IMEKO TC6 International Conference on Metrology and Digital Transformation
September 19 – September 21, 2022, Berlin, Germany

EXPRESSIVE TYPE SYSTEMS FOR METROLOGY

Conor McBride [a], *Georgi Nakov* [a], *Fredrik Nordvall Forsberg* [a,*]

[a]Department of Computer and Information Sciences, University of Strathclyde, Glasgow, UK
[*]Corresponding author. Email address: `fredrik.nordvall-forsberg@strath.ac.uk`

Abstract – Modern programming language type systems help programmers write correct software, and the software they intended to write. We show how expressive types can be used to encode dimension and units of measure information, which can be used to avoid dimensional mistakes and guide software construction, and how types can even help to generate code automatically, which eliminates a whole class of bugs.

Keywords: type systems, correctness, programming languages

1. INTRODUCTION

The digital transformation of metrology offers both challenges and opportunities. With increased software usage and complexity, there is also a need to increase trust in the computations performed — how do we know that the software is doing what is expected of it? Computer Science offers a wide range of formal methods and verification techniques to tackle this challenge. As always, there is a balance to be struck between how much additional effort is required from the user, and how useful the verification procedure can be. Our main thesis is that a lot can be achieved by simply "bridging the semantic gap" between human and machine: current common practice is for computers to mindlessly execute instructions, without understanding for what purpose. This means that any verification must happen after the fact, by a separate process. What if, instead, there would be some way of communicating our intent as we are writing our software? Then the machine could help us write it, rather than just tell us off for getting it wrong after the fact...

We advocate for the use of type systems as a lightweight method to communicate intent. *Dependently typed programming languages* are a new breed of languages which have type systems which are expressive and strong enough to use types to encode the meaning of programs to whatever degree of precision is needed. We can ensure that types can be automatically checked at compile-time, and so they provide machine-certification of the program's behaviour at low-cost. Concretely, we therefore get both *lightweight* and *machine-certified* trust in the correctness of software.

In the metrology domain, in particular, we can make good use of implicit tacit knowledge such as dimensional correctness to help the computer help us. This is work currently done by humans, but there is no reason why it could not be done automatically by a machine instead. As a small case study, we show that by turning informal comments about the expected input format of a data into machine-readable form, we can not only check that given data conforms to the format, but automatically generate Matlab code for reading from disk and converting to appropriate units, thus eliminating a source of bugs and increasing trust in the software.

Our goals are similar to other software projects for calculation with physical quantities [Fos13, Hal20], but we put additional emphasis on the use of types as a convenient method of communication between human and machine.

2. TYPE SYSTEMS AS LIGHTWEIGHT FORMAL METHODS

In programming languages such as Fortran or C, types such as floating point numbers or integers are used to help the compiler with memory layout. A pleasant side effect is that basic errors such as trying to divide an integer by a string can be detected and reported at compile time, rather than at run time. While useful for avoiding disastrous results, this is quite a negative view on types: they are *against errors*, but are they also *for something*?

This century, types can be used to make an active contribution by offering guidance *during* the program construction process, not just criticism afterwards. We pay upfront by stating the type of the program we want to write, but are then paid back in the machine being able to use those types to offer suggestions for functions to call, or even generate code for boring parts of the program. The space within which we search for programs is correspondingly smaller and better structured.

For such help to be meaningful, the types available need to be sufficiently expressive; it is usually not very instructive to be told that we need to for example supply an integer, nor is it going to be very helpful for the compiler to generate a floating point number for us. As another example, consider a type whose elements are matrices. As given, this is again not very helpful — a matrix can, after all, be seen as a (structured) collection of numbers, and we just said that numbers in themselves do not carry much meaning. But we can *refine* our type of matrices to a type $\mathsf{Matrix}(n, m)$ which keeps track of the size $n \times m$ of the matrix: e.g., the type of matrix multiplication can be usefully expressed as

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

i.e. insisting that the sizes of the input matrices are compatible, and determining size of the output matrix. Furthermore, if we were trying to write a program to implement matrix multiplication, the above type would give us helpful hints on what we need to produce.

For another example, consider implementing a program that creates a block matrix by putting two given matrices next to each other. It is natural to give it type

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

i.e. we insist that both input matrices have the same number of rows, and the number of columns in the output matrix is the sum of the number of columns in the input matrices. We see that computations such as $m+k$ naturally arise in types — to do a proper job classifying such programs as meaningful, our systems must thus allow values and computations to occur in types. Such type systems are called *dependent type systems* [BD09], as types can depend on values. They give us enough expressive power to meaningfully communicate our intentions to the compiler.

3. UNITS OF MEASURE USING TYPES

The metrology domain is perhaps especially well suited for the use of types to guarantee correctness, because the prevalent use of dimensions (such as length and time) and units of measure (such as metres and seconds) in many respects play the same role as types: it is not dimensionally correct to add a metre and a second. It thus seems natural to use type systems to reduce dimension checking to type checking. Indeed, many mainstream languages have support for units, implemented using a wide range of techniques, from static types to dynamic run-time checks (see Bennich-Björkman and McKeever's survey [BBM18] for more):

- Microsoft's F# [Ken10] has units of measure built in to static type checking;
- C++'s Boost Units library [SW10] uses templates to check units statically;
- Java has a proposed API adding classes for dimensioned quantities [DKS21], but run-time casts are inevitable;
- Haskell's type system can now encode basic units of measure as a library [ME14] or a typechecker plugin [Gun15];
- Python libraries such as Pint [pin22] cannot do static checking of dimensional correctness, but implement run-time checks instead. Similarly MATLAB has support for dynamic unit checking using the Symbolic Math Toolbox [Mat22].

However many of these solutions provide no static guarantees, or rely on rather ad-hoc extensions of the type system, often with hard to understand error messages as a result. Encoding dimensions using dependent types is a more principled way to include dimension checking in a programming language. We briefly describe how we implemented a typechecker including dimensions here [MNF22].

First, how are we to represent dimensions themselves? Following Kennedy [Ken95], we fix a set D of fundamental dimensions (such as length L, time T and mass M). We may multiply or divide dimensions (for example forming mass per time squared M/T^2), and the order of dimensions do not matter (mass times length $\mathsf{M} \cdot \mathsf{L}$ is the same as length times mass $\mathsf{L} \cdot \mathsf{M}$). These considerations leads us to model dimensions as elements of the free Abelian group over the set of fundamental dimensions D [Sim94, §8.1].

For typechecking, we need to be able to decide if two given dimensions are equal or not. This is made easier by a *normal form* for elements of the free group: we first (arbitrarily) impose a total order on the fundamental dimensions D (for example mass before length before time $\mathsf{M} < \mathsf{L} < \mathsf{T}$). Any dimension may be given as a finite product of distinct fundamental dimensions, in the chosen order, raised to nonzero integral powers. Hence to check equality of dimensions $d \overset{?}{=} d'$, we can reduce d and d' to normal forms $d = \mathsf{M}^{n_0} \cdot \mathsf{L}^{n_1} \cdot \mathsf{T}^{n_2}$, $d' = \mathsf{M}^{n_0'} \cdot \mathsf{L}^{n_1'} \cdot \mathsf{T}^{n_2'}$, and then straightforwardly check equality of the exponents $n_i \overset{?}{=} n_i'$, rather than applying the group axioms directly.

With equality of dimensions in place, the crucial step in making dimension checking part of type checking is now to allow *abstract* dimensions [WO91]: addition is not length-specific, but works in one *arbitrary* dimension, which can stand for any dimension in particular. Similarly, multiplication and division of quantities multiplies and divides arbitrary dimensions respectively. By giving addition and multiplication these types, and taking our refined notion of equality of dimensions into account, dimension checking simply becomes type checking. Gundry [Gun11] has shown that the property of programs still having most general types is retained in this setting.

As discussed by e.g. Hall [Hal22], dimension checking seems to be more fundamental than "unit checking". When dimensions are encoded in types, units can be introduced as "smart constructors" such as Watt `_W` : $\mathbb{R} \to Q\,(\mathsf{ML}^2\mathsf{T}^{-3})$. If this is the only way to introduce quantities, we can ensure that only meaningful expressions enter the system. Similarly, by only allowing the extraction of an actual number at dimensionless types (which can for example be achieved by dividing a quantity by a unit constant), only physically meaningful information can flow out of the system.

4. USING TYPES TO AUTOMATICALLY GENERATE CODE

Types are not just a stick to be beaten with when one makes a mistake; they can also act as a carrot, for example by enabling code generation. As a simple demonstration of this principle, we have developed a program that automatically generates code for reading and validating input data based on type declarations. The implementation is available at `https://github.com/g-nakov/mgen`.

Many metrology software packages come with careful

```
ivals :
  nlayer : contains the @number of layers (2 or 3) in the sample
  lams   :  array of thermal conductivities of layer @nlayer (in @W m^-2 K^-1).
  kappas : contains radius of
    - sample (in @cm)
    - laser (in @mm)
    - measuring (in @mm)
  Rs : heat transfer coefficient for losses in  @W m^-2 K^-1
    - from front face
    - from rear face
    - curved side face
  cps : specific heat capacities in @J kg^-1 K^-1
    - of the front face
    - of the rear face
    - of the curved side face
  tflash : duration of laser flash in @ms
```

Fig. 1: Formal input data description example

descriptions of input formats in their documentation, usually describing what input is required (e.g. "thermal conductivities"), in what form (e.g. "an array with an entry for each layer"), and in what unit (e.g. "$\mathsf{Wm^{-2}K^{-1}}$"). However these are written for humans, not machines, and consequently the code to read the inputs and convert them to the internal units used, if applicable, is also written by humans. This is typically fiddly code, with perhaps nested loops, and many opportunities for off-by-one errors to slip in.

Our approach is instead to make the description of the input format formal, so that it can be understood by a machine, which can then write the code for reading the inputs. In practice, this requires minimal changes to the description — mostly ensuring that the required data is actually present.

An example input description is displayed in Fig. 1. An input is declared with its name (for example `ivals` and `nlayer`), followed by a colon '`:`', followed by its description, which is for the benefits of humans. An input is either a composite object (such as `ivals`), a scalar field (such as `nlayer`), or an array (such as `kappas`). Details about inputs which are important for the machine are tagged with an `@` symbol, such as if an input is a number (for example, `nlayer` is tagged as a `@number`), or an array of a certain length (for example, `lams` is tagged as an array of size `@nlayer`). Later inputs can refer to earlier inputs for their description (for example, the description of `lams` refer to `nlayer`) — we are making full use of dependent types by allowing later entries to *depend* on earlier ones. Each non-number field entry has a unit attached to it, again indicated by an `@` symbol. These can either be attached to individual fields of an array (such as for the array `kappas`), or uniformly for the whole array (such as the array `Rs`). Also note that we allow SI derived units such as Watt `W` — we convert these to their standard form in terms of SI base units internally.

Given an input description, we first validate that it is sensible: that array lengths are numbers, that field names are not repeated, and that each scalar field has a unit. This way, we can catch simple mistakes in the input description such as typos or undeclared input fields.

After validating the input description, we can generate code for reading input data following it. We currently generate Matlab code, but there is nothing Matlab specific about what we do — it would be possible to generate code for most programming languages. For the input description from Fig. 1, we generate the following code:

```
function ivals = getinputsfromfile(fname);
f1 = fopen(fname);
c1 = textscan(f1, '%f');
src = c1{1};
fclose(f1);

rPtr = 1;
ivals.nlayer = src[rPtr];
rPtr = rPtr + 1;
for i = 1:ivals.nlayer
  ivals.lams[i] = 1e3 * src[rPtr+i];
end
rPtr = rPtr + ivals.nlayer;
ivals.kappas[1] = 1e-2 * src[rPtr+1];
for i = 2:3
  ivals.kappas[i] = 1e-3 * src[rPtr+i];
end
rPtr = rPtr + 3;
for i = 1:3
  ivals.Rs[i] = 1e3 * src[rPtr+i];
  ivals.cps[i+3] = src[rPtr+i+3];
end
rPtr = rPtr + 6;
ivals.tflash = 1e-3 * src[rPtr];
```

We make sure to generate fresh variable names for the read pointer `rPtr`, the file handle `f1` and the file contents `c1` and `src`. The rest of the names are guaranteed to be non-clashing, since we have validated the description. We then sequentially read the data, advancing the read pointer as we go along. We use the unit information to scale data into the units used internally in the program. Note also that we have taken the opportunity to merge the loops for `ivals.kappas` and `ivals.Rs` into a single loop. These are exactly the kind of code transformations that are easy to get wrong if done manually — in contrast, we can reason generically that this transformation will always be correct. As a result, the generated code looks like similar to code that one would write by hand, but without the risk of making for example an off-by-one error somewhere.

5. CONCLUSIONS AND FUTURE WORK

Type systems could be a powerful tool in the digitalisation of metrology. By exploiting advances in dependent type systems, we have shown that we can strengthen our ability to reason about dimensional correctness, and also bridge the gap between human-readable semantic specifications of data, and the actual code representing it in a specific programming environment. Crucially, we were able to reap these benefits with minimal additional costs — we put to good use already existing typecheckers without having to rewrite the infrastructure in place from scratch.

We have chosen a straightforward treatment of dimensions as elements of a free group, and units as constants; this choice does not accurately disambiguate for example radians $\mathtt{rad} = \mathsf{mm}^{-1}$ and square radians $\mathtt{sr} = \mathsf{m}^2\mathsf{m}^{-2}$, even though they are of very different nature. However we stress that this is not an inherent limitation in the methodology of using types for dimensions — dimensionless quantity ratios can if necessary be tracked separately in types, using the same principles as presented here, which we hope to do in the future. Overall, the work reported here is part of a larger project to incorporate dependent types in Matlab programs for correctness checking, including dimensional correctness.

ACKNOWLEDGEMENTS

Thanks to Alistair Forbes, Keith Lines and Ian Smith for discussions about this work. Funding: supported by the UK National Physical Laboratory Measurement Fellowship project "Dependent types for trustworthy tools".

REFERENCES

[BBM18] Oscar Bennich-Björkman and Steve McKeever. The next 700 unit of measurement checkers. In *SLE '18*, pages 121–132. ACM, 2018.

[BD09] Ana Bove and Peter Dybjer. *Dependent Types at Work*, pages 57–99. Springer, 2009.

[DKS21] Jean-Marie Dautelle, Werner Keil, and Otavio Santana. JSR 385: Units of measurement. `https://unitsofmeasurement.github.io/`, 2021.

[Fos13] Marcus P. Foster. Quantities, units and computing. *Computer Standards & Interfaces*, 35(5):529–535, 2013.

[Gun11] Adam Gundry. Type inference for units of measure. In *Pre-proceedings of the 12th International Symposium on Trends in Functional Programming (TFP'11)*, pages 17–35, 2011.

[Gun15] Adam Gundry. A typechecker plugin for units of measure: Domain-specific constraint solving in GHC Haskell. In *Haskell '15*, pages 11–22. ACM, 2015.

[Hal20] B. D. Hall. Software for calculation with physical quantities. In *2020 IEEE International Workshop on Metrology for Industry 4.0 IoT*, pages 458–463, 2020.

[Hal22] B. D. Hall. *Software representation of measured physical quantities*, pages 273–284. World Scientific, 2022.

[Ken95] Andrew Kennedy. *Programming languages and dimensions*. PhD thesis, University of Cambridge, United Kingdom, 1995.

[Ken10] Andrew Kennedy. *Types for Units-of-Measure: Theory and Practice*, pages 268–305. Springer, 2010.

[Mat22] Mathworks. MATLAB units of measurement. `https://mathworks.com/help/symbolic/units-of-measurement.html`, 2022.

[ME14] Takayuki Muranushi and Richard Eisenberg. Experience report: Type-checking polymorphic units for astrophysics research in Haskell. In *Haskell '14*, pages 31–38. ACM, 2014.

[MNF22] Conor McBride and Fredrik Nordvall Forsberg. Type systems for programs respecting dimensions. In Franco Pavese, Forbes Alistair, Nien-Fan Zhang, and Anna Chunovkina, editors, *Advanced Mathematical and Computational Tools in Metrology and Testing XII*, volume 90 of *Advances in Mathematics for Applied Sciences*, pages 331–345. World Scientific, 2022.

[pin22] Pint: makes units easy. `https://pint.readthedocs.io/`, 2022.

[Sim94] Charles C. Sims. *Computation with Finitely Presented Groups*, volume 48 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1994.

[SW10] Matthias C. Schabel and Steven Watanabe. Boost C++ libraries, chapter 42 (Boost.Units 1.1.0). `https://www.boost.org/doc/libs/1_79_0/doc/html/boost_units.html`, 2010.

[WO91] Mitchell Wand and Patrick O'Keefe. Automatic dimensional inference. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 479–486. MIT Press, 1991.