Electronic Communications of the EASST
Volume X (2014)

Proceedings of the Automated Verification of Critical Systems (AVoCS 2013)

Physical Type Tracking through Minimal Source-Code Annotation

Dave Donaghy and Tom Crick

2 pages

Guest Editors: Helen Treharne, Steve Schneider
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

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 business 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 aribtrary associated operation sets.

However, one often deals with software objects that ultimately represent numerical entities corresponding to real-world measurements, even through very 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 precent the erroneous addition of values representing speed (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: compiler, plug-in, verification

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.)

This is true, but 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 under-statement to say that tracking the types and content of hese data represents the whole job of software development. Indeed, whole paradigms (for example, object oriented development) may be thought of as addressing this one 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 add 10 metres to 20 kilograms; 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 know *slightly* more physics that the authot to recall the physical meaning of a unit of ”kilogram metre”, but there is one.)

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 iff 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 thethe 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 restricting the operations allowed on it: no additional sofwtare 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.

4 Open Questions

We can ask the following questions:

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