Journal of Computational Science 9 (2015) 156–162

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Evolving Fortran types with inferred units-of-measure

Dominic Orchard [a,*], Andrew Rice [b], Oleg Oshmyan [b]

[a] Department of Computing, Imperial College London, United Kingdom
[b] Computer Laboratory, University of Cambridge, United Kingdom

ARTICLE INFO

Article history:
Available online 18 April 2015

Keywords:
Units-of-measure
Dimension typing
Type systems
Verification
Code base evolution
Fortran
Language design

ABSTRACT

Dimensional analysis is a well known technique for checking the consistency of equations involving physical quantities, constituting a kind of type system. Various type systems for dimensional analysis, and its refinement to units-of-measure, have been proposed. In this paper, we detail the design and implementation of a units-of-measure system for Fortran, provided as a pre-processor. Our system is designed to aid adding units to existing code base: units may be polymorphic and can be inferred. Furthermore, we introduce a technique for reporting to the user a set of *critical variables* which should be explicitly annotated with units to get the maximum amount of unit information with the minimal number of explicit declarations. This aids adoption of our type system to existing code bases, of which there are many in computational science projects.

1. Introduction

Type systems are one of the most popular static techniques for recognizing and rejecting large classes of programming error. A common analogy for types is of physical quantities (*e.g.*, in [2]), where type checking excludes, for example, the non-sensical addition of non-comparable quantities such as adding 3 m to 2 J; they have different *dimensions* (length *vs.* energy) and different *units* (metres *vs.* joules). This analogy between types and dimensions/units goes deeper. The approach of *dimensional analysis* checks the consistency of formulae involving physical quantities, acting as a kind of type system (performed by hand, long before computers). Various automatic type-system-like approaches have been proposed for including dimensional analysis in programming languages (*e.g.* [10] is a famous paper detailing one such approach, which also cites much of the relevant history of other systems).

Failing to ensure that the dimensions (or units) of values are correctly matched can be disastrous. An extreme example of this is the uncaught unit mismatch which led to the destruction of the Mars Climate Orbiter [20]. Many programs in computational science are also sensitive to this kind of error since they focus on modelling the physical world. The software for the Mars Orbiter had orders of magnitude more resources devoted to the robustness and correctness of code than is possible in normal scientific research circumstances. It therefore seems inevitable that these errors are likely in computational science too.

The importance of units is often directly acknowledged in source code. We have seen source files carefully commented with the units and dimensions of each variable and parameter. We have also watched programmers trying to use this information: a process of scrolling up and down, repeatedly referring to the unit specification of each parameter. Incorporating units into the type system would move the onus of responsibility from the programmer to the compiler.

A recent ISO standards proposal (N1969) for Fortran introduces a units-of-measure system which follows Fortran's tradition of explicitness [7]. Every variable declaration must have an explicit unit declaration and every composite unit (*e.g.*, metres times seconds) must itself be explicitly declared. This imposes the extra burden of annotating variables directly on the programmer. As an example, we studied two medium-sized models (roughly 10,000 lines of code each) and found roughly a 1:10 ratio between variable declarations and lines of code. Thus, adding explicit units of measure to a project with 10,000 lines of code means manually adding 1000 unit declarations. This is prohibitively large.

In this paper, we show how the bulk of this work can be done automatically based on a few manual annotations. This approach might be used to automatically add N1969 annotations to a codebase or in an Integrated Development Environment (IDE) to inform the programmer of the units as they code. Our approach is to add a validation step prior to compilation: our tool takes annotated Fortran code and validates the units. The annotations can then be automatically removed and the program compiled as normal using the preferred compiler.

* Corresponding author.
E-mail addresses: d.orchard@imperial.ac.uk (D. Orchard), andrew.rice@cl.cam.ac.uk (A. Rice).

http://dx.doi.org/10.1016/j.jocs.2015.04.018

We describe a lightweight extension to Fortran's type system for polymorphic units-of-measure (Section 2) and explain the inference process which reduces the amount of explicit declaration required (Section 3). By default, it is always possible to infer all variables as "unitless" if no explicit unit declarations are given. However, this is not useful. In order to minimise the task of adding explicit unit declarations, our system can automatically identify a minimal set of variables for which an explicit annotation is needed (Section 4). We evaluate our approach on a number of small but useful examples Section 5) and show we can reduce the burden of explicit annotation by roughly 80%. We compare our approach with existing proposals and argue that our system is more lightweight and requires less programmer effort (Section 6).

The general idea and approach of inferring units-of-measure is already well established. Instead the contribution of this paper is in the application of this technique to Fortran and existing code base, helping to evolve the language and co-evolve existing code via inference and our method for identifying which variables require manual annotation.

The type checker, inference, and analysis described here are implemented as part of the CamFort project, a research infrastructure for the analysis, transformation, refactoring, and extension of Fortran [14]. CamFort is open-source and available online.[1] Our long term interest is in how software engineering interacts with the scientific method and how techniques from programming language theory and design can be beneficially applied [15]. The present paper is a contribution in this space.

Example. Fig. 1 shows a simple Fortran program which computes (one-dimensional) velocity (`v`) and speed (`s`) from a given distance (`x`) and time (`t`). As a use case of our tool, the programmer initially runs the analysis phase of CamFort (Fig. 1(a)) and is told that only `x` and `t` need be annotated. Fig. 1(b) shows the syntax used by the programmer to add `m` (metres) and `s` (seconds) units respectively to the distance and time variables. CamFort then infers the units of `v` and `s` automatically from the program itself and inserts those into the code (without disturbing any formatting/comments).

2. Units-of-measure for Fortran

Unit attributes In our extensions, units-of-measure can be explicitly declared for variables similarly to types and other attributes of variables. Our extension adds the attribute `unit`, which is shown in the above example (Fig. 1). The `unit` attribute takes a single unit expression as an argument, the syntax of which is defined by the following grammar (where the right-hand side shows an example of the syntax):

(*grammar*)		(*description*)	(*example*)
name ::=	$[a-zA-Z]+$	*unitnames; regularexpression*	*m, metres*...
$\mathbb{R}$ ::=	$\mathbb{Z}$	*integerconstants*	$1, 2, -2\ldots$
|	$\mathbb{Z}/\mathbb{Z}$	*fractionoftwointegers*	$2/3, 4/2\ldots$
u, v ::=	ϵ	*empty— equivalenttounitless*	x
|	1	*unitless*	$unit(1) :: x$
|	*name*	*unitidentifier*	$unit(m) :: x$
|	$u**(\mathbb{R})$	*rationalpower*	$unit(s**(1/2)) :: x$
|	$u\,v$	*product*	$unit(ms**2) :: x$
|	u/v	*division*	$unit(m/s**3) :: x$

Identifiers for unit names are not themselves explicitly declared. For example, a unit attribute `unit(m)` implicitly introduces the unit named `m` to the program, where any other uses of `m` as a unit in the program denote the same unit.

[1] http://www.cl.cam.ac.uk/research/dtg/naps.

```
real :: x, t, v, s                    Critical variables:
v = x / t        --analysis-->        line 1: x, t
s = abs(v)
```

(a) Step 1: CamFort reports on critical variables for annotation

```
real, unit(m) :: x                    real, unit(m) :: x
real, unit(s) :: t                    real, unit(s) :: t
real          :: v, s  --inference and output-->  real, unit(m/s) :: v, s
v = x / t                             v = x / t
s = abs(v)                            s = abs(v)
```

(b) Step 2: CamFort infers unit declarations for remaining variables

Fig. 1. Example.

A `unit` attribute can be given to any type, not just numerical types (this differs from others, *e.g.*, [10]). In practice, numerical types tend to benefit the most from unit attributes, but there are some situations where it is useful to ascribe units to non-numerical types, *e.g.*, to string representations of numerical values or to booleans for grouping related control variables.

An empty unit expression is equivalent to a unitless specification, *i.e.*, `unit()=unit(1)`. Any variable which does not have an explicit unit declaration will have its unit inferred.

Unit declarations Named aliases for unit expressions can be declared in the declarations part of a Fortran file with the following syntax:

$$decls ::= \ldots \mid unit :: name = u \qquad (namedalias) \quad unit :: speed = m/s$$

During unit checking, any occurrences of a derived unit name are replaced by their declared unit expression. Hence in the unit checker, an alias is indistinguishable from its defining unit expression. A global check ensures that no named aliases conflict (*e.g.*, redefine) each other.

Type system Fig. 2 describes the type system of CamFort in a standard declarative and inductive way, defining the relation $\Gamma \vdash F : u$, where Γ is a map from program variables to their unit and F is a Fortran expression of unit u. The type system definition (and its implementation) extends the visible syntax of units with some additional constructs: (1) function types $(u_1, \ldots, u_n \to v)$ *i.e.*, the unit specification of a Fortran function with n formal parameters (or *dummy variables* in Fortran parlance) of units $u_1 \ldots u_n$ and result unit v, (2) variable placeholders for units, written α (3) universal quantification $\forall \alpha \,.\, u$ for unit polymorphism. Fig. 2 shows the polymorphic unit types of some core Fortran intrinsic operators. When a unit is associated with a value type (*e.g.*, `integer`) we write $u[t]$ for a value type t as in rule (real-pow). The (int-pow) and (rational-pow) rules raise their unit to the power provided by a static constant.

Polymorphism in our unit system follows a similar approach to that of types in the polymorphic λ-calculus [18], though we restrict universal quantification to the top-level of a unit expression (*i.e.*, not nested). The introduction of universal quantification (unit *generalisation*) occurs only when a function is defined. The complementary (spec) rule, specialises a universally quantified unit by substituting a unit v for the variable α. By the form of the (app) rule, a polymorphic function must be specialised first before it is applied. For example:

$$(\text{app})\frac{(\text{spec})[\alpha \mapsto m]\frac{\Gamma \vdash abs : \forall \alpha.\alpha \to \alpha}{\Gamma \vdash abs : m \to m} \quad (\text{var})\frac{(x:m) \in \Gamma}{\Gamma \vdash x : m}}{\Gamma \vdash abs(x) : m}$$

Unit polymorphism example A key part of our unit type system is that it provides polymorphic unit support on top of Fortran's monomorphic type system. As an example, Fig. 3(a) defines a `square` function without any unit annotations. Under the typing scheme described in this section, then