University of Kent

Kent Academic Repository

Danish, Matthew, Allamanis, Miltiadis, Brockschmidt, Marc, Rice, Andrew and Orchard, Dominic A. (2019) *Learning Units-of-Measure from Scientific Code*. In: 2019 IEEE/ACM 14th International Workshop on Software Engineering for Science (SE4Science). Proceedings: 2019 IEEE/ACM 14th International Workshop on Software Engineering for Science (SE4Science 2019). . pp. 43-46. IEEE ISBN 978-1-7281-2277-9. E-ISBN 978-1-7281-2276-2.

Downloaded from

https://kar.kent.ac.uk/79924/ The University of Kent's Academic Repository KAR

The version of record is available from

https://doi.org/10.1109/SE4Science.2019.00013

This document version

Author's Accepted Manuscript

DOI for this version

Licence for this version

UNSPECIFIED

Additional information

Versions of research works

Versions of Record

If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

Author Accepted Manuscripts

If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in ***Title of Journal*** , Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries

If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our Take Down policy (available from https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies).

Learning units-of-measure from scientific code

Matthew Danish
University of Cambridge
Cambridge, UK
mrd45@cam.ac.uk

Miltiadis Allamanis
Microsoft Research
Cambridge, UK
miallama@microsoft.com

Marc Brockschmidt
Microsoft Research
Cambridge, UK
mabrocks@microsoft.com

Andrew Rice
University of Cambridge
Cambridge, UK
acr31@cam.ac.uk

Dominic Orchard
University of Kent
Canterbury, UK
d.a.orchard@kent.ac.uk

***Abstract*—CamFort is our multi-purpose tool for lightweight analysis and verification of scientific Fortran code. One core feature provides units-of-measure verification (dimensional analysis) of programs, where users partially annotate programs with units-of-measure from which our tool checks consistency and infers any missing specifications. However, many users find it onerous to provide units-of-measure information for existing code, even in part. We have noted however that there are often many common patterns and clues about the intended units-of-measure contained within variable names, comments, and surrounding code context. In this work-in-progress paper, we describe how we are adapting our approach, leveraging machine-learning techniques to reconstruct units-of-measure information automatically thus saving programmer effort and increasing the likelihood of adoption.**

***Index Terms*—units-of-measure, verification, machine learning**

I. Introduction

Scientific computing and computational models are playing an increasingly important role in research, industry, and policy. However, such programs are just as error prone as other kinds of software and their reliance on complex numerical routines makes testing difficult. Very simple errors, such as a flipped minus sign, can have significant impact, e.g. leading to retractions from premier journals [1]. Lightweight verification techniques have the potential to help in this situation through tools that have a low specification burden [2]. Since scientific models usually handle numerical quantities with physical meaning, a helpful lightweight technique is to statically verify the consistency of units-of-measure in computations. Units-of-measure mistakes have famously led to high-stakes disasters, such as the NASA Mars Climate Orbiter disappearing during orbital insertion. An investigation pinned the blame on a units-of-measure conflict, with ground-based software sending numbers in Imperial units instead of metric units, dropping the probe into the atmosphere of Mars [3].

Checking the consistency of dimensions, or their refinement as units-of-measure, is a method long employed by scientists working with pen and paper [4]. Informal discussion with scientists and examination of scientific program code has shown that dimensionality and units-of-measure reasoning in code does take place, but usually only by hand [5]. Formal dimensions/units checking has not been widely adopted in scientific programming despite the familiarity of the technique and a multitude of proposed systems for automated verification [6, 7, 8, 9, 10]. One potential explanation for this is the burden of applying units annotations to existing codebases.

In this work-in-progress paper, we describe how we are exploring the idea of using machine learning techniques to automatically infer units-of-measure annotations. We give examples from existing open-source models demonstrating potential sources of information and we identify challenges arising from the shortage of labelled training data and the importance of a good user experience.

a) Static analysis for units-of-measure: CamFort is an open-source Fortran analysis and verification tool [11] that provides units-of-measure inference and checking as one of its core features. To do this it generates and solves constraints derived from program variable relationships. Typically, the high level of interdependence between variables means that units-of-measure can be inferred for most variables from a relatively small set of programmer-provided annotations. In one study of real-world programs [12], approximately 80% of annotations could be automatically inferred if given a 'critical' subset of manual annotations. CamFort also supports polymorphic units-of-measure annotations: functions that operate generically in relation to units can be used with any units as input, provided the usage is consistent with the rest of the code.

b) Machine learning for code: Machine learning is increasingly applied to source code [13] with the goal of driving new software engineering tools and program analyses. The core enabler behind this is the observation that the vast majority of source code is more than mere instructions for a computer, it is also meant to be text that is read by the human beings who maintain the code. For example, identifier names and code comments are a rich source of information to software engineers about the implemented functionality, but meaningless to the compiler and most software analysis programs.

II. Annotating existing code-bases

As an example of CamFort's approach, Listing 1 shows the elementary equation of ballistics transcribed into Fortran with units-of-measure annotations as comments given explicitly before some of the variable's typing declarations. Units are expressed in terms of unit names (e.g., `metre`, `sec`) combined by multiplication, division, and scalar exponentiation.

CamFort provides an inference mode which attempts to infer suitable annotations for variables in a program. We can also request that CamFort automatically inserts the inferred

```
real, parameter :: x0 = 0
!= unit metre / sec :: v0
real, parameter :: v0 = 20
!= unit metre / sec**2 :: a
real, parameter :: a = -9.8
!= unit metre :: x
real :: x
real :: t
x = 0.5 * a * t * t + v0 * t + x0
```

Listing 1: Example Fortran code with CamFort units-of-measure annotations

annotations into the code at the relevant positions, to save effort. In this case, the following missing specifications are inferred for Listing 1:

```
unit metre :: x0
unit sec :: t
```

A. Inferring units-of-measure in Fortran with ML

By combining existing static analyses for inferring units-of-measure with machine learning methods, we seek to fuse ambiguous, yet useful, information from identifier names and comments with the constraints generated by the units inference system and thus mostly automate the annotation process.

a) Characteristics of the inference domain: The above example (Listing 1) demonstrates the style of specifications that are generated, where units-of-measure terms are considered a commutative group over a set of unit names.

The 'skeleton' of a units specification for a variable can usually be automatically inferred from its usage constraints. For example, if Listing 1 had no user annotations, then our system would look at the relationships in the code and would infer, for example, that variable `a` must have units that look like α/β^2 for some units α and β. This provides a skeleton (a partial annotation) that can be completed by substituting actual units (such as `metre` and `sec` respectively) into the placeholders labelled α and β, and may provide a useful starting point for more sophisticated analysis.

b) Information sources: Beyond the usual constraints generated by our inference, there are many additional sources of information that can allow us to probabilistically reason about the units-of-measure, especially in scientific code:

- Identifier names, such as variable and function names, can often be very indicative about the values they contain. For example, a variable named `distance` probably contains distance-related values. Or, as seen in Listing 2, a variable named `TEMP` is possibly temperature, but in other cases the name simply means 'temporary' variable. Notably, `distance` would imply only the dimensionality of the variable (a length) rather than its units-of-measure. However, knowing the dimensionality restricts possibilities for the unit. Machine learning techniques should easily learn the categorisation of units-of-measure based on their dimensionality, given appropriate training data.

- Scientific code is often annotated with comments that helpfully indicate relevant information with regards to the quantities and computations used, such as in Listings 2 and 3. However, such comments are free-text and cannot be canonically parsed. By employing machine learning methods that have been used in Natural Language Processing, we may be able to gain useful hints about the units-of-measure of the annotated code.

```
! Viscosity [Pa s] of air as a function of
↪  temp (K).
! Sutherland eqn. (ref. pp 25 in Hinds)
VISC = 1.458d-6 * (TEMP)**(1.5d0) / ( TEMP +
↪  110.4d0 )
```

Listing 2: Example Fortran statement from the GEOS-Chem project: `geos-chem.org`.

```
REAL BTEMP      ! in degK
REAL BPRESS     ! in Pa
REAL VSP        ! mean molecular speed (m/s)
REAL DG         ! gas-phase moleular diffusion
                ! coefficient (m2/s)
```

Listing 3: Example Fortran variable declarations extracted from the WRF project: `github.com/wrf-model/WRF`.

```
TempK    = TempC + 273.15
RT       = RGasConstant * TempK
```

Listing 4: Example code from the SHYFEM model: `github.com/SHYFEM-model/shyfem`.

- Usage patterns of variables, such as interaction with constants, provide further hints towards inferring units-of-measure. For example, adding the constant 273.15 probably indicates a conversion between Celsius and Kelvin (Listing 4). Such constants are plentiful in scientific code.

- Execution-time instrumentation of code might additionally provide useful probabilistic hints. For example, a variable of Earth-weather simulation representing temperature in Celsius will fall within a fairly limited range of values.

B. Challenges and opportunities

One important issue is that only small amounts of labelled-data are available for training. This requires us to find data-efficient methods that will allow us to learn with the minimal amount of training data. Such methods are actively being explored in machine learning and include domain adaptation, active, semi-supervised and unsupervised learning.

One way to mitigate a shortage of labelled data could be to provide to an inference algorithm a knowledge base of facts about the units-of-measure for common constants and scaling factors, e.g., Avogadro's constant. Knowledge of common scaling factors (e.g., 1000 has the unit $\mathtt{m/km}$) might also be suitably generalised to capture schemas of naming conventions (e.g. 1000 has units $u/\mathtt{k}u$ for any unit names u).

Self-supervised or unsupervised techniques can learn salient features about code elements and assist units inference methods, without annotated data. Inspired by pre-trained representation learning methods in NLP, such as BERT [14] and

ELMo [15], methods used for learning distributed vector representations (embeddings) can be repurposed for variable naming [16, 17]. These methods take advantage of the fact that variables that appear within similar contexts often have similar names, roles and functions within a program. Such methods have been shown to learn semantically rich representations and may be useful for other downstream tasks, including unit inference.

Of course, fully unsupervised methods alone cannot solve the units inference problem. Pre-trained representations however provide a starting point for semi-supervised, active learning and domain adaptation methods. Active learning aims to reduce the annotation burden by interacting with humans, minimising the number of manual annotations needed while maximising the information received per annotation. Simultaneously, domain adaptation methods might allow us to transfer learned information across programs of different domains, further reducing the necessary annotation effort. For example, despite the differences between a fluid dynamics simulator for wind turbines and a meteorological forecasting program, many physical quantities and concepts appear in both.

At the same time, contrary to many standard machine learning scenarios, inferring units-of-measure also presents a unique opportunity: the formal structure of source code can inform the learning process rendering it more data-efficient compared to other, less structured tasks. For example, an active learning approach might take advantage of the minimal set of needed unit annotations [5] to further improve the efficiency of the learning process. Unsupervised methods, such as RefiNym [18], have shown some evidence towards the importance of a program's structure. For example, RefiNym qualitatively shows that static dataflow analysis can help an unsupervised method to nominally refine numeric variables based on the physical quantities they represent; JSNICE [19] can exploit JavaScript program structure to predict variable types using a special type of factor graph. Based on those ideas, one opportunity is to combine hard constraints derived by static analysis with soft, probabilistic information within factor graphs and other structured prediction methods, taking advantage of all available information sources for units-of-measure inference.

Despite these promises, the shape and form of our data introduces a challenging setting for existing machine learning methods. For example, despite the prevalence of polymorphism in programming languages, we are not aware of any machine learning methods that can learn such a concept. Similar concepts such as hyponym and hypernym representations are only now starting to be tackled by machine learning methods in Natural Language Processing.

The final set of challenges lie with the user experience. The user interface with machine learning-generated suggestions must be convenient and simple. Although annotations can be checked for consistency, we do not expect that machine learning methods will always be able to decipher the intended units-of-measure. Instead, the uncertainty and rationale of the (probabilistic) selections needs to be accurately communicated to the user who will make the final decision about the correct annotations for each case; an interactive approach is likely.

III. Related Work

Units-of-measure inference can be considered a form of type inference. Interestingly, various authors have explored the application of machine learning to type inference. JSNICE [19] and DEEPTYPER [20] probabilistically infer types of JavaScript code through variable usage patterns and variable names. However, these methods can only predict a relatively small set of predefined types, often excluding user-defined types or other rarely used types and rely on relatively large supervised corpora of type-annotated code.

Units-of-measure annotations have been well-explored as an extension to Hindley-Milner-style type systems [8]. The most mainstream implementation is in the F# language [21]. Units of measure may be declared or aliased using a special syntax:

```
[<Measure>] type cm
```

Types and literal values are annotated directly, e.g.:

```
let cm2inch(x:float<cm>) = x / 2.54<cm/inch>
```

The types are stratified: the `float` type cannot be unified with a units-annotated `float<u>` type, only with `float<1>`. This makes transitioning existing code-bases difficult because adding units annotations to a function means that you have to annotate all of its call-sites as well. Similarly, a type-checker plugin for Haskell provides units-of-measure checking, reusing Haskell's type inference algorithm with a custom solver [22].

Several other libraries leverage their host's advanced type system to provide statically-checked units-of-measure. For example, Squants provides units-of-measure for Scala with a hierarchy of dimensions with associated units, conversion and static checking (see squants.com). C++ has Boost::Units (see boost.org) and JScience is a scientific programming package for Java that includes a units-of-measure library (see jscience.org). All of these offer statically-checked units integrated with the host language. However, this comes at the price of inference, which is not possible since the hosts' type systems cannot reason over commutative groups in the same way as F# or CamFort.

IV. Conclusion

Units-of-measure errors in software can be costly, e.g., undermining the validity of conclusions drawn from scientific models, or leading to catastrophic outcomes in control software. Formally verifying units-of-measure is therefore an attractive goal. Furthermore, verifying units statically is appealing because it imposes no runtime cost. However, adoption remains low despite the range of published approaches.

Tools like CamFort ease the burden of adding units-of-measure to existing code by providing a system of units inference. However, our experience is that the amount of annotation required is still perceived by developers as a significant overhead. Thus, as described here, our aim is to leverage techniques from machine learning to reduce this burden, allowing concrete unit annotations to automatically generated. We presented examples from existing open-source models that

show potential sources of information and explored some of the novel challenges that might arise.

We are currently working with scientists adding units-of-measure annotations manually to existing code but the shortage of labelled data remains a challenge. We are interested in exploring other ideas to address this.

V. Acknowledgements

This work has been supported by Grant EP/M026124/1 from the Engineering and Physical Sciences Research Council.

References

[1] Z. Merali, "Computational science: error, why scientific programming does not compute," *Nature*, no. 467, October 2010.

[2] D. Orchard and A. Rice, "A computational science agenda for programming language research," *Procedia Computer Science*, vol. 29, pp. 713 – 727, 2014, International Conference on Computational Science.

[3] A. G. Stephenson, D. R. Mulville, F. H. Bauer, G. A. Dukeman, P. Norvig, L. S. LaPiana, P. J. Rutledge, D. Folta, and R. Sackheim, "Mars Climate Orbiter Mishap Investigation Board phase I report," NASA, Tech. Rep., 1999.

[4] E. O. Macagno, "Historico-critical review of dimensional analysis," *Journal of the Franklin Institute*, vol. 292, no. 6, pp. 391–402, 1971.

[5] D. Orchard, A. Rice, and O. Oshmyan, "Evolving Fortran types with inferred units-of-measure," *Journal of Computational Science*, vol. 9, no. Supplement C, pp. 156 – 162, 2015, computational Science at the Gates of Nature.

[6] J.-P. Ore, C. Detweiler, and S. Elbaum, "Lightweight detection of physical unit inconsistencies without program annotations," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 341–351.

[7] L. Jiang and Z. Su, "Osprey: a practical type system for validating dimensional unit correctness of C programs," in *Proceedings of the 28th International Conference on Software Engineering*. ACM, 2006, pp. 262–271.

[8] A. J. Kennedy, "Programming languages and dimensions," University of Cambridge, Computer Laboratory, Tech. Rep., 1996.

[9] W. Snyder, "ISO/IEC JTC1/SC22/WG5 N1969: Units of measure for numerical quantities," International Organization for Standardization, Tech. Rep., 2013. [Online]. Available: http://wg5-fortran.org/N1951-N2000/N1969.pdf

[10] M. Hills, F. Chen, and G. Roşu, "A rewriting logic approach to static checking of units of measurement in C," in *Ninth International Workshop on Rule-Based Programming*. Elsevier, 2008, pp. 51–67.

[11] D. Orchard, M. Danish, M. Contrastin, and A. Rice, "CamFort," http://camfort.github.io/, 2019, accessed: 2019-01-23.

[12] D. A. Orchard, A. C. Rice, and O. Oshmyan, "Evolving Fortran types with inferred units-of-measure," *J. Comput. Science*, vol. 9, pp. 156–162, 2015.

[13] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[15] M. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proceedings of Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, vol. 1, 2018, pp. 2227–2237.

[16] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 38–49.

[17] R. Bavishi, M. Pradel, and K. Sen, "Context2Name: A deep learning-based approach to infer natural variable names from usage contexts," *arXiv preprint arXiv:1809.05193*, 2018.

[18] S. K. Dash, M. Allamanis, and E. T. Barr, "RefiNym: using names to refine types," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 107–117.

[19] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from big code," in *ACM SIGPLAN Notices*, vol. 50, no. 1. ACM, 2015, pp. 111–124.

[20] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 152–162.

[21] A. Kennedy, *Types for units-of-measure: theory and practice*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 268–305.

[22] A. Gundry, "A typechecker plugin for units of measure: domain-specific constraint solving in GHC Haskell," in *ACM SIGPLAN Notices*, vol. 50, no. 12. ACM, 2015, pp. 11–22.