XXIV IMEKO World Congress "Think Metrology"
August 26 - 29, 2024, Hamburg, Germany

LabMate: a prospectus for types for MATLAB

Conor McBride [1], Georgi Nakov [1], Fredrik Nordvall Forsberg [1], André Videla [1], Alistair Forbes [2], Keith Lines [2]

[1] University of Strathclyde , UK
[2] National Physical Laboratory, UK

Abstract - Many computations in science and engineering are implemented in the programming language MATLAB. However the high-level meaning of such MATLAB programs stays informal, which can lead to implementation errors and bugs, for example relating to incompatible units of measure for quantities, or incompatible sizes of matrices at runtime. We are in the process of developing LabMate, which is a tool for reifying current informal programmer practices into a language of formal comments. These comments are ignored by MATLAB, but acted on and checked by LabMate. We outline the design principles behind LabMate, our current progress, and our future plans.

Keywords: MATLAB, software correctness, dimensional consistency, type theory

1. Introduction

MATLAB is a key workhorse in many scientific and engineering disciplines that are heavily reliant on numerical methods. It helps us do powerful things. However, as with all software, MATLAB code may contain implementation errors and bugs. In good programming practice for MATLAB developers, programmers use comments to make clear what physical systems their data concern and how the data should be interpreted, specifying, e.g., units of measure for quantities. Regrettably, none of this rich and often disciplined metadata is perceptible to MATLAB, which instead enforces the compatibility of producers and consumers of data by run-time checking of tags which indicate only machine representation, not any form of meaning. In reality, the programmers document meaning for each other's benefit but keep the machine in the dark.

To rectify this situation, we are developing LabMate, which is a tool to reify current virtuous engineering practices as a formal language of MATLAB comments. MATLAB remains in the dark, but LabMate reads, assesses, and transforms MATLAB programs in accordance with these comments. Behind the scenes, LabMate is retrofitting an expressive type system to MATLAB, with more of the meaning of programs recorded in their types. This gives a lightweight and low-cost way for MATLAB programmers to express their intent, and in a language they are already working in, rather than starting over from scratch. While type systems and their theory is a well established discipline of computer science, the development of LabMate has several novel advances on the algebraic structure required to classify matrices and the meanings of the quantities therein, e.g., their physical dimensions. LabMate brings advanced type systems to effectiveness within, rather than instead of, existing scientific and engineering toolchains and practices.

LabMate is under constant development; we encourage interested readers to get in touch for access to the latest version. At time of writing, the necessary core type system is in place: we are busily extending the variety of MATLAB expressions it can make sense of, and the constraints it can solve. This paper effectively documents how far we have come, and the roadmap to our next milestone.

2. LabMate in action

The following greatly simplified, but still realistic, example from metrology will help illustrate the principles of how LabMate will work. Consider one of the calculations required for measuring resistors using a cryogenic current comparator bridge [1].

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 the offset and drift from the voltage readings. Therefore, the applied signal (bridge energisation) is a square wave of *reversals*. A calibration factor is also applied, dividing the signal into two parts with different amplitudes.

The recorded output follows the input signal, superimposed on a detector with noise, offset and drift. Figure 1 provides an example of recorded output, using simulated data.

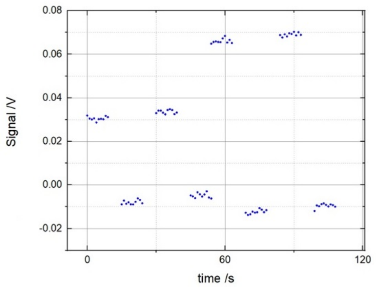

Fig.1. Simulated example of recorded output.

The calculation of offset and drift is achieved using a least squares line fit calculated by solving a series of simultaneous equations, listed below. Note that although offset and drift differ for calibration and non-calibration parts of the input signal, the drift is the same in both cases.

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

The matrix left of $\times$ is made up of the following values:

- 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 column contains time values.

The column vector right of $\times$ and left of $=$ contains values to be calculated using the line fit which then determine the measured value of the resistor by a calculation which is beyond the scope of this paper. The column vector right of $=$ contains voltage readings.

This calculation can be converted into a MATLAB problem using the builtin left division operator $\backslash$ to compute a least squares fit. Let us step through how we intend to use LabMate to help us deliver this solution.

We begin by declaring the types of the input data: the `times` and the `voltages`. Our example calculation uses twelve time and voltage inputs, arranged as two rows to be read from a file. A real measurement would involve considerably larger datasets.

```
%> times    :: [ 1 x 12 ] double
%> voltages :: [ 1 x 12 ] double
```

Lines starting with ‘`%>`’ are interpreted as ordinary comments by MATLAB, but as *directives* by LabMate, here giving us enough information not only to insist that the values of `times` and `voltages` *conform* to the given sizes, but to determine *how* to read them from a file. We would issue the further directive

```
%> readfrom "inputs.txt" times voltages
```

LabMate is a program *transducer*: its output is a modified version of its input, allowing it to respond to directives with information in comments or by generating code. Here we would expect to see something like

```
%> readfrom "inputs.txt" times voltages
%<{
h=fopen("inputs.txt");
c=textscan(h,'%f');
fclose(h);
src = c{1};
readPtr = 1;
for i = 1:12
  times(i) = src(readPtr);
  readPtr = readPtr + 1;
end
for i = 1:12
  voltages(i) = src(readPtr);
  readPtr = readPtr + 1;
end
%<}
```

where the comment lines ‘`%<{`’ and ‘`%<}`’ visually delimit the extent of the code generated by LabMate, so that humans may safely ignore it. LabMate knows enough about scope in MATLAB to ensure that auxiliary variable naming never creates conflict. We have kept to one datum per line for simplicity, but we are free to extend our format specification language as we see fit [2].

Next, let us build up the matrix in the calculation. The method gives the first four columns a pattern of blocks, pasted in a grid.

```
%> z3 :: [ 3 x 1 ] double
z3 = [ 0; 0; 0 ]
%> i3 :: [ 3 x 1 ] double
i3 = [ 1; 1; 1 ]

ddnc = [ i3  z3  i3  z3
        -i3  z3  i3  z3
         z3  i3  z3  i3
         z3 -i3  z3  i3 ]
```

LabMate will check that this pasting strategy delivers a rectangular matrix, well in advance of MATLAB runtime. Pasting on the measured times (transposed to a column) completes the matrix. Let us ask for its type. LabMate responds to information-seeking directives by writing comments of its own, beginning with ‘`%<`’.

```
M = [ ddnc times' ]
%> typeof M
%< M :: [ 12 x 5 ] double
```

Now that we have assembled the data, we may solve the equations and ask for the type of the solution.

```
x = M \ voltages
%> typeof x
```

Did you notice the error in the above code? The vector `voltages` should be of size 12×1, but is erroneously of size 1×12 instead. LabMate will detect this error and respond:

```
x = M \ voltages
%< unsolved constraint 12 =? 1
%> typeof x
%< the type of x is quite a puzzle
```

That is, for `x` to have a sensible type, we need `M` and `voltages` to have the same number of rows. We forgot to transpose `voltages`! If we make the necessary repair, we should now see something more satisfying.

```
x = M \ voltages'
%> typeof x
%< x :: [ 5 x 1 ] double
```

This way, LabMate can detect matrix size errors at development time, rather than at runtime — running the original code would indeed crash with the error:

```
Matrix dimensions must agree
```

3. Types for matrices

Under the hood, LabMate translates selected well behaved MATLAB expressions to typed expressions in its own core theory of types and values. As matrices feature heavily in MATLAB code, the types of matrices play a central role in this theory. In particular, we acknowledge that the rows and columns of a matrix might pertain to distinct individual entities, rather as the rows and columns of a spreadsheet do. Internally, a matrix type looks like

$$\mathsf{Matrix}\,R\,C\,E\,rs\,cs$$

We thus parametrize matrix types by two arbitrary 'header' types R, for the individuals the rows concern, and C for the columns. We may then give two lists, $rs = [r_1, \ldots, r_m] \in \mathsf{List}\,R$ (so that each $r_i \in R$) and $cs = [c_1, \ldots, c_n] \in \mathsf{List}\,C$ of those individuals. A parametrized type $E(r, c)$ gives the type of entry for any combination of row and column individual, with $E(r_i, c_j)$ being the type of the entry $e_{i,j}$. In the diagram below, the es inside the rectangle are the matrix entries computed by the MATLAB code. The rs and cs outside the rectangle are *meta*data coming from the matrix type, known only to LabMate, and not available as data to MATLAB.

$$\begin{array}{c|ccccc|}
 & c_1 & \cdots & c_j & \cdots & c_n \\ \hline
r_1 & e_{1,1} & & e_{1,j} & & e_{1,n} \\
\vdots & & & & & \\
r_i & e_{i,1} & & e_{i,j} & & e_{i,n} \\
\vdots & & & & & \\
r_m & e_{m,1} & & e_{m,j} & & e_{m,n} \\ \hline
\end{array}$$

To check that $\mathsf{Matrix}\,R\,C\,E\,rs\,cs$ is a valid type, LabMate must:

1. check that R and C are both valid types;
2. check that $E(r, c)$ is a valid type whenever $r \in R$ and $c \in C$;
3. check that $rs \in \mathsf{List}\,R$ and $cs \in \mathsf{List}\,C$.

A notable special case is when both R and C are the unit type $R = C = \mathbf{1}$, that is, the type with exactly one element. $\mathsf{List}\,\mathbf{1}$ amounts to a mere 'tally', counting indistinct things, which we allow to be written as a number in decimal notation. In this case, there is only one type of entries $E(r, c)$ (because both r and c must be the unique element of the unit type), and the only information available in the lists $rs : \mathsf{List}\,\mathbf{1}$ and $cs : \mathsf{List}\,\mathbf{1}$ are their lengths. Thus, we have recovered $\mathsf{Matrix}\,\mathbf{1}\,\mathbf{1}\,E\,m\,n$ as the type of $m \times n$ matrices with entries of type E, where m and n are the unique lists over the unit type of length m and n respectively. That is what we render in LabMate directives and responses as

$$\texttt{[}\ m\ \texttt{x}\ n\ \texttt{]}\ E$$

as in our `M :: [ 12 x 5 ] double` in Section 2.

So much for what it is to be a valid matrix type: what does it mean to be a valid matrix *in* such a type?

Firstly, a singleton entry matrix must have singleton header lists $[r]$ and $[c]$.

$$\begin{array}{c|c|}
 & c \\ \hline
r & e \in E(r,c) \\ \hline
\end{array}
\quad \in \mathsf{Matrix}\,R\,C\,E\,[r]\,[c]$$

Secondly, to check a horizontal pasting of matrices, we must ensure that the header and entry types match, that the row headers are the same for left and right, and that the column headers can be split into a prefix for the left matrix and a suffix for the right. In other words, the column headers for the whole are the concatenation ($+\!\!+$) of the column headers for the parts.

$$\begin{array}{c|c|c|}
 & cs_A & cs_b \\ \hline
rs & \begin{array}{c} A \in \\ \mathsf{Matrix}\,R\,C\,E\,rs\,cs_A \end{array} & \begin{array}{c} B \in \\ \mathsf{Matrix}\,R\,C\,E\,rs\,cs_B \end{array} \\ \hline
\end{array}$$

$$\in \mathsf{Matrix}\,R\,C\,E\,rs\,(cs_A +\!\!+ cs_B)$$

Thirdly, vertical pasting is similar, except that it is the row headers which split into prefix and suffix while the column headers are shared.

$$\begin{array}{c|c|}
 & cs \\ \hline
rs_A & \begin{array}{c} A \in \\ \mathsf{Matrix}\,R\,C\,E\,rs_A\,cs \end{array} \\ \hline
rs_B & \begin{array}{c} B \in \\ \mathsf{Matrix}\,R\,C\,E\,rs_B\,cs \end{array} \\ \hline
\end{array}
\in
\begin{array}{l} \mathsf{Matrix}\,R\,C\,E \\ \quad (rs_A +\!\!+ rs_B)\,cs \end{array}$$

Now that we have matrices, we must explain how to typecheck matrix operations.

To check if we may add matrices A and B, LabMate must:

1. check that A and B both have the *same* matrix type $\mathsf{Matrix}\, R\, C\, E\, rs\, cs$;
2. check that *every* entry type $E(r, c)$ admits addition.

In the short term, LabMate can get by with a hard-coded table of which types (e.g. `double`) admit addition, but more flexible and general approaches to operator overloading (e.g., Haskell's 'type classes' [3]) are readily adoptable in due course.

Multplication is rather more involved, of course. The usual requirement that the matrix sizes 'meet in the middle' generalises straightforwardly to lists of headers,

$$\begin{array}{c|c|} & ms \\ \hline rs & \begin{array}{c} A \in \\ \mathsf{Matrix}\, R\, M\, E_1\, rs\, ms \end{array} \\ \hline \end{array} \times \begin{array}{c|c|} & cs \\ \hline ms & \begin{array}{c} B \in \\ \mathsf{Matrix}\, M\, C\, E_2 \\ ms\, cs \end{array} \\ \hline \end{array}$$

$$\in \quad \begin{array}{c|c|} & cs \\ \hline rs & \begin{array}{c} \mathsf{Matrix}\, R\, C\, E_3 \\ rs\, cs \end{array} \\ \hline \end{array}$$

but we must also check that the entry types are compatible. Specifically, LabMate must:

1. check, for all r, m and c, that if $a \in E_1(r, m)$ and $b \in E_2(m, c)$, then $a \times b \in E_3(r, c)$;
2. check that *every* entry type $E_3(r, c)$ admits addition.

These conditions are straightforward in the special case where $R = C = \mathbf{1}$ and the Es are standard numerical types. There are, however, more motivating possibilities.

Of particular interest is the case when the header types, R and C, are types of physical dimensions, with $E(r, c)$ being the type of physical quantities of dimension $r^{-1} \cdot c$. Such quantities admit addition only if the dimensions match. Meanwhile, their multiplication correspondingly multiplies the dimensions. This way, we can reduce dimensional consistency checking à la Kennedy [4] to typechecking, but also for programs involving matrices of non-uniform dimension, following the work of Hart [5]. We will see in Section 5 how this can be helpful.

Checking the compatibility between operations and data boils down to checking equality of types. LabMate types are *dependent* types, in the style of the state-of-the-art functional programming languages Agda and Idris [6, 7, 8]. Dependent types express the reality that data validity is often a *relative* notion, with the compatibility requirements for matrix multiplication being a standard example. Types which mention (hence depend on) some values (e.g., our header lists) are used to classify other values (e.g., our matrices) with respect to them.

Equality of types therefore demands some notion of equality for the values they mention. We equip every type in our core theory with an algorithmic decision procedure testing when two expressions in the type are sufficiently obviously equal to guarantee compatibility of types. Languages like Agda and Idris generate their notion of expression equality by extending the evaluation rules used to compute with values at runtime to cope with expressions involving variables by getting stuck whenever a variable prevents a condition from being tested. In MATLAB, the matrix pastings `[[A B] C]` and `[A [B C]]` both mean the same thing as `[A B C]`, but for us to see that requires *algebraic* properties of the $+\!\!+$ operator for our header lists. The state-of-the-art languages turn up to this algebra fight armed only with arithmetic! They force their programmers to give explicit proofs of type compatibility, doing the algebra by hand. We would not dream of inflicting this bureaucracy on LabMate users, especially when the small amount of algebra we need is perfectly straightforward [9].

We incorporate the specific algebraic theories we need to manage matrices and dimensions into LabMate's expression equality. Indeed, we also consider when *matrix* expressions are equal. We reassociate horizontal and vertical pastings, so that

$$\begin{array}{|c|}\hline A \\ \hline C \\ \hline \end{array}\begin{array}{|c|}\hline B \\ \hline D \\ \hline \end{array} = \begin{array}{c} \begin{array}{|c|c|}\hline A & B \\ \hline \end{array} \\ \begin{array}{|c|c|}\hline C & D \\ \hline \end{array} \end{array}$$

4. Implementation and Current Status

In contrast to most other typecheckers, LabMate works using a *transducer* model of interaction: it inputs MATLAB code with formal comments, and outputs a new version of its input, responding to the comments, as if it were a development collaborator. It thus needs to make sure that it retains as much of the input formatting as possible. Making sense of MATLAB code presented a significant reverse engineering challenge, as the MATLAB syntax is largely specified informally and by example.

The meaning of whitespace in MATLAB is highly context-sensitive — e.g., `(2 - 2)`, `[2 - 2]` and `(2 -2)` all meaning zero, but `[2 -2]` being a 2×1 matrix. Another precursor to typechecking is the deduction of variable scope: as proof of concept, we implemented the directive

```
%> rename x y
```

which requests LabMate to rename all and only the occurrences of the `x` in scope at that point to `y`, but only if that does not conflict with other things called `y`. To our surprise, we discovered that a similar facility offered by MATLAB's own editor sometimes changes the semantics of programs, which is presumably not what the user intends. Writing a typechecker, we cannot afford the luxury of waiting until runtime to figure out which `x` is which.

LabMate elaborates MATLAB expressions and commands into terms of its own internal core type theory. Although the simple examples in this paper use *constant* sizes and dimensions, any practicable library requires size, header and dimension expressions to contain *parameters*. For example, if `A` is $i \times j$ and `B` is $i \times k$, then

```
[ A B
  B A ]
```

should be a valid matrix pasting, because $j + k = k + j$. Our notion of equality for numbers thus includes tacit applications of commutative and associative laws. Physical dimensions are modelled as the free Abelian group generated by some base set of dimensions [10], so we decide that theory, too. The core theory we need is in place.

The actual typechecking is implemented as a stack based virtual machine which traverses the syntax tree, deducing its translation into the core theory by propagating type information, generating typing constraints, and solving them by unification [11]. In order to support a much more expressive language of types, we must move on from the classic requirement that all constraints can be solved or rejected on first inspection and hence that the program can be processed exactly once and in a specific order [12]. Our virtual machine supports repeated refocusing to wherever in the program there is progress to be made. Moreover, as MATLAB notation is somewhat overloaded (e.g., `*` can mean either scalar or matrix multiplication), so LabMate elaboration has a (limited) need for backtracking search, so that, too, is facilitated.

After the virtual machine has solved as many problems as it can, we analyse its final state to reconstruct the output source code, perhaps with inserted warnings and error messages, or additional responses to queries and generated code.

5. Supporting dimensional consistency

Coming back to the example from Section 2 again, we aim to give both `M` and `voltages` more precise types, also taking their physical dimensions into account,

```
%> M :: [ 12 x k::D<-[{},{},{},{},T] ] Q(k)
%> voltages :: [ 1 x 12 ] Q(V)
```

but we had better unpack what we mean.

Firstly, `M`'s row header type remains **1**, so our row header list is still abbreviated by a number.

Secondly, we see `Q(`d`)` as the type of *quantities* with physical dimension d, represented in MATLAB as a mere `double` with standardised units. We shall define a type `D` of dimensions presently. The notation

```
k::D<-[{},{},{},{},T]
```

will tell LabMate that the column header type is `D`, and that the matrix entry type `Q(k)` varies in accordance with the header, locally named `k`, which is drawn from the given list of dimensions. But what type is `D`? We will have defined, in LabMate directives, rather than MATLAB,

```
%{>
Base =
  ['Time, 'Length, 'Mass, 'Current,
   'Temperature, 'Amount, 'Luminosity]
D = Abel(Enum(Base))
%}
```

where `Base` is a list of symbolic constants standing for base dimensions, `Enum` converts any such list to a finite enumeration type, and `Abel` constructs the free Abelian group on any type of generators, with unit `{}`, singleton generators `{`g`}`, multiplication `*`, division `/`, and exponentiation by integer constants `^`. We further define

```
%{>
T = {'Time}
V = {'Mass}*{'Length}^2/T^3/{'Current}
%}
```

and `voltages` becomes a row vector uniformly containing quantities in `Q(V)`.

With this refinement in place

```
x = M \ voltages'
%> typeof x
```

will yield

```
%< x :: [ k::D<-[{},{},{},{},T] ] Q(V/k)
```

In other words, `x` is a column vector comprising four quantities in `Q(V)` and a fifth in `Q(V/T)`. LabMate has thus propagated the dimensional variation from the type of `M` to the type of `x`, while retaining the uniformity of `V` in the type of `voltages`. Checking multiplicability for matrices of quantities `Q(`$d(i,j)$`)` and `Q(`$d(j,k)$`)` amounts to checking that $d(i,j) \star d(j,k)$ is independent of j. In our example, there is no dimensional variation in the 'middle' of the inputs, so multiplicability is evident. There is a degree of freedom to multiply headers by an arbitrary dimension and compensate in the entry type, but we avoid ambiguity by sticking to Hart's discipline [5], namely

1. ensure any list of dimensions starts with the unit, `{}`;
2. divide by dimensions coming from a row header;
3. multiply by dimensions coming from a column header.

Valid operations which respect Hart's discipline will always have header lists which meet in the middle exactly and yield outputs which retain Hart's discipline [13].

6. Conclusions and future work

We have illustrated our prospectus for LabMate, a third-party programming assistant for MATLAB. Its type system offers expressivity well beyond the dynamic type tagging that MATLAB does itself. Typechecking with LabMate is no ironclad guarantee of complete correctness, but it is relatively lightweight and has the potential to rule out large classes of silly mistake, including dimensional inconsistency.

However, if all LabMate does is complain about mistakes, it will not appeal to programmers in a hurry. We have already shown a simple example where LabMate does not merely check but rather generates code using its types as specification. There is considerable potential to push further in this direction, little by little. For example, we could equip LabMate with its own expression language for calculating with quantities which have not only dimensions but *units*, compiling to MATLAB with appropriate conversion factors correctly inserted. Our mission is, ever so gently, to tilt the balance towards programming at a higher level of structure and meaning in LabMate's directive language.

So far, our focus has been on calculation with matrices, as this is what makes MATLAB unique as a programming language. In the future, we would like to also extend our coverage to conditionals and loops, which might necessitate some approximation of what is known about program fragments at runtime. We also have our work cut out to retrofit LabMate types to MATLAB *libraries*.

Inspired again by Kennedy's work — his gradualist reformation of PHP [14] — we have set out on this journey to bring the benefits of expressive type systems to scientists and engineers *where they are*. It would be unrealistic of us to expect practitioners to abandon their existing toolchains and learn whole new programming styles and languages that, in any case, exist only as academic research prototypes. Moreover, it has very much *not* turned out that, in addressing MATLAB, we were merely *applying* type system research already well established. This project has pushed us in new directions and generated raised expectations of the *algebra* to be done by the typechecker. There is so much to learn from applications of dependent types in the wild!

Acknowledgements

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

Thanks to NPL colleague Nick Fletcher for his help and guidance providing the example from Section 2. Thanks also to NPL colleagues Peter Harris, Louise Wright and Ian Smith for reviewing this paper, and to Professor Neil Ghani for his support.

REFERENCES

[1] J M Williams, T J B M Janssen, G Rietveld, and E Houtzager. An automated cryogenic current comparator resistance ratio bridge for routine resistance measurements. *Metrologia*, 47(3):167, 2010.

[2] Conor McBride, Georgi Nakov, and Fredrik Nordvall Forsberg. Measuring with confidence: leveraging expressive type systems for correct-by-construction software. *Acta IMEKO*, 12(1):1–5, 2023.

[3] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '89)*, pages 60–76. ACM, 1989.

[4] Andrew Kennedy. Types for units-of-measure: Theory and practice. In Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók, editors, *Central European Functional Programming School (CEFP 2009), Revised Selected Lectures*, volume 6299 of *Lecture Notes in Computer Science*, pages 268–305. Springer, 2009.

[5] George W. Hart. *Multidimensional Analysis*. Springer, 1995.

[6] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Available at https://www.cs.nott.ac.uk/~psztxa/publ/ydtm.pdf, 2005.

[7] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[8] Edwin Brady. Idris 2: Quantitative type theory in practice. In Anders Møller and Manu Sridharan, editors, *Proceeding of the 35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194, pages 9:1–9:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[9] Guillaume Allais, Conor McBride, and Pierre Boutillier. New equations for neutral terms: a sound and complete decision procedure, formalized. In *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-Typed Programming (DTP '13)*, pages 13–24. ACM, 2013.

[10] Charles C. Sims. *Computation with Finitely Presented Groups*, volume 48 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1994.

[11] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23—41, 1965.

[12] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '82*, POPL '82. ACM Press, 1982.

[13] Conor McBride and Fredrik Nordvall Forsberg. Type systems for programs respecting dimensions. In Franco Pavese, Forbes Alistair, Nien-Fan Zhang, and Anna Chunovkina, editors, *AMCTM XII*, pages 331–345. World Scientific, 2022.

[14] Andrew Kennedy. Driving types into PHP (invited talk). In Sam Lindley and Brent A. Yorgey, editors, *Type-Driven Development 2017*, page 1. ACM, 2017.