NPL
National Physical Laboratory

NPL REPORT MS 60

FUNCTIONAL PROGRAMMING (WITH SOME TYPE THEORY) FOR METROLOGY

ISSUE 1.0

ANDRÉ VIDELA, KEITH LINES

MARCH 2025

Functional Programming (with some Type Theory) for Metrology

André Videla[1], Keith Lines[2]

[1] Mathematically Structured Programming Group, University of Strathclyde

[2] Data Science Department, NPL

ABSTRACT

Results are presented of an investigation by the NPL Data Science department and the Mathematically Structured Programming Group of the University of Strathclyde into how functional programming and type theory could increase trustworthiness of software for metrology.

Examples of software that help provide NPL measurement services are explored. Functional languages, such as Haskell and Idris 2 (which provides dependent types), can be used to develop code that is more concise, and more self-evidently correctly implement underlying mathematical models, than more traditional imperative languages such as Python or Visual Basic.

This report is not a detailed introduction to functional programming or type theory. Sufficient background information is provided to ensure readers unfamiliar with these subject can follow the case studies and the authors' conclusions. References giving further details are provided.

DOCUMENT HISTORY

Issue	Date	Author(s)	Change(s)
1.0	31/03/2025	André Videla, Keith Lines	Initial issued version.

ISSN 1754-2960

https://doi.org/10.47120/npl.MS60

National Physical Laboratory
Hampton Road, Teddington, Middlesex, TW11 0LW

This work was funded by the UK Government's Department for Science, Innovation & Technology through the UK's National Measurement System programmes.

Approved on behalf of NPLML by Spencer Thomas,
Science Area Leader, Data Science Department

GLOSSARY

Term	Definition
Dependent type	Reference *Wikipedia* [1]: > In computer science and logic, a dependent type is a type whose definition depends on a value. Also see section 3 of this report.
Functional programming vs. imperative programming	Reference *Haskell website* [2]: > In functional programming, programs are executed by evaluating expressions, in contrast with imperative programming where programs are composed of statements which change global state when executed. Also see section 2 of this report.
Programming paradigm	Reference *P. Van Roy* [3]: > A programming paradigm is an approach to programming a computer based on a coherent set of principles or a mathematical theory. The paradigms of interest in this report are *functional programming* and *imperative programming*.
Theoretical computer science (TCS)	Reference *UK Research and Innovation* [4]: > This area explores the fundamental and foundational aspects of computers and computation. Aiming to improve understanding of computation and its capabilities, limitations and future potential, this research area encompasses research around logic and semantics, and the study of algorithms, complexity and automata. Some theoretical computer scientists may object to the term “computers” in the above definition. Reference *Wikipedia* [5]: > Theoretical computer science (TCS) is a subset of general computer science and mathematics that focuses on mathematical aspects of computer science such as the theory of computation, formal language theory, the lambda calculus and type theory.

Term	Definition
Trustworthy	*BS 10754-1:2018* [6] defines trustworthy as: > Appropriately addresses safety, reliability, availability, resilience and security issues. The standard refers to these issues as the five *facets* of trustworthiness.
Type theory	Reference Wikipedia [7]: > . . . the academic study of type systems.
Type system	Reference Wikipedia [8]: > . . . a set of rules that assigns a property called a type (for example, integer, floating point, string) to every term (a word, phrase, or other set of symbols). Usually the terms are various language constructs of a computer program, such as variables, expressions, functions, or modules. A type system dictates the operations that can be performed on a term. For variables, the type system determines the allowed values of that term.

CONTENTS

1 INTRODUCTION

This report summarises work undertaken as part of a wider collaboration between NPL's Data Science Department [9] and the Mathematically Structured Programming Group of the University of Strathclyde [10]. The aims of this work included:

- Demonstrate whether *functional programming* and *dependent types* could be applied to help increase trustworthiness of metrology software developed within NPL.
- Help determine whether the underlying theoretical computer science could be made accessible to and applicable by metrologists.
- Make the case for the value of some knowledge of theoretical computer science for anyone who codes. An appreciation can help write code that is more maintainable and trustworthy.

1.1 READERSHIP

This report is largely, but not exclusively, aimed at metrologists accustomed to using programming languages like LabVIEW [11], Python [12] or MATLAB [13] for data acquisition and analysis. We present how modern techniques of computer science can help the day-to-day work of the scientist and outline possible uses and research areas for future tools and programming languages.

Providing detailed introductions to functional programming, type theory, dependent types etc. would be well beyond the scope of this report. Also it would not be a sensible use of resources, as many excellent introductions have already been written.

Instead, an overview of these subjects is presented (using some examples from metrology). The overview is broad but contains sufficient detail to allow the case studies to be followed at a reasonably high level. Numerous references provide further details.

1.2 HISTORICAL CONTEXT

NPL's Data Science department and its predecessors have a long history of employing computer scientists, and collaborating with computer scientists outside of NPL. The development of the Pilot ACE [14] is arguably the most famous example, but there are many others [15].

Input to the development of the ALGOL [16] and ADA [17] programming languages benefited software development at NPL. NPL's sponsorship of Christopher Stratchey and Robert Milne's development of programming language semantics [18] should also be noted.

As noted in [15], present-day work in this area could hardly be as dramatic. But this report presents an argument for NPL maintaining a presence in the fields of computer science and programming language development. The NPL Data Science department should be somewhere that metrologists and computer scientists can communicate fruitfully.

1.3 REPORT STRUCTURE

The report is structured as follows:

- Sections 2 and 3 provide background information required to follow the case studies. Readers familiar with functional programming and type theory could skip these sections. But their opinions on the approach taken would be of interest to the authors.
- Sections 4 and 5 describe the case studies. The report ends by drawing some conclusions and providing suggestions for further work.

- Appendices provide further background information. Appendix B provides a reference list intended for readers unfamiliar with functional programming and type theory.

2 Functional Programming

Functional programming is a *paradigm* in which the principal means of programming is to manipulate *functions*, rather than execute instructions. Code written in a *functional language* can be more concise and more closely resemble mathematical models of physical systems than code written in one of the traditional *imperative languages* such as C++ or Python. One difference between these paradigms may be summarised as follows. For mathematical calculations:

- An imperative program consists of a sequence of instructions telling the computer *how* to perform a calculation.
- A functional program consists of a *description* of a calculation, something more akin to an equation.

As Peter Landin stated in his seminal 1966 paper [19]:

> LISP brought the class of entities that are denoted by expressions a programmer can write nearer to those that arise in models of physical systems and in mathematical and logical systems.

The above statement is even more true of modern-day functional languages, such as Haskell and Idris 2 as it is of LISP [20]. It is interesting to note that functional programming is being used as a means of teaching physics [21–23].

2.1 Introductory Example

In figure 1 we introduce the concept of functional programming using a side-by-side comparison of implementations of the factorial function in Python (iterative) and Haskell (functional). This example is commonly used, but still worth studying:

Figure 1: Compare factorial implementations

A key point to consider is the different role of "=" in both implementations:

- In Python "=" is not an expression of mathematical equality. It is a *statement* telling the computer to assign a value to the variable `fact`. After each assignment, i.e. the initialisation to 1 and after each execution of the loop, `fact` will contain a different value.

  In other words, after each assignment the *state* of the computer has been changed. That is to say, the value of a memory location will have been altered.

- In the Haskell code "=" *expresses mathematical equality*, it is not a statement telling the computer to perform an action. Instead factorial is defined using two expressions:
  - An expression that the factorial of 0 is equal to 1
  - For all other input values, call them `n`, their factorial is equal to `n` multiplied by the factorial of `n - 1`.

As long as `n ≥ 0` this definition will "unravel" into a series of so-called recursive calls, ending with the factorial of 0, which we have defined as 1. But see the note below...

– A similar example is provided in further detail in section 1.5 of [24].

– At the head of the function, the *function type* `factorial :: Int -> Int` expresses that factorial inputs an integer and returns an integer.

NOTE: This implementation of factorial was chosen as a simple example of functional programming. But inputting a negative integer would result in a stack overflow, rather than a helpful message saying that factorials of negative integers cannot be calculated. More trustworthy Haskell implementations of factorial can be written.

2.2 Further Details

Functional programming comes with benefits such as powerful abstraction mechanisms, smaller and clearer code, free of the repetition of boilerplate code [25], and a smaller space for programming errors thanks to a reduced cognitive load.

Functional programming is not one thing, but a collection of features and practices that some languages embrace. Among those we note:

- The presence and use of *first-class* functions. That is to say functions can be passed as parameters to, or returned as a value by, other functions. The related concept is that of *higher order functions* which take functions as parameters and/or returns a function as a value.
- Infrastructure to control *side effects*. As noted in [24], *pure* functions take all their inputs as arguments and generate all their outputs as results. Other matters such as reading keyboard or sensor input, outputting to a screen or using global variables are side effects. Pure functions are easier to reason about formally than functions containing side effects. Functional languages such as Haskell provide infrastructure to use side effects while functions remain pure. Reference [24] provides further details for Haskell.
- A powerful type system.
- A lean but expressive syntax.

Different programming languages optimise for different aspects. A language such as LISP is considered functional thanks to its use of first-class functions and strong metaprogramming [26] capabilities.

2.3 Languages Used In This Study

For the study summarised in this report we selected functional languages Haskell [27] and Idris 2 [28]. Haskell was chosen for its industrial-strength compiler and libraries. As Haskell does not provide dependent types, Idris 2 was also selected as it offers many of the same features as Haskell. Also, the first author of this report is involved in the maintenance of the Idris 2 compiler.

For non-functional languages, MATLAB [13] was selected because it is widely used at NPL. Python [12] was chosen because of its ubiquitousness.

2.4 Trustworthiness

We study the practice of programming through the lens of *trustworthiness*. In this report, we use the definition of trustworthiness provided by standard BS 10754-1:2018 [6]:

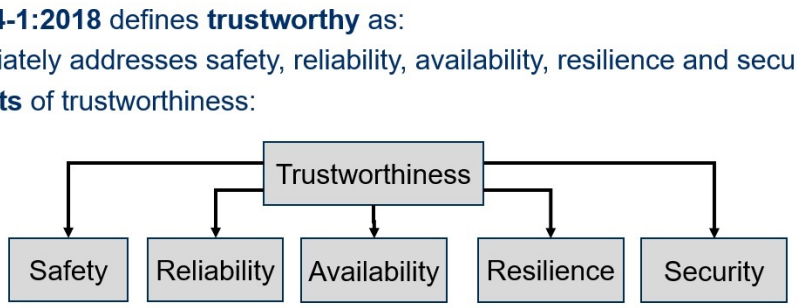

Figure 2: Definition of trustworthy

A trustworthy piece of software can be relied on to do what it is meant to do, no more, no less. Obviously, that requires a language for describing the intent of the program, before the program is written. And this description needs to match what is run in the computer, regardless of the platform or version number of the software. The specification and implementation need to be in synchronisation.

We identify the following issues with ensuring trustworthiness of software written at NPL:

- If a specification is provided, in general, it will be written in prose and checking that the code abides by it can be difficult.
- Source code is not resilient through time due to software updates and ever-changing data formats. This need for new requirements creates churn and duplicates work, which also increases the chance for errors.
- Known sources of errors cannot be accounted for during the development of the program, even if we know about them in advance thanks to domain specific knowledge. For metrology, those errors are typically related to linear algebra or dimensional analysis.

We show how functional programming and advanced type systems can mitigate those issues. Our methodology relies on porting existing programs into Haskell and Idris 2, two functional programming languages known for their expressivity and ability to optimise idiomatic code efficiently.

3 Type Theory and Dependent Types

As stated in the introduction, some of the work presented in this report falls into the domain of theoretical computer science (TCS). A key challenge was to determine whether the relevant subjects could be made accessible to a metrology audience.

For the reader familiar with coding, the task of beginning to understand the theory could be divided into the following steps.

3.1 Step 1: Think about what you know already

A type system is defined in Wikipedia [8] as:

- . . . a set of rules that assigns a property called a *type* (integer, floating point, string) to every term.
- Usually, the terms are language constructs such as variables, expressions, functions. . .
- . . . dictates the operations that can be performed on a term

NOTE: Type systems detect basic errors, such as dividing an integer by a string, at compile time. It is this feature that we want to extend to checking whether statements are dimensionally correct at *compile time*.

3.1.1 Example

Consider the following 64-bit stream (little endian):

```
01111010 10010100 01001110 10100001 11010011 11000011 11101100 00111111
```

- Is it IEEE double-precision floating point number 0.8989046240354945?
- Is it unsigned long long int 4606271832604972154?
- Is it a sequence of characters? zN¡ÓÃì?
- Or is it just a bitstream, perhaps one whose randomness we seek to determine?

The type determines how to interpret the bitstream, including the operations that can be performed on it.

3.2 Step 2: Think about the bigger picture

Type theory is defined as "the academic study of type systems" [7]. It is important to understand that type theory did not begin with the development of modern-day (digital) computers. In fact, it sprang from an ongoing quest to find a formal system [29] as a basis for mathematics. The TCS came first. To summarise:

- This quest goes back at least as far 300 BC with Euclidean geometry [30] and its *axioms*. But the modern-day phase began with Leibniz in 1667.
- Arguably the most (in)famous such system is that presented in Whitehead and Russell's epic Principia Mathematica [31].
- As part of the work leading to the Principia Mathematica Russell introduced *the Doctrine of Types* to help address the famous set theoretical paradox:

  Let $R = \{x | x \notin x\}$ **then** $R \in R \Leftrightarrow R \notin R$

- To date, the most widely-accepted formal system underpinning mathematics is Zermelo-Frankel set theory with the axiom of choice (ZFC) [32].

Further details are presented in the BCS webinar "The Next 700 Domain Specific Languages" listed in appendix B.

3.3 Step 3: Start being judgemental

Judgements are not statements *in* the type theory, but statements *about* the type theory. Simplified forms presented in [33] are listed in figure 3:

_____ is a _____	**_____ is the same _____ as _____**
_____ is a type	**_____ and _____ are the same type**

Figure 3: Judgements simplified

3.4 Step 4: Compare type theory to set theory and predicate logic

Set theory and type theory share enough superficial similarities for an understanding of one to help begin understanding the other. As Klev notes in [34]:

> There might, for instance, be a set containing the author of this paper, the Mona Lisa, the boolean value true, and the empty set. The notion of set may thus be said to be formal in the sense of topic-neutral: a set is not tied to any one topic, any one conceptual sphere, but can have elements from arbitrary conceptual spheres.
>
> Types, by contrast, may be thought of as being just these topics or conceptual spheres themselves.

Figure 4 below summarises some key differences:

	Set theory (e.g., ZFC)	Type theory (e.g., Martin-Löf)
Foundations	Built on **predicate logic**, e.g.: {(x, y) | x ∈ ℕ ∈ Λ y ∈ ℕ Λ y = 2x}	**Stand-alone.** Built on its own foundations
Primitive notion	Sets	Functions
Functions	Defined using **cartesian products**, e.g.: double = {(1,2),(2,4),(3,6)…} ⊆ ℕ × ℕ	Defined using **lambda expressions**, e.g.: λ x • 2*x
Theorems	Use **axioms** and **inference rules** to prove theorems either **TRUE** or **FALSE**	Use **inference rules** to **construct** proofs of theorems. **No axioms** (as such), uses inference rules without **premises** instead.

Figure 4: Compare set theory to type theory

3.4.1 "∈" versus ":"

It is important to bear in mind that statements of set membership and type membership have different meanings. Consider the following statements: $n \in \mathbb{N}$ and n : Nat

The first is saying that n is a member of the set of natural numbers (an entity that is considered to already exist somehow). The second is a statement of how "n" will be constructed. It is also a *judgement* of the first kind listed in figure 3 (":" being equivalent to "is a").

A helpful analogy may be to consider that, at some level of abstraction, a set would be a means of modelling a packet of biscuits (if each biscuit is considered to be a unique entity). To determine the *type* of biscuit, it will be necessary to read the label on the packet.

3.5 Step 5: Think about relevance to metrology

As fascinating as philosophical debates about the foundations of mathematics [35] etc. may be, they are not a justified use in themselves of the budgets and resources at National Measurement Institutes. We now consider the relevance to metrology. Kennedy notes in [36] that "Units-of-measure are to science what types are to programming". As Conor McBride and Fredrik Nordvall Forsberg note in [37, 38]:

> The casual conceptual conflation of physical quantities with mere numbers is a standard computational practice resulting in lamentable errors. A common approach to addressing this problem is to bundle numbers with units in compound data structures, then check dynamically whether arithmetic operations make sense before performing them. However, that approach pushes to run time what is much better managed at the time of a program's construction.

It is interesting to note the use of the type theoretical term "construction" rather than "development". They continue:

> Type systems are natural tools for mechanising the concept of dimensional analysis, yielding a lightweight approach to data integrity for metrology. Indeed, there is already work in this direction, with dimension types implemented in mainstream languages.
>
> None support static units for compound data structures, such as vectors and matrices, except in overly uniform ways (e.g. matrices of quantities all in the same dimension).

3.5.1 Example: Resistance Calibrations

Consider the example metrology calculation in figure 5, from software used to calculate resistance measurements. This example is explored in detail in reference [39]:

$$\begin{pmatrix} d_{\mathrm{nc}1} & d_{\mathrm{c}1} & n_1 & c_1 & t_1 \\ d_{\mathrm{nc}2} & d_{\mathrm{c}2} & n_2 & c_2 & t_2 \\ & \vdots & & & \\ d_{\mathrm{nc}N} & d_{\mathrm{c}N} & n_N & c_N & t_N \end{pmatrix} \times \begin{pmatrix} a \\ a_{cal} \\ c \\ c_{cal} \\ m \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{pmatrix}$$

Figure 5: Metrology example

This calculation may be summarised as follows:

$$\Big(Known\ stimulus\Big) \times \Big(Unknown\ parameters\Big) = \Big(Measured\ response\Big)$$

As noted in [39] an electric current with a known value (the applied signal) is passed through the resistor being measured, and a series of voltage readings are taken at fixed time intervals (the recorded output). After a specified number of readings are taken, the direction of the input signal must be reversed in order to separate offset and drift from the voltage readings. A calibration factor is also applied, dividing the signal into two parts with different amplitudes. The column vector right of × and left of = contains values to be calculated using a line fit which then determines the measured value of the resistor using a calculation which is beyond the scope of this report.

In the matrix in figure 5:

- The values in the first four columns are purely numeric and have no associated dimension.
- The first column will contain either 1 or -1 for non-calibration voltage readings (i.e., those to which the calibration factor has not been applied) depending on the direction of the input signal. For calibration readings the value will be 0.
- The second column contains equivalent values for voltage readings in which the calibration factor has been applied. I.e. 0 for non-calibration readings and 1 or -1 depending on the input signal direction for calibration readings.
- The third column contains 1 for non-calibration readings and 0 for calibration.
- The fourth column is the opposite of the third.
- The fifth columns contains values having the associated dimension of time.

The type of the contents of the matrix and both column vectors could all be floating point. Adding one unit type to all contents of each matrix / vector would not be helpful.

But, looking at the matrix, what if the type of each column were a function from the column number to an appropriate combination of numeric and unit types? I.e. for the first four columns a combination of integer and, as are the values are simply numbers, unit **1** (see section 5.4.7 of the 9th edition of the SI Brochure [40]). The final column contains floating point values with unit seconds.

That is to say, the column number is an *index into a family of types*. That is exactly what dependent types offer.

There is another important issue here, which is that in figure 5 the unit types of the component quantities in the matrices and vectors must be consistent with how they are combined. Consider the first unit vector, for example, a, a_{cal}, c, c_{cal} must have the unit type of voltage because that is the type of the component quantities in the rightmost vector. Dependent types help by the type of each *row element* being a function from the row number to an appropriate combination of numeric and unit types.

4 Case Studies for Functional Programming

The goal of this work is to evaluate the following:

- Can a functional programming language provide equivalent affordances compared to the existing popular languages in data analysis such as Python? We are especially interested in comparing syntax, libraries ecosystem and numerical accuracy.
- What class of errors can we avoid using a language such as Haskell for data analysis?
- In which ways does Haskell fall short of Python?

NOTE: This section contains some Haskell notation without explanation: please refer to [27] and [24] for explanations.

4.1 Uncertainty Evaluation: The Law of Propagation of Uncertainty

The first case study concerns a straightforward but vital calculation for a national measurement institute such as NPL. A measurement result is incomplete without a quantitative statement about the quality of the measured value (in the form of an uncertainty), and hence the importance to metrology of making such statements trustworthy.

Figure 6 shows an example of such a statement:

Value	Uncertainty	Mean Date
1·000 003 97 kΩ	± 0·05 ppm	4 September 2023

The reported expanded uncertainty is based on a standard uncertainty multiplied by a coverage factor $k = 2$, providing a coverage probability of approximately 95 %.

Figure 6: Example of statement of uncertainty

The above statement says there is an approximately 95 % probability that the true value, rather than the measured value, lies within the specified range. The *expanded uncertainty*, quoted in parts per million, is equal to the *standard uncertainty* multiplied by the coverage factor $k = 2$. The above statement is also based on the values of the measured quantity being normally distributed:

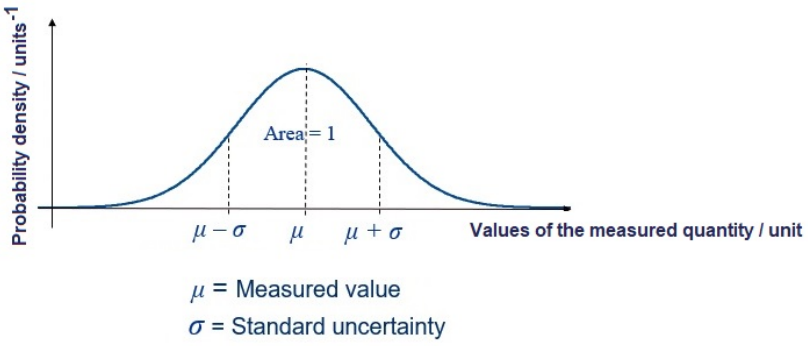

Figure 7: Standard uncertainty, normal distribution

Often there is a need to combine standard uncertainties from a variety of inputs (e.g., an array of sensors). The *law of propagation of uncertainty* (LPU) [41–43] provides a means of combining these uncertainties. The following formula applies if the inputs are independent:

$$u_y = \sqrt{\sum_{j=1}^{N} c_j^2 u_j^2}$$

where:

- u_y is the standard uncertainty of the output quantity.
- $N > 0$ is the number of input quantities, indexed by j
- c_j is a sensitivity coefficient associated with input j
- u_j is a standard uncertainty associated with input j
- The following assumptions are made:
  - There is one output quantity.
  - The input quantities are independent.
  - The output quantity can be expressed as an explicit function of the input quantities: $Y = f(X_1, \cdots, X_N)$

4.1.1 LPU: MATLAB Implementation

NOTE: Complete listings of the code in the following sections can be found in appendix C.

The above formula can be coded in MATLAB as follows:

```
uy = 0;
for j = 1:N
    uy = uy + c(j)^2*u(j)^2;
end % for j
uy = sqrt(uy);
```

or more succinctly:

```
uy = sqrt((c.^2)*(ux.^2)');
```

where `(ux.^2)'` is a transpose of the vector `(ux.^2)`. Therefore, provided `c` and `ux` are both row vectors containing the same number of components, the multiplication operator will return a 1×1 matrix containing the sum of the squared values of the vector components.

Although straightforward, this calculation displays some important properties for software that implements it:

- In the first for loop-based (i.e. iterative) implementation the variable `uy` is first calculated as the variance and then overwritten by the standard deviation. So, its "meaning" changes as the code is executed. The correct answer is generated, so are such matters worth consideration? This report argues that is definitely the case.
- In the second, more functional in appearance (although it is an assignment statement not a statement of mathematical equality) implementation the following statements apply because `c` and `u` are row vectors:
  - The programming language needs to handle *overloading* of notation for multiple types, including vectors and scalars. See below for a definition of overloading.
  - The multiplication operator can only work when both vectors are *dimensionally consistent*. In this example, as noted above, both inputs have to be row vectors of the same length.

4.1.2 Haskell: Hackage Package Data.Matrix

The following Haskell implementation uses the Data.Matrix package [44] downloaded from the Hackage Package Repository. `c` and `ux` are both matrices containing one row. Wrapping the matrix multiplication function `multStd` in backticks allows it to be used as *infix notation*, i.e. placed between the matrices being multiplied.

`fmap` applies a function to each matrix element. Mapping function `\x -> x*x` squares the matrix contents. As `multStd` returns a 1×1 matrix, `fmap` is also required to obtain the square root.

```
uy :: Matrix Double
uy = fmap sqrt ((fmap (\x -> x*x) c) `multStd` (fmap (\x -> x*x) (transpose ux)))
```

4.1.3 Haskell: Vectors

As an alternative to the above implementation a vector type using Haskell lists can be declared and the calculation implemented as follows:

```
uy :: Double
uy = sqrt ((c.^2) `sumMultVec` (ux.^2))
```

where `sumMultVec` inputs two row vectors and outputs a scalar value. This value is the result of multiplying the equivalent components of the input vectors then adding the components of the resulting vector:

```
sumMultVec :: (Floating i) => Vec i -> Vec i -> i
sumMultVec a b = sum (asList (a * b))
```

where:

- The first line declares the function type.
- The section before `=>` is called a *class constraint*. `Floating` is the class of Haskell floating-point number types, i.e., `Float` (single-precision) and `Double` (double-precision). `i` is a *type variable*. Further information on class constraints can be found in chapter 3 of [24].
- `Vec i` is a type defined using Haskell lists. The class constraint ensures that the type of the vector components can be either of the Haskell floating-point number types. The type of the value output by `sumMultVec` can also be either of these types.
- All vector component types and the type of the output value must be consistent. That is to say all components of all vectors and the output value are of type `Float` or all contents of all vectors and the output value are of type `Double`.
- Functions whose type definitions contain one or more class constraints are said to be *overloaded*. That is to say they can be applied to parameters of more than one type.

- Backticks allows the function name be to placed between the vectors
- Point-wise versions of the power and multiplications operators will need to be defined. See the code listing in appendix C.4.1 for further details.

  NOTE: In Haskell, operators are functions that input two parameters and are written using infix notation. Operator names have to consist of non-alphanumeric characters.

Finally, we notice that we are unable to check that both vectors have the same length. Our `Vec` type is only parameterised over the type of the content, but not the length of the array. Calling the following code will result in a value of 0:

```
worksButShouldNot :: Double
worksButShouldNot = (Vec [1,2,3]) `sumMultVec` (Vec [])
```

4.1.4 Idris 2: Sized Vectors

To ensure both vectors are the same length, we need to augment our `Vec` type with the vector length. Idris 2 provides this feature with *dependent types*. The code implementing this calculation in Idris 2 is listed below. There are differences with the Haskell syntax (e.g., `:` in function definitions, rather than `::`) but they should not cause readability problems:

```
-- Assign sensitivity coefficients.
c : Vect 4 Double
c = [0.5, 0.4, 0.3, 0.2]

-- Assign uncertainties associated with estimates of input quantities.
ux : Vect 4 Double
ux = [0.1, 0.2, 0.3, 0.4]

uy : Double
uy = sqrt ((c.^2) `sumMultVec` (ux.^2))
```

The type declaration `Vect 4 Double` gives a first glimpse of dependent types.

The $*$ operator now has type:

```
-- n is the length of the vector
(*): Num a => Vect n a -> Vect n a -> Vect n a
```

where `n` is the vector length, it has type natural number. This version of the operator ensures that both vectors given in the argument are the same size, and that the resulting vector is also the same size. The round brackets indicate that $*$ is an operator.

Languages that provide *dependent types* offer a trustworthy means of implementing the above operator. A dependent type is determined by something that is not a type, in this example the parameter `n`. For such languages, the ability to define such types is built into the type system.

Another way to think of dependent types is that they promote types to being *first-class citizens* [45]. That is to say an "entity which supports all the operations generally available to other entities". It is interesting to note that the concept of first-class citizens has its origins in the definition of ALGOL, in which NPL had a role.

The definition of `sumMultVec` is now:

```
-- sumMultVec inputs two row vectors of the same length and outputs
-- a scalar value. This value is the result of multiplying the equivalent
-- components of the input vectors then adding the components of the
-- resulting vector.
-- NOTE: Use Data.Vector.(*) xs ys, because compiler complains of
--       ambiguity if using xs * ys
sumMultVec : Num a => Vect n a -> Vect n a -> a
sumMultVec xs ys = sum (toList (Data.Vector.(*) xs ys))
```

4.1.5 Haskell: Sized Vectors

Haskell does not feature dependent types, but has enough infrastructure around its type system that we can implement sized-arrays regardless. For completeness we list an example implementation in appendix C.6.

Many programming languages provide the ability to program lists (or arrays) of a particular size. However, this feature is often not easily customizable or user-definable. For example, it may be possible to specify list lengths at compile time but not at runtime.

4.1.6 Python with NumPy library

To complete this case study we now consider an implementation of the calculation using Python and the NumPy library [46] version 2.2.2:

```
c = np.square(c)
ux = np.square(ux)
uy = np.sqrt(np.matmul(c,ux.transpose()))
```

NOTE: `c` and `ux` are used to contain the squares of `c` and `ux`. The formula for `uy` looks wrong written in these terms, and that takes away from the trustworthiness of the code in terms of its correspondence to the underlying statistics. The correct answer is generated, so are such matters worth consideration? This report argues that yes they are worth consideration.

4.2 PARSING

Another activity that is routinely performed in data analysis is data acquisition and parsing (in this case uploading measurement results from a file with a known format). Parsing is often considered a "solved problem" thanks to tools such as Antlr [47], BNFC [48], Bison [49], and more. In day-to-day life, those tools are not necessarily the easiest to deploy, and programmers often roll out their own parsing code instead.

However, parsing is a tedious exercise as it is error prone and hard to debug. Using Haskell, we hope to find a happy medium between heavy-weight tools like Bison and customizable and easy to integrate hand-written code.

The structure we set out to parse looks like this:

```
45
81.33 93.11 'val1'
91.34 55.11 'val2'
...
```

That is, a header and a list of triples separated by spaces. We can write a grammar for it in Backus–Naur form [50–53]:

```
file ::= <header> "\n" <rows>
header ::= <num>
rows ::= <row> "\n" <rows>| ""
row ::= <num> <num> <name>
```

```
name ::= '<str>'
```

However, you will probably agree that defining such syntax, passing it to Bison, generating some source-code, compiling it, and then calling it through some form of foreign function interface (FFI) [54] is quite cumbersome in itself.

Haskell makes use of *parser combinators* to build parsers in a declarative but programmatic way. The above grammar can be parsed with the following program:

```
parseRow :: Parser DataRow
parseRow = DataRow <$> token double
                   <*> token double
                   <*> token (stringBetween '\'')

parseData :: Parser DataFile
parseData =
  DataFile <$> token decimal <*> many parseRow
```

The function `parseRow` is to be read as "To build a DataRow, first parse a double, then a double then a string between single quotes". The second function `parseData` parses the whole file by expecting a decimal number and multiple rows and reads similarly as "To build a DataFile, first parse a decimal, then multiple rows".

The types `Parser DataFile` and `Parser DataRow` capture the fact that those functions are parsers and produce the data requested. This ensures there is no confusion as to what the data is supposed to be. The types themselves are defined as named pairs and triples:

```
data DataFile = DataFile { header :: Int, content :: [DataRow] }

data DataRow = DataRow { val1 :: Double, val2 :: Double, name :: Text }
```

As expected a `DataFile` is a pair of a header and a list of rows, a `DataRow` is a triple of two `Double` and a string.

This example showcases two forms of trustworthiness: legibility and type-safety (the software will, of course, still need to be tested). Thanks to its declarative nature, the code is easy to read and understand, giving the programmer, and her peers, strong confidence that the code is indeed doing what is expected from it. Accompanying this is the type guarantee, as long as the data is parsed into a `DataFile`, the rest of the program need not change, allowing the parser to change while retaining the confidence that the rest of the program works.

4.3 GravCalc2

GravCalc2 [55] is our prototype program to establish the benefits and drawbacks of redeveloping an existing program into one with more safeguards. It is used by the NPL Gas and Particle Metrology Team.

GravCalc2 is a software program designed to calculate the amount fraction and uncertainty of all components in a gravimetrically-prepared gas mixture, using the method outlined in ISO 6142 [56].

The ISO 6142 method determines the amount fraction, in mol/mol, and uncertainty also in mol/mol of each component given knowledge of:

- The mass of each component added to the mixture, and its uncertainty.
- The purity of each component, and the uncertainty of the purity analysis.
- The molar mass of each species, and its uncertainty.

NOTE:

- Component mass is measured in grams, purity in moles per mole and molar mass in grams per mole.

- The components used to create the final mixture are taken from a number of parent cylinders.
- The uncertainty calculation is beyond the scope of this report.

GravCalc2 is written in Visual Basic 6. For this case study the original GravCalc2 functional specification document (in which the mathematical formulas are listed) and a Microsoft Excel spreadsheet (in which some GravCalc2 calculations were implemented), were used to develop the functional programming version. This approach helped ensure the functional version was not influenced by the imperative (Visual Basic) version.

The spreadsheet splits the computation into multiple hidden columns, making changes and checks difficult to perform. Although the flexibility of spreadsheet software allows for rapid prototyping and intuitive handling of raw data, it also allows for mistakes when molar masses and meters per second are added together.

4.3.1 Haskell implementation

The first step is to implement our program in Haskell, our functional programming language of choice. We can summarize the program to write with the following equation:

$$y_k = \frac{\sum_{j=1}^{n} \frac{m_j y_{k,j}}{\sum_{i \in I(:j)} y_{i,j} M_i}}{\sum_{j=1}^{n} \frac{m_j}{\sum_{i \in I(:j)} y_{i,j} M_i}}$$

where:

- y_k is the amount fraction of component k
- m_j is the mass taken from parent cylinder j
- $y_{k,j}$ is the amount fraction of component k in parent cylinder j
- $y_{i,j}$ is the amount fraction of component i in parent cylinder j
- M_i is the molar mass of component i
- $I_{i,j}$ is the indices of those components in parent cylinder j
- n is the number of parent cylinders

In an imperative programming language, this calculation would be achieved by nesting multiple for-loops, one for traversing our outer array, and two to compute each sum.

In Haskell we can achieve the same goal using generic functions such as `map` and `sum` which abstract away the need to iterate through elements and results in the following implementation:

```
gasMixCalc' components cylinderMasses =
  fmap computeRatio components
  where
    computeRatio :: Component a -> (String, a)
    computeRatio (Component name _ y) = (name, ratio (V y) )

    massTimesConcentration :: V a
    massTimesConcentration
        = fmap sum
        $ transposeV
        $ fmap (\c -> fmap (molMass c *)
               (V $ concentrations c))
        $ components

    totalMassRatio :: V a
    totalMassRatio = (cylinderMasses / massTimesConcentration)

    ratio :: V a -> a
    ratio concentrations =
      sum (totalMassRatio * concentrations) / sum totalMassRatio
```

This program makes use of a type `V` for *vectors*, for which we have given implementations of point-wise versions of operations on doubles like `(*) (/)` and `(+)`. It is seen that the overall implementation looks similar to the mathematical specification, albeit with more intermediate names to split the code into readable chunks. The fraction given in the specification can be seen in the `ratio` function and the fact that we have one concentration for each component in our array is given by the fact that we are mapping (with `fmap`) our `components` argument.

In conclusion, This result is arguably even more generic and less imperative than the mathematical description since it even does away with the manual indexing of sums. And the types permit us to be flexible over the representation that we use for the data, as we will see subsequently. Indeed, one might be surprised to see no mention of numerical types such as `Double` or `Int`, but Haskell's *typeclass* features allows us to be generic over the concrete implementation of the data, while still capturing the meaning of the specification.

Whenever the program is called, the caller will supply values of type `Double` or `Int` and the program will then specialise to this implementation. One can approximate the same feature in a language such as Java with *generics* and *interfaces*.

4.3.2 Haskell: Adding dimensional analysis

We can go further by using advanced features from the Haskell type system and add dimension checking of the program using the `units` library [57]. This library makes use of singletons[1] and type families[2] to approximate dependent types. They are used to annotate values with the dimension they represent. This dimension must then match the operations that we are performing on them.

Equipped with this library we can write our existing program using units to ensure our program respects units and dimensions.

[1] A singleton is the name given to a type that has a single unique inhabitant.
For example one could imagine the type "Three" to have only the inhabitant `3` as its only value.

[2] Type families are an advanced feature of the Haskell type system designed to perform computation on types.

```
gasMixCalc components cylinderMasses =
  fmap computeRatio components
  where
    computeRatio :: Component (AmountOfSubstance SI) a
                 -> (String, a)
    computeRatio (Component nm _ y) = (nm, ratio (V y))

    massTimesConcentration :: [AmountOfSubstance SI a]
    massTimesConcentration
        = fmap sum'
        $ transpose
        $ fmap (\c -> fmap (molMass c |*) (concentrations c))
        $ components

    totalMassRatio :: V a
    totalMassRatio = V $ fmap extract
        (zipWith (|/|) cylinderMasses massTimesConcentration)

    ratio :: V a -> a
    ratio concentrations =
      sum (totalMassRatio * concentrations) / sum totalMassRatio
```

The changes are minimal, we replace `(/)` by a dimension-aware operation `zipWith (|/|)`, replace multiplication by a dimension-aware equivalent `(|*)` and use different sum and transpose variants.

The biggest differences are in the types of the values we are manipulating. Where before we were dealing with a dimensionless value `a`, we now see `AmountOfSubstance SI a` which indicates that the value is now semantically attached to a dimension in the SI system.

If we were to make a dimension mistake, we would see a compiler error. Typically, if we use volume instead of molar mass we get the cryptic error:

- ```
  Couldn't match type:
      '[ 'F
           Data.Dimensions.SI.AmountOfSubstance One,
         'F Data.Dimensions.SI.Length ('P ('P ('P 'Zero)))]
      with: '[]
  ```

It requires some familiarity with the library but means that the program expected a dimensionless value, but got one with units mol/m^3. While the quality of error messages leaves much to be desired, the fact that we get an error to begin with is to be lauded since the existing infrastructure is unable to detect such mistakes.

4.3.3 F#

F# [58] is a functional programming language developed by Microsoft that offers some dimensional analysis natively. F# is well suited to Microsoft Windows.

The dimensional analysis of F# inserts itself nicely in existing code, replacing the type `decimal` by `decimal<mol>` indicating that the decimal value is an amount of substance. Similarly, algebraic features of units are properly handled, since dividing two values with the same dimensions results in a dimension-less value. As can attest the function `div` that divides pointwise two vectors

```
let (/!) (left: decimal<'a> array) (right: decimal<'a> array): decimal array =
      Array.map (fun (x, y) -> x / y) (Array.zip left right)
```

This syntax will be familiar to programmers accustomed to generics in Java. Here, the `<'a>` is a reference to a generic unit, and so the function `div` takes two vectors with two identical units and returns a vector of dimensionless values.
```

The rest of the program is very similar to the version seen above.

```
let calc (components : Component array,masses: decimal<mol> array): (string * decimal) array
    let massTimesConcentration: decimal<mol> array
      =  components
      |> Array.map (fun c -> Array.map ((*) c.molMass) c.concentration)
      |> Array.transpose
      |> Array.map Array.sum

    let totalMassRatio: decimal array =
      masses /! massTimesConcentration

    let ratio (concentrations: decimal array): decimal =
        Array.sum (totalMassRatio *! concentrations) / Array.sum totalMassRatio


    let computeRatio (y : Component) : string * decimal =
      (y.name, ratio y.concentration)
    Array.map computeRatio components
```

F# offers the benefit that the dimensional analysis is baked into the compiler, rather than encoded using more general features, the errors are much more precise and actionable, rendering the experience of using dimensional analysis at lot more pleasant. This ergonomical aspect is not to be understated since a safe tool is useless if users find the less safe alternative more pleasant to use.

However, F# does not have any way to talk about vector and matrix dimensions, so while the language is better at catching and diagnosing errors involving mismatched units, it cannot catch errors regarding matrices and vectors of different sizes.

4.3.4 Conclusion

One aspect that we cannot see in those code samples is the process taken in order to obtain the working program.

It is worth noting that writing the Haskell version only took a couple of hours, most of which was spent understanding the Microsoft Excel spreadsheet and the mathematical specification.

Additionally, a lot of time was spent debugging dimension errors due to incorrect assumptions. This highlights the importance of types further since the original specification could not accurately communicate the appropriate semantics for the program.

This gap between code and semantics is particularly visible when dealing with errors in a strongly-typed environment. The Haskell unit library in particular makes no effort to provide a possible fix for errors which renders the debugging of programs time-consuming. In those scenarios, the programmer cannot know if the error is in the original code, in the original specification, in the units used, or in the ported code. Despite this difficulty, it is remarkable that a system such as Haskell, that was not designed to track dimension errors, is able to correctly ensure units are used in a consistent manner across the program at compile-time, something Python is unable to do.

While most users probably will not be programming in that paradigm any time soon, it does not mean the approach is without merit. One way to bridge the gap between existing practice and the benefits of statically checking dimensions, could come from a shared central library of reusable components written in a dimension-checked programming language, and exposing untyped-bindings to languages such as Python, MATLAB or C.

5 What Can Dependent Types Offer?

We now explore the use of dependent types for metrology and data science purposes. As shown in the previous section 4, there is a lot we can achieve with functional programming only, but can we do more with dependent types?

5.1 Motivation

Using Haskell, we demonstrated that functional programming provides a compelling user experience with regards to a certain class of errors. Haskell's type system in particular also enjoys a great deal of flexibility thanks to its very powerful polymorphism and type inference. The Haskell ecosystem also fulfills a lot of use-cases in a great range of areas, from data acquisition, to analysis to visualisation.

However, we require greater infrastructure around proofs and type-driven development than that provided by Haskell.

Arguably, those problems are a far cry from the kind of problems we encounter in every-day data science (syntax errors, runtime errors, unrepeatable results), but it turns out we already have the technology to address them: Dependent types.

With dependent types we can write proofs about our programs that *guarantee* without a shadow of a doubt that it behaves the way we expect and using type-level programming we can further abstract over patterns we see emerge in Haskell, simplifying further existing code and removing sources of errors.

Just like in Haskell, we can write dimension-safe matrix multiplication code. For this calculation we use the `Vect` type, a type of lists that carry their lengths in their type.

```
data Vect: Nat -> Type -> Type where
  Nil : Vect 0 a
  (::) : a -> Vect n a -> Vect (S n) a
```

From this definition, we can define matrices:

```
Matrix : Nat -> Nat -> Type -> Type
Matrix m n a = Vect m (Vect n a)
```

Notice how the matrix definition itself is not a bespoke "type alias", like you can find in languages like Haskell, or Rust [59], but a *function definition* like any other.

From this definition of matrix, we can write the type of `transpose` and `multiply` and see that the size information is correctly propagated through the type.

```
transpose : Matrix m n a -> Matrix n m a
multiply : Matrix m n a -> Matrix n o a -> Matrix m o a
```

This ability to write programs that were previously exclusively available at the type-level using the same functionality as regular programs is what forms the basis for dependently-typed programming.

Using such systems we can write proofs about programs, for example that transposing a matrix twice is equivalent to doing nothing:

```
transposeTwice : (mx : Matrix m n a) -> transpose (transpose mx) = mx
```

5.2 Data Acquisition

We have seen how we can leverage Haskell's declarative libraries to write parsers for data acquisition. We can do better using Idris 2 where, thanks to dependent types, the separation between data description and data acquisition is blurred in a way that enables the programmer to do both at once.

We see this in the `DJSON` [60] library developed by the first author of this report using Idris 2. This library allows users to directly manipulate JavaScript Object Notation (JSON) values without first parsing, all while ensuring proper type-safety. By declaring what field we expect to access in the type we allow safe indexing of record fields and automatic parsing of values.

In the following example, we define the type of calculation data as a record that contains multiple fields

```
CalcFields : List FieldType
CalcFields =
  [ "Voltage/V" :! List Double
  , "Current/A" :! List Double
  , "Operator_ID" :! String
  , "Serial_No_Primary" :! String
  , "Primary_Type" :! String
  , "Primary_Nominal_Value" :! String
  , "Serial_No_Secondary" :! String
  , "Secondary_Type" :! String
  , "Secondary_Nominal_Value" :! String
  , "Balance_Gain" :! Double
  ]
```

We can write a value of this type by building a record and the type system will ensure that each field is provided

```
calcData : Record CalcFields
calcData =
  [ "Voltage/V" :-: [ 0.000115960, 0.00234658, 0.00434667, 0.00356969]
  , "Current/A" :-: [-0.72024757,-0.32035671,-0.32827205, -0.33232034]
  , "Operator_ID" :-: "FGY"
  , "Serial_No_Primary" :-: "3468u"
  , "Primary_Type" :-: "XINR1"
  , "Primary_Nominal_Value" :-: "100R"
  , "Serial_No_Secondary" :-: "3447h"
  , "Secondary_Type" :-: "ZINR1"
  , "Secondary_Nominal_Value" :-: "100R"
  , "Balance_Gain" :-: 1
  ]
```

The type ensures that the fields are appropriately populated. If a field were to be misspelt (such as `Current/A` in the example below) or if a field value were to be populated with the wrong type, the compiler will raise an error catching a breach in the advertised type of the data.

```
failing #"Mismatch between: "Currnet/A" and "Current/A"."#
  calcData : Record CalcFields
  calcData =
    [ "Voltage/V" :-: [ 0.00011596, 0.00234658, 0.00434667, 0.00356969]
    , "Currnet/A" :-: [-0.72024757,-0.32035671,-0.32827205, -0.33232034]
    , ...

failing #"Can't find an implementation for FromString (List Double)"#
  calcData : Record CalcFields
  calcData =
    [ "Voltage/V" :-: " 0.00011596, 0.00234658, 0.00434667, 0.00356969"
    , "Current/A" :-: [-0.72024757,-0.32035671,-0.32827205, -0.33232034]
    , ...
```

We can extract fields out of records using the projection function `get` and give the field name:

```
operatorID : String
operatorID = calcData.get "Operator_ID"
```

Of course, the compiler can detect when a typographic error has been made when obtaining the values of a field before executing the program. In the following code, the error raised indicates that the compiler cannot find a field `Opeartor_ID` for the record `calcData`.

```
failing #"Can't find an implementation for HasField "Opeartor_ID""#
  operatorID : String
  operatorID = calcData.get "Opeartor_ID"
```

The library exposes a generic function to parse any JSON value that matches a given format provided every field of the format can be parsed:

```
parseJSONString : (json : String) -> (schema : List FieldType) ->
    (parsers : All (FromJSON . type) schema) => Maybe (Record schema)
parseJSONString json schema = JSON.parse json >>= parseJSON
```

We can specialise it to parse our calculation data format:

```
parseCalc : String -> Maybe (Record CalcFields)
parseCalc str = parseJSONString str CalcFields
```

In the following listing we show how to parse arbitrary data using our `parseCalc` function. The information given by the type of the record is enough to correctly parse the data correctly and malformed data produces errors when parsed.

```
testParserSuccess : Maybe (Record CalcFields)
testParserSuccess = parseCalc validData
  where
    validData : String
    validData = """
      {"Voltage/V":[1.1596e-4,0.00234658,0.00434667,0.00356969],
       "Current/A":[-0.72024757,-0.32035671,-0.32827205,-0.33232034],
       "Operator_ID":"FGY",
       "Serial_No_Primary":"3468u",
       "Primary_Type":"XINR1",
       "Primary_Nominal_Value":"100R",
       "Serial_No_Secondary":"3447h",
       "Secondary_Type":"ZINR1",
       "Secondary_Nominal_Value":"100R",
       "Balance_Gain":1.0}
      """

testParserFail : Maybe (Record CalcFields)
testParserFail = parseCalc "empty"
```

This familiar yet verified implementation of records brings together two aspects of formal verification: type checking and programming ergonomics.

Type checking ensures that errors due to type mismatch do not occur, and ergonomics ensure that what the intent of the programmer is unambiguously communicated to the computer, without sources of confusion or uncertainty like excessive boilerplate.

5.3 Formal verification

Programming with dependent types provides the necessary infrastructure to write proofs alongside programs.

The proofs we can build are limited to proofs in *constructive logic*. This design space allows to write proofs such as associativity of operations like $+$ on $\mathbb{N}$.

Those kinds of proofs, while useful to have, are not the ones that we expect to find useful in the domain of data science. Dependent types provide another kind of proofs that are closer to what programmers expect and is generally referred to as *correct-by-construction* design.

5.4 Abstracting over representations

One of the benefits of dependent types is that they combine multiple approaches to programming in a way that is either impossible or very difficult in other languages.

In the previous implementation, we have to chose to use different operators for plain numbers and numbers with dimensions. Using dependent types, we can write one single program and handle both plain numbers and numbers that can have a measurement unit attached to them.

We do this by using *graded numbers*, numbers with a piece of metadata attached to them, this metadata is called the *grade*. Graded numbers will follow certain rules, for example the grade of two numbers we add must be the same, and result in a number with the same grade.

This is done so that, if the grade represents units of measurement, then adding two numbers with a measurement unit does not change the unit.

For multiplication of graded numbers, the grade of the arguments can be anything but the resulting grade is the multiplication of the two input grades. Again using our measurement unit intuition, multiplying two "seconds" result in "seconds squared", and that is the behaviour we aim to reproduce.

For division, the two numbers can again have two arbitrary grades, but the resulting number will have is grade be given by the division of grades. Such that if we divide a length by a duration we obtain a "metre per second" grade.

Unfortunately, we cannot achieve this level of abstraction using the existing `Num` interface. This is because given a number type `a`, the `Num` exposes the following functions:

```
(+) :  a -> a -> a

(*) :  a -> a -> a

(/) :  a -> a -> a
```

Those functions do not have the appropriate type for explaining how the grade propagates differently between addition and multiplication, therefore, we need a new interface for graded numbers. It will have the following methods:

```
(+~) :  f u a -> f u a -> f u a

(*~) :  f u1 a -> f u1 a -> f (u1 `mul` u2) a

(/~) :  f u1 a -> f u2 a -> f (u1 `div` u2) a
```

This interface is graded over a grade `u` and relies on operations `mul` and `div`. This suggests the grade we need must be a *group* with two operations that are inverses of each other. We define this in Idris 2 using the `VerifiedGroup` interface. This interface contains the multiplication and the *inverse* operation, such that multiplying an element by its inverse results in the neutral element.

```
interface VerifiedGroup (0 a : Type) where
  constructor MkVerifiedGroup
  m : a -> a -> a
  u : a
  inv : a -> a
  m_u : (x : a) -> x `m` u = x
  u_m : (x : a) -> u `m` x = x
  m_m : (x, y, z : a) -> x `m` (y `m` z) = (x `m` y) `m` z
  m_i : (x : a) -> x `m` inv x = u
  i_m : (x : a) -> inv x `m` x = u
  d_inv : (x, y : a) -> inv (x `m` y) = inv x `m` inv y
  i_i : (x : a) -> inv (inv x) = x
```

The interface contains the necessary proofs that go along with the group axioms, we define `div` as `div` x $y \mapsto$ `m` x `inv`(y) , and the proof that dividing a number by itself results in the neutral element is given by the proof `m_i`.

Is it left to explain precisely how this helps with encoding dimensions. If we list all the SI dimensions, see section 2.3.3 of [40], we obtain a vector:

[time, length, mass, current, temperature, amount of substance, luminosity]

Using this vector, we can imagine attaching an integer to each dimension corresponding to the *power* to which this dimension is raised.

A dimensionless value would therefore have an associated vector `[0,0,0,0,0,0,0]`.

A value measured in seconds would an associated dimension vector `[1,0,0,0,0,0,0]`.

Division is represented by a negative power, so a measurement in metres per second squared would have an associated dimension vector `[-2,1,0,0,0,0,0]`.

It is those vectors we are going to use as the *grade* for our numbers.

The interface for graded numbers follows the list we gave, we define a neutral element called `zero`, and three operations for graded addition, graded multiplication, and graded division.

```
interface  (vg : VerifiedGroup grade) =>
  GradedNum grade (0 f : grade -> Type) | f where
  constructor MkGradedNum

  zero : {0 g : grade} -> f g

  -- The grade needs to match for both arguments of the addition
  (+~) : {0 g1 : grade} ->     f g1 -> f g1 -> f g1

  -- For multiplication, the grade is combined using the group operation
  (*~) : {0 g1, g2 : grade} -> f g1 -> f g2 -> f (g1 `m` g2)

  -- For division, the grade is combined using the inverse operation
  (/~) : {0 g1, g2 : grade} -> f g1 -> f g2 -> f (g1 `div` g2)
```

Using this interface, we can update our concentration calculation to use graded numbers instead of plain numbers and it will be able to keep track and propagate our units correctly. The code for the graded version of the calculation looks the same but the type now requires a graded number instead of plain numbers.

```
gasMixCalc : {0 g : Type} -> {0 num : g -> Type} ->
             (gn : GradedNum g num) =>
             {0 gr : g} ->
             {sampleCount, massCount : Nat} ->
             Vect sampleCount (Component massCount (num gr) (num u)) ->
             Vect massCount (num gr) ->
             Vect sampleCount (String, num u )
```

We can now share the same code for programs with units and without.

5.4.1 Plain numbers in a graded interface

The above example does not actually address the promise that we can use the same program with multiple types. It was claimed that we could use plain numbers, but in the end we still had to rewrite our code to handle grades. So how can we recover the original ability to work with plain `Double` and lists?

For this we use the following insight: a plain `Double` is the same as a graded `Double` where the grade is *trivial*. Indeed, a trivial grade that carries no information does not allow us to distinguish between multiple `Double` and so is the same as using plain double. We implement this idea by providing an instance of graded numbers and choosing the `Unit` type as the grade:

```
[gnum] VerifiedGroup gr => Fractional a => GradedNum Unit (\x : gr => a) where
  zero = 0
  (+~) a b = a + b
  (*~) a b = a * b
  (/~) a b = a / b
```

As you can see, graded addition, multiplication and division is achieved by using regular addition multiplication and division.

With this interface defined, we can simply replace the type of graded quantities by doubles, and the program typechecks and runs just fine:

```
gasMixCalc @{gnum} {gr = ()} testComponents testCylinderMasses
```

5.4.2 Graded numbers with units

As previously explained, to represent units, we chose a canonical ordering of dimensions, and use a vector of integers to represent the power applied to each dimension. This vector forms a group using the underlying group-property of integer with addition to combine numbers and subtraction as the inverse operation. We build an instance of `VerifiedGroup` for vectors and use it to instantiate our calculation. We also need to provide a grade for our value that represents a single mole. We do this by defining the grade `SI` as a vector of integers 7 integers, one for each dimension.

```
SI : Type
-- Length, amount, mass, time, current, temperature, luminosity
SI = Vect 7 Zed

-- The grade for a mole
Mole : SI
Mole = [0,1,0, 0, 0, 0, 0]

gasMixCalc @{gnum} {gr = Mole} testComponents testCylinderMasses
```

The surprise is not in the result of the comuptation, but in the ability for the same program to typecheck and ensure that dimensions are respected while remaining compatible with plain dimension-less values.

Because both programs use `Double` as the numerical representation, even if the grade is different, the output is the same:

```
("D4siloxane", 3.0538938292634736e-8)
("D5siloxane", 4.898188716595061e-6)
("D3siloxane", 1.4061866312654097e-9)
("D6siloxane", 3.13138244265386e-8)
("C10unknown", 4.634035401883751e-11)
("Ar", 4.999975192529968e-7)
("CO", 2.999985115517981e-10)
("O2", 4.999975192529968e-9)
("CxHy", 4.999975192529968e-9)
("H2O", 4.999975192529968e-9)
("N2", 0.999994520208565)
("NO", 4.999975192529969e-10)
("SO2", 4.999975192529969e-10)
("methane", 9.999950385059938e-10)
("H2", 9.999950385059938e-10)
```

This confirms that the computational content of the program is the same although the semantics for values are different.

5.5 Stencil-based computation

Another important aspect of evaluating dependent types is the ergonomical benefits its rich capabilities enable. By blending the distinction between metaprogramming [26] and programming we obtain a language that is highly flexible and lends itself to design libraries that expose a purpose-built domain-specific language that is familiar and expressive for that domain.

As an example, convolution is a common operation performed on image data for image processing, data analysis, and machine learning purposes. Because of its importance, convolutions need to be easy to use, understand and debug.

Using dependent types, we can build a framework for convolution based on stencils in a way that would require metaprogramming in other languages. Avoiding metaprogramming comes with a number of benefits:

- Firstly, the concepts and constructs are the same, ensuring that the user does not need to learn any new tools to understand how to use the library.
- Secondly, and as a consequence of that, the debugging tools remains the same as the ones already used by the programmer, the errors are familiar and the ways to address them are the same. Errors introduced with libraries using metaprogramming are notoriously complex, showing errors in code that was generated rather than code that was written by the user directly.
- Finally, metaprogramming comes at a performance cost in terms of type checking. Avoiding it minimises compilation time, which also helps with debugging and testing loops.

In the most common case, we want to pick a "shape" of data, and perform an operation on the data using that shape. This shape is the *kernel* of the convolution. For practical reasons, the kernel is represented as a square matrix, but in principle there is no reason not to pick some more sophisticated shapes.

A kernel is therefore a matrix applied to a region around a single coordinate in a bigger matrix, typically pixel values for an image in image processing.

This application can be done naively by iterating over every coordinate of the data and applying the kernel to the region centered at that coordinate. This iteration can easily be abstracted away by

making the computation be entirely determined by the kernel when it is multiplied with the region of interest.

This is easily done with existing programming language and infrastructure, we can do better and generalise over not only linear kernels but *stencil computation* where a stencil is used to "cut-out" parts of the data, perform an arbitrary computation, and place the result at the center coordinate.

In this model, kernels are a subset of the more general *stencil-based computation* domain. When we apply a kernel in a convolution, we only ever perform a dot product of a submatrix to the kernel. But stencil-based computation offers a more general form of convolution by defining a general function `Shape -> Value`.

Here we reproduce some part of the paper on stencil-based computation in Haskell by Orchard et al. [61] but in a way that does not require any template metaprogramming or macro.

A first approximation of this exercise would be to implement stencils as lists and kernels as functions matching on those lists. For example, a Laplace kernel, defined using a 5-point stencil, could be implemented with a list of 5 elements and a function matching on those 5 elements. The layout of the list represents the shape of the stencil in 2 dimensions:

```
Cross : Type -> Type
Cross = Vect 5

CrossKer : Cross a -> a
CrossKer [    t,
           l, c, r,
              b     ] = ?program
```

This approach is wholly unsatisfactory because the same program could be written on a single line `CrossKer [t, l, c, r, b] = ?program` entirely loosing the shape information that makes the stencil so recognisable.

We therefore need additional typing to ensure that the shape is rendered properly, at least in the most natural way of pattern matching on values.

For this purpose, we introduce a notion of a region of interest with single points of interest. A point of interest is no different from a boolean value indicating if we are interested in using this specific point in the stencil, or not. The *region* of interest is then a n-dimensional matrix of those points of interest. For example, the Laplace Kernel can be characterised by the following region of interest:

```
data Interest = Y | N

CrossShape : Matrix 3 3 Interest
CrossShape = [[N, Y, N],
              [Y, Y, Y],
              [N, Y, N]]
```

where `Y` represents interest in the point and `N` says that the point is dropped.

This example explicitly makes use of a 2D matrix for the region of interest, and therefore, we can use a 2D *predicate* on matrices to generate the appropriate patten for it.

```
I : Type -> Interest -> Type
I x Y = x
I x N = Unit

Region2D : {0 w, h: Nat} -> Matrix w h Interest -> Type -> Type
Region2D xs ty = All (All (I ty)) xs
```

This type is then used to define the kernel of the convolution as a function `Region2D kernel a -> a`. Here is an example:

```
crossComputation : Region CrossShape a -> a
crossComputation t@[[(), x1, ()],
                    [x2, x3, x4],
                    [(), x5, ()]] = ?programCross
```

As you can see, we can only match on the values above and below the center point `x3`. Attempting to match on the corners results in a *Unit* value carrying no data. In this way, there is no manner in which to misuse or incorrectly use indices that were not intended by the region of interest.

Using the same facilities we can define all sorts of regions of interest, for example here is one that captures a center point and its corners:

```
XShape : Matrix 3 3 Interest
XShape = [[Y, N, Y],
          [N, Y, N],
          [Y, N, Y]]
```

Or one that ignores the center point and only captures the surroundings:

```
OShape : Matrix 3 3 Interest
OShape = [[Y, Y, Y],
          [Y, N, Y],
          [Y, Y, Y]]
```

With those different shapes we can define non-linear stencil computation in such a way that the stencil gives us precisely the information we need, and no more. Such that the types drive the computation in a safe and easy to understand way. What is more the visual nature of the stencil is reproduced in code allowing readers to readily understand what stencil is applied.

```
crossShaped : XShape Int -> Int
crossShaped [[x , () ,v ],
             [(),  w, ()],
             [z,  (), a]] = ?here
```

Inspecting the hole `here` shows that we only have access to the variables marked by `XShape`.

```
   x : Int
   v : Int
   w : Int
   z : Int
   a : Int
------------------------------
here : Int
```

With those variables in scope we can implement our convolution operations without any ambiguity related to the shape used.

```
crossShaped : "X" Int -> Int
crossShaped [[x , () ,v ],
             [(),  w, ()],
             [z,  (), a]] = x + v + z + w + 5 * a
```

6 CLOSING REMARKS

The practice of functional programming could be very helpful in bridging the gap between specification and implementation.

Advanced type system features are in need of further development if they are to be applied to the domain of metrology. As a result, the programming experience is much less enjoyable than it could be. Improvements one could make include:

- Build a compiler that has native support for dependent dimensional analysis.
- Attach this compiler to existing workflows such as Jupyter notebooks [62].
- Do not allow unchecked code to run.
- Make the runtime compatible with existing debugging tools.

Our work to date leads us to believe that dependently-typed programming languages, such as Idris 2, are capable of handling metrology-related programs, but the culture around them as statically-compiled theorem-proving capable language clashes with the existing practice around scientific programming (e.g., using Python, MATLAB or Microsoft Excel). The points raised above should help bridge this cultural gap.

6.1 Next Steps

Thinking more immediately, the following steps can be taken:

- Develop further case studies, there is no shortage of candidate software.
- Build an ecosystem of metrology-related libraries that scientists can rely on. Evaluate the trustworthiness of those libraries and use tools to improve them. Advertise the libraries and their level of trustworthiness, to further cultivate a culture of code correctness and reuse.

7 Acknowledgements

This work was undertaken jointly by the National Physical Laboratory's Data Science department as part of Data Science's Tools for Trustworthiness National Measurement System (NMS) project 2023 – 2024 and the Mathematically Structured Programming Group of the University of Strathclyde as part of the Trust[4] project.

Thanks to colleagues in the Gas Metrology and Electromagnetic Measurements group for help with providing case studies: Nick Allen, Lucy Culleton, Nick Fletcher, Colin Porter, Ben Thornton and Scott Wilkins.

Thanks to Peter Harris, Louise Wright and Spencer Thomas for reviewing this report. Thanks also to ex-colleague Michael Stevens for his development of GravCalc2.

REFERENCES

[1] Wikipedia. *Dependent type*. https://en.wikipedia.org/wiki/Dependent_type.

[2] Haskell.org. *What is functional programming?* https://wiki.haskell.org/Functional_programming.

[3] P. Van Roy. *The principal programming paradigms*. https://www.info.ucl.ac.be/~pvr/paradigmsDIAGRAMeng201.pdf.

[4] UK Research and Innovation. *Area of investment and support: Theoretical computer science*. https://www.ukri.org/what-we-do/browse-our-areas-of-investment-and-support/theoretical-computer-science/.

[5] Wikipedia. *Theoretical computer science*. https://en.wikipedia.org/wiki/Theoretical_computer_science.

[6] BSI. *Information technology — Systems trustworthiness*. BS 10754-1:2018. https://knowledge.bsigroup.com/products/information-technology-systems-trustworthiness-governance-and-management-specification?version=standard. BSI Group, 2018.

[7] Wikipedia. *Type theory*. https://en.wikipedia.org/wiki/Type_theory.

[8] Wikipedia. *Type system*. https://en.wikipedia.org/wiki/Type_system.

[9] *Data science - NPL*. https://www.npl.co.uk/data-science.

[10] *Mathematically Structured Programming Group, Computer and Information Sciences, University of Strathclyde*. https://msp.cis.strath.ac.uk/.

[11] NI formerly National Instruments. *NI home page*. Retrieved 15/03/2024 from ni.com. https://www.ni.com.

[12] Python. *Python Software Foundation*. https://www.python.org/.

[13] MathWorks. *MATLAB home page*. https://www.mathworks.com/.

[14] *Alan Turing's work was instrumental in placing NPL at the forefront of computer technology*. https://www.npl.co.uk/famous/alan-turing.

[15] D. Yates. *Turing's Legacy: A History of Computing at the National Physical Laboratory 1945-1995*. The Science Museum, London, 1997. ISBN: 0901805947.

[16] Peter Naur (Editor) et al. including *M. Woodger from NPL*. "Report on the Algorithmic Language ALGOL 60". In: *Communications Of the Association for Computing Machinery* (1960). https://dl.acm.org/doi/10.1145/367236.367262.

[17] B. A. Wichmann et al. "Guidance for the use of the Ada programming language in high integrity systems". In: *ACM SIGAda Ada Letters* XVIII (1998). Retrieved 09/03/2024 from ACM. https://doi.org/10.1145/290214.290222.

[18] R. Milne and C. Strachey. *A theory of programming language semantics*. Chapman and Hall, 1976. ISBN: 0412142600.

[19] P. Landin. "The next 700 programming languages". In: *Communications of the ACM* 9 (1966). https://dl.acm.org/doi/10.1145/365230.365257.

[20] Common Lisp Foundation. *Welcome to Common-Lisp.net*. https://common-lisp.net/.

[21] S. N. Walck. "Learn Physics by Programming in Haskell". In: *EPTCS* 170 (2014). https://doi.org/10.48550/arXiv.1412.4880, pp. 442–455.

[22] S. N. Walck. "Learn Quantum Mechanics with Haskell". In: *EPTCS* 230 (2016). https://doi.org/10.48550/arXiv.1611.09471, pp. 31–46.

[23] S. N. Walck. *Learn Physics with Functional Programming: A Hands-on Guide to Exploring Physics with Haskell*. No Starch Press, 2023. ISBN: 9781718501669.

[24] G. Hutton. *Programming in Haskell, 2nd Edition*. Cambridge University Press, 2016. ISBN: 9781316626221.

[25] Wikipedia. *Boilerplate code*. https://en.wikipedia.org/wiki/Boilerplate_code.

[26] Wikipedia. *Metaprogramming*. https://en.wikipedia.org/wiki/Metaprogramming.

[27] Haskell.org. *Haskell. An advanced, purely functional programming language*. https://www.haskell.org/.

[28] The Idris Community. *Documentation for the Idris 2 Language*. https://idris2.readthedocs.io/en/latest/.

[29] Wikipedia. *Formal system*. https://en.wikipedia.org/wiki/Formal_system.

[30] Euclid translated with introduction by Sir Thomas L. Heath. *Euclid The Thirteen Books of the Elements*. Dover Publications Inc., 1956. ISBN: 9780486600888.

[31] A. N. Whitehead and B. Russel. *Principia Mathematica*. Cambridge University Press, 1913. ISBN: 1603861823.

[32] Wolfram Mathworld. *Zermelo-Fraenkel Set Theory*. https://mathworld.wolfram.com/Zermelo-FraenkelSetTheory.html.

[33] D. Freidman and D. Christiansen. *The Little Typer*. https://thelittletyper.com/. MIT Press, 2018. ISBN: 9780262536431.

[34] A. Klev. "A Comparison of Type Theory with Set Theory". In: *Reflections on the Foundations of Mathematics Univalent Foundations, Set Theory and General Thoughts*. Ed. by D. Kant S. Centrone and D. Sarikaya. https://doi.org/10.1007/978-3-030-15655-8_12. Cham: Springer International Publishing, 2019, pp. 271–292. ISBN: 9783030156558.

[35] E. Snapper. "The three crises in Mathematics: Logicism, Formalism and Intuitionism". In: *Mathematics magazine* (1979). https://www.jstor.org/stable/2689412.

[36] A. Kennedy. *Programming languages and dimensions*. Tech. rep. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-391.pdf. University of Cambridge, 1996.

[37] C. McBride and F. Nordvall Forsberg. *Dimensionally correct by construction: Type systems for programs | BCS FACS SG*. https://www.youtube.com/watch?v=DVDvIoz9vE0. Webinar for BCS introducing concept of dependent types. 2021.

[38] C. McBride. G. Nakov and F. Nordvall Forsberg. "Measuring with confidence: leveraging expressive type systems for correct-by-construction software". In: *Acta IMEKO* 12 (2023). https://acta.imeko.org/index.php/acta-imeko/article/view/1412.

[39] Conor McBride et al. "LabMate: A prospectus for types for MATLAB". In: *Measurement: Sensors* (2025), p. 101460. ISSN: 2665-9174. URL: https://www.sciencedirect.com/science/article/pii/S2665917424004367.

[40] BIPM. *SI Brochure: The International System of Units (SI), 9th edition*. https://www.bipm.org/en/publications/si-brochure. BIPM, 2019, revised 2022. ISBN: 9789282222720.

[41] S. Bell. *Measurement Good Practice Guide No. 11 (Issue 2) A Beginner's Guide to Uncertainty of Measurements*. Tech. rep. https://www.npl.co.uk/gpgs/beginners-guide-measurement-uncertainty-gpg11. National Physical Laboratory UK, 2001.

[42] M.G. Cox and P.M. Harris. *Software Support for Metrology Best Practice Guide No. 6 Uncertainty Evaluation*. Tech. rep. https://eprintspublications.npl.co.uk/4655/1/MS6.pdf. National Physical Laboratory UK, 2010.

[43] BIPM et al. *JCGM GUM-1:2023 Guide to the expression of uncertainty in measurement — Part 1: Introduction*. Joint Committee for Guides in Metrology, JCGM GUM-1:2023. URL: https://www.bipm.org/en/committees/jc/jcgm/publications.

[44] matrix-0.3.6.3: A native implementation of matrix operations. *Haskell matrix operations*. https://hackage.haskell.org/package/matrix-0.3.6.3/docs/Data-Matrix.html.

[45] Wikipedia. *First class citizen*. https://en.wikipedia.org/wiki/First-class_citizen.

[46] NumPy. *NumPy Library*. https://numpy.org/.

[47] Terence Parr. *ANTLR (ANother Tool for Language Recognition)*. https://www.antlr.org.

[48] Andreas Abel et al. *BNFC: A compiler front-end generator*. https://hackage.haskell.org/package/BNFC.

[49] Free Software Foundation Inc. *GNU Bison*. https://www.gnu.org/software/bison/.

[50] Wikipedia. *Backus–Naur form*. https://en.wikipedia.org/wiki/Backus-Naur_form.

[51] ISO. *Information technology - Syntactic metalanguage - Extended BNF*. ISO/IEC 47977:1996. International Organization for Standardization, 1996.

[52] J. Wells D. Quinlan and F. Kamareddine. "BNF-Style Notation as It Is Actually Used". In: *Intelligent Computer Mathematics*. Ed. by C. Kaliszyk. E. Brady. A. Kohlhase. and C. Sacerdoti Coen. Springer International Publishing, 2019, pp. 187–204. ISBN: 9783030232504.

[53] J. Backus. "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference". In: *IFIP Congress 1959* (1959). https://www.softwarepreservation.org/projects/ALGOL/paper/Backus-Syntax_and_Semantics_of_Proposed_IAL.pdf.

[54] Wikipedia. *Foreign function interface*. https://en.wikipedia.org/wiki/Foreign_function_interface.

[55] *Gas metrology software*. https://www.npl.co.uk/products-services/gas/gas-metrology-software.

[56] ISO. *Gas analysis. Preparation of calibration gas mixtures. Part 1: Gravimetric method for Class I mixture*. ISO 6142-1:2015. International Organization for Standardization, 2015.

[57] units: A domain-specific type system for dimensional analysis. *Haskell units*. https://hackage.haskell.org/package/units.

[58] F# Software Foundation. *F# Software Foundation*. https://fsharp.org/.

[59] Rust Team. *Rust Programming Language*. https://www.rust-lang.org/.

[60] Andre Videla. *DJson. A library for handling well typed json objects using extensible records*. https://gitlab.com/avidela/djson/.

[61] D. Orchard and A. Mycroft. "Efficient and Correct Stencil Computation via Pattern Matching and Static Typing". In: *Proceedings IFIP Working Conference on Domain-Specific Languages, DSL 2011* (2011). https://doi.org/10.4204/EPTCS.66.4, pp. 68–92.

[62] Project Jupyter. *Jupyter Notebook Documentation*. https://jupyter-notebook.readthedocs.io/en/latest/.

[63] D. Turner. *Some history of functional programming languages. BCS Peter Landin Semantics Seminar 2019*. https://www.youtube.com/watch?v=ezFZIPuSQU8.

[64] S. Nicholas. *David Turner obituary*. https://www.theguardian.com/technology/2023/nov/24/david-turner-obituary.

[65] D. Turner. *Some History of Functional Programming Languages*. https://www.cs.kent.ac.uk/people/staff/dat/tfp12/tfp12.pdf. An invited lecture given at St Andrews University, 12 June 2012.

[66] N. Wu. *The Next 700 Domain Specific Languages. BCS Peter Landin Semantics Seminar 2022*. https://www.youtube.com/watch?v=y0EjeY02joI.

[67] Stephen Cole Kleene. *Mathematical Logic*. Dover edition first published in 2002. Dover, 1967. ISBN: 0486425339.

[68] G. Michaelson. *An introduction to functional programming through Lambda Calculus*. Dover Publications, 2011. ISBN: 0486478831.

[69] Anon. *A Brief History of Functional Programming*. https://www.cse.psu.edu/~gxt29/historyOfFP/historyOfFP.html.

[70] Wikipedia. *Entscheidungsproblem*. https://en.wikipedia.org/wiki/Entscheidungsproblem.

[71] A. M. Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* 42 (1937). https://turingarchive.kings.cam.ac.uk/publications-lectures-and-talks-amtb/amt-b-12.

[72] J. McCarthy. *History of Lisp*. http://jmc.stanford.edu/articles/lisp.html. 1979.

[73] Oldest.org. *10 Oldest Programming Languages Still in Use*. https://www.oldest.org/technology/programming-languages/.

[74] R. Milner. “How ML evolved”. In: *ML/Hope/LCF Newsletter* 1 (1982). https://www.pure.ed.ac.uk/ws/portalfiles/portal/17084823/Milner_R_1982_How_ML_Evolved.pdf, pp. 25–34.

[75] Research Software Limited. *Miranda. A non-strict polyporphic functional language*. https://www.cs.kent.ac.uk/people/staff/dat/miranda/.

[76] P. Hudak et al. “A history of Haskell: being lazy with class”. In: *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages* (2007). https://doi.org/10.1145/1238844.1238856 also retrived 10/03/2024 from Microsoft https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/history.pdf, 12-1–12–55.

[77] Haskell wiki. *Haskell in industry*. Retrieved 19/02/2024 from wiki.haskell.org. https://wiki.haskell.org/Haskell_in_industry.

[78] Blocksplained. *Haskell: A powerful language for blockchain development and beyond*. https://adapulse.io/haskell-a-powerful-language-for-blockchain-development-and-beyond/.

[79] Monday Morning Haskell. *Machine Learning in Haskell*. https://mmhaskell.com/machine-learning.

[80] The Agda Team. *Welcome to Agda’s documentation!* https://agda.readthedocs.io/.

[81] P. Dybjer A. Bove and U. Norell. “A Brief Overview of Agda – A Functional Language with Dependent Types”. In: *Conference: Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Ed. by S. Berghofer et al. https://www.researchgate.net/publication/221302211. Springer, 2009, pp. 73–78. ISBN: 364203358X.

[82] Simon Peyton Jones. “Associated types with class”. In: *POPL ’05: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*. ACM Press, Jan. 2005, pp. 1–13. ISBN: 1-58113-830-X. URL: https://www.microsoft.com/en-us/research/publication/associated-types-with-class/.

[83] X. Leroy. *Objective Caml 1.00*. https://inbox.vuxu.org/caml-list/199605091427.QAA28631@pauillac.inria.fr/.

[84] Microsoft. *TypeScript Documentation*. https://www.typescriptlang.org/.

[85] Stephen Wolfram. *What is a Turing Machine?* https://www.wolframscience.com/prizes/tm23/turingmachine.html.

[86] Wikipedia. *Turing completeness*. https://en.wikipedia.org/wiki/Turing_completeness.

[87] Wolfram Mathworld. *Church-Turing Thesis*. https://mathworld.wolfram.com/Church-TuringThesis.html.

[88] BCS (formerly the British Computer Society). *FACS (Formal Aspects of Computing Science) group*. https://www.bcs.org/membership-and-registrations/member-communities/facs-formal-aspects-of-computing-science-group/.

[89] A. Harry. *Formal Methods Fact File VDM and Z*. John Wiley, 1966. ISBN: 0471940062.

[90] B. Sitnikovski. *Introduction to dependent types with Idris*. Apress Media LLC, 2023. ISBN: 9781484292587.

[91] A. Helwer. *The missing prelude to The Little Typer’s trickiest Chapter*. https://ahelwer.ca/post/2022-10-13-little-typer-ch9/.

[92] E. Brady. *Type-Driven Development with Idris*. https://www.manning.com/books/type-driven-development-with-idris. Manning Publications, 2017. ISBN: 9781617293023.

A FUNCTIONAL PROGRAMMING: A VERY BRIEF HISTORY

The following is a very brief timeline listing some (but far from all, due to the scope of the report) highlights in the history of functional programming. References are provided for further reading.

For a thorough description of the origins of functional programming see the BCS 2019 Peter Landin Semantics Seminar [63] delivered by David Turner [64]. Also see [65].

The BCS 2022 Peter Landin Semantics Seminar [66], delivered by Nicholas Wu, provides further details of connections between the work of Church and Turing. For more technical details see §41 of [67].

- *Late 1920s Alonzo Church:* Lambda Calculus [68]:
  - Common ancestor of all functional programming languages [69]
  - *1936:* The decision problem (Entscheidungsproblem) [70].
    - Alan Turing used *a-machines* ("automatic machines", renamed *Turing machines* by Church) to prove the Entscheidungsproblem cannot be solved [71]. But Church had already arrived at same result using lambda calculus.
- *1958 John McCarthy:* List processing (LISP) [20, 72]:
  - Closely linked to early developments in artificial intelligence.
  - Second oldest high-level programming language still in use today [73].
- *1966 Peter Landin:* If you see what I mean (ISWIM) [19]:
  - A proposal, a collection of ideas.
  - Never fully implemented, perhaps never intended to be.
  - Contains features used in modern-day languages (e.g., offside rule in Python).
- *1973 Robin Milner:* Meta Language (ML) [74]:
  - *NOTE:* In functional programming, ML can standard for Meta Language, as well as machine learning.
  - Ancestor of F# language.
- *1985 David Turner:* Miranda [75]
  - Lazy, purely functional, no side effects (e.g., no setting of global variables).
  - Originally proprietary, now open-source.
  - *Strong influence* on languages that followed, such as Haskell.
- *1990 A Committee:* Haskell [27]
  - Open-source. Proposed during 1987 Functional Programming Languages and Computer Architecture (FPCA '87) in Portland, Oregon [76].
  - Users include Facebook, Google, Microsoft (a key sponsor) [77].
  - Used in blockchain [78] and machine learning [79] applications.
- *1999 Ulf Norell, Catarina Coquand* Agda [80].
  - *Dependently typed* language [81].
- *2007 Edwin Brady:* Idris [28]
  - *Dependently typed* language.
- *2005 M. Chakravarty, G. Keller, and S. Peyton Jones:* Associated Type Synonyms [82]

  - Paved the way for Type families.
  - Type families were implemented in 2007 in GHC 6.8.1
  - Kickstarted the work on dependent haskell.
- *1996 X, Leroy:* OCaml [83]
  - Combines object-oriented patterns with ML
  - Offers a wide design space with
  - Shows inspiration from ML, Haskell, OCaml.
- *2012 G. Hoare:* Rust [59]
  - Industrial-strength compiler that has no need for runtime garbage collection.
  - Revitalized the domain of type-safe low-level programming.
  - Shows inspiration from ML, Haskell, OCaml.
- *2012 Microsoft:* TypeScript [84]
  - Allows gradual typing of JavaScript code
  - Showcases a restricted amount of dependent typing

A.1 Turing Machines: Further Notes

Turing machines are described by Stephen Wolfram [85] as:

> The original idealized model of a computer. Equivalent to modern electronic computers at a certain theoretical level.

Further useful definitions are provided in [86]:

> A programming language is said to be Turing-complete or computationally universal if it can be used to simulate any Turing machine.
>
> No physical system can have infinite memory, but if the limitation of finite memory is ignored, most programming languages are otherwise Turing-complete.

Lambda calculus is Turing complete.

To quote Wolfram Mathworld [87]:

> The *Church-Turing thesis* says that any real-world computation can be translated into an equivalent computation involving a Turing machine.

B Introductory Reference List

The following references proved useful to the second author for becoming familiar with the subject covered in this report. It is not presented as a *definitive* reference list. It is intended to be a collection of introductory references, useful for non-specialists to familiarise themselves with subjects covered in this report.

B.1 Background Information

- Webinars recorded for the BCS FACS (Formal Aspects of Computing Science) group [88]:
  - *The Next 700 Domain Specific Languages. BCS Peter Landin Semantics Seminar 2022 by Nicholas Wu*: https://www.youtube.com/watch?v=y0EjeY02joI
  - *NPL's Experience with Formal Aspects by Keith Lines*: https://www.youtube.com/watch?v=CjKIxwv-z6U
- *Turing's Legacy: A History of Computing at the National Physical Laboratory 1945-1995 by David Yates* [15].

B.2 Logic

- *Formal Methods Fact File VDM and Z by Andrew Harry* [89]
- *Mathematical Logic by Stephen Cole Kleene* [67]

B.3 Functional Programming and Lambda Calculus

- Webinar recorded for the BCS:
  - *Some history of functional programming languages. BCS Peter Landin Semantics Seminar 2019 by David Turner*: https://www.youtube.com/watch?v=ezFZIPuSQU8
- The Haskell website provides an abundance of reference material: https://www.haskell.org/documentation/
- *Programming in Haskell by Graham Hutton* [24]. A link to a related course can be found on the Haskell website link above.
- *An introduction to functional programming through Lambda Calculus by Greg Michaelson* [68].
- *Learn Physics with Functional Programming: A Hands-on Guide to Exploring Physics with Haskell by Scott N. Walck* [23] will surely be of interest to most readers of this report!

B.4 Dependent Types

- Webinars recorded for the BCS:
  - *Dimensionally correct by construction: Type systems for programs by Conor McBride and Fredrik Nordvall Forsberg*: https://www.youtube.com/watch?v=DVDvIoz9vE0
  - *Dependent types for practical use by André Videla*: https://www.youtube.com/watch?v=txJiuDM2gUM
- Webinar recorded for the Strange Loop Conference 2018:
  - *A Little Taste of Dependent Types by David Christiansen* https://www.youtube.com/watch?v=VxINoKFm-S4
- *Introduction to dependent types with Idris by Boro Sitnikovski* [90]

- *The Little Typer by Daniel P. Friedman and David Thrane Christiansen, foreword by Robert Harper and afterword by Conor McBride* [33]
- *The missing prelude to The Little Typer's trickiest chapter by Andrew Helwer* [91]
- *Type-Driven Development with Idris by Edwin Brady* [92]
- *Documentation for the Idris 2 Language by The Idris Community* [28]

C CODE LISTINGS

C.1 END-USER LICENCE AGREEMENT

All code listed in this report is subject to the following end-user license agreement.

SOFTWARE USER LICENCE AGREEMENT

This licence agreement (Licence) is a legal agreement between you (Licensee or you) and NPL Management Limited of Hampton Road, Teddington, Middlesex, TW11 0LW, United Kingdom (Licensor, us or we) for:

- Code listed in this report (Software); and
- This report (Documents).

We license use of the Software and Documents to you on the basis of this Licence. We do not sell the Software or Documents to you. We remain the owners of the Software and Documents at all times.

OPERATING SYSTEM REQUIREMENTS: THIS SOFTWARE REQUIRES A COMPUTER WITH A MINIMUM OF 200 MB OF DISK SPACE AND 64 MB OF MEMORY.

You should print a copy of this Licence for future reference.

1. GRANT AND SCOPE OF LICENCE

1.1 In consideration of payment by you of the agreed licence fee and you agreeing to abide by the terms of this Licence, we grant to you a non-exclusive, non-transferable licence to use the Software and the Documents on the terms of this Licence.

1.2 You may:

- (a) download, install and use the Software for your internal business purposes only:
  - (i) on one central processing unit (CPU) if the Licence is a single-user licence or the Software is for single use; or
  - (ii) if the Licence is a multi-user or network licence, by the number of concurrent users agreed;
- (b) provided you comply with the provisions in Condition 2, make up to one copy of the Software for back-up purposes only;
- (c) might receive and use any free supplementary software code or update of the Software incorporating “patches” and corrections of errors as may be provided by us from time to time; and
- (d) use any Documents in support of the use permitted under Condition 1.2 and make up to one copy of the Documents as are reasonably necessary for its lawful use.

2. RESTRICTIONS

2.1 Except as expressly set out in this Licence or as permitted by any local law, you undertake:

- (a) not to copy the Software or Documents except where such copying is incidental to normal use of the Software, or where it is necessary for the purpose of back-up or operational security;
- (b) not to rent, lease, sub-license, loan, translate, merge, adapt, vary or modify the Software or Documents;

- (c) not to make alterations to, or modifications of, the whole or any part of the Software, nor permit the Software or any part of it to be combined with, or become incorporated in, any other programs;
- (d) not to disassemble, decompile, reverse-engineer or create derivative works based on the whole or any part of the Software nor attempt to do any such thing except to the extent that (by virtue of section 296A of the Copyright, Designs and Patents Act 1988) such actions cannot be prohibited because they are essential for the purpose of achieving inter-operability of the Software with another software program, and provided that the information obtained by you during such activities:
  - (i) is used only for the purpose of achieving inter-operability of the Software with another software program; and
  - (ii) is not unnecessarily disclosed or communicated without our prior written consent to any third party; and
  - (iii) is not used to create any software which is substantially similar to the Software;
- (e) to keep all copies of the Software secure and to maintain accurate and up-to-date records of the number and locations of all copies of the Software;
- (f) to supervise and control use of the Software and ensure that the Software is used by your employees and representatives in accordance with the terms of this Licence;
- (g) to include our copyright notice on all entire and partial copies you make of the Software on any medium;
- (h) not to provide or otherwise make available the Software in whole or in part (including but not limited to program listings, object and source program listings, object code and source code), in any form to any person without prior written consent from us;
  - (i) to comply with all applicable technology control or export laws and regulations; and
  - (j) not use the Software via any communications network or by means of remote access.

3. INTELLECTUAL PROPERTY RIGHTS

3.1 You acknowledge that all intellectual property rights in the Software and the Documents anywhere in the world belong to us, that rights in the Software are licensed (not sold) to you, and that you have no rights in, or to, the Software or the Documents other than the right to use them in accordance with the terms of this Licence.

3.2 You acknowledge that you have no right to have access to the Software in source code form.

4. LIMITED WARRANTY

THE LICENSED SOFTWARE AND DOCUMENTATION ARE PROVIDED ON AN “AS IS” BASIS. NPL MAKES NO WARRANTIES OR REPRESENTATIONS RELATING TO THE LICENSED SOFTWARE AND DOCUMENTATION, EXPRESS OR IMPLIED, STATUTORY OR OTHERWISE, AND EXPRESSLY EXCLUDES THE WARRANTY OF NON-INFRINGEMENT OF THIRD-PARTY RIGHTS, FITNESS FOR A PARTICULAR PURPOSE OR MERCHANTABILITY. NPL DOES NOT WARRANT THAT THE LICENSED SOFTWARE AND DOCUMENTATION WILL SATISFY THE LICENSEE’S REQUIREMENTS, THAT THE LICENSED SOFTWARE AND DOCUMENTATION IS WITHOUT DEFECT OR ERROR OR THAT OPERATION OF THE LICENSED SOFTWARE WILL BE UNINTERRUPTED.

5. LIMITATION OF LIABILITY

5.1 You acknowledge that the Software has not been developed to meet your individual requirements, including any particular cybersecurity requirements you might be subject to under law

or otherwise, and that it is therefore your responsibility to ensure that the facilities and functions of the Software as described in the Documents meet your requirements.

5.2 We only supply the Software and Documents for internal use by your business, and you agree not to use the Software or Documents for any re-sale purposes.

5.3 We shall not in any circumstances whatever be liable to you, whether in contract, tort (including negligence), breach of statutory duty, or otherwise, arising under or in connection with the Licence for:

- (a) loss of profits, sales, business, or revenue;
- (b) business interruption;
- (c) loss of anticipated savings;
- (d) wasted expenditure;
- (e) loss or corruption of data or information;
- (f) loss of business opportunity, goodwill or reputation; where any of the losses set out in Condition 5.3(a) to Condition 5.3(f) are direct or indirect; or
- (g) any special, indirect or consequential loss, damage, charges or expenses.

5.4 Other than the losses set out in Condition 5.3 (for which we are not liable), our maximum aggregate liability under or in connection with this Licence whether in contract, tort (including negligence) or otherwise, shall in all circumstances be limited to a sum equal to £5,000. This maximum cap does not apply to Condition 5.5.

5.5 Nothing in this Licence shall limit or exclude our liability for:

- (a) death or personal injury resulting from our negligence;
- (b) fraud or fraudulent misrepresentation;
- (c) any other liability that cannot be excluded or limited by English law.

5.6 This Licence sets out the full extent of our obligations and liabilities in respect of the supply of the Software and Documents. Except as expressly stated in this Licence, there are no conditions, warranties, representations or other terms, express or implied, that are binding on us. Any condition, warranty, representation or other term concerning the supply of the Software and Documents which might otherwise be implied into, or incorporated in, this Licence whether by statute, common law or otherwise, is excluded to the fullest extent permitted by law.

6. TERMINATION

6.1 We may terminate this Licence immediately by written notice to you if you commit a material or persistent breach of this Licence which you fail to remedy (if remediable) within 14 days after the service of written notice requiring you to do so.

6.2 On termination for any reason:

- (a) all rights granted to you under this Licence shall cease;
- (b) you must immediately cease all activities authorised by this Licence; and
- (c) you must immediately and permanently delete or remove the Software from all computer equipment in your possession, and immediately destroy or return to us (at our option) all copies of the Software and Documents then in your possession, custody or control and, in the case of destruction, certify to us that you have done so.

7. COMMUNICATIONS BETWEEN US

7.1 We may update the terms of this Licence at any time on notice to you in accordance with this Condition 7. Your continued use of the Software and Documents following the deemed receipt and

service of the notice under Condition 7.3 shall constitute your acceptance to the terms of this Licence, as varied. If you do not wish to accept the terms of the Licence (as varied) you must immediately stop using and accessing the Software and Document on the deemed receipt and service of the notice.

7.2 If we have to contact you, we will do so by email in accordance with your order for the Software.

7.3 Note that any notice:

- (a) given by us to you will be deemed received and properly served 24 hours after it is first posted on our website or 24 hours after an email is sent; and
- (b) given by you to us will be deemed received and properly served 24 hours after an email is sent.

7.4 In proving the service of any notice, it will be sufficient to prove, in the case of posting on our website, that the website was generally accessible to the public for a period of 24 hours after the first posting of the notice and, in the case of an email, that such email was sent to the email address of the recipient given for these purposes.

8. EVENTS OUTSIDE OUR CONTROL

8.1 We will not be liable or responsible for any failure to perform, or delay in performance of, any of our obligations under this Licence that is caused by an Event Outside Our Control. An Event Outside Our Control is defined below in 8.2.

8.2 An Event Outside Our Control means any act or event beyond our reasonable control, including without limitation failure of public or private telecommunications networks.

8.3 If an Event Outside Our Control takes place that affects the performance of our obligations under this Licence:

- (a) our obligations under this Licence will be suspended and the time for performance of our obligations will be extended for the duration of the Event Outside Our Control; and
- (b) we will use our reasonable endeavours to find a solution by which our obligations under this Licence may be performed despite the Event Outside Our Control.

9. HOW WE MAY USE YOUR PERSONAL INFORMATION

Under data protection legislation, we are required to provide you with certain information about who we are, how we process the personal data of those individuals who use the Software and the Documents and for what purposes and those individuals' rights in relation to their personal data and how to exercise them. This information is provided in this link and it is important that you read that information.

10. OTHER IMPORTANT TERMS

10.1 We may transfer our rights and obligations under this Licence to another organisation, but this will not affect your rights or our obligations under this Licence.

10.2 You may only transfer your rights or your obligations under this Licence to another person if we agree in writing.

10.3 This Licence and any document expressly referred to in it constitutes the entire agreement between us and supersedes and extinguishes all previous and contemporaneous agreements, promises, assurances and understandings between us, whether written or oral, relating to its subject matter.

10.4 You acknowledge that in entering into this Licence you do not rely on and shall have no remedies in respect of any statement, representation, assurance or warranty (whether made

innocently or negligently) that is not set out in this Licence or any document expressly referred to in it.

10.5 You agree that you shall have no claim for innocent or negligent misrepresentation or negligent misstatement based on any statement in this Licence or any document expressly referred to in it.

10.6 A waiver of any right or remedy is only effective if given in writing and shall not be deemed a waiver of any subsequent right or remedy.

10.7 A delay or failure to exercise, or the single or partial exercise of, any right or remedy does not waive that or any other right or remedy, nor does it prevent or restrict the further exercise of that or any other right or remedy.

10.8 Each of the conditions of this Licence operates separately. If any court or competent authority decides that any of them are unlawful or unenforceable, the remaining conditions will remain in full force and effect.

10.9 This Licence, its subject matter and its formation (and any non-contractual disputes or claims) are governed by English law. We both irrevocably agree to the exclusive jurisdiction of the courts of England and Wales.

C.2 Law of Propagation of Uncertainty: MATLAB Implementation

C.2.1 File ImplementUncertaintyEvaluationGUM.m

```
% ------------------------------------------------------------------------
% ImplementUncertaintyEvaluationGUM.m
%
% Script to implement uncertainty evaluation according to the
% Guide to the Expression of Uncertainty in Measurement (GUM).
%
% The following assumptions are made:
% - There is one output quantity.
% - There are N input quantities.
% - The input quantities are independent.
% - The output quantity can be expressed as an explicit function of the
%   input quantities: Y = f(X1, ..., XN).
%
% Author:       I M Smith, Data Science department, NPL.
% Created:      26/10/2022.
% Last amended: 18/02/2025. Added further comments and uncommented the
%                           more efficient calculation.
%                           Keith Lines 25/04/2023
%                           Assign c and ux as row vectors.
% ------------------------------------------------------------------------

% Assign number of input quantities.
  N = 4;

% Assign sensitivity coefficients. Assign as row vector.
  c = [0.5, 0.4, 0.3, 0.2];

% Assign uncertainties associated with estimates of input quantities.
% Assign as row vector.
  ux = [0.1, 0.2, 0.3, 0.4];

% Apply law of propagation of uncertainty (LPU) to evaluate standard
% uncertainty associated with estimate of output quantity.
%
% NOTE: The variable uy is first calculated as the variance and then
% overwritten by the standard deviation. So, its ‘‘meaning’’ changes as
% the code is executed. The correct answer is generated, so are such
% matters worth consideration?
  uy = 0;
  for j = 1:N
    uy = uy + c(j)^2*ux(j)^2;
  end % for j
  uy = sqrt(uy);

% This calculation can be written more efficiently, as below.
% The .^2 operator squares each element of the vector.
%
% Transposing the second vector of squared values results in a 1xN vector
% being multiplied by an Nx1 vector, i.e. a 1x1 scalar that consists of
% the sum of the vector elements.
uy_efficient = sqrt((c.^2)*(ux.^2)');
% ------------------------------------------------------------------------
```

C.3 LAW OF PROPAGATION OF UNCERTAINTY: HASKELL WITH HACKAGE

C.3.1 File Main.hs

```
module Main where
-----------------------------------------------------------------------
-- Description: Combine standard uncertanties according to
--              law of propagation of uncertainty from BIPM
--              Guide to the Expression of Uncertainty in Measurement (GUM)
--
--              The following assumptions are made:
--              - There are N input quantities.
--              - There is one output quantity.
--              - The input quantities are independent.
--              - The output quantity can be expressed as an explicit
--                function of the input quantities: Y = f(X1, ..., XN).
--
-- Author:  Keith Lines, Data Science department, NPL.
--          From MATLAB code and comments by Ian Smith,
--                                     Data Science department, NPL.
--
-- Version: 0.1.1
--
-- Amendment History
-- -----------------
-- Version Date        Name           Description
-- ------- ----------  -------------  ---------------------------------
-- 0.1.0   08/02/2025  Keith Lines    INITIAL VERSION
--
-- 0.1.1   18/02/2025  Keith Lines    Modify definition of uy, to make
--                                    more consistent with MATLAB
--                                    implementation.
-----------------------------------------------------------------------

-- Use Hackage matrix-0.3.6.3: A native implementation of matrix operations.
-- See lpu.cabal for further details.
import Data.Matrix

-- Assign sensitivity coefficients. Assign as row vector.
c :: Matrix Double
c = fromLists [[0.5, 0.4, 0.3, 0.2]]

-- Assign uncertainties associated with estimates of input quantities.
-- Assign as row vector.
ux :: Matrix Double
ux = fromLists [[0.1, 0.2, 0.3, 0.4]]

uy :: Matrix Double
uy = fmap sqrt ((fmap (\x -> x*x) c) `multStd` (fmap (\x -> x*x) (transpose ux)))

main :: IO ()
main = do
  putStrLn "Law of propagation of uncertainty (LPU)"
  putStrLn "Haskell example version 0.1.1 with Hackage."
  putStrLn (prettyMatrix uy)
```

C.4 LAW OF PROPAGATION OF UNCERTAINTY: HASKELL WITH VECTORS

C.4.1 File Vector.hs

```
{-# LANGUAGE DerivingStrategies #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
module Data.Vector where
-------------------------------------------------------------------------
-- Description: Create vector type and related functions and operations.
--              Overload power, addition and multiplication operators
--              to create point-wise operations for vectors.
--
-- Author:  André Videla
--
-- Version: 0.4.0
--
-- Amendment History
-- -----------------
-- Version Date        Name           Description
-- ------- ----------  -------------  ---------------------------------
-- 0.1.0   04/11/2022  André Videla   INITIAL VERSION
--
-- 0.2.0   08/02/2025  Keith Lines    Following modifications made:
--                                    1) Removed export list from
--                                       module declaration.
--                                    2) Replaced function 'fn' with
--                                       sumVec
--
-- 0.3.0   08/02/2025  Keith Lines    Removed point-wise square root
--                                    operator as no longer required.
--
-- 0.4.0   19/02/2025  Keith Lines    Replaced sumVec with sumMultVec.
-------------------------------------------------------------------------

-- Point-wise power operation is left associative with the second
-- highest level of binding.
infixl 8 .^

-- First we declare a newtype that will wrap a plain list. We need this for implementing
-- `Num` on vectors
newtype Vec a = Vec { asList :: [a] }
  deriving newtype Functor
  deriving stock Show


-- We need a point-wise power operation on vectors `i` must be some sort of integer
-- for this function to make sense. This restriction is expressed with the `Integral`
-- typeclass.
(.^) :: (Num n, Integral i) => Vec n -> i -> Vec n
(.^) ls i = fmap (^i) ls


-- The Num instance for `Vec` allows us to write `x * y` for point-wise multiplication
-- of two vectors.
instance Num a => Num (Vec a) where
  (+) a b = Vec (zipWith (+) (asList a) (asList b))
```

```
  (*) a b = Vec (zipWith (*) (asList a) (asList b))
  negate = fmap negate
  abs = fmap abs
  signum = fmap signum
  fromInteger = Vec . pure . fromInteger


-- sumMultVec inputs two row vectors and outputs a scalar value.
-- This value is the result of multiplying the equivalent components of the
---input vectors then adding the components of the resulting vector.
sumMultVec :: (Floating i) => Vec i -> Vec i -> i
sumMultVec a b = sum (asList (a * b))


-- This is an example of where Haskell's type system falls short
worksButShouldNot :: Double
worksButShouldNot = (Vec [1,2,3]) `sumMultVec` (Vec [])
```

C.4.2 File Main.hs

```
module Main where
-------------------------------------------------------------------------
-- Description: Combine standard uncertanties according to
--              law of propagation of uncertainty from BIPM
--              Guide to the Expression of Uncertainty in Measurement (GUM)
--
--              The following assumptions are made:
--              - There are N input quantities.
--              - There is one output quantity.
--              - The input quantities are independent.
--              - The output quantity can be expressed as an explicit
--                function of the input quantities: Y = f(X1, ..., XN).
--
-- Author:  Keith Lines, Data Science department, NPL.
--          From MATLAB code and comments by Ian Smith,
--                                     Data Science department, NPL.
--
-- Version: 0.2.0
--
-- Amendment History
-- -----------------
-- Version Date        Name           Description
-- ------- ----------  -------------  ---------------------------------
-- 0.1.0   08/02/2025  Keith Lines    INITIAL VERSION
--
-- 0.2.0   19/02/2025  Keith Lines    Use sumMultVec
-------------------------------------------------------------------------

-- Use implementation of vectors by André Videla.
-- Overloads operators ^ and * for applying to elements of vectors.
import Data.Vector


-- Assign sensitivity coefficients.
c :: Vec Double
c = Vec [0.5, 0.4, 0.3, 0.2]
```

```
-- Assign uncertainties associated with estimates of input quantities.
ux :: Vec Double
ux = Vec [0.1, 0.2, 0.3, 0.4]

uy :: Double
uy = sqrt ((c.^2) `sumMultVec` (ux.^2))

main :: IO ()
main = do
  putStrLn "Law of propagation of uncertainty (LPU)"
  putStrLn "Haskell example version 0.2.0 with Andre vectors."
  putStrLn (show uy)
```

C.5 LAW OF PROPAGATION OF UNCERTAINTY: IDRIS 2 WITH VECTORS

C.5.1 File Numerical.idr

```
module Data.Numerical
--------------------------------------------------------------------------
-- Description: Add further numerical functions.
--
-- Author:  André Videla
-- Created: 04/11/2022
--
-- Version: 0.1.0
--
-- Amendment History
-- -----------------
-- Version Date        Name           Description
-- ------- ----------  -------------  --------------------------------
-- 0.1.0   04/11/2022  André Videla   INITIAL VERSION
--------------------------------------------------------------------------

infixl 9 ^

export
(^) : Num i => i -> Nat -> i
(^) x Z = 1
(^) x (S n) = x * (x ^ n)

public export
interface Floating a where
  sqrt : a -> a

export
Floating Double where
  sqrt = Prelude.sqrt
```

C.5.2 File Vector.idr

```
module Data.Vector
--------------------------------------------------------------------
-- Description: Add functions and operations to vector type provided
--              by Idris2.
--
-- Author:  André Videla
-- Created: 04/11/2022
--
-- Version: 0.2.0
--
-- Amendment History
-- -----------------
-- Version Date        Name           Description
-- ------- ----------  -------------  ---------------------------------
-- 0.1.0   04/11/2022  André Videla   INITIAL VERSION
--
-- 0.2.0   19/02/2025  Keith Lines    Removed point-wise square root
--                                    function, as no longer required.
--                                    Added function sumMultVec.
--------------------------------------------------------------------

import public Data.Vect
import Data.Numerical

-- Operator is right associative and with the highest level
-- of binding.
infixl 9 .^

public export
(.^) : Num a => Vect n a -> Nat -> Vect n a
(.^) xs n = map ( ^ n) xs

public export
(*), (+): Num a => Vect n a -> Vect n a -> Vect n a
xs * ys = zipWith (*) xs ys
xs + ys = zipWith (+) xs ys

-- sumMultVec inputs two row vectors of the same length and outputs
-- a scalar value. This value is the result of multiplying the equivalent
-- components of the input vectors then adding the components of the
-- resulting vector.
-- NOTE: Use Data.Vector.(*) xs ys, because compiler complains of
--       ambiguity if using xs * ys
public export
sumMultVec : Num a => Vect n a -> Vect n a -> a
sumMultVec xs ys = sum (toList (Data.Vector.(*) xs ys))

public export
concat : Vect n a -> Vect m a -> Vect (n + m) a
concat [] ys = ys
concat (x :: xs) ys = x :: concat xs ys
```

C.5.3 File Main.idr

```
module Main
--------------------------------------------------------------------------
-- Description: Combine standard uncertanties according to
--              law of propagation of uncertainty from BIPM
--              Guide to the Expression of Uncertainty in Measurement (GUM)
--
--              The following assumptions are made:
--              - There are N input quantities.
--              - There is one output quantity.
--              - The input quantities are independent.
--              - The output quantity can be expressed as an explicit
--                function of the input quantities: Y = f(X1, ..., XN).
--
-- Author:  Keith Lines, Data Science department, NPL.
--          From MATLAB code and comments by Ian Smith,
--                                      Data Science department, NPL.
--
-- Version: 0.2.0
--
-- Amendment History
-- -----------------
-- Version Date        Name           Description
-- ------- ----------  -------------  ---------------------------------
-- 0.1.0   09/02/2025  Keith Lines    INITIAL VERSION
--
-- 0.2.0   19/02/2025  Keith Lines    Use function sumMultVec.
--------------------------------------------------------------------------

import Data.Vector

-- Assign sensitivity coefficients.
c : Vect 4 Double
c = [0.5, 0.4, 0.3, 0.2]

-- Assign uncertainties associated with estimates of input quantities.
ux : Vect 4 Double
ux = [0.1, 0.2, 0.3, 0.4]

uy : Double
uy = sqrt ((c.^2) `sumMultVec` (ux.^2))

main : IO ()
main = do putStrLn "Law of propagation of uncertainty (LPU)"
          putStrLn "Idris2 example version 0.2.0 with Andre vectors."
          putStrLn (show uy)
```

C.6 Law of Propagation of Uncertainty: Sized Vectors in Haskell

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
module Dependent.Vector (Vec(..), zipWith) where
-------------------------------------------------------------------------
-- Description: Define sized vectors using Haskell.
--
-- Author:  André Videla
--
-- Version: 0.1.0
--
-- Amendment History
-- -----------------
-- Version Date        Name           Description
-- ------- ----------  -------------  --------------------------------
-- 0.1.0   03/11/2022  André Videla   INITIAL VERSION
-------------------------------------------------------------------------

import Data.Nat
import Data.Kind
import Prelude hiding (zipWith)

infixr 6 :-

-- Vectors with lengths
data Vec :: Nat -> Type -> Type where
  -- The empty vector has length ‘Z‘
  Nil :: Vec 'Z a
  -- Cons cell has a value of type ‘a‘, a vector of length n as tail
  -- and is itself of length (S n)
  (:-) :: a -> Vec n a -> Vec ('S n) a

-- Zip over two vectors of length ‘n‘ preserves the length
zipWith :: (a -> b -> c) -> Vec n a -> Vec n b -> Vec n c
zipWith _ Nil Nil = Nil
zipWith f (x :- xs) (y :- ys) = f x y :- zipWith f xs ys

-- Vectors with length are functors
instance Functor (Vec n) where
  fmap f Nil = Nil
  fmap f (x :- xs) = f x :- fmap f xs

-- point-wise exponential preserves length
(.^) :: (Num a, Integral i) => Vec n a -> i -> Vec n a
(.^) ls i = fmap (^i) ls

-- We give a ‘Num‘ instance in order to use ‘*‘ ‘+‘ etc
instance Num a => Num (Vec n a) where
  (+) a b = zipWith (+) a b
  (*) a b = zipWith (*) a b
  negate = fmap negate
  abs = fmap abs
  signum = fmap signum
  -- we leave ‘fromInteger‘ undefined because there is no easy way
```

```
-- to replicate ‘n‘ copies of the given integer
fromInteger = undefined
```

C.7 LAW OF PROPAGATION OF UNCERTAINTY: PYTHON WITH NUMPY

```
import numpy as np
import numpy as np
# ------------------------------------------------------------------
# Description: Combine standard uncertanties according to
#              law of propagation of uncertainty from BIPM
#              Guide to the Expression of Uncertainty in Measurement (GUM)
#
#              The following assumptions are made:
#              - There are N input quantities.
#              - There is one output quantity.
#              - The input quantities are independent.
#              - The output quantity can be expressed as an explicit
#                function of the input quantities: Y = f(X1, ..., XN).
#
# Author:  Keith Lines, Data Science department, NPL.
#          From MATLAB code and comments by Ian Smith,
#                                     Data Science department, NPL.
#
# Version: 0.1.1
#
# Amendment History
# -----------------
# Version Date        Name           Description
# ------- ---------   -------------  ------------------------------------
#  0.1.0  11/02/2024  Keith Lines    INITIAL VERSION
#
#  0.1.1  17/02/2024  Keith Lines    Add comments regarding trustworthiness.
# ------------------------------------------------------------------

# Assign sensitivity coefficients.
c = np.array([0.5, 0.4, 0.3, 0.2])
c = np.square(c)


# Assign uncertainties associated with estimates of input quantities.
# Note this vector is a column vector (implemented in this case as a
# transpose of a row vector).
ux = np.array([0.1, 0.2, 0.3, 0.4])
ux = np.square(ux)

uy = np.sqrt(np.matmul(c,ux.transpose()))

# NOTE: Although the correct result is generated, c and ux are
# used to contain the squares of c and ux. The formula for uy looks
# wrong written in terms of the notation c and ux, and that takes away
# from the trustworthiness of the code in terms of its correspondence
# to the underlying statistics. The correct answer is generated, so are
# such matters worth consideration? This report argues that yes they are
# worth consideration. [Thanks to Peter Harris]
print("Law of propagation of uncertainty (LPU)")
print("Python example version 0.1.1 with NumPy")
print(f'{uy:.17f}')
```