Cronfa - Swansea University Open Access Repository

_____

This is an author produced version of a paper published in:
Proceedings of 14th International Workshop on Automated Verification of Critical Systems

_____

Cronfa URL for this paper:

http://cronfa.swan.ac.uk/Record/cronfa43773

_____

Conference contribution :

Donaghy, D. & Crick, T. (2014). *Physical Type Tracking through Minimal Source-Code Annotation.* Proceedings of 14th International Workshop on Automated Verification of Critical Systems, Enschede, Netherlands: University of Twente.

_____

http://www.swansea.ac.uk/library/researchsupport/ris-support/

Proceedings of the
14th International Workshop on
Automated Verification of Critical Systems (AVoCS 2014)

Physical Type Tracking through Minimal Source-Code Annotation

Dave Donaghy and Tom Crick

3 pages

Physical Type Tracking through Minimal Source-Code Annotation

Dave Donaghy[1] and Tom Crick[2]

[1] dave.donaghy@hp.com
HP Bristol, UK

[2] tcrick@cardiffmet.ac.uk
Department of Computing
Cardiff Metropolitan University, UK

Abstract: One of many common artefacts of complex software systems that often needs to be tracked through the entirety of the software system is the underlying type to which numerical variables refer. Commonly-used languages used in industry provide complex mechanisms through which general objects are associated to a given type: for example, the *class* (and *template*) mechanisms in Python (and C++) are extremely rich mechanisms for the construction of types with almost entirely arbitrary associated operation sets.

However, one often deals with software objects that ultimately represent numerical entities corresponding to real-world measurements, even through standardised SI units: metres per second, kilogram metres per second-squared, etc. In such situations, one can be left with insufficient and ineffective type-checking: for example, the C *double* type will not prevent the erroneous addition of values representing velocity (with SI units *metre per second*) to values representing mass (SI unit *kilogram*).

We present an addition to the C language, defined through the existing `attribute` mechanism, that allows automatic control of physical types at compile-time; the only requirement is that individual variables be identified at declaration time with appropriate SI (or similar) units.

Keywords: Verification, Software Engineering, Type-Checking, Units, Compilers, Plug-ins

1 Introduction

Large (and indeed small) software systems typically track data, stored in a variety of different types; in fact, "bytes in, bytes out" is a fairly accurate description of a massive portion of the functionality of large software systems. While this is clearly true, it is of course an extraordinary over-simplification: the nature of the data we track through software systems ultimately maps all of the data that humankind has, can, or ever will, encompass.

It is not much of an understatement to say that tracking the types and content of these data represents the whole job of software development. Indeed, whole paradigms (for example, object-oriented development) may be thought of as addressing this one significant issue. However, such paradigms, while rich and functional, can also be cumbersome.

2 Simple Techniques for Data Tracking

In certain scenarios, the nature of the data we track may make it amenable to simpler representation: from a mathematical point of view, while a C++ class representing a command-line instruction to be parsed and executed *is not* a mathematical object in any useful sense; whereas a C++ object representing the distance from a geographical point to another most assuredly is.

Nevertheless, it is common to either use the same complex, powerful techniques to track these mathematical objects as to track non-mathematical ones; or indeed not to effectively track them at all. To contrast that idea, it might be possible to track these fundamentally mathematical items in C-like languages in ways that allow minimal additional effort at development time, and no maintenance effort at all.

For example, there is no meaningful way in which one can (or should) add 10 metres to 20 kilograms [Fos13]; there *is*, however, a way in which we can multiply the two: the result has a value of 200, and a unit of "kilogram metre" (one has to understand *slightly* more physics that the authors to recall the precise physical meaning of a unit of "kilogram metre", but one certainly exists) [Bur14]. Units-of-measure are to science what types are to programming; over the years, a number of attempts have been suggested to extend programming languages with support for static checking of units-of-measure [Ken97, EB02, Ken09], alongside bespoke languages (e.g. Frink [Eli08]).

3 Implicit Type Operation and Restricted Types

Imagine, then, that a variable declaration can be tagged with a *unit*; that two variables can be arbitrarily multiplied; that two variables can be added if and only if their underlying *units* match exactly. With these abilities in place, we could simply manipulate all physical types (kilograms, metres, farads etc) as built-in numerical types, without resorting to the complex *class* mechanisms that we might otherwise need.

We might call such a declaration a *restricted type*: being based on a built-in type (or indeed, any arbitrary type) it is then further restricted by constraining the operations allowed on it: no additional software is written to define the type, and the only distinction between it and the type from which it is defined is this: that with the new type, either the software compiled and behaves identically, or it does not compile at all [DC14]. While similar features exist for certain programming paradigms (for example, functional e.g. Haskell), there are significant opportunities for their implementation in the languages commonly used for industrial software development. Such a feature could improve the code quality for critical systems, as well as detecting errors earlier, lowering software development costs.

4 Open Questions

We can ask the following questions to frame future work in this area:

1. Which existing definitions of complex types (for example, C++ classes) can be replaced with restricted types?

2. By how much might software development effort be reduced using such techniques?

3. How much more effective might such techniques be at detecting software errors?

Bibliography

[Bur14] Bureau International des Poids et Mesures. The International System of Units (SI). http://www.bipm.org/en/si/, 2014.

[DC14] D. Donaghy, T. Crick. No-Test Classes in C through Restricted Types. In *Proceedings of the 14th International Workshop on Automated Verification of Critical Systems (AVoCS 2014)*. 2014.

[EB02] M. Erwig, M. M. Burnett. Adding Apples and Oranges. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL'02)*. Lecture Notes in Computer Science 2257, pp. 173–191. Springer, 2002.

[Eli08] A. Eliasen. Frink. http://futureboy.us/frinkdocs, 2008.

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

[Ken97] A. J. Kennedy. Relational parametricity and units of measure. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*. Pp. 442–455. ACM Press, 1997.

[Ken09] A. J. Kennedy. Types for units-of-measure: theory and practice. In *Proceedings of the 3rd Central European Functional Programming School (CEFP'09)*. Lecture Notes in Computer Science 6299, pp. 268–305. Springer, 2009.