{\Gamma \vdash x \times y\colon a \cdot b \cdot P!u \times P!w \text{ per } Q!v \times Q!z}$$

$$\frac{\Gamma \vdash x\colon P! \text{ per } Q!, \quad \Gamma \vdash y\colon 1}{\Gamma \vdash x \,\hat{}\, y\colon P! \text{ per } Q!}$$

$$\frac{\Gamma \vdash x\colon a \cdot P!u \text{ per } Q!v, \quad \Gamma \vdash y\colon b \cdot Q!v \text{ per } R!w}{\Gamma \vdash x \cdot y\colon a \cdot b \cdot P!u \text{ per } Q!w}$$

$$\frac{\Gamma \vdash x\colon a, \quad \Gamma \vdash y\colon b \cdot P!u \text{ per } Q!v}{\Gamma \vdash x \cdot_c y\colon a \cdot b \cdot P!u \text{ per } Q!v}$$

$$\frac{\Gamma \vdash x\colon a \cdot P!u \text{ per } Q!v}{\Gamma \vdash x^T\colon a \cdot Q!v^{-1} \text{ per } P!u^{-1}}$$

$$\frac{\Gamma \vdash x\colon a \cdot P!u \text{ per } Q!v}{\Gamma \vdash x^{-1}\colon a^{-1} \cdot P!u^{-1} \text{ per } Q!v^{-1}}$$

$$\frac{\Gamma \vdash x\colon a \cdot P!u \text{ per } Q!v}{\Gamma \vdash x^{\sim}\colon a^{-1} \cdot Q!v \text{ per } P!u}$$

Figure 6: Types for numerical array operations corresponding with the evaluation rules from Figure 1.

$$\frac{\Gamma \vdash x\colon a \cdot P!u \text{ per } Q!v, \quad \Gamma \vdash i\colon P, \quad \Gamma \vdash j\colon Q}{\Gamma \vdash \text{getnum}\ x\ i\ j\colon 1}$$

$$\frac{\Gamma \vdash x\colon a \cdot P! \text{ per } Q!, \quad \Gamma \vdash i\colon P, \quad \Gamma \vdash j\colon Q}{\Gamma \vdash \text{getunit}\ x\ i\ j\colon a}$$

$$\frac{\Gamma \vdash x\colon a \cdot P! \text{ per } Q!v \quad \Gamma \vdash i\colon P}{\Gamma \vdash \text{row}\ x\ i\colon a \text{ per } Q!v}$$

$$\frac{\Gamma \vdash x\colon a \cdot P!u \text{ per } Q!, \quad \Gamma \vdash j\colon Q}{\Gamma \vdash \text{column}\ x\ j\colon a \cdot P!u}$$

$$\frac{\Gamma \vdash x\colon a \cdot P!u \text{ per } Q!v}{\Gamma \vdash \text{unitfactor}\ x\colon a}$$

$$\frac{\Gamma \vdash x\colon a \cdot P!u \text{ per } Q!v}{\Gamma \vdash \text{rowunit}\ x\colon P!u}$$

$$\frac{\Gamma \vdash x\colon a \cdot P!u \text{ per } Q!v}{\Gamma \vdash \text{columnunit}\ x\colon Q!v}$$

$$\frac{\Gamma \vdash x\colon a \cdot P!u \text{ per } Q!v}{\Gamma \vdash \text{magnitude}\ x\colon P! \text{ per } Q!}$$

Figure 7: Types of array operations corresponding with the evaluation rules from Figure 2.

As usual the let binding introduces type variables.

A context Γ contains type statements of the form $x\colon \sigma$. Type statement $\Gamma \vdash e\colon \tau$ means expression e has type τ given the types in context Γ. The following derivation rules are defined.

$$\frac{x\colon \sigma \in \Gamma}{\Gamma \vdash x\colon \sigma} \qquad \text{(Var)}$$

$$\frac{\Gamma \cup \{x\colon \tau\} \vdash e\colon \tau'}{\Gamma \vdash (\lambda x.e)\colon \tau \to \tau'} \qquad \text{(Abs)}$$

$$\frac{\Gamma \vdash e\colon \tau' \to \tau \qquad \Gamma \vdash e'\colon \tau'}{\Gamma \vdash (e\ e')\colon \tau} \qquad \text{(App)}$$

$$\frac{\Gamma \vdash e\colon \sigma \qquad \Gamma \cup \{x\colon \sigma\} \vdash e'\colon \tau}{\Gamma \vdash (\text{let}\ x = e\ \text{in}\ e')\colon \tau} \qquad \text{(Let)}$$

Next a rule for the index set variables is added to the generalization and instantiation rules for type variables and units variables.

$$\frac{\Gamma \vdash e\colon \sigma}{\Gamma \vdash e\colon \forall t.\sigma} \qquad \frac{\Gamma \vdash e\colon \sigma}{\Gamma \vdash e\colon \forall u.\sigma} \qquad \frac{\Gamma \vdash e\colon \sigma}{\Gamma \vdash e\colon \forall p.\sigma} \qquad \text{(Gen)}$$

$$\frac{\Gamma \vdash e\colon \forall t.\sigma}{\Gamma \vdash e\colon \sigma[t \to \tau]} \qquad \frac{\Gamma \vdash e\colon \forall u.\sigma}{\Gamma \vdash e\colon \sigma[u \to \mu]} \qquad \frac{\Gamma \vdash e\colon \forall p.\sigma}{\Gamma \vdash e\colon \sigma[p \to \varepsilon]} \qquad \text{(Inst)}$$

These rules handle the three kinds of variables.

Finally type rules for the linear algebra operators are defined. Figure 6 gives type rules for common linear algebra operations, All operator types are the most general ones except the type of the exponent operator. A comparison with the evaluation rules from Figure 1 shows that all other types follow the evaluation rules exactly, but that the type for the exponent operator is dimensionless and lacks terms for the exponent. The unit of the result depends on the value of the exponent i and the matrix type cannot express that, because that would require dependent typing. The current types are correct, but they are special cases. The element-wise power is restricted to dimensionless matrices.

The restrictions of the exponent operator is mitigated by the ability to define static element-wise powers like the element-wise square and cube functions. For example the square function is simply the element-wise product of its argument with itself. This function can be typed correctly because the exponent is known at compile time.

```
define element-wise_square(x) = x .* x
↪ element-wise_square :: for_index D,E: for_unit a,b,c:
      (a*D!b per E!c) → a^2*D!b^2 per E!c^2
```

All units are squared as they are supposed to be. Similarly a `cube` function can be defined and so on, but generic power functions would require dependent typing. Functions like square and cubes have been sufficient for all cases we have encountered.

A second difference between the evaluation and the type rules is in the operator for scaling. It is customary to overload the matrix product with scaling, but the types of these two operators cannot be expressed in a single rule. Therefore the type variant has special operator $\cdot_c$ for scaling. It expects a scalar as first argument and scales the second argument with it.

A major difference between the operators for units of measurement from Figure 2 and the type rules from Figure 7 is that the functions to access array elements are restricted to matrices with

$$\frac{}{\Gamma \vdash |a| : a}$$

$$\frac{}{\Gamma \vdash |P!u| : P!u}$$

$$\frac{\Gamma \vdash x : a \cdot P!u \text{ per } Q!v, \quad \Gamma \vdash y : a' \cdot P'!u' \text{ per } Q'!v'}{\Gamma \vdash x \otimes y : a \cdot a' \cdot P!u \otimes P'!u' \text{ per } Q!v \otimes Q'!v'}$$

$$\frac{\Gamma \vdash x : a \cdot P_1!u_1 \otimes \ldots \otimes P_i! \otimes \ldots \otimes P_k!u_k \text{ per } Q!v}{\Gamma \vdash x \downarrow i : a \cdot P_1!u_1 \otimes \ldots \otimes P_k!u_k \text{ per } Q!v}$$

$$\frac{\Gamma \vdash x : a \cdot P!u, \quad \Gamma \vdash y : b \cdot P!v}{\Gamma \vdash \texttt{conv}\, x\, y : a/b \cdot P!u \text{ per } P!v}$$

Figure 8: Types of array operations corresponding with the evaluation rules from Figure 3.

homogeneous units of measurement. Since the type system relies on index-free constructions we cannot expect statically typed indexing. Parametric types are homogeneous as was explained in Section 2, but since a matrix can be factored into a magnitude and a unit matrix, any indexing operations can always be performed on the magnitudes so no expressiveness is lost. However, the type system is most beneficial when using an index-free and functional programming style.

Figure 8 gives type rules for the tensor and conversion primitives. They follow the mapping from the matrix type to dimensioned arrays.

5.3 Type Erasure

The dimensioned arrays and the matrix type can be used in various ways to create a unit-aware matrix language. A completely unit aware language can read and write in proper units of measurement, check the unit correctness of computations, do conversions, etc. The extended arrays have all the information to do these tasks, but the extra bookkeeping results in some performance overhead. The matrix type catches errors early, so when the units are already shown correct at compile time the checks can be omitted. But the matrix type can also improve safety without runtime support. In that case input and output with units of measurement is impossible, but the computed numbers are the same as when the units would be attached. The only problematic case is conversion. If the conversion matrix is not known at compile time, then the runtime system must compute the proper conversion factors at runtime.

The runtime overhead can drastically be reduced if the unit computation can be decoupled from the number computations. The combination of units and numbers in dimensioned arrays leads to constant boxing and unboxing. Instead we can use the compile time type and generate for each function two functions, one for the runtime type and one for the numbers. For a function of type $X \to Y$ the actual argument's runtime type is matched with type X and substituted in Y. This yields exactly the same runtime type as the boxed version but far more efficient. In a 3D growth model for a biology application the overhead for boxed units of measurement was around 10%. In the decoupled version this dropped to 0.01%. The relatively high overhead for the boxed version was expected because all the geometry vectors in this case contain just three numbers.

6. THE PACIOLI LANGUAGE

The Pacioli language is a proof of concept for a unit-aware statically typed matrix language. Implementing Pacioli for various environments gave valuable insight in details of the matrix type and the required support from the runtime system. The matrix type fits well in a parametric polymorphic type system and besides the matrix type the compiler is fairly standard. The matrix type is a sizable but localized extension of the type system. The effect on the runtime system is more substantial. The runtime needs to implement primitive functions and data types, conversion factors for units of measurement, unit symbols, etc. This requires implementing mechanisms to fetch data like index sets and unit vectors dynamically.

Pacioli has its own unit-aware runtime system called the Matrix Virtual Machine (MVM), but to experiment with various runtime options the compiler can also generate Matlab and JavaScript code. All three runtime implementations use existing linear algebra libraries to run the code that was generated by the compiler.

The Matlab runtime supports all required primitives, but without units of measurement. The generated Matlab code is not unit-aware but since the type system already guarantees unit correct operations it is still unit correct. Any computation will produce the same numbers, the only difference is that the units symbols are missing from the output. Obviously in this case there is no overhead from the units of measurement.

The Matrix Virtual Machine (MVM) is Pacioli's own light-weight unit aware runtime system. It interprets a language that maps by design exactly on the compiler's internal representation. Pacioli's unit-aware MVM uses an existing matrix implementation from a library and tags any matrix instance with its type. Having the type available makes it easy to determine a matrix' units and do all the tasks necessary for a unit-aware runtime system.

The JavaScript runtime library allows Pacioli scripts to be run in the browser. Vectors and matrices can be displayed in a web page with all numbers annotated with units of measurement. Furthermore, graphical representations like charts and 3D models are supported, all with automatic support for units of measurement where applicable.

In the JavaScript implementation the unit computation is completely decoupled from the numeric computation and reuses the types that were derived by the compiler. This reduces the overhead to an insignificant fraction of the total computation.

6.1 Case: Fourier Motzkin

A typical example where the distinction between matrices as linear transformations and matrices as general data structure is apparent is the Fourier-Motzkin algorithm. The algorithm is an adaptation of Martinez and Silva's algorithm to compute all invariants of a generalized Petri Net [5]. Let matrix **A** be an m×n integer matrix. The algorithm computes all positive integer vectors $\mathbf{y}^T$ such that $\mathbf{y}^T \cdot \mathbf{A} = 0$.

```
function FOURIER_MOTZKIN(A)
    B ← I_m
    for i := 1 to m do
        (1) Append to matrix [A|B] all rows which results as
        positive least common multiple linear combinations of
        row pairs out of [A|B] that both annul the i-th column
        of A and are minimal rows in B
        (2) Eliminate from matrix [A|B] the rows in which
        the i-th column of A is non null
    end for
    return B's rows
end function
```

The use of matrices in the algorithm is problematic. First matrices **A** and **B** are glued together. The effect is that operations on **A** are also carried out on **B**. Next this matrix grows and shrinks, until it is broken down into a list at the end. The growing and shrinking matrix is not a linear transformation and cannot be given a meaningful matrix type.

To enable meaningful types the Pacioli version replaces the matrix by a list of column vector pairs. It does not create matrix $[\mathbf{A}|\mathbf{I}]$ but creates the set of columns vector pairs $\{(\mathbf{a}_k, \mathbf{i}_k) \mid k < n\}$. Scaling is lifted to such pairs by $\alpha(\mathbf{a}, \mathbf{b}) = (\alpha\mathbf{a}, \alpha\mathbf{b})$ and addition by $(\mathbf{a}, \mathbf{b}) + (\mathbf{c}, \mathbf{d}) = (\mathbf{a}+\mathbf{c}, \mathbf{b}+\mathbf{d})$. With these changes the algorithm can be written using standard idiom for lists:

```
define Fourier_Motzkin(matrix) =
    let
        rows = row_domain(matrix),
        pairs = zip(columns(matrix),
                    columns(right_identity(matrix)))
    in
        [b | (_, b) ∈ loop_list(pairs, eliminate, rows)]
    end

↪fourier_motzkin ::
    for_index P,Q: for_unit a,P!u: (a*P!u per Q!) → List(Q!)
```

Function eliminate implements the growing and shrinking of the list. Its type and the type of the Fourier Motzkin function are derived correctly without any annotation.

7. RELATED WORK

The efficiency and correctness of functional implementations of matrix and vector operations is a large and active research topic, but despite the numerical nature of the domain it is rather unconnected from language support for units of measurement. Type inference for matrix languages like Matlab is continuously improved to optimize code and remove run-time overhead, or to increase software quality or reuse. In general purpose functional languages the implementation of linear algebra operations is a typical use case for array optimizations. What most of these approaches have in common is some form of shape inference or algebraic rewriting.

Extending language syntax or typing rules with units of measurement is a common approach to increase the safety of numerical software. Recent related work besides Kennedy's unit inference is the MetaGen extension of Java that statically checks units of measure [2]. Dimensions and units can be formulated in a nominally typed object-oriented language through the use of statically typed meta-classes. Another recent development is the Osprey tool [9]. This tool extends the C programming language with type annotations that can be statically checked for errors in units of measurement. The prototype was extensively validated on mature code bases of significant size and discovered many errors. The advantages of polymorphic units is demonstrated in the application of a Haskell unit library to astrophysics research [15]. The library is dimension-monomorphic but unit-polymorphic, which means that the library interface specifies the dimensions (e.g. length), but a library user is free to call the library with any units he wants (e.g. feet or metre). Kennedy's units in F# and Pacioli's units are dimension-polymorphic. However, according to Kennedy the choice between units of dimensions is mostly a matter of taste, because the relations that hold between dimensions also holds between the units in which those dimensions are measured. None of these approaches provide support for data with heterogeneous units of measurement, or support for matrices or linear algebra.

Many type inference techniques have been developed to eliminate runtime checks or to improve code generation for matrix languages. Various Matlab compilers use a lattice-based type inference technique based on fixed-point solutions that was influenced by the FALCON and other APL and SETL compilers. The Matlab D compiler is a parallel Matlab compiler that uses a telescoping languages framework to generate Fortran with MPI from Matlab scripts [7]. Type analysis is performed to obtain high performance parallelism from high-level Matlab scripts. MaJIC (Matlab Just-In-Time Compiler) employs a combination of just-in-time and speculative ahead-of-time compilation [3]. It consists of an extremely fast type inference engine that makes a conservative estimate of the types of expressions. Its goal is to remove as much runtime overhead as possible without sacrificing the interactive nature of Matlab. The MAGICA type inference engine overcomes limitations of the lattice-based type inference technique by modeling the language's shape semantics with an algebraic system and applying term rewriting techniques to evaluate expressions under this algebra [10]. Like unit inference with unit variables, this enables the deduction of valuable shape information even when array extents are compile-time unknowns. However, this information cannot always be derived and is much less refined than unit inference. Specific support for linear algebra in Haskell is promoted as statically typed linear algebra in [6]. It aims to catch errors in matrix and vector operations by statically verifying properties specific to linear algebra. However, no checks beyond array shape are performed, and it is for instance mentioned that forgetting to invert a matrix would not be detected by statically typed linear algebra. This paper's matrix type would detect it, just as in the missing transpose case in Section 2.2. The advantages of statically typed linear algebra is shown in the linear algebra library interface from [1]. It verifies the consistency of matrix operations by generative phantom types in ML.

In general purpose functional languages the implementation of efficient data types like functional arrays is continuously improved. Support for operations like those from linear algebra is a typical use case for optimizations or verification. Functional matrix operations are explicitly discussed as application of higher-order vectorization in [14]. A matrix is represented as an array of rows and operations on it are transformed into vectorized form. This form is implemented efficiently in Data Parallel Haskell (DPH). In [4] the level of abstraction for sparse matrix programs is raised from imperative code with loops to functional programs with comprehensions and limited reductions. It then uses Isabelle/HOL to verify full functional correctness of programs. In the Single Assignment C (SaC) array language , shape inference helps prevent runtime errors and also improves the code generation with respect to runtime efficiency [16]. The type system uses a hierarchy of array types, containing varying levels of shape information, and infers shapes as specific as possible. Type checks are postponed to runtime when exact shapes cannot be inferred statically. The shape information allows SaC to generate code that is competitive with hand-optimized code. A novel approach to multi-dimensional arrays in Haskell is presented in [11]. The purely functional arrays support reuse through shape polymorphism, which is embedded in Haskell's type system using type classes and type families. The embedding in Haskell and the strong typing and purity gives it advantages over languages like APL, J and Matlab. Although this approach is more tailored towards matrices, it still suffers from the restriction to homogeneous elements. The parametric array type expects an element type, so it cannot express different units of measurement for different elements. All these approaches share with Pacioli the emphasis on functional index-free code.

8. CONCLUSION

Explicit support for linear algebra operations in array programming can increase the safety of numerical software. We developed dimensioned arrays and a corresponding matrix type that infers types up to dimensioned vector spaces. Dimensioned arrays increase safety of numerical software by guaranteeing unit correct operations. The matrix type allows sound and complete type inference for matrix expressions up to complete dimensioned vector spaces. The type's naming scheme provides sufficient information about shapes and units of measurement to infer the most general type for linear algebra expressions. The inference technique is based on unit unification; the abstraction from units to vector spaces follows from the correspondence between heterogeneous units of measurement and complete dimensioned vector spaces. Besides the practical benefit of adding units of measurement and dimensional analysis, the matrix type's inference of dimensioned vector spaces guarantees safe operations beyond the possibilities of current shape analysis.

Experience from the extension of the Pacioli language with a prototype implementation of the matrix type has shown that type inference for linear algebra is practically feasible and that the matrix type is well-suited for a type system with parametric polymorphism and a functional index-free coding style. The language was applied to problems from a variety of domains and successfully embedded in various environments. The general index sets of the matrix type and operations on it combine well with parametric typing. The largest effect of the static typing is that instead of flexible indexing facilities the language features more functional idioms like lists, tuples and comprehensions.

The matrix type can type almost all numerical operators, but it is more strict than normal arrays for array manipulation. The overloaded matrix product is split into an operator for scaling and an operator for the dot product, because the units in these operators cannot be given a most general common type by a single rule. Operators using indices cannot be typed so easily as the index-free operators, because operators for indexing lead to homogeneous units. However, code using indexing can always be applied to a vector's magnitudes. Since a dimensioned vector or matrix can always be factored into a magnitude vector and a unit vector and broken down into its elements, it follows that indexing is even possible for heterogeneous units, but at the expense of safety. This shows that the language is not limited in any fundamental way, although it favors a more functional style. The functional idiom however increases the type safety.

The units of measurement scale well, and the overhead is negligible when the static type system derives the units at compile time. Pacioli's own runtime system MVM uses boxed values and although it scales well, it still suffers from some overhead. The generated JavaScript code computes the units of measurement separately from the numbers and reuses the types that were inferred by the compiler. This eliminates the overhead almost completely while providing full support for units of measurement.

9. REFERENCES

[1] A. Abe and E. Sumii. A simple and practical linear algebra library interface with static size checking. In O. Kiselyov and J. Garrigue, editors, *Proceedings ML Family/OCaml Users and Developers workshops, Gothenburg, Sweden, September 4-5, 2014*, volume 198 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–21. Open Publishing Association, 2015.

[2] E. E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, and G. L. S. Jr. Object-oriented units of measurement. In J. M. Vlissides and D. C. Schmidt, editors, *OOPSLA*, pages 384–403. ACM, 2004.

[3] G. S. Almasi and D. A. Padua. Majic: Compiling Matlab for speed and responsiveness. In J. Knoop and L. J. Hendren, editors, *PLDI*, pages 294–303. ACM, 2002.

[4] G. Arnold, J. Hölzl, A. S. Köksal, R. Bodík, and M. Sagiv. Specifying and verifying sparse matrix codes. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 249–260, New York, NY, USA, 2010. ACM.

[5] J. M. Colom and M. Silva. Convex geometry and semiflows in p/t nets: A comparative study of algorithms for computation of minimal p-semiflows. In *Proceedings on Advances in Petri Nets 1990*, APN 90, pages 79–112, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[6] F. Eaton. Statically typed linear algebra in Haskell. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 120–121, New York, NY, USA, 2006. ACM.

[7] M. Fletcher, C. McCosh, G. Jin, and K. Kennedy. Compiling parallel Matlab for general distributions using telescoping languages. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, volume 4, pages IV–1193–IV–1196, 2007.

[8] G. W. Hart. The theory of dimensioned matrices. In *Proceedings of the 5th Society for Industrial and Applied Mathematics Conference on Applied Linear Algebra*, 1994.

[9] L. Jiang and Z. Su. Osprey: a practical type system for validating dimensional unit correctness of C programs. *Software Engineering, International Conference on*, 0:262–271, 2006.

[10] P. G. Joisha and P. Banerjee. An algebraic array shape inference system for Matlab&Reg;. *ACM Trans. Program. Lang. Syst.*, 28(5):848–907, Sept. 2006.

[11] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM.

[12] A. Kennedy. Dimension types. In D. Sannella, editor, *ESOP*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 1994.

[13] A. Kennedy. Types for units-of-measure: Theory and practice. In Z. Horváth, R. Plasmeijer, and V. Zsók, editors, *CEFP*, volume 6299 of *Lecture Notes in Computer Science*, pages 268–305. Springer, 2009.

[14] B. Lippmeier, M. M. Chakravarty, G. Keller, R. Leshchinskiy, and S. Peyton Jones. Work efficient higher-order vectorisation. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 259–270, New York, NY, USA, 2012. ACM.

[15] T. Muranushi and R. A. Eisenberg. Experience report: Type-checking polymorphic units for astrophysics research in haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 31–38, New York, NY, USA, 2014. ACM.

[16] S.-B. Scholz. Single Assignment C: Efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, Nov. 2003.