Type Inference for Linear Algebra with Units of Measurement

P.R. Griffioen

March 5, 2013

Abstract

Refining types of numerical data with units of measurement has the potential to increase safety of programming languages but is restricted to homogeneous units when checked statically with parametric polymorphism. We lift units to vectors and show how type inference of linear algebra expressions can statically determine safety for data with heterogeneous units. The typing is based on the dyadic product of units that is found in linear transformations and the corresponding vector spaces. Since it is a refinement of Kennedy's types for units we automatically obtain a unification algorithm, which gives principal types for linear algebra. The extension of unit-unification to numerical data with heterogeneous units makes the safety of statically checked numerical expressions available to a significantly larger set of use-cases.

Contents

Chapter 1

Introduction

Correct computation with units of measurement can be statically guaranteed with parametric polymorphism, as was demonstrated by Andrew Kennedy in [Ken94, Ken96]. A polymorphic type like `[a]` denotes the real numbers with unit `a`. Examples of typed numbers are the following declarations of quantities `mass` and `acceleration`

```
mass :: [kg];
acceleration :: [m/s^2];
```

Inhabitants of type `[kg]` are real number with unit kilogram. This unit is one of the seven base units of the International Standard of Units [dPeM]. Other units can be derived from the base units by algebraic expressions. Type `[m/s^2]` is for example derived from base units `m` (metre) and `s` (second).

Amazingly, unit types have a most general unifier. The type system infers a principal type for any numerical expression if and only if the expression is unit correct. To start, types are assigned to primitives like the usual sum and product functions:

```
+ :: forall a: [a] x [a] -> [a]
* :: forall a,b: [a] x [b] -> [a*b]
```

The first rule expresses that only numbers with the same unit can be summed. The second rule states that the unit of a product is the product of the arguments' units. The rules enable the correct type to be inferred, for example for the definition of `force` as the product of `mass` and `acceleration`. User input is indicated by prompt `>` and the system responds with the inferred type:

```
> define force = mass * acceleration
force :: [kg*m/s^2];
```

Standard Damas-Milner type inference [DM82, Pie02] is syntactical and does not allow such a semantic expression in types. This ability to semantically infer the proper units provides extremely powerful support for numerical expressions.

However, a shortcoming of polymorphic types is that they cannot express types for data with heterogeneous units. Consider the recipe for apple pie from Figure 1.1. It is a miniature example of the "explosion" and "netting" problem in requirements planning from operations research [Elm63]. Such a

Product	Product	BoM
butter	pastry	360.00 g/kg
flour	pastry	550.00 g/kg
pastry	apple_pie	0.40 kg
apples	apple_pie	700.00 g
sugar	apple_pie	225.00 g
butter	apple_pie	115.00 g
apple_pie	piece_of_pie	0.12

Figure 1.1: The bill of material from a recipe for apple pie. Each entry is the amount of a product going directly into one unit of a containing product. The heterogeneous units are problematic for a parametric polymorphic type, which is homogeneous by nature.

bill of material is a numerical relation between products that is conveniently represented by a matrix. The units in the entries are an essential part of a bill of material but problematic for polymorphic types. It is impossible to type all matrix entries statically when the data is dynamic and unknown at compile time, and any polymorphic matrix type `Matrix([a])` is homogeneous and therefore insufficient to express the units.

Using named vectors of units we exploit the mathematical structure of units in a matrix to obtain a matrix type that statically guarantees unit-correct matrix operations. We extend Milner's polymorphic type with a dedicated syntax for matrices with units. In this scheme the bill of material could be typed as follows.

```
BoM :: [Product!bom_unit per Product!bom_unit]
```

The brackets indicate the matrix type. The expression before the 'per' specifies the matrix' row dimension and the part after the 'per' the column dimension (see Figure 1.3 for some examples). The `bom_unit` part in the term `Product!bom_unit` names a vector of units that is indexed by elements from index set `Product`. The exclamation mark is a token that separates the index set name and the name of the unit vector and thus provides a simple naming scheme. In this case the vector of units is column `bom_unit` from Figure 1.2. In this column product `butter` for example has unit `gram` and product `pastry` has unit `kg`, and therefore the first entry in Figure 1.1 has unit `g/kg`. The advantage of this scheme is that using names like `bom_unit` we can reason about the units in a matrix without knowing the individual units in each entry. The extra indirection from the vector names solves the problem with heterogeneous units, and because units in matrices are always a dyadic (outer) product of two vectors [Har, Har95], the type is sufficient to type any matrix.

Thanks to the semantic nature of units the matrix type allows typing of matrix operations. The most general matrix type in our syntax is

```
forall a,P,u,Q,v: [a * P!u per Q!v]
```

For any matrix `A` of this type it means that matrix entry $\mathtt{A}_{ij}$ is a number with units $\mathtt{a}\cdot(\mathtt{P!u})_i/(\mathtt{Q!v})_j$. The singleton factor `a` is not only convenient to factor out common units, but also allows typing of singleton numbers with a 1×1 matrix. Variables `P` and `Q` refer to index sets and can be substituted with identifiers

Product	trade_unit	bom_unit	pur_price	sal_price	sales
butter	pound	gram	2.00 $/lb	-	-
flour	kilogram	gram	2.50 $/kg	-	-
apples	kilogram	gram	0.90 $/kg	5.00 $/kg	2.00 kg
sugar	kilogram	gram	0.80 $/kg	-	-
pastry	kilogram	kilogram	-	-	-
apple_pie	-	-	-	20.00 $	25.00
piece_of_pie	-	-	-	2.50 $	100.00

Figure 1.2: A table for index set `Product` containing unit columns `trade_unit` and `bom_unit` and various columns with numerical quantities. Column `sales` is the sales amount and has units `trade_unit`. Columns `pur_price` and `sal_price` have units `dollar` per `trade_unit`. Unit column `bom_unit` is used in the bill of material from Figure 1.1.

like `Product` above. Variables `u` and `v` refer to vectors of units for `P` and `Q` as explained above. They are semantic, as is `a`, meaning unit expressions are allowed. For example when all entries are squared in the above type the result will be

```
forall a,P,u,Q,v: [a^2 * P!u^2 per Q!v^2]
```

This shows that operations on the units in the matrix type are allowed, just as Kennedy's polymorphic number type allows operations at the type level. We present types for the basic linear algebra operators with this syntax.

The key to the expressiveness of the matrix type is that an index set unit combination like `Product!bom_unit` more or less names a vector space. Practically, the index sets map nicely on database tables or other relational settings, and the units provide much added value at run-time, but the most powerful feature is the static safety provided by type inference. To illustrate this point consider how the type inference algorithm computes a type for the commutator from Lie algebras. The dot in the definition denotes matrix multiplication.

```
> define commutator(x,y) = x.y - y.x

commutator :: forall u2, u1, E0, u0:
                [u1 * E0!u0 per E0!u0] x
                  [u2 * E0!u0 per E0!u0]
                    -> [u1*u2 * E0!u0 per E0!u0]
```

In all three matrices the index sets are `E0` in the row and column direction and the units are `E0!u0` in the row and column direction. This shows we must be working with square matrices. The example also illustrates dimensional parallelism, which will be defined in Chapter 4. We expect the operator to be closed with respect to the vector spaces, but the inferred type is more general than intended. Not only should the matrices be square, but also should `u1 = u2 = 1` hold. The inferred type is more general and correctly also includes the dimensionally parallel cases. The example shows that the type provides much interesting feedback without actual concrete index sets and units. In our experience, more than signaling errors the type system turns out to be extremely

`[a]`	scalar
`[I!]`	dimensionless column vector
`[I!u]`	dimensioned column vector
`[a·I!]`	homogeneous dimensioned column vector
`[1 per J!v]`	dimensioned row vector
`[I! per J!]`	dimensionless rectangular matrix
`[I!u per I!v]`	square matrix
`[I!u per I!u]`	even more square matrix

Figure 1.3: Examples of matrix types. The brackets indicate the matrix type. The expression before the 'per' specifies the matrix' row dimension and the part after the 'per' the column dimension. The exclamation mark is a token that separates an index set name and the name of and unit vector. Scalars are 1×1 matrices and vectors are $n\times1$ or $1\times n$ matrices.

helpful in providing insight into transformations between vector spaces and other vector manipulations.

We generalize the type to the multi-linear case via matricized tensors. Unification for the matrix type is a direct decomposition to unification of its parts. The two index set names can be directly unified with standard unification, whereas the three unit parts can be unified with Kennedy's unit unification. With tensors there are not always exactly five parts because the number of indices varies. This difficulty is solved by matricization of tensors. Multiple indices are replaced by two compound matrix indices, which makes matrix unification applicable again.

The research results documented in this report are i) an extension of Milner's parametric polymorphic type system with a matrix type based on units of measurement that has principal types, and ii) a generalization of the matrix type to tensors for the multi-linear case, and iii) types for basic primitive linear algebra operators, and iv) an experimental language design as a proof of concept. The small experimental matrix language is presented in chapter 2 to further introduce the concepts of the matrix type. In the form of a user interaction the bill of material case is explored. After this extended introduction the work of Kennedy and other related literature is discussed in chapter 3. This chapter gives background information on units of measurement and its use in a type system. The underlying mathematics is very basic and briefly explained. chapter 4 reviews how units are lifted to vectors and how this refines vector spaces. We discuss all work from [Har, Har95] that is relevant for out matrix type and explain the generalization to the multi-linear case. In chapter 5 the reviewed material is combined into a polymorphic type system based on Milner and Damas. We show that the statically typed operators for vectors and matrices supports principal types. We conclude with some example applications and a conclusion.

Chapter 2

An Experimental Unit-based Matrix Language

As an experiment we designed and implemented a matrix language with the matrix type described in the introduction. The language was applied to various use-cases to test the feasibility of language support for numbers with heterogeneous units. Specific questions from the language perspective concerned the capabilities of the type system. Could the linear algebra operators be given useful types? And could the language handle numerical data with heterogeneous units like the bill of material? Is the type useful for library functions where the type typically has type variables?

The language consists of some syntactic sugar on top of the lambda calculus and has linear algebra operators as primitive functions. Matrices are the only values and types of expressions are inferred.

The most distinguishing features of the language are the indexing of matrices by any index set rather than by the customary natural numbers and the way this is incorporated into the matrix type. This choice is not only made to have a practical naming scheme but also to prevent errors with matrix sizes. The role of index set names like `P` and `Q` is the same as the roles of the numbers m and n in the $\mathbb{R}^{m\times n}$ notation from mathematics. This notation is a form of typing and prevents errors with matrix sizes. Matrix multiplication is for example typed by $\mathbb{R}^{m\times k} \times \mathbb{R}^{k\times n} \to \mathbb{R}^{m\times n}$. With our matrix type we assume two matrices have the same size if and only if they have the same index set name. The matrix type further extends this with units and prevents invalid operations on them, but errors with sizes are prevented by the index set names and they are therefore crucial for the matrix type.

The matrix type allows the expression of size constraints and units constraints. A square matrix with different units in the row and column dimension is written as `[P!u per P!v]` and a square matrix with equal units as `[P!u per P!u]`. A dimension-less matrix is `[P per Q]`. Special index set name `1` indicates the absence of a row or column dimension and is always dimensionless. A column vector is written as `[P!u per 1]`, a row vector as `[1 per P!u]` and a singleton matrix as `[1 per 1]`. This shows how index set names allow

```
dot :: forall a,I,u,K,v,b,J,w:
  [a·I!u per K!v] x [b·K!v per J!w] -> [a·b·I!u per J!w]
sum :: forall a,I,u,J,v:
  [a·I!u per J!v] x [a·I!u per J!v] -> [a·I!u per J!v]
mult :: forall a,I,u,J,v,b,w,z:
  [a·I!u per J!v] x [b·I!w per J!z] -> [a·b·I!u·I!w per J!z·J!v]

scale :: forall a,b,I,u,J,v: [a] x [b·I!u per J!v] -> [a·b·I!u per J!v]
power :: forall I,u: [I!u per I!u] x [1] -> [I!u per I!u]
expt  :: forall I,J: [I! per J!] x [1] -> [I! per J!]

negative   :: forall a,I,u,J,v: [a·I!u per J!v] -> [a·I!u per J!v]
reciprocal :: forall a,I,u,J,v: [a·I!u per J!v] -> [1/I!u per a/J!v]
transpose  :: forall a,I,u,J,v: [a·I!u per J!v] -> [a/J!v per 1/I!u]
dim_inv    :: forall a,I,u,J,v: [a·I!u per J!v] -> [J!v per a·I!u]

left_ident  :: forall a,P,u,Q,v: [a·P!u per Q!v] -> [P!u per P!u]
right_ident :: forall a,P,u,Q,v: [a·P!u per Q!v] -> [Q!v per Q!v]

solve :: forall a,b,P,u,Q,v,R,w:
  [a·P!u per Q!v] x [b·P!u per R!w] -> [b·Q!v per a·R!w]
```

Figure 2.1: Types for basic primitive linear algebra functions. The first three are used for the implementation of the binary operators '`.`', '`+`', '`*`'. Function `mult` is elementwise matrix multiplication. Different use cases would require different additions to this list of primitives. Functions `left_ident`, `right_ident` and `solve` are used in the bill of material case from section 2.1. The details of the matrix type syntax is given in section 5.2

expression of constraints in the row and column dimensions. See Figure 1.3 for some examples.

Additionally the matrix type has a scalar unit factor for singletons and homogeneous matrices. The basic matrix type from the previous paragraph only allows unit-less singletons. To make it an extension of single numbers it requires a scalar unit factor. Say `a` is a single unit then the general form of the matrix type is `[a * P!u per Q!v]`. The scalar factor also allows homogeneous vectors and matrices by multiplying a unit-less basic matrix type with a scalar unit. For example a homogeneous column vector with elements from index set P each having unit `a` is `[a * P per 1]`. As a special case we write `[a]` for singleton `[a * 1 per 1]`.

An overview of primitive functions is shown in Figure 2.1. A justification for these types is given in section 4.4 when units in matrices are discussed. The arithmetic operators `+`, `-`, `*` and `/` desugar into element-wise `sum`, `multiply`, `negative` and `reciprocal`. Function `product` is the matrix product and written as infix dot. Function `scale` scales a matrix by a number. Function `transpose` is written as postfix operator `^T` and function `reciprocal` is written as postfix operator `^R`. Function `solve` computes a solution to $\mathbf{A} \cdot \mathbf{x} = \mathbf{B}$. Its type follows from `product`'s type. The identity functions gives the left and right identity of a given matrix. Their types also follow from `product`'s type.

Examples are shown in the form of a user interaction. Input is indicated by prompt `>`. For definitions the system responds with a type assignment using the `::` syntax. For expressions the response is a value. Type declarations for used quantities and functions will be given in the text when needed.

2.1 Bill of Material Case

The bill of material case was chosen because it is a challenging real-world problem involving units, with a well-known linear algebra solution. It highlights some specific issues from linear algebra that we hoped to address.

1. The explosion of the bill of material requires a matrix inverse. Will the correct types be inferred without annotations?

2. Volumes and prices are dual notions. Can this distinction be expressed and will units be handled correctly?

3. The data in Figure 1.2 is in `trade_unit`s while the bill of material is in `bom_unit`s. Can units errors be prevented?

The type system handled these and other issues satisfactory. The remainder of this chapter explores the bill of material case and demonstrates important language features.

We assume that the following types are declared for the quantities from Figure 1.2.

```
BoM :: [Product!bom_unit per Product!bom_unit]
sales :: [Product!trade_unit]
pur_price :: [usd per Product!trade_unit]
sal_price :: [usd per Product!trade_unit]
```

For example, the unit of `pur_price` is read as the amount of dollar per product trade unit. It is easily verified that the numerical data in the table satisfies these types.

Product prices and volume are dual notions that can be multiplied to give total sales. Vector `sal_price` is a row vector with `Product!trade_unit` in the column direction. Vector `sales` is not a row vector, but a column vector with `Product!trade_unit` in the row direction. The fact that prices are row vectors and that volume is a column vectors shows that prices and volumes are dual notions. Their types allow them to be multiplied.

```
> define revenue = sal_price . sales
revenue :: [dollar]

> revenue
760 $
```

If the details are required instead of the total then an element-wise product can be used but it requires some care with the dimensions. An element-wise product between the prices and the sales yields the sales per product, but because of the duality this requires a transpose on the volume. The type system will complain if the transpose is forgotten.

```
> define revenue_details = sal_price * sales
Error while unifying matrices
   [u107 per Product!u103]
and
   [Product!trade_unit]
index sets do not match: 1 and Product
```

The error message correctly complains that an attempt is made to multiply a row vector with a column vector. With the transpose we get the expected result.

```
> define revenue_details = sal_price * sales^T
revenue_details :: [dollar per Product]

> revenue_details

 Product       |  Value
---------------+------------
 Apples        |   10.00 $
 Apple Pie     |  500.00 $
 Piece of Pie  |  250.00 $
```

This time expression gives the correct output. A sketch of its type derivation in abstract form is

$$\dfrac{\texttt{v: [a per P!u]} \qquad \dfrac{\texttt{w: [P!u]}}{\texttt{w\^{}T: [1 per P!u\^{}-1]}}}{\texttt{v * w\^{}T: [a per P]}}$$

To compute the expenses we have to overcome the difference of units in the bill of material and the trade units in the prices. We assume that conversion matrix `conv` exists with the following type declaration.

```
conv :: Product!trade_unit per Product!bom_unit
```

In this case the conversion is well defined but this needs not always be the case. The existence of the conversion between bom units and trade units depends on run-time data and will fail if columns `bom_unit` and `trade_unit` from Figure 1.2 contain incompatible units. In general the existence of conversion can never be guaranteed if it depends on run-time data. For that reason we didn't add a construction for conversion to the language expressions but instead introduced a special declaration. Conversions are the only language construction that may fail at run-time.

Converting the bill of material requires a multiplication from the left and from the right. It is convenient to create a function to do that.

```
> define convert(x) = conv . x . conv^R^T
convert :: forall u:
 [u * Product!bom_unit per Product!bom_unit]
  -> [u * Product!trade_unit per Product!trade_unit]
```

The combination `^R^T` of the reciprocal and the transpose is called the dimensional inverse. It has a prominent role in dimensioned matrices as we shall see in section 4.3. The function's type shows the correct inferred type. Unit variable `u` indicates that any type scaled by a single unit is also an acceptable argument. This correctly shows that function convert is slightly more general than a conversion from `bom_unit` to `trade_unit`.

With the converted matrix the bill of material can be "exploded". An elegant solution is the use of Leontief matrices $\mathbf{A} + \mathbf{A^2} + \mathbf{A^3} + \ldots$ [Elm63, Str88]. This series computes what could be called a numerical transitive closure of the numerical relation in the bill of material. The series can be computed with the inverse of $(\mathbf{I} - \mathbf{A})$ so we need the inverse function in our language. Function `inverse` can be defined using the `solve` and `left_ident` primitives.

```
> define inverse(x) = solve(x, left_ident(x))
inverse :: forall a,P,u,Q,v:
  [a * P!u per Q!v] -> [a^-1 * Q!v per P!u]

> define leontief(x) =
    let I = left_ident(x) in
      inverse(I - x) - I
leontief :: forall P,u:
  [P!u per P!u] -> [P!u per P!u]
```

Function `leontief` computes the needed series. The series are only defined for square matrices and the system infers that correctly. Together with the `convert` function it allows the netting problem to be solved.

```
> eBoM = leontief(convert(BoM))
eBoM :: [Product!trade_unit per Product!trade_unit]

> eBoM

 Product       | Product       |   Value
---------------+---------------+------------
```

```
Sugar         | Piece of pie  |  0.02700 kg
Sugar         | Apple pie     |  0.22500 kg
Apple pie     | Piece of pie  |  0.12000
Apples        | Piece of pie  |  0.00060 kg
Apples        | Apple pie     |  0.00500 kg
Pastry        | Piece of pie  |  0.04800 kg
Pastry        | Apple pie     |  0.40000 kg
Flour         | Piece of pie  |  0.02640 kg
Flour         | Apple pie     |  0.22000 kg
Flour         | Pastry        |  0.55000
Butter        | Piece of pie  |  0.06852 lb
Butter        | Apple pie     |  0.57100 lb
Butter        | Pastry        |  0.79366 lb/kg
```

The result is the complete transitive part-of relation in trade units.

Multiplying the exploded BoM with the sales volume gives the input volume of the ingredients. Both are in `trade_units` and can safely be multiplied. From the types can also be seen that the sales volume has to be multiplied from the right.

```
> purchases = eBoM . sales
purchases :: [Product!trade_unit]
```

Dually the exploded bom can be multiplied from the left to aggregate the purchase prices. This again shows that volumes and prices are dual notions.

```
> cost = pur_price . eBoM
cost :: [dollar per Product!trade_unit]
```

Multiplying prices from the right would give a type error.

Finally we can compute the complete expenses on ingredients in two different ways. Using the input and the purchase prices or the sales and the product cost.

```
> pur_price . purchases
243.21 $

> cost . sales
243.21 $
```

Chapter 3

Units and Types

Units of measurement have a long history in physics that leads back via Maxwel to Fourier and even further. Birkhof is a classic reference for a mathematical treatment of units [Bir60]. According to him Fourier was the first to note that there are certain fundamental units of which every physical quantity has certain "dimensions", to be written as exponents [Fou22].

An international standard of units has been defined, called Le Système international d'unités [dPeM], and is maintained by Le Bureau International des Poids et Mesures. Internationally this is known as The International System of Units, or SI. The document describes all aspects of the SI. For precise definitions the text refers to the International vocabulary of metrology [BIP]. We follow the international standard of units (SI) for the core concepts regarding systems of units and explain them briefly in the next section.

Andrew Kennedy created parametric polymorphic types with units and demonstrated how unit unification computes principal types. His PhD thesis is the most rigorous work on units in programming languages [Ken96] and recently his ideas were added to the F# programming language. A good overview of his work and a nice hands-on introduction to units in F# is in [Ken09]. We'll summarize his main results and describe his unification algorithm.

Extending language syntax or typing rules with units of measurement is a common approach to unit safety for software. Recent related work is the MetaGen extension of Java that statically checks units of measure [ACL04]. Dimensions and units can be formulated in a nominally typed object-oriented language through the use of statically typed meta-classes. This enables both parametric and inheritance polymorphism for dimensions and units. MetaGen is an extension of Java's MixGen with a meta-class that provides the algebraic properties necessary to model dimensions and units accurately.

Another recent development is the Osprey tool [SJ06]. This tool extends the C programming language with type annotations that can be statically checked for errors in units of measurement. The standard type checking algorithms are extended with techniques like constraint solving and Gaussian Elimination to provide the required algebraic properties. The prototype was extensively validated on mature code bases of significant size and discovered many errors.

As far as we know there is no language that support type inference for matrices or similar structures. MetaGen and Osprey do not provide explicit support for data with heterogeneous units of measurement, or other support for

matrices or linear algebra in general.

3.1 Preliminaries

A group is a triple $(S, *, e)$ with S a set, $*$ an associative binary operator on S, and e an identity element such that $e * a = a * e = a$ and for each $a \in S$ some $a' \in S$ exists satisfying $a * a' = a' * a = e$. This inverse a' is unique so it is a well-defined operation. A group is abelian or commutative when the operator commutes.

A subset of the elements of an abelian group forms a basis if any element from the group can be uniquely written as a linear combination of the basis' elements. An abelian group with a basis is called free.

A field $(S, +, 0, \cdot, 1)$ is the combination of an abelian group $(S, +, 0)$ and an abelian group $(S, \cdot, 1)$ satisfying $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$. Convention is to write the additive inverse as $-a$ and the multiplicative inverse as a^{-1}.

3.2 Units of Measurement

A *quantity* is a property of a phenomenon, body or substance, where the property has a unique magnitude that can be expressed as a number and a reference. The value of a quantity is the product of a number and a *unit*. The unit is a particular example of the quantity concerned. A *prefix* is a scaling factor of a unit. A unit has at most one prefix. The prefix together with the unit forms a new unit with its own identity.

Quantities are organized in a *system of dimensions*. As a matter of convenience some quantities are called base quantities and all other quantities are derived from them. Typical base quantities are length, mass, time, etc. Each base quantity is its own *dimension*, and each derived quantity's dimension follows from the derivation [dPeM, BIP].

Kennedy [Ken96] provides syntax to derive new dimensions from base dimensions and new units from base and units. Both cases are very similar so only units are discussed. Units are derived from given base units like metre, gram, second, etc. Let's write μ for units. The syntax is

μ	$::= u$	unit variables
	$\mid \mathcal{U}$	base units
	$\mid \mathbb{R}^+$	positive numbers
	$\mid \mu\mu$	product
	$\mid \mu^{-1}$	inverse

A *system of units* is a set of base units and derived units together with their prefixes, defined in accordance with given rules, for a given system of quantities.

Mathematically the units of measurement are a free abelian group. We will use symbol $\mathbb{U}$ to denote this group.

$$\text{UnifyUnits}(\mu_0, \mu_1) = \text{UnitUnify}(\mu_0 \cdot \mu_1^{-1})$$

$$\begin{array}{l}
\text{UnitUnify}(\mu) = \\
\quad \text{let nf}(\mu) = u_1^{x_1} \cdots u_m^{x_m} \cdot \mathcal{U}_1^{y_1} \cdots \mathcal{U}_n^{y_n} \\
\quad \text{where } |x_1| \leq \cdots \leq |x_m| \\
\quad \text{in} \\
\qquad \text{if } m = 0 \text{ and } n \neq 0 \text{ then fail} \\
\qquad \text{else if } m = 1 \text{ and } x_1 | y_i \text{ for all } i \text{ then} \\
\qquad\quad \{u_1 \mapsto \mathcal{U}_1^{-y_1/x_1} \cdots \mathcal{U}_n^{-y_n/x_1}\} \\
\qquad \text{else if } m = 1 \text{ otherwise then fail} \\
\qquad \text{else } S \circ U \text{ where} \\
\qquad\quad U = \{u_1 \mapsto u_1 \cdot u_2^{-\lfloor x_2/x_1 \rfloor} \cdots u_m^{-\lfloor x_m/x_1 \rfloor} \cdot \\
\qquad\qquad\qquad\qquad \cdot \mathcal{U}_1^{-\lfloor y_1/x_1 \rfloor} \cdots \mathcal{U}_n^{-\lfloor y_n/x_1 \rfloor}\} \\
\qquad\quad S = \text{UnitUnify}(U(\mu))
\end{array}$$

Figure 3.1: Andrew Kennedy's unification algorithm for units. It 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.

3.3 Types and Unification

Let $\mathbb{R}$ be the field of real numbers and let $\mathbb{U}$ be the abelian group of units. Maxwell's idea to treat a quantity as a product of a number [Roc98] and a unit means a quantity is from product space $\mathbb{R} \times \mathbb{U}$. Such a combination of a number and a unit is called a *dimensioned scalar*. A pairs like (r, u) is used to denote a dimensioned scalar with magnitude $r \in \mathbb{R}$ and unit $u \in \mathbb{U}$. When $u = 1$ we call the quantity dimension-less. Lifting the operators from $\mathbb{R}$ and $\mathbb{U}$ defines the common arithmetic operators on this combined space $\mathbb{R} \times \mathbb{U}$. Although addition is absent for units, equal units can be added via the addition of $\mathbb{R}$. Kennedy gave polymorphic types for the common arithmetic operators. The $\forall u$ syntax introduces quantified units variable u. This syntax is also used in this paper and defined later on.

$+ : \forall u.u \times u \to u$	addition
$* : \forall u.\forall v.u \times v \to uv$	multiplication
$- : \forall u.u \to u$	negation
$(\_)^{-1} : \forall u.u \to u^{-1}$	reciprocal

These type rules accurately capture compatible operations with units.

Units have most general unifier with a unification algorithm because it is a free abelian group. Algebraically a free abelian group is a $\mathbb{Z}$-module, which means it is isomorphic with the functions to $\mathbb{Z}$. For the free multiplicative abelian group of units this means that any units can be written as a power

product $\Pi i : \mathcal{U}_i^{x_i}$ where $\mathcal{U}_i$ is one of the base units and $x_i \in \mathbb{Z}$ is its power. The powers x_i represent a map $\mathcal{U} \to \mathbb{Z}$ that completely describes the unit. The property that any element of a free abelian group has a unique representation in the base elements is a crucial condition for unit unification.

Unification is based on normal form $\text{nf}(\mu)$, which is a power product that also takes unit variables into account. Let μ be a unit with unit variables u_i and base units $\mathcal{U}_i$. The normal form $\text{nf}(\mu)$ is the power product $u_1^{x_1} \cdots u_m^{x_m} \cdot \mathcal{U}_1^{y_1} \cdots \mathcal{U}_n^{y_n}$ with $x_i, y_i \in \mathbb{Z}$ and $|x_1| \leq \cdots \leq |x_m|$. Kennedy's unification algorithm uses the normal in the computation of the most general unifier for two units. The algorithm, shown in Figure 3.1, is based on Knuth's adaptation of Euclid's greatest common divisor algorithm. This sound and complete semantic unification algorithm computes the most general unifier given two units, or it fails if no such unifier exists.

The development of the matrix type depends on the property that the base units in $\mathcal{U}$ can have additional structure as long as the group remains free. Base elements can be compound structures, for example the combination of a prefix and a base-unit, but crucial is that variables only identify such a compound, not an operator or any substructure in the compound. For example a kilogram is an entity on its own and for the unification algorithm it doesn't matter that it is a pair, as long as equality is defined. For the matrix type the set of base units $\mathcal{U}$ will be extended with the names of dimension vectors and with tuples $\langle x_0, \ldots, x_n \rangle$ with each x_i the name of a dimension vector. These changes have no effect on the unification algorithm. There is a still a unique representation in the base elements, the group remains free and the unification algorithm can be applied.

Chapter 4

Units in Matrices

This chapter describes the structure units of measurement in matrices. It involves a generalization from dimensioned scalars from $\mathbb{R} \times \mathbb{U}$ to dimensioned matrices from $(\mathbb{R} \times \mathbb{U})^{m \times n}$. A dimensioned vector coincides with a $(\mathbb{R} \times \mathbb{U})^{n \times 1}$ matrix and a dimensioned co-vector with $(\mathbb{R} \times \mathbb{U})^{1 \times n}$. The principle that only numbers with the same unit can be summed affects most matrix and vector operations since it directly restricts linear combinations and the dot product.

All results are from [Har, Har95]. Just as in the discussion of unit unification we summarize the material from the unit perspective instead of the dimension perspective used by Hart. However, we follow Harts' naming conventions and treat dimension as synonymous for unit.

4.1 Preliminaries

Vectors are written in boldface, matrices are written in bold uppercase and scalars in regular font.

A vector space over a field of scalars $(F, +, 0, \cdot, 1)$ is an abelian group $(V, +, 0)$ extended with scalar multiplication $\cdot$ satisfying for all scalars $a, b \in F$ and for all vectors $\mathbf{v}, \mathbf{w} \in V$

$a \cdot (\mathbf{v} + \mathbf{w}) = a \cdot \mathbf{v} + a \cdot \mathbf{w}$	distributivity over vector addition
$(a + b) \cdot \mathbf{v} = a \cdot \mathbf{v} + b \cdot \mathbf{v}$	distributivity over field addition
$a \cdot (b \cdot \mathbf{v}) = (a \cdot b) \cdot \mathbf{v}$	compatibility of multiplication
$1 \cdot \mathbf{v} = \mathbf{v}$	identity of multiplication

Further we use the following products.

$\langle \mathbf{v}, \mathbf{w} \rangle = \mathbf{v}^T \cdot \mathbf{w}$	inner product
$\mathbf{v} \circ \mathbf{w} = \mathbf{v} \cdot \mathbf{w}^T$	outer product
$(\mathbf{A} \otimes \mathbf{B})_{(ip+k)(jq+l)} = \mathbf{A}_{ij} \cdot \mathbf{B}_{kl}$	Kronecker product

The element-wise (or Hadamard) product is written as $\mathbf{A} * \mathbf{B}$ and the element-wise reciprocal (not the inverse!) as $\mathbf{A}^{-1}$.

Multi-linear algebra generalizes a matrix to a tensor. A matrix has exactly one co-variant and one contra-variant index, whereas a tensor can have any number of co-variant and contra-variant indices. Convention is to write co-variant

indices in an super-script position and contra-variant in sub-script position like $\mathcal{X}^{i_1 \cdots i_k}_{j_1 \cdots j_l}$. A tensor with one upper and one lower index like $\mathcal{X}^j_i$ is a matrix, $\mathcal{X}^i$ is a column vector and $\mathcal{X}_j$ a row vector.

4.2 Matrices with Units

A *dimensioned vector* is an n-tuple in which each entry is a dimensioned scalar. Any dimensioned vector belongs to a *complete dimensioned vector space* defined as the set of all vectors with the same units in the corresponding components. A subspace of a vector space is a subset that is closed under addition and under scalar product by a dimension-less scalar. A *dimensioned vector space* is a subspace of a complete dimensioned vector space.

For our purpose it is convenient to separate the magnitudes from the units and switch from space $(\mathbb{R} \times \mathbb{U})^{m \times n}$ to the isomorphic space $\mathbb{R}^{m \times n} \times \mathbb{U}^{m \times n}$. Let $(\mathbf{R}, \mathbf{U})$ be a dimensioned matrix with magnitudes $\mathbf{R}$ and units $\mathbf{U}$. The units are called the *dimensional form* of the matrix and can be extracted with function dim.

Definition 1. For any dimensioned matrix $\mathbf{A} = (\mathbf{R}, \mathbf{U})$

$$\dim(\mathbf{A}) = \mathbf{U} \qquad \text{dimensional form}$$

Using the dimensional form Hart defines two natural equivalence relations on dimensional matrices. 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.

Definition 2. For all matrices $\mathbf{A}$ and $\mathbf{B}$

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

Definition 3. For all matrices $\mathbf{A}$ and $\mathbf{B}$

$$\mathbf{A} \approx \mathbf{B} \;\Leftrightarrow\; \exists c : \mathbf{A} \sim c\mathbf{B} \qquad \text{dimensional parallelism}$$

The equivalence relations are used to define the vector space to which a given vector belongs.

Definition 4. 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 $\mathbf{v} \sim \mathbf{w}$.

Definition 5. 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.

Such $\mathbf{v}$-spaces correspond to the index set unit combinations from the matrix type. An index set unit combination `P!u` in the matrix type names a complete dimensioned vector space. We will call this the `P!u`-space

4.3 Dimensional Inverse and the Structure of Units

When a matrix is regarded simply as a rectangular data structure it can contain any units, but when it is restricted to valid linear transformations then the units are restricted as well. The intuition behind the matrix type `[P!u per P!v]` is that the matrix transforms vectors from the `P!v`-space to vectors from the `P!u`-space. Before this can be made precise we need the concept of the dimensional inverse. This concept is central to the theory and explains how to read the `per` in the matrix type.

Definition 6. The *dimensional inverse* $\mathbf{A}^\sim$ of a matrix is defined by

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

The intuition is that it combines a reciprocal and a transpose. We can also write the dimensional inverse as $(\mathbf{A}^{-1})^T$ or $(\mathbf{A}^T)^{-1}$.

Hart's main results about the units in valid linear transformations use the dimensional inverse. A matrix is called *multipliable* when the structure of its units allow unit-correct linear transformations. The main results are:

Theorem 1. For any matrix $\mathbf{A}$

$$\mathbf{A} \text{ is multipliable } \Leftrightarrow \exists \mathbf{v}, \mathbf{w} : \mathbf{A} \sim \mathbf{v} \cdot \mathbf{w}^\sim$$

Theorem 2. Any matrix of a linear transformation from a $\mathbf{v}$-space to a $\mathbf{w}$-space has the form $\mathbf{w} \cdot \mathbf{v}^\sim$

The form of the units in the last theorem is used in the matrix type. The form $\mathbf{v} \cdot \mathbf{w}^\sim$ is written as `v per w`. Reformulated with this `per` notation the theorem states that any linear transformation from a `Q!v`-space to a `P!u`-space has the form `[P!u per Q!v]`.

4.4 Types for Linear Algebra

With the theory from the previous sections the types from Figure 2.1 can be justified.

The element-wise `sum` simply require that the units of its arguments and the units of its result are the all same. The same applies to function `negative`. Functions `multiply` and `reciprocal` are base on the following property:

Property 1. For all dimension vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ and $\mathbf{d}$:

$$(\mathbf{a} \cdot \mathbf{b}^\sim) * (\mathbf{c} \cdot \mathbf{d}^\sim) = (\mathbf{a} * \mathbf{c}) \cdot (\mathbf{b} * \mathbf{d})^\sim$$

The type for the matrix product follows from the following property

Property 2. For all matrices $\mathbf{A} \sim \mathbf{a} \cdot \mathbf{b}^\sim$ and $\mathbf{B} \sim \mathbf{c} \cdot \mathbf{d}^\sim$:

1. $\mathbf{A} \cdot \mathbf{B}$ is defined iff $\mathbf{b} \approx \mathbf{c}$

2. $\mathbf{A} \cdot \mathbf{B} \sim \mathbf{a} \cdot \mathbf{d}^\sim$ if it is defined

If matrices are viewed as linear transformation then this rule states that a transformation from a **d**-space to a **c**-space followed by a transformation from a **b**-space to an **a**-space requires that the intermediate space must agree and that the result is a transformation from the **d**-space to the **a**-space.

The rule for the identity matrices follow directly from the previous property since identity matrices are trivial linear transformations. The type for `solve` also follows from the property. Function `solve` computes the solution to $\mathbf{A} \cdot \mathbf{x} = \mathbf{B}$. The stated type follows from the rule for matrix products.

4.5 Multi-linear algebra

The type system works with matricized tensors for multi-dimensional cases. The difficulty with tensors is handling its variadic indices. To solve this issue a tensor is treated as a matrix with compound indices and projection matrices project the contents for the various indices. The Kronecker product combines indices and plays a central role in the matricized treatment of multiple dimensions.

The units in a tensor require a generalization of the binary dimensional inverse to the n-ary case. In the previous section it was explained that the units in a multipliable matrix are of the form $\mathbf{v} \cdot \mathbf{w}^\sim$. The matrix product does not generalize well to tensors so we use identities $\mathbf{v} \cdot \mathbf{w}^\sim = \mathbf{v} \cdot (\mathbf{w}^{-1})^T = \mathbf{v} \circ \mathbf{w}^{-1}$ to change to the dyadic product. This dyadic product is a matrix of rank 1 and higher rank matrices are sums of such dyadic products [Rom08]. Similarly a tensor is a sum of rank 1 tensors. A matrix has units like $\mathbf{v} \circ \mathbf{w}^{-1}$ while for example a tensor with two co-variant and three contra-variant indices has units like $\mathbf{v} \circ \mathbf{w} \circ \mathbf{x}^{-1} \circ \mathbf{y}^{-1} \circ \mathbf{z}^{-1}$. Any tensor can always be organized so that the reciprocals are grouped at the end. Thus the units in a tensor are a product a any number of co-variant vectors "divided" by any number of contra-variant vectors.

Using the Kronecker product the example tensor can be matricized into $(\mathbf{v} \otimes \mathbf{w}) \cdot (\mathbf{x} \otimes \mathbf{y} \otimes \mathbf{z})^\sim$. Here $\mathbf{v} \otimes \mathbf{w}$ and $\mathbf{x} \otimes \mathbf{y} \otimes \mathbf{z}$ are dimension vectors and the entire result is a matrix. This construction is expressed in the following proposition.

Property 3. For all $\mathbf{u} \cdot \mathbf{v}^\sim$ and $\mathbf{w} \cdot \mathbf{z}^\sim$

$$\mathbf{u} \cdot \mathbf{v}^\sim \otimes \mathbf{w} \cdot \mathbf{z}^\sim = (\mathbf{u} \otimes \mathbf{w}) \cdot (\mathbf{v} \otimes \mathbf{z})^\sim$$

Proof. $(\mathbf{u} \cdot \mathbf{v}^\sim \otimes \mathbf{w} \cdot \mathbf{z}^\sim)_{(ip+k)(jq+l)} = (\mathbf{u} \cdot \mathbf{v}^\sim)_{ij} \cdot (\mathbf{w} \cdot \mathbf{z}^\sim)_{kl} = \mathbf{u}_i \cdot \mathbf{v}_j^{-1} \cdot \mathbf{w}_k \cdot \mathbf{z}_l^{-1} = (\mathbf{u} \otimes \mathbf{w})_{(ip+k)} \cdot (\mathbf{v} \otimes \mathbf{z})^{-1}_{(jq+l)} = ((\mathbf{u} \otimes \mathbf{w}) \cdot (\mathbf{v} \otimes \mathbf{z})^\sim)_{(ip+k)(jq+l)}$ □

The advantage of the matricized form is that difficult tensors are replaced by manageable matrices. The type system from the next section supports multiple dimensions but does not have operators for tensors neither does it allow type variables in compound indices. Projection matrices on matricized tensors provide the language primitive needed to do tensor manipulations. This is sufficient to guarantee the existence of a most general unifier.

The dyadic products of unit vectors from the previous paragraph are a free abelian group for elementwise multiplication. Multiplication is lifted to binary dyadic products like

$$(\mathbf{v} \circ \mathbf{w}) \times (\mathbf{x} \circ \mathbf{y}) = (\mathbf{v} \times \mathbf{x}) \circ (\mathbf{w} \times \mathbf{y})$$

and generalized to the n-ary case. Product $1 \circ 1 \circ \ldots \circ 1$ is the unit, with each 1 of appropriate size. Basis elements are products with ones everywhere except one base unit vector. For example product $\mathbf{x}^{-1} \circ \mathbf{y}^3 \circ \mathbf{z}^2$ is written as $(\mathbf{x} \circ 1 \circ 1)^{-1} \times (1 \circ \mathbf{y} \circ 1)^3 \times (1 \circ 1 \circ \mathbf{z})^2$. Product $\mathbf{x} \circ 1 \circ 1$ is an example of a base element. Any product can be uniquely written in such base elements, making the group free.

The set of all tensor bases is isomorphic to $\mathcal{U} \times \mathbb{N}$. Pair ($\mathbf{u}$,i) means a product with $\mathbf{u}$ at the i-th position and 1s everywhere else. The base of $\mathbb{U}$ is extended with $\mathcal{U} \times \mathbb{N}$.

Chapter 5

Typed Linear Algebra

The basics of the type system follows Damas-Milner [DM82, Pie02]. First syntax is provided for expressions and types. Milner's polymorphic type is extended with special syntax for matrices. Multi-linear types require some care to ensure that the conditions for correct unit unification are satisfied. Next type inference is defined for matrices. The extended type system yields principal types.

5.1 Language

The syntax for expressions e is exactly the same as Damas' syntax. Lambda abstraction and function application are the core primitives. As usual the let binding will provide a mechanism to introduce type variables.

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

A type τ is a variable, a function, a pair, or a matrix. The special syntax for matrices is a dedicated extension for the implementation of the matrix type.

τ	$::= t$	type variable
	$\mid \tau \to \tau$	function
	$\mid \tau \times \tau$	pair/tuple
	$\mid \mathrm{Mat}\langle \mu, \gamma, \mu, \gamma, \mu \rangle$	matrix type

Here γ is a list of index set names or a variable.

The matrix type data structure captures the five elements from the most general matrix type `[a * P!u per Q!v]`. This type maps to $\mathrm{Mat}\langle \mathtt{a}, \mathtt{P}, \mathtt{u}, \mathtt{Q}, \mathtt{v} \rangle$. The first argument is a scalar unit, the second and third an index set with units for the row direction, and the fourth and the fifth an index set with units for the column direction. The units are extended with bases as explained in Section 4.5. The next section explains how the language's syntax for matrix types maps to this matrix type data structure.

5.2 Surface Syntax

The language's surface syntax for matrix types is more convenient than the internal representation from the previous paragraphs. In this syntax units can be omitted in the dimensionless case and multiple indices for tensors can be indicated by a comma separated list. A matrix type is `[`δ `per` δ`]` or `[`δ`]` where the syntax for the matrix dimensions δ is as follows.

δ	$::= 1$	One
	$\mid \mu$	scalar unit
	$\mid x!y$	unit vector name
	$\mid \delta^{\mathbb{N}}$	Power
	$\mid \delta * \delta$	Product
	$\mid \delta \,/\, \delta$	Division
	$\mid \delta, \delta$	Kronecker product

Here $x!y$ is a pair of identifiers. These can also be type variables.

The surface syntax is mapped to the matrix type from the previous section. For example the following variation on a matrix type from one of the later examples in section 6.1, with two co-variant and three contra-variant indices

```
[Comm!unit^2, Year per a*Year, b*Region, c/Comm!unit]
```

maps to the following internal representation

$$\mathrm{Mat}\langle \mathtt{a}^{-1}\mathrm{b}^{-1}\mathrm{c}^{-1}, \langle \mathtt{Comm}, \mathtt{Year}\rangle, \langle \mathtt{unit,0}\rangle^2, \langle \mathtt{Year,Region,Comm}\rangle, \langle \mathtt{unit,2}\rangle^{-1}\rangle$$

In this representation the five indices are reduced to two compound indices, each with a compound unit. In general let $\phi(x)$ mean the matrix type from expression x.

$$\begin{aligned}
\phi(1) &= \mathrm{Mat}\langle 1, \langle\rangle, 1, \langle\rangle, 1\rangle \\
\phi(\mu) &= \mathrm{Mat}\langle \mu, \langle\rangle, 1, \langle\rangle, 1\rangle \\
\phi(g!) &= \mathrm{Mat}\langle 1, \langle g\rangle, 1, \langle\rangle, 1\rangle \\
\phi(g!h) &= \mathrm{Mat}\langle 1, \langle g\rangle, \langle h, 0\rangle, \langle\rangle, 1\rangle \\
\phi(x^n) &= \phi(x)^n \\
\phi(x * y) &= \phi(x) * \phi(y) \\
\phi(x, y) &= \phi(x) \otimes \phi(y) \\
\phi(x \text{ per } y) &= \phi(x) \text{ per } \phi(y)
\end{aligned}$$

where the appropriate operations on matrix types in the last four cases are assumed. The 'per' combines the vectors into a matrix. Note that its arguments are always scalars or vectors. The fist four cases only create scalars and vectors because the last arguments are always $\langle\rangle$ and 1, and the operators besides 'per' don't create matrices.

$$\text{UnifyMatrices}(\tau, \tau') = S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1$$

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

Figure 5.1: Unification for matrices. The unit parts are unified by Kennedy's semantic UnifyUnits algorithm, the index set parts by standard syntactic unification. The result is the composition of the partial results.

5.3 Type Inference

A type schema introduces quantified variables. Index set variables are added to the units variables and the normal type variables.

$\sigma ::= \tau$	type
$\mid \forall t.\sigma$	type quantification
$\mid \forall u.\sigma$	unit quantification
$\mid \forall p.\sigma$	index set quantification

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

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

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

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

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

As usual the scope of the type variables is determined by the introduction of type schemes σ in the let construction. Finally a rule for the index set variables

is added to the generalization and instantiation rules for type variables and units variables.

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

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

At this point the only deviation from Damas and Milner and Kennedy is the index set variables. With this in place the unification of the matrix type can be defined.

Unification is defined component-wise. The three cases with units are unified with the unification algorithm for units, the other two with Damas-Milner style unification, as shown in Figure 5.1. The correctness of this matrix type unification follows directly from the correctness of unit and standard unification. The extensions to the syntax have no impact on the properties of the group of units.

Theorem 3. Type inference for the polymorphic type extended with matrix types is sound and complete.

Proof. Kennedy's extension to Damas' type inference algorithm is sound and complete. Since the changes for the matrix type are confined to the base units $\mathcal{U}$ as explained in section 3.3 the abelian group of units remains free. Since unification is composable the result follows from each of the five unification occurrences in Figure 5.1. □

5.4 Projections and Conversions

For conversions and projections special constructors are added to the language. These constructors expect type expressions as arguments and have the following types:

$$\begin{aligned} &\texttt{convert}(\varepsilon,\mu_0,\mu_1) \;::\; [\varepsilon\texttt{!}\mu_0 \;\texttt{per}\; \varepsilon\texttt{!}\mu_1] \\ &\texttt{project}(I \;\texttt{per}\; I) \;::\; [I \;\texttt{per}\; I] \end{aligned}$$

In our experimental language these constructors can be used to declare these special matrices, but they are not part of the expression language.

Conversions have the following definition.

$$\texttt{convert}(\varepsilon,\mu_0,\mu_1)_{ij} = \begin{cases} (\varepsilon\texttt{!}\mu_1)_i/(\varepsilon\texttt{!}\mu_0)_j & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

The existence of a conversion matrix cannot be checked before the units are known. If this is not before run-time then a run-time error is the only option. Conversions are the only construction with possible run-time errors.

The construction of projections may also fail, but the validity of projection matrices can be checked at compile time. Projections are defined as follows.

$$\texttt{project}(\varepsilon_0\texttt{!}\mu_0 \;\texttt{per}\; \varepsilon_1\texttt{!}\mu_1)_{ij} = \begin{cases} 1 & \text{if } i,j \text{ in the projection} \\ 0 & \text{otherwise} \end{cases}$$

Details for the construction of projection matrices can be found in [MO11]. The definition shows that each entry in a projection matrix is dimension-less. If the type does not agree then the projection does not exist. Since we can do type inference we can do this check at compile time.

5.5 Runtime support

A unit-aware runtime system is capable of interaction in the proper units, and detection of operations on incompatible units. Interaction can be with the user or with external systems. Of every incoming and outgoing number the unit should be known, either explicitly or implicitly. Detection of incompatible operations means that no computations with mismatching units can be performed unnoticed. What the runtime system needs to do depends strongly on the language and what for example a compiler already can do, but interaction in proper units and detection of incompatibilities are the two major tasks of a unit-aware runtime system.

In the previous section it was argued that the non-existence of conversion matrices is the only possible run-time error. This means that all unit operations can be checked at compile time, except conversions and communication with the outside world.

A second issue is efficiency of handling units for numerical data collections. Efficiency is gained if units are not associated with individual numbers but with matrices. The separation of units and magnitudes makes it easy to add units to a matrix data structure. Since the units in a matrix require only two vectors this storage is efficient. The operators on such run-time types are also efficient, for example matrix multiplication is just updating some references. Using these properties it stores less information and makes units easier to handle.

Chapter 6

Examples

The following type assertion was deduced by the unification algorithm. It shows that the function requires and produces square matrices.

```
> define f(x) = (x + x^R)
f :: forall P,u: [P!u per P!u] -> [P!u per P!u]
```

Let's follow the unification manually by letting `x` be a matrix of type `[a * P!u per Q!v]`. Then dimensional inverse `x^R` is a matrix of type `[a^-1 Q!v per P!u]`. Unification of the sum requires `P` = `Q`, `u`=`v`, and `a`=`a^-1`. Unification of the units adds conclusion `a`=`1`. This means `x` is of type `[P!u per P!u]`. This example demonstrates the equational reasoning that occurs during unification.

A more meaningful example is the definition of the inner product via the sum of the element-wise product. Assume function `total` has type `[a * P per Q] -> [a]`. This is an example of a type with homogeneous units. The inner product can be defined as.

```
> define inner(x,y) = total(x*y)
inner :: forall a,P,u,Q,v:
           [a * P!u per Q!v] x
             [b * P!u^-1 per Q!v^-1] -> [a*b]
```

It is infered that the result is a singleton matrix. We interpret that as a single number. Note that the arguments do not have to be vectors. They can also be matrices, as long as their row and column spaces are reciprocal.

This variation on the definition of the norm is also infered as expected:

```
> define norm(x) = sqrt(total(x*x))
norm :: forall a,P,Q: [a * P per Q] -> [a]
```

It shows that the norm requires a homogeneous matrix. The derivation is similar to the previous one for the inner product. It is assumed that `sqrt :: [a^2] -> [a]` or `sqrt :: [a^2 * P!u^2 per Q!v^2] -> [a * P!u per Q!v]`.

6.1 Commodities Case

The second example shows the multi-dimensional case. It is an adaptation of the OLAP example from [MO11] that demonstrates heterogeneous units of measurement in multi-dimensional numerical data.

For the multi-dimensional case we have the following numerical data store

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

The store contains sales data per commodity, year and region. From these three dimension only commodities have units.

Commodity	units ...
Gold	ounce
Oil	barrel
...	...

The following multi-dimensional types are assumed.

```
sales :: [dollar per Comm,Year,Region]
amount :: [Comm!unit,Year,Region]
```

The type system helps with the construction of projections. Say we want to project commodities from the sales column. A projection matrix has to match the `Comm * Year * Region` part of `sales`'s type and the result has to be `dollar` per `Comm`. Combining this leads to projection matrix `P0` and the sales per commodity can be projected by multiplying with it from the right.

```
P0 :: [Comm,Year,Region per Comm]

> define sales_per_commodity = sales . P0
sales_per_commodity :: [dollar per Com]

>sales_per_commodity

 Commodity  |    sales
------------+---------------
 Oil        |   92,000.00 $
 Gold       |  178,000.00 $
```

Multiple dimensions can also be projected. Matrix `P1` is a correct projection matrix for years and commodities. The result is of type `Year,Comm!unit`.

```
P1 :: [Year,Comm!unit per Comm!unit,Year,Region]

> P1 . amount

 Year  Commodity  |    amount
------------------+---------------
 1990  Oil        |  2,550.00 bbl
 1990  Gold       |    385.00 oz
 1991  Gold       |     34.00 oz
```

Chapter 7

Conclusion

The combination of Damas-Milner type inference and Kennedy's unit inference applied to Hart's dimensioned matrices enables inference of principal types for linear algebra expressions. The static type system's symbolic reasoning provides powerful feedback at compile time and guarantees absence of unit errors at runtime. The only run-time error that can occur is when a conversion matrix does not exist.

The ability of the matrix type to handle heterogeneous units in addition to homogeneous units makes static safety of numerical expressions available to a much larger set of use-cases. The type is applicable in domains ranging from symbolic algebra to data warehousing. The examples show that even without concrete units the semantic type system provides powerful symbolic information about vector spaces. But the type also provides advantages for the implementation of units of measurement at runtime. The storage of units at runtime can be more efficient since a dimensioned matrix only requires two vectors of units that can be kept separate from the magnitudes. Another consequence is that computations of units can be done separately from the more expensive matrix computations.

In further research we plan to improve the implementation of the experimental design and apply it to more use-cases. Questions from a language implementation perspective concern the efficiency and performance of numerical expressions with units. The units require extra administrative work at run-time but it might also benefit from the improved type information by generating better compiled code. Computations might for example be distribution over multiple nodes via block matrices and the type system might provide interesting information to implement this. Finally we hope to gain more insight by applying the type to various fields. Examples are typed libraries for matrix languages, typed OLAP and data warehouses, typed services, etc.

Bibliography

[ACL04] Eric E. Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L. Steele Jr. Object-oriented units of measurement. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA*, pages 384–403. ACM, 2004.

[BIP] BIPM. International vocabulary of metrology - basic and general concepts.

[Bir60] G. Birkhoff. *Hydrodynamics: a study in logic, fact, and similitude.* Princeton University Press, 1960.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[dPeM] Bureau International des Poids et Mesures. *The International System of Units (SI).*

[Elm63] Salah E. Elmaghraby. A note on the 'explosion' and 'netting' problems in the planning of materials requirements. *Operations Research*, Vol. 11, No. 4, pp. 530-535, 1963.

[Fou22] J.B.J. Fourier. *Théorie analytique de la chaleur.* Chez Firmin Didot, père et fils, 1822.

[Har] George W. Hart. The theory of dimensioned matrices.

[Har95] G.W. Hart. *Multidimensional analysis: algebras and systems for science and engineering.* Springer-Verlag, 1995.

[Ken94] Andrew Kennedy. Dimension types. In Donald Sannella, editor, *ESOP*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 1994.

[Ken96] Andrew J. Kennedy. *Programming Languages and Dimensions.* PhD thesis, University of Cambridge, 1996.

[Ken09] Andrew Kennedy. Types for units-of-measure: Theory and practice. In Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók, editors, *CEFP*, volume 6299 of *Lecture Notes in Computer Science*, pages 268–305. Springer, 2009.

[MO11] H.D. Macedo and J.N. Oliveira. Do the middle letters of "olap" stand for linear algebra ("la")?, 2011. Journal paper (submitted July 2011); information available from the authors' websites.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Roc98] J.J. Roche. *The mathematics of measurement: a critical history*. Athlone Press, 1998.

[Rom08] Steven Roman. *Advanced Linear Algebra*. Springer New York, 3rd edition, 2008.

[SJ06] Lingxiao Jiang and Zhendong Su Osprey: a practical type system for validating dimensional unit correctness of c programs. *Software Engineering, International Conference on*, 0:262–271, 2006.

[Str88] Gilbert Strang. *Linear Algebra and Its Applications*. Brooks Cole, February 1988.