Author's copy: to appear at ICCS'15 and in Journal of Computational Science.

Evolving Fortran types with inferred units-of-measure

Dominic Orchard[1], Andrew Rice[2], Oleg Oshmyan

[1] Department of Computing, Imperial College London
d.orchard@imperial.ac.uk
[2] Computer Laboratory, University of Cambridge
andrew.rice@cl.cam.ac.uk

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.

Keywords: units-of-measure, dimension typing, type systems, verification, code base evolution, Fortran, language design

1 Introduction

Type systems are one of the most popular static techniques for recognising and rejecting large classes of programming error. A common analogy for types is of physical quantities (*e.g.*, in [Bar93]), where type checking excludes, for example, the non-sensical addition of non-comparable quantities such as adding 3 metres to 2 joules; 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.* [Ken94] 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 [SMB+99]. 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 [ISO13a]. 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 1,000 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.

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 [OR13]. 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 [OR14]. The present paper is a contribution in this space.

Example Figure 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 (Figure 1(a)) and is told that only `x` and `t` need be annotated. Figure 1(b) shows the syntax used by the programmer to add `m`

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

```
real :: x, t, v, s        --analysis-->   Critical variables:
v = x / t                                 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

Figure 1: Example

(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 (Figure 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*)
$name ::= [a-zA-Z]+$	*unit names; regular expression*	`m`, `metres` ...
$\mathbb{R} ::= \mathbb{Z}$	*integer constants*	`1`, `2`, `-2` ...
$\mid \mathbb{Z}/\mathbb{Z}$	*fraction of two integers*	`2/3`, `4/2` ...
$u, v ::= \epsilon$	*empty—equivalent to unitless*	`x`
$\mid$ `1`	*unitless*	`unit(1) :: x`
$\mid name$	*unit identifier*	`unit(m) :: x`
$\mid u$`**(`$\mathbb{R}$`)`	*rational power*	`unit(s**(1/2)) :: x`
$\mid u\,v$	*product*	`unit(m s**2) :: x`
$\mid u/v$	*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.

A `unit` attribute can be given to any type, not just numerical types (this differs from others, *e.g.*, [Ken94]). 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.

$$\text{(var)}\frac{(\mathtt{x} : u) \in \Gamma}{\Gamma \vdash \mathtt{x} : u} \quad \text{(app)}\frac{\Gamma \vdash f : u_1, \dots u_n \to v \quad \Gamma \vdash F_i : u_i}{\Gamma \vdash f(F_1, \dots, F_n) : v} \quad \text{(spec)}\frac{\Gamma \vdash F : \forall \alpha . u}{\Gamma \vdash F : u[\alpha \mapsto v]}$$

$$\text{(int-pow)}\frac{\Gamma \vdash F : u \quad \Gamma \vdash n : 1[\mathtt{integer}] \quad \mathit{static}(n)}{\Gamma \vdash F\mathtt{**}n : u\mathtt{**}n} \quad \text{(real-pow)}\frac{\Gamma \vdash F : u \quad n : 1[\mathtt{real}]}{\Gamma \vdash F\mathtt{**}n : 1}$$

$$\text{(rational-pow)}\frac{\Gamma \vdash F : u \quad \Gamma \vdash p : u_2 \quad \Gamma \vdash q : u_2 \quad \mathit{static}(p, q)}{\Gamma \vdash \mathtt{RATIONAL_POWER}(F, p, q) : u\mathtt{**}(p/q)}$$

conf-op	$: \forall \alpha.\ \alpha \to \alpha \to \alpha$		
rel-op	$: \forall \alpha.\ \alpha \to \alpha \to 1$	**op**	$= \{$**conf-op**, **rel-op**, `*`, `/`, **not**, **abs**$\}$
`*`	$: \forall \alpha_1, \alpha_2.\ \alpha_1 \to \alpha_2 \to (\alpha_1 \alpha_2)$	**conf-op**	$= \{$`+`, `-`, `//`, `.AND.`, `.OR.`$\}$
`/`	$: \forall \alpha_1, \alpha_2.\ \alpha_1 \to \alpha_2 \to (\alpha_1 / \alpha_2)$	**rel-op**	$= \{$`==`, `/=`, `<`, `>`, `<=`, `>=`$\}$
`not, abs`	$: \forall \alpha.\ \alpha \to \alpha$		

Figure 2: Typing rules for units-of-measure

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

$\mathit{decls} ::= \ \dots \ |$ `unit` $:: \mathit{name} = u$ *(named alias)* `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 Figure 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 $(u_1, \dots, u_n \to v)$ *i.e.*, the unit specification of a Fortran function with n formal parameters (or *dummy variables* in Fortran parlance) of units $u_1 \dots u_n$ and result unit v, (2) variable placeholders for units, written α (3) universal quantification $\forall \alpha . u$ for unit polymorphism. Figure 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 [Pie02], 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:

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

```
real function square(y)
    real y
    square = y * y
end function
```

(a) Polymorphic square function

```
real, unit(m) :: x
real, unit(s) :: t
a = square(x)
b = square(t)
```

(b) Use of square function

```
real function scale_square(y)
    real y
    real, unit(m) :: k
    scale_square = y * y * k
end function
```

(c) Monomorphic scaling factor

Figure 3: Functions with unit polymorphism

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, Figure 3(a) defines a `square` function without any unit annotations. Under the typing scheme described in this section, then `square` : $\forall u.[u]\texttt{real} \rightarrow [u\texttt{**2}]\texttt{real}$. Figure 3(b) shows a program fragment using square with two different units. These are inferred as `m**2` and `s**2` respectively by specialising the type of `square`. As an example of a function which combines both unit polymorphism with monomorphic units, Figure 3(c) defines a function which squares its input then scales by a real number of unit `m`. By our typing scheme, `scale_square` : $\forall u.[u]\texttt{real} \rightarrow [\texttt{m}\, u\texttt{**2}]\texttt{real}$, which exposes constant scaling by a real of unit `m` inside the function.

3 Inference

Inference of units is done through Gaussian elimination, similar to the work of Kennedy [Ken94]. The idea is that the type system described in the previous section can be used to generate a series of constraints on unit terms which can be treated as linear equations and solved using the standard Gaussian elimination method. Here we briefly outline our technique through two examples, one for a monomorphic program, and the other for a polymorphic program.

Monomorphic example Figure 4(a) shows a simple program and Figure 4(b) the corresponding constraints generated from the rules of the units-of-measure system. Each constraint is turned into a linear equation (sum of scalar-variable products) by taking logarithms, *e.g.*:

$$u_{\texttt{volume}} = u_{\texttt{pi}} \cdot u^2_{\texttt{radius}} \cdot u_{\texttt{height}} \quad \xrightarrow{\log} \quad \log u_{\texttt{volume}} = \log u_{\texttt{pi}} + 2 \log u_{\texttt{radius}} + \log u_{\texttt{height}}$$

This system of linear equations is then represented as a matrix in the type checker, where each equation is a row and each column is a log variable $\log u_v$ (for $\log u_v$ we write just v for the column headings here). Gaussian elimination is then applied by scaling a row by a non-zero scalar, adding one row to another, or swapping rows. These operations are applied until the

```
real, unit(m)  :: radius, height
real, unit(kg) :: mass
real :: density, volume, pi = acos(-1.0)
volume = pi * radius**2 * height
mass = volume * density
```

(a) Unit monomorphic source program

$$u_{\texttt{radius}} = \texttt{m}, \quad u_{\texttt{height}} = \texttt{m}, \quad u_{\texttt{mass}} = \texttt{kg}$$
$$u_{\texttt{pi}} = u_{\texttt{acos(-1.0)}}, \quad u_{\texttt{acos(-1.0)}} = 1$$
$$u_{\texttt{volume}} = u_{\texttt{pi}} \cdot u^2_{\texttt{radius}} \cdot u_{\texttt{height}}$$
$$u_{\texttt{mass}} = u_{\texttt{volume}} \cdot u_{\texttt{density}}$$

(b) Constraints generated from (a)

Figure 4: Constraints generated from a monomorphic program

`pi`	`acos(-1.0)`	`radius`	`height`	`mass`	`volume`	`density`	
1	-1	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	$\log \mathtt{m}$
0	0	0	1	0	0	0	$\log \mathtt{m}$
0	0	0	0	1	0	0	$\log \mathtt{kg}$
-1	0	-2	-1	0	1	0	0
0	0	0	0	1	-1	-1	0

$\longrightarrow$

`pi`	`acos(-1.0)`	`radius`	`height`	`mass`	`volume`	`density`	
1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	$\log \mathtt{m}$
0	0	0	1	0	0	0	$\log \mathtt{m}$
0	0	0	0	1	0	0	$\log \mathtt{kg}$
0	0	0	0	0	1	0	$3 \log \mathtt{m}$
0	0	0	0	0	0	1	$\log \mathtt{kg} - 3 \log \mathtt{m}$

Figure 5: Gaussian elimination applied to the linearised constraints of Figure 4(b)

matrix is in *row echelon form*, where all entries to the left of the diagonal are zero. Figure 5 shows this transformation for our example monomorphic program.

A matrix in *reduced* row echelon form has zero in every entry apart from its diagonal (like the above). This represents a unique solution to the system of equations. In this case, we have a unique solution for the typing of the program, where every inferred type is then added into the program. For example $\log u_{\mathtt{volume}} = 3 \log \mathtt{m}$ and so $u_{\mathtt{volume}} = \mathtt{m}^3$.

Polymorphic example To accommodate polymorphism in the Gaussian elimination procedure, we extend the usual technique slightly. As an example, consider the polymorphic `square` function in Figure 3(a), and its use in Figure 3(b) with two variables of different units.

Functions and subroutines in a program are analysed and a set of constraints is built and reduced using Gaussian elimination. This results in a relationship between the units of the parameters and the unit of the result. This relationship, which we call a *procedure constraint*, results in a constraint on units. The procedure constraint for `square` is $2 \log u_{\mathtt{square\#0}} = \log u_{\mathtt{=square}}$, meaning the log-unit of the result is two times the log-unit of the first (and only) parameter.

For every procedure call a new constraint (matrix row) is added based on the corresponding procedure constraint by copying the parameter coefficients to the columns for the corresponding arguments and copying the result coefficient to the column of the calling expression. This step corresponds to the (spec) rule in Figure 2; this new constraint introduces a unit specialisation.

If there are local variables in the procedure which require annotation then CamFort identifies these when deriving the procedure constraint. These can then be annotated by the programmer as required. This approach is sufficient for all cases except if the units of the local variable depend on the units of the parameters. The CamFort syntax currently does not allow a programmer to express this polymorphism. We plan to address this in future work.

4 Guided annotation

Consider an expression `a + b + c`. In the units system described above, this expression elicits the constraints that `a`, `b`, and `c` have equal units. Without any concrete unit given to any of these variables, the inference procedure can only infer they are *unitless*. But to give a concrete, unitful type requires only a single explicit unit annotation for one variable, not all.

In order to reduce the burden on programmers adopting our units-of-measure system and evolving their existing code, our tool includes a feature for reporting on "critical" subsets of

the variables in a program which, if given an explicit annotation, provide a solution without any unnecessary defaulting to *unitless*. This was shown in Figure 1(a). Here we outline the procedure, which builds on the Gaussian elimination procedure described in Section 3.

Consider the program fragment `e = a + b*c*d` in which only `d` has an explicit unit declaration as unit `m`. For this program, the system generates the following constraints and corresponding linear constraints (by applying the logarithm and rearranging variables to the left):

$$\begin{array}{ccc} u_a = u_e & u_a = u_b u_c u_d & u_d = \mathtt{m} \\ log\, u_a - log\, u_e = 0 & log\, u_a - log\, u_b - log\, u_c - log\, u_d = 0 & log\, u_d = log\, \mathtt{m} \end{array} \quad (1)$$

The linear constraints are represented via the following matrix (on the left) which is then reduced into row echelon form (on the right):

$$\begin{array}{ccccc|c} \mathtt{a} & \mathtt{b} & \mathtt{c} & \mathtt{d} & \mathtt{e} & \\ \hline 1 & 0 & 0 & 0 & -1 & 0 \\ 1 & -1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & log\,\mathtt{m} \end{array} \xrightarrow{row\ echelon} \begin{array}{ccccc|c} \mathtt{a} & \mathtt{b} & \mathtt{c} & \mathtt{d} & \mathtt{e} & \\ \hline 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & -1 & -1 & 0 & 1 & log\,\mathtt{m} \\ 0 & 0 & 0 & 1 & 0 & log\,\mathtt{m} \end{array} \quad (2)$$

If there are non-zero values on the leading diagonal of the matrix then we can solve for all variables (this is the back-substitution phase of Gaussian Elimination). Therefore, a zero value on the diagonal corresponds to an unknown variable. In the right matrix above, we can follow the leading diagonal for variables `a` and `b`, but the third row has no value to determine `c`. Instead the column for `d` has the leading non-zero coefficient, so we record `c` as missing and continue. We then find that `e` is missing (it has no row with a leading non-zero coefficient) and so record this too and stop. Variables `c` and `e` are therefore reported as being *critical variables*.

Definition [Critical variables, formally] Let m_i be the number of the first column in the row i with a non-zero coefficient, assigning the value of ∞ if all columns are zero (or undefined). The *critical variable set* C has the property that $\forall k$ such that $m_i < k < m_{i+1}$ then $v_k \in C$ for each row i where $m_{i+1} > m_i + 1$.

There are often many possible solutions for C, but each will provide equivalent information.

Example In the matrix above m_1,m_2,m_3,m_4,m_5 are the values 1,2,4,∞,∞ respectively. When $i = 2$ we have $m_3 = 4$ and $m_2 = 2$ so $m_3 > m_2 + 1$. C therefore contains v_k for $2 < k < 4$ (*i.e.*, $k = 3$). Also, when $i = 3$ we have $m_4 = \infty$ and $m_3 = 4$ and so we add v_k for $4 < k < \infty$ (*i.e.*, $k = 5$) to C. Therefore the critical variable set for is $\{v_3, v_5\}$ which are the variables $\{\mathtt{c}, \mathtt{e}\}$.

An interesting nuance to the critical variable analysis is deciding what units to infer for literals; a literal constant in a program might be unitless (*e.g.*, a scalar translation) or not. There is no single correct choice which covers all situations and so we provide an option to control the default assumption made by CamFort. We illustrate the three available choices via the example of Celsius-Fahrenheit conversion: `s = 1.8; a = 32.0; f = s * c + a`.

- **Polymorphic** literals are assumed polymorphic. In this case the possible critical variable sets are $\{f, s\}$, $\{f, c\}$, $\{s, c\}$, $\{s, a\}$, $\{c, a\}$. This is the safest option as it minimises the number of values assumed to be unitless, but in turn will require the most annotation.
- **Unitless** literals are assumed all to be unitless. In this case no further annotation is required for our example since this forces all quantities to be unitless.
- **Mixed** literals are assumed to require units if used in a **conf-op** or a **rel-op** (see Figure 2) and to be unitless otherwise. This captures the intuition that we add a value with units (+ is a **conf-op**) but we multiply by a unitless scalar. This option leaves the possible critical value sets as $\{f\}$,$\{c\}$,$\{32.0\}$, each requiring less annotation than the polymorphic case.

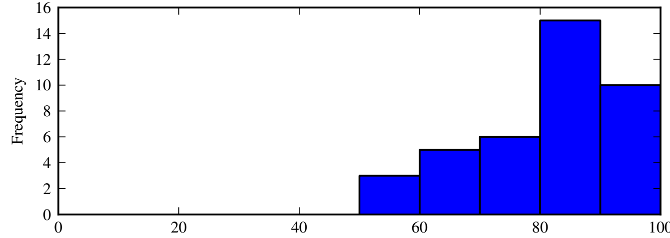

Figure 6: Distribution of % annotation savings

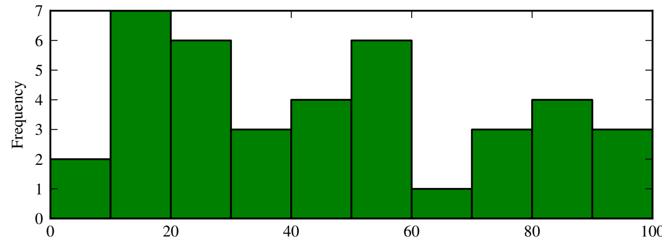

Figure 7: Distribution of % unit coverage

5 Evaluation

Our evaluation of CamFort considers 43 numerical Fortran programs taken from a well-known computational physics textbook [Pan06]. This provides an excellent corpus of small numerical methods and models, between 50 and 200 lines of code, which can benefit from units-of-measure.

We excluded a few programs that use MPI since CamFort cannot usefully process these at this point. This is due to a lack of syntactic support for polymorphic annotation of procedures—without this we would have to process and annotate the entire MPI library in order to progress. As mentioned, we will be addressing this limitation in future work. All other programs in the book were processed, barring four with difficult to parse data formatting.

The first question we investigated was whether the inference process actually results in a saving in programmer effort, compared to annotating every variable with a unit. For this we analysed each program and recorded the total number of declared variables (t) and the size of the critical variable set ($|C|$). From this we computed the percentage annotation saving $s = (1 - |C|/t) * 100$. Figure 6 shows the distribution of annotation savings (s) across the corpus. The median saving was 82.4% (3 sf.). We see that the use of CamFort can significantly reduce the amount of annotation effort required for many programs.

Our second question was to what extent is dimension typing useful for scientific computing. To understand this we annotated every variable that was reported critical and counted the number of variables which subsequently had a unit inferred which is not *unitless*. Our intuition is that since unitless variables can be combined together arbitrarily they do not benefit from the extra guarantees provided by the units-of-measure system. Therefore, if the vast majority of the variables in a program are unitless then the value of unit typing to that program is small. We therefore recorded the total number of variables which were given units after inference (u, which includes C). We computed the *unit coverage* $a = u/t * 100$.

Figure 7 shows the distribution of unit coverage (a). The median coverage percentage was 42.8% (3sf.), but ranging from 0% to 100% in some cases. We found that the programs which benefited most from dimension typing involved lots of polymorphic intrinsics (multiplication,

```
unit :: metre                               ! explicit declaration of new units
unit :: centimetre = metre * 100            ! conversion unit
unit :: volume = metre ** 3                 ! composite unit
real, unit(centimetre) :: radius, height    ! new type attribute unit(...)
real, unit(volume) :: volume
read *,radius, height
volume = cylinder_volume(radius,height)

function cylinder_volume(radius,height)
  unit, abstract :: length, volume = length**3 ! unit-polymorphism in arguments
  real, unit(length) :: radius, height
  real, unit(volume) :: cylinder_volume
  real, parameter :: PI = ACOS(-1.0)
  cylinder_volume = PI * radius**2 * height
end function
```

Figure 8: A simple program conforming to the N1969 proposal

divisions, abs). Conversely, programs which used more trigonometric functions seem to benefit less from this approach, since they constrained units to be unitless. Whilst the distribution of unit coverage results is wide, its median of roughly 40% shows the general usefulness of unit typing and its potential to aid program correctness.

6 Comparison with N1969

The Fortran programming language is internationally standardised by ISO/IEC JTC1/SC22. In April 2013, the working group received a proposal for adding native units of measure to Fortran, identified by N1969 [ISO13a] (with associated presentation N1970 [ISO13b]). CamFort syntax is based on that of N1969. We make a comparison here.

Figure 8 shows an example program conforming to N1969 syntax. Our alterations to this syntax focus on simplicity and reducing the burden on the programmer. Extending CamFort to generate code which is compliant with N1969 is straightforward.

Explicit unit declaration N1969 requires that all units are explicitly declared and named. This has the benefit of protecting the programmer from typos when declaring variables but imposes an extra burden when converting existing code. Although it is sometimes the case that a new name for a complicated composite unit can aid clarity we don't believe this is always the case: a programmer might well prefer to write `m/s` instead of `speed`.

Therefore CamFort does not require the explicit declaration of units. Instead, a new unit name is introduced implicitly on first use. For cases where a new name would improve clarity, we provide optional unit declaration which introduces a unit alias (see Section 2).

Kinds of unit N1969 units can be either *atomic*, *composite* (combining existing units through multiplication/division) or *conversions* (linear scaling and translation existing units). The first two (also supported by CamFort) are essential to dimensional analysis. Conversion units allow automatic, compiler-generated conversion code when the programmer moves between units. Instead, we prefer distinct fundamental units with explicit conversions. This better matches existing practices and avoids obscuring potential numerical issues created by the conversion.

Unit polymorphism in arguments The keyword `abstract` can be used by an N1969 programmer to declare that the unit of a function parameter is *independent*, *i.e.*, polymorphic.

Any dependent units can then be expressed in terms of these abstract units. In CamFort, no special syntax is required for this. The details of polymorphic units are simply inferred. A disadvantage with this approach in CamFort is that unit polymorphic functions therefore lack any unit specification/signature that describes their polymorphism. It is currently possible to use our tool in a *query* mode to ask for the unit type of a function, but a better scheme would introduce syntax for describing polymorphic unit types explicitly. This is future work.

Rational power Occasionally it is necessary to raise a value to a non-integer power. One example might be calculating the length of an edge from the area of a square. CamFort and N1969 both permit this through the use of a new intrinsic function `RATIONAL_POWER` which raises its first argument to a rational power specified as a numerator and a denominator. Both systems require that the power required be specified statically (*i.e.*, available at compile time).

Unitless N1969 provides a built-in unit `UNITLESS` for use with scalar constants. In CamFort we call this `1`. In addition to the built-in unit N1969 also provides an intrinsic coercion function (also called `UNITLESS`) which strips the units from its argument. We have so far not seen the need for this in our experiments. However, if needed, such a feature is a trivial extension of our typing rules—in the context of the typing rules (but not execution) this is just the same as raising a value to the power 1.0 (a real constant), via rule *(real-pow)* in Figure 2.

7 Related work

Despite the clear benefits of automatic units-of-measure inference/checking, this feature is relatively uncommon in programming languages. One of the most well-known and well-developed is the system provided by the functional programming language F#, which provides both polymorphism and inference [Ken10] and is based on the earlier work of Kennedy for the ML language [Ken96, Ken94]. The functional language Haskell also has various forms of polymorphic, inferred units-of-measure typing provided internally by building on Haskell's rich type system (such as the work of Muranushi and Eisenberg [ME14]) or with some additional simple compiler extensions to improve the type checking facilities, as in the work of Gundry [Gun13, Chapter 3][Gun15, Gun11]. The Fortress language was designed to include units of measure from the very start (although unfortunately development of this language has been halted as of 2012) [ACH+05]. C++ has some support for static unit checking via the use of a library.[2] A previous system for Fortran by Petty, provides a dynamic approach to unit checking (via overloading) [Pet01]. The static approach used here, and in the other tools mentioned above, has the advantage of not incurring any runtime overhead and providing safety guarantees about all execution paths (not just those that have been encountered during testing).

For other languages there are a variety of external tools (in the style of pre-processors, similar to CamFort's approach) for adding units-of-measure to languages. For example, Osprey for C [JS06] and SimCheck [RS10] for Simulink.

An alternate tool for C, by Guo and McCaman, provides an interactive process for users to specify units following an automatic constraint solving process [GM05]. This has similar aims to our own work: to ease adding units to a program via inference, reducing the annotation burden on the programmer. They evaluate their tool on various programs and note the number of "basic units" interactively requested from the user and the number of variables in the program. For a test program whose size is comparable to our own tests, they report a ratio of 4:33

[2] `http://tuoml.sourceforge.net/`

between explicitly given units and variables in the program: equivalent to roughly 88% unit coverage by our measure in Section 5, similar to our median coverage result. Their other test programs are larger (500-60k lines) which makes it hard to compare coverage. They report results equivalent to between 99% and 89%, though larger programs likely contain significant portions of "unitless" code. Further work for us it experiment with larger code base. We believe having units as part of the syntax (as in CamFort) is important for adoption (rather than this information being external, *e.g.*, via an interactive tool) as this interacts more naturally with standard development practices (*cf.* Java annotations, which replaced external XML files with inline comment-based syntax, *e.g.* for the Spring framework).

8 Conclusion and further work

We have described an extension to the Fortran language which allows automatic verification of units, and by extension dimensions, in a program. Given the prevalence of physical quantities in computational science software we argue that this provides a useful means to increase our confidence in the correctness of our models. We believe that automatic verification tools will become more and more pertinent as the complexity of scientific models continues to increase [OR14].

Our system, CamFort, is complementary to the current standards proposal for adding units of measure to Fortran. Our contribution is to add the significant benefit of automatically inferring units where possible rather than requiring explicit annotation. We envisage that CamFort could be used in two different ways: 1) as a pre-processor which validates units before stripping the annotations in preparation for compilation with a standard compiler; or 2) as a migration tool to N1969—CamFort can automatically infer units for approximately 80% of the variables in our tests, requiring only 20% manual annotations.

The concept of inferring units of measure has been established in the research literature for a long time. However, it has not yet been adopted despite its obvious applicability to scientific computing. Our intention with CamFort is to lower the barrier to adoption by showing in detail how this approach can be used with Fortran without affecting existing workflows.

Further work Currently we use a simple, hand-rolled implementation of Gaussian elimination. Other tools use off-the-shelf solvers. For example, Osprey (units-of-measure system for C) uses LAPACK and has shorter type checking times [JS06]. One avenue of future work is to improve the performance of CamFort, possibly using LAPACK for the solver engine.

Although CamFort will infer polymorphic unit signatures, there is no syntax for representing this polymorphism in the source code. There are times when it would be very useful to do so. For example, to specify the behaviour of external functions. We would also like to consider a 'transparent' syntax for units which embeds the annotations within Fortran comments. The benefit of this would be that code which is verified with CamFort can still be compiled with traditional tool chains without pre-processing.

We also intend to investigate how CamFort performs in practical use through user studies. One possibility is that a more interactive approach is required with the programmer. This might take the form of a REPL for querying unit information and inference.

Acknowledgments Many thanks to Alan Mycroft for helpful discussions, and to Raoul-Gabriel Urma and the participants of the Workshop on Programming Language Evolution 2014 for their comments on an earlier informal talk about this work [UOM14]. This work was supported in part by a Google Focussed Research Award and by EPSRC grant EP/K011715/1.

References

[ACH+05] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. The Fortress language specification. *Sun Microsystems*, 139:140, 2005.

[Bar93] H. P. Barendregt. Lambda calculi with types, Handbook of logic in comp. sci. (vol. 2), 1993.

[GM05] Philip Guo and Stephen McCamant. Annotation-less unit type inference for C. *Final Project, 6.883: Program Analysis, CSAIL, MIT*, 2005.

[Gun11] Adam Gundry. Type Inference for Units of Measure. In Ricardo Peña and Marko van Eekelen, editors, *Draft Proceedings of the 12th International Symposium on Trends in Functional Programming (TFP '11)*, pages 17–35, 2011. SIC-07/11, Dept. Computer Systems and Computing, Universidad Complutense de Madrid.

[Gun13] Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[Gun15] Adam Gundry. A Typechecker Plugin for Units of Measure, 2015. Under review, `http://adam.gundry.co.uk/pub/typechecker-plugins/`.

[ISO13a] ISO/IEC JTC1/SC22/WG5, Units of measure for numerical quantities, April 2013. N1696, `ftp://ftp.nag.co.uk/sc22wg5/N1951-N2000/N1969.pdf`.

[ISO13b] ISO/IEC JTC1/SC22/WG5 N1970, Units of Measure in Fortran, 2013. N1970, `ftp://ftp.nag.co.uk/sc22wg5/N1951-N2000/N1970.pdf`.

[JS06] Lingxiao Jiang and Zhendong Su. Osprey: a practical type system for validating dimensional unit correctness of C programs. In *Proceedings of ICSE*, pages 262–271. ACM, 2006.

[Ken94] Andrew Kennedy. Dimension types. In *European Sympsium on Programming Languages and Systems (ESOP)*, pages 348–362. Springer Berlin Heidelberg, 1994.

[Ken96] Andrew John Kennedy. *Programming languages and dimensions*. Number 391. University of Cambridge, Computer Laboratory, 1996.

[Ken10] Andrew Kennedy. Types for units-of-measure: Theory and practice. In *Central European Functional Programming School*, pages 268–305. Springer, 2010.

[ME14] Takayuki Muranushi and Richard A Eisenberg. Experience report: Type-checking polymorphic units for astrophysics research in haskell. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*, pages 31–38. ACM, 2014.

[OR13] Dominic Orchard and Andrew Rice. Upgrading Fortran source code using automatic refactoring. In *Proceedings of 6th Workshop on Refactoring Tools, WRT 2013*. ACM, 2013.

[OR14] Dominic Orchard and Andrew Rice. A computational science agenda for programming language research. *Procedia Computer Science*, 29(0):713 – 727, 2014. ICCS.

[Pan06] Tao Pang. *An Introduction to Computational Physics*. Cambridge University Press, 2006.

[Pet01] Grant W Petty. Automated computation and consistency checking of physical dimensions and units in scientific programs. *Software: Practice and Experience*, 31(11):1067–1076, 2001.

[Pie02] B.C. Pierce. *Types and programming languages*. MIT press, 2002.

[RS10] Pritam Roy and Natarajan Shankar. SimCheck: An Expressive Type System for Simulink. In *NASA Formal Methods*, pages 149–160, 2010.

[SMB+99] A. G Stephenson, D. R Mulville, F. H Bauer, G. A Dukeman, P. Norvig, LS LaPiana, PJ Rutledge, D Folta, and R Sackheim. Mars Climate Orbiter Mishap Investigation Board Phase I Report, 44 pp. *NASA, Washington, DC*, 1999.

[UOM14] Raoul-Gabriel Urma, Dominic Orchard, and Alan Mycroft. Programming language evolution workshop report. In *Proceedings of the 1st Workshop on Programming Language Evolution*, pages 1–3. ACM, 2014.