ACTA IMEKO
ISSN: 2221-870X
March 2023, Volume 12, Number 1, 1 - 5

Measuring with confidence: leveraging expressive type systems for correct-by-construction software

Conor McBride[1], Georgi Nakov[1], Fredrik Nordvall Forsberg[1]

[1] *Department of Computer and Information Sciences, University of Strathclyde, Glasgow, UK*

ABSTRACT
Modern programming language type systems help programmers write correct software, and furthermore helps them write the software they actually 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.

Section: RESEARCH PAPER

Keywords: type systems; correctness; metrology; programming languages

Citation: Conor McBride, Georgi Nakov, Fredrik Nordvall Forsberg, Measuring with confidence: leveraging expressive type systems for correct-by-construction software, Acta IMEKO, vol. 12, no. 1, article 15, March 2023, identifier: IMEKO-ACTA-12 (2023)-01-15

Section Editor: Daniel Hutzschenreuter, PTB, Germany

Received November 19, 2022; **In final form** February 28, 2023; **Published** March 2023

Funding: supported by the UK National Physical Laboratory Measurement Fellowship project “Dependent types for trustworthy tools”.

Corresponding author: Georgi Nakov, e-mail: georgi.nakov@strath.ac.uk

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…

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 data into machine-readable form, we can not only check that given data conforms to the format, but automatically generate 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 [1], [2], but we put additional emphasis on the use of types as a convenient method of communication between human and machine. Thus, our focus is on how dependently typed programming languages can provide a sound basis for developing software for metrology rather than describing any such software in detail.

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*?

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* [3], 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, just like it is not data type correct to add an integer and a string. 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 [4] for more):

- Microsoft's F# [5] has units of measure built-in to static type checking;
- C++'s Boost Units library [6] uses templates to check units statically;
- Java has a proposed API adding classes for dimensioned quantities [7], but run-time casts are inevitable;
- Haskell's type system can now encode basic units of measure as a library [8] or a typechecker plugin [9];
- Python libraries such as Pint [10] 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 [11].

Many of these solutions however provide no static guarantees, or rely on rather ad-hoc extensions of the type system, often with unintelligible error messages as a result. Encoding dimensions using dependent types is a more principled way to include dimension checking in a programming language. In the rest of this section, we briefly describe how we implemented a typechecker including dimensions [12].

First, how are we to represent dimensions themselves? Following Kennedy [13], 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 $M \cdot L$ is the same as length times mass $L \cdot M$). These considerations lead us to model dimensions as elements of the free Abelian group over the set of fundamental dimensions D [14].

For type checking, 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 $M < L < T$). 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 to also allow *abstract* dimensions [15]: 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 [16] has shown that the property of programs still having most general types is retained in this setting.

As discussed by e.g., Hall [17], 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 $\_\mathsf{W} \colon \mathsf{R} \to Q(\mathsf{M}\ \mathsf{L}^2\ \mathsf{T}^{-3})$ — the type of this function says that for any real number r, we get a quantity

```
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 faced
    - of the curved side face
  tflash : duration of laser flash in @ms
```

Figure 1. Formal input data description.

r W of dimension M L^2 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.

The described approach is agnostic in the concrete choice of fundamental dimensions; indeed, if one wants to distinguish between torque and energy, for example, which both have the same dimension M L^2 T^{-2} in the International System of Units (SI) [18], then one could introduce separate dimensions for them. A more principled approach would be to extend our system utilising *aspects* from the M-layer representation of quantities [19], but we use a conceptually simple representation here, as our aim is to demonstrate how dimension checking can be internalised as type checking.

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 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., “W m^{-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 Figure 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 and Python code, but there is nothing particular about the choice of languages — it would be possible to cover most programming languages. For the input description from Figure 1, we generate the following Matlab code — the corresponding Python code can be found in Figure 2.

```
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];
```

```
class Ivals():
  __slots__ = ("lams", "kappas", "rs", "cps", "tflash", "nlayer")

  def __init__(self):
    self.lams = {}
    self.kappas = {}
    self.rs = {}
    self.cps = {}

ivals = Ivals()
with open(fname, 'r') as f1:
  src = f1.readlines()
  rPtr = 0
  ivals.nlayer = int(src[rPtr])
  rPtr = rPtr + 1
  ivals.tflash = 1e-3 * float(src[rPtr])
  rPtr = rPtr + 1
  for i in range(0, 3):
    ivals.cps[i] = float(src[rPtr + i])
    ivals.rs[i+3] = 1e3 * float(src[rPtr + i + 3])
  rPtr = rPtr + 6
  ivals.kappas[0] = 1e-2 * float(src[rPtr + 0])
  for i in range(1, 3):
     ivals.kappas[i] = 1e-3 * float(src[rPtr+i])
     rPtr = rPtr + 3
  for i in range(0, ivals.nlayer):
    ivals.lams[i] = 1e3 * float(src[rPtr + i])
```

Figure 2. Generated Python code from the data description in Figure 1.

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.

Another guiding principle for our code generation tool is convenience of use of the input data afterwards — the whole exercise would be pointless if the user manually would have to convert the data into a different data type after reading. Hence, we make sure to store the data in a data type that is "natural" for the chosen language, and which allows for a direct access of the fields following the input data description. In Matlab, *structure arrays* readily meet these requirements, but we have to take extra care to explicitly generate the needed classes in Python.

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 $\mathrm{rad}\ = \mathrm{m}\,\mathrm{m}^{-1}$ and square radians $\mathrm{sr} = \mathrm{m}^2\,\mathrm{m}^{-2}$, 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. The authors would also like to thank the attendants of the first IMEKO TC6 International Conference on Metrology and Digital Transformation for the useful suggestions and comments on a presentation of this work, in particular the suggestion to extend code generation also to Python. Funding: supported by the UK National Physical Laboratory Measurement Fellowship project "Dependent types for trustworthy tools".

6. REFERENCES

[1] M. P. Foster, Quantities, units and computing, Computer Standards & Interfaces 35 (2013), pp. 529–535. DOI: 10.1016/j.csi.2013.02.001

[2] B. D. Hall, Software for calculation with physical quantities, 2020 IEEE International Workshop on Metrology for Industry 4.0 IoT, Rome, Italy, 2020, pp. 458–463. DOI: 10.1109/MetroInd4.0IoT48571.2020.9138281

[3] A. Bove, P. Dybjer, Dependent types at work, LerNet ALFA Summer School 2008 Revised Tutorial Lectures (2009), pp. 57–99. DOI: 10.1007/978-3-642-03153-3_2

[4] O. Bennich-Björkman, S. McKeever, The next 700 Unit of measurement checkers, 11th ACM SIGPLAN Int. Conference on

Software Engineering, Boston, USA, 2018. pp. 121–132. DOI: 10.1145/3276604.3276613

[5] A. Kennedy, Types for units-of-measure: theory and practice. Central European Functional Programming School 2009, Revised Selected Lectures (2010), pp. 268–305. DOI: 10.1007/978-3-642-17685-2_8

[6] M. C. Schabel, S. Watanabe, Boost C++ Libraries, Chapter 42 (Boost.Units 1.1.0), 2010. Online [Accessed 13 January 2023] https://www.boost.org/doc/libs/1_81_0/doc/html/boost_units.html

[7] J.-M. Dautelle, W. Keil, O. Santana, JSR 385: Units of measurement, 2021. Online [Accessed 13 January 2023] https://unitsofmeasurement.github.io/pages/about.html

[8] T. Muranushi, R. Eisenberg, Experience report: type-checking polymorphic units for astrophysics research in Haskell, 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, 2014, pp. 31–38. DOI: 10.1145/2633357.2633362

[9] A. Gundry, A typechecker plugin for units of measure: domain-specific constraint solving in GHC Haskell, 2015 ACM SIGPLAN symposium on Haskell, Vancouver, Canada, 2015, pp. 11–22. DOI: 10.1145/2804302.2804305

[10] Pint Developers, Pint: makes units easy, 2022. Online [Accessed 13 January 2023] https://pint.readthedocs.io/

[11] Mathworks, MATLAB units of measurement, 2022. Online [Accessed 13 January 2023] https://www.mathworks.com/help/symbolic/units-of-measurement.html

[12] C. McBride, F. Nordvall Forsberg, Type systems for programs respecting dimensions, in: Advanced Mathematical and Computational Tools in Metrology and Testing XII. F. Pavese, A. B. Forbes, N. F. Zhang, A. G. Chunovkina (editors). World Scientific, Singapore, 2022, ISBN 978-981-124-237-3, pp. 331–345. DOI: 10.1142/9789811242380_0020

[13] A. Kennedy, Programming languages and dimensions, Ph.D. dissertation, University of Cambridge, 1995.

[14] C. C. Sims, Computation with Finitely Presented Groups, Cambridge University Press, Cambridge, 1994, ISBN 978-051-157-470-2. DOI: 10.1017/CBO9780511574702

[15] M. Wand, P. O'Keefe, Automatic dimensional inference, in: Computational Logic: Essays in Honor of Alan Robinson. J.-L. Lassez, G. Plotkin (editors), MIT Press, Cambridge, Massachusetts, 1991, ISBN 978-0-262-12156-9, pp. 479–486.

[16] A. Gundry, Type inference, Haskell and dependent Types, Ph.D. dissertation, University of Strathclyde, 2013.

[17] B. D. Hall, Software representation of measured physical quantities, in: Advanced Mathematical and Computational Tools in Metrology and Testing XII. F. Pavese, A. B. Forbes, N. F. Zhang, A. G. Chunovkina (editors). World Scientific, Singapore, 2022, ISBN 978-981-124-237-3, pp. 273-284. DOI: 10.1142/9789811242380_0016

[18] BIPM, The International System of Units ('The SI Brochure') (Ninth ed.). Bureau International des Poids et Mesures, 2019. Online [Accessed 13 January 2023] http://www.bipm.org/en/si/si_brochure/

[19] B. D. Hall, M. Kuster, Representing quantities and units in digital systems, Measurement: Sensors 23 (2022), 6 pp. DOI: 10.1016/j.measen.2022.100387