COVER SHEET

Cook, Phil and Fidge, Colin and Hemer, David (2006) Well-Measuring Programs. In Han, Jun and Staples, Mark, Eds. *Proceedings Australian Software Engineering Conference (ASWEC 2006)*, pages pp. 253-261, Sydney.

Accessed from http://eprints.qut.edu.au

Copyright 2006 IEEE

Well-Measuring Programs

Phil Cook
School of Information Technology and
Electrical Engineering,
The University of Queensland
email: philc@itee.uq.edu.au

Colin Fidge
School of Software Engineering and
Data Communications,
Queensland University of Technology
email: c.fidge@qut.edu.au

David Hemer
School of Computer Science,
University of Adelaide
email: hemer@cs.adelaide.edu.au

Abstract

Any program that measures quantities from its physical environment must compute using correct and consistent units of measurement. Such a program is described here as well-measuring. *In many systems, particularly embedded control software, paying inadequate attention to units of measurement can result in catastrophe. Unfortunately, current programming languages and tools provide little aid to the programmer attempting to establish or verify the well-measuring property.*

We present a program analysis technique for inferring and checking the units used within a program. The technique combines traditional Hindley-Milner-style type inference with the use of Static Single Assignment (SSA) form to enable analysis of imperative programs.

1 Introduction

On 23 September 1999 NASA's Mars Climate Orbiter was lost during its insertion into Martian orbit. The root cause of this system failure was later identified as the use of inconsistent units of measurement for course correction data sent to the probe [15]. Impulse values for course corrections were calculated by ground-based software in U.S. customary units, but the probe expected the data in SI units. Ultimately, this caused the orbiter to fly too close to Mars, probably burning up in the atmosphere.

This incident served as a costly reminder of the lesson many of us learned from emphatic physics professors: calculations involving physical quantities must be carried out using consistent units of measurement! Scientists and engineers apply this lesson routinely and use dimensional analysis to check the consistency of their equations. However, in the development of complex software-intensive systems, such disciplined handling of measures is uncommon. At best, units are noted as comments. At worst, they are not documented at all.

The difficulties in ensuring correct manipulation of measurements are a particular challenge in the development and maintenance of embedded control software. Such software is commonly charged with reading measurements from sensors, computing derived quantities, and using those to exercise control over the system's physical environment. Examples of such systems abound in areas such as aeronautics, medicine, transport, robotics, and power generation. Moreover, due to the nature of the physical control such systems exert, failure can cost both life and livelihood. As NASA's experience [14, 15] demonstrates, such failure is often only 1.09 yard.metre^{-1} away!

One of the obstacles to successful treatment of units in software is the lack of assistance from current programming languages and tools in this area. Many other common programming errors can be detected either by the language implementation itself, or by associated tools. The obvious analogue to unit-correctness is type-correctness: a well understood and useful property that is enforced by many languages. However, programming language types are no substitute for a system of units. Knowing that variable x is of type integer still does not tell us whether it denotes x feet or x metres.

In this paper we define what it means for a program to be unit-correct, or *well-measuring*, and how to go about automatically demonstrating this property. There are a number of requirements we assume that set this work apart from re-

Proceedings of the 2006 Australian Software Engineering Conference (ASWEC'06)
1530-0803/06 $20.00 © 2006 IEEE

lated efforts. Programs subjected to our analysis may be

- written in conventional imperative languages, as are commonly used in embedded systems (we use C here but the approach is equally applicable to languages such as Java or Ada),
- large existing bodies of code, including legacy programs for which there is little or no documentation on assumed units extant, and
- part of a larger system comprising both hardware and software elements in which achieving compatibility of interfaces is crucial.

As such, we cannot rely on extending a language with new type constructs [7], nor can we confine ourselves to inference of units in a functional language [9]. Furthermore, we choose not to burden programmers with the need for excessive annotation of their programs with unit information [17].

The method we demonstrate combines type inference techniques that are common in functional programming, with a novel use of Static Single Assignment (SSA) form, as found in compiler theory [13]. The core of the unit inference mechanism is typical of Hindley-Milner systems [12]. In particular, we rely heavily on the use of type variables. Such type systems, however, are designed to be applied to functional programs (or some form of the λ-calculus). We use SSA form in a novel way to allow us to analyse imperative programs in a functional style.

The use of type variables in inferring units not only gives us polymorphism, it also provides significant flexibility to the programmer. The units used in a program may be either inferred (yielding units containing type variables), or checked against a specification of the program's anticipated I/O environment. In the latter case, the specification may be partial or complete. Ultimately, only a minimum number of annotations must be added to a program to produce useful results. This encourages programmers to use the same level of mathematical expressiveness that a physicist would, while assisting them to achieve an appropriate level of rigour in their handling of units.

In the next section we review some terminology and related work. A motivational example is presented and analysed in Sect. 3. Section 4 defines the well-measuring property. The two parts of our analysis technique (translating programs to a functional form and unit inference) are discussed in Sections 5 and 6.

2 Measures in science and software

Measurement is the process of assigning a numerical value to some physical quantity. Each quantity we may wish to measure can be assigned a *dimension*, which may be formed from a small set of base dimensions (e.g., if mass (M), length (L), and time (T) are our base dimensions, then the dimension of force is $\mathrm{M.L.T}^{-2}$). Some quantities (e.g., angle) are dimensionless. A *unit of measurement* is some scale which we may use to assign numerical values to quantities. Similarly to dimensions, we may define a *system of units* in terms of a small set of base units from which all other units can be derived (e.g., if the units of mass, length, and time are kg, m, and s, respectively, then force can be measured in $\mathrm{kg.m.s}^{-2}$). A value which is not a measure of a physical quantity is unitless. Units constrain the arithmetic operations that may be performed on measurements. Addition, subtraction, and comparison of two measurements require identical units for both arguments. Multiplication of two measurements produces a measurement where the units are multiplied.

The modern scientific community has adopted the SI system [2] as its *lingua franca* for measurements, though other systems (e.g., US customary units) persist due to economic and cultural inertia.

Early attempts to support consistent handling of measurements in programs focussed on extensions to the type systems of existing programming languages [8, 5, 10, 4]. These type systems were typically limited in the units they could express. In contrast, House [7] proposes a more sophisticated system with support for polymorphism. In this paper, we have not assumed the luxury of extending a programming language specifically for the purpose of handling units. Instead, we rely only on a minimal number of programmer-supplied annotations of unit information, and employ a more sophisticated type inference system for determining the units used in the rest of the program.

The type inference system we use is similar to that used in the implementation of functional languages such as ML [12]. Kennedy [9] and Wand and O'Keefe [18] independently observed that such type inference systems can be easily extended to support dimensional analysis. We agree with these authors that a Hindley-Milner-style type system is a powerful foundation for reasoning about the units used in a program. However, embedded control software is usually written in imperative languages which are not immediately amenable to this form of analysis. We provide a novel translation technique to allow us to reason about this important class of programs.

Recently, Rosu and Chen [17] have described a verification technique that deals with units of measurement. Although their technique is powerful, it requires onerous annotation of programs with unit information. In contrast, our technique only requires annotation of some subset of the numeric constants used in a program.

In this paper, we focus on units of measurement used in programs. Other software artifacts can also benefit from

IEEE COMPUTER SOCIETY

systematic treatment of units and dimensions. Hayes and Mahony [6] discuss units in the context of formal specifications.

3 Motivation

In this section we introduce as a motivating example a piece of embedded software which is required to be well-measuring.

3.1 Example

Our example program is designed to run on board a moving vehicle, determining the vehicle's position based on measurements from the environment. To this end, the vehicle is equipped with a GPS receiver. The system also comprises an accelerometer (which measures the force applied to the vehicle in its direction of travel) and a rate gyroscope (which measures the rate of change of the vehicle's heading). These sensors are used to perform a dead reckoning calculation of the vehicle's position whenever GPS data is absent or not trustworthy. The program communicates with an input device to calibrate certain measurements at the beginning of its mission. Finally, the program outputs the vehicle's position on a display device.

The source code for the program is shown in Fig. 1. The program makes use of special "I/O variables" (underlined in the listing) to denote input and output actions. In practice, embedded programs use various mechanisms for communicating with other hardware and software elements of a system. Our I/O variables are simple abstractions of these language-, machine-, and application-specific mechanisms.

The program first establishes its initial configuration by accessing certain calibration inputs. It then proceeds to poll its input devices once every 10 ms. The period of the main loop is controlled by the `wait_for_clock_tick()` function, which relies on either clock interrupts or operating system services to synchronise the program.

In each period the program checks to see if the GPS receiver is able to provide a reliable position by reading the `gps_quality` I/O variable. This provides a measure of the quality of the GPS readings available, with positive values indicating sufficient accuracy. (In practice, GPS receivers produce such quality metrics based on the triangulation geometry of the satellites currently in view [11].) If the value of this input is positive, the readings from the GPS receiver are taken. If not, then the accelerometer and gyroscope are used to calculate the position using dead reckoning. (The nature of these calculations implies that any errors, due to either sensor imprecision or mishandling of units, will have a cumulative effect.)

```
1  static const double PERIOD = 0.01;
2  int eastings, northings, heading,
3      speed, mass;
4
5  main()
6  {
7    /* Calibrate measurements */
8    eastings = eastings_calibration;
9    northings = northings_calibration;
10   heading = heading_calibration;
11   mass = mass_calibration;
12   speed = 0;  /* Assume the vehicle is
13                  stationary */
14
15   while (1) {
16     double a, s;
17
18     wait_for_clock_tick();
19
20     /* Read force from accelerometer &
21        calculate acceleration */
22     a = accelerometer / mass;
23
24     if (gps_quality > 0) {
25       /* GPS is available - use the
26          position provided by it */
27       eastings = gps_eastings;
28       northings = gps_northings;
29     } else {
30       /* Estimate position using
31          dead reckoning */
32       s = speed * PERIOD +
33             0.5 * a * PERIOD * PERIOD;
34       eastings = eastings +
35                    s * cos(heading);
36       northings = northings +
37                    s * sin(heading);
38     }
39
40     /* Calculate new speed & heading */
41     speed = speed + a * PERIOD;
42     heading = heading + rate_gyro*PERIOD;
43
44     /* Display position */
45     display_eastings = eastings;
46     display_northings = northings;
47   }
48 }
```

Figure 1. Navigation program

3.2 Analysing the example

From inspecting this program alone, it might seem that we cannot say much about the units of measurement with which it works. We can, however, determine that the program handles measurements *consistently*, even without knowing precisely which units are involved. We can associate symbolic units with each program variable and analyse the arithmetic of the program to determine the relationships between these. This allows us to infer properties such as "if `gps_eastings` is in units of U, then `display_eastings` must also be in units of U". Furthermore, if we were armed

with more information about the units of some (but not necessarily all) of the program's I/O variables, we could constrain the units of the other variables even further using substitution and simplification.

Consider the statement on line 22 of Fig. 1, which calculates the vehicle's acceleration. Assume that the accelerometer produces values in units of U, and we wish to determine the units of `a`, denoted V. We can infer from the form of the statement that $V = U.A^{-1}$, where A represents the units of the variable `mass`. To determine A we can examine which assignments to `mass` can flow to this point of the program. In this case, there is only one: the initalisation of `mass` from the calibration interface. If `mass_calibration` is assumed to be in units of W, then we can say that $A = W$, and therefore $V = U.W^{-1}$. Now, assume that the accelerometer produces values in newtons (or $\mathrm{kg.m.s}^{-2}$) and the mass is calibrated in kilograms. If we substitute these units for U and W and perform some simplifications, we arrive at: $V = \mathrm{kg.m.s}^{-2}.\mathrm{kg}^{-1} = \mathrm{m.s}^{-2}$. This outcome provides us with some assurance that the acceleration value is being computed correctly.

The process of using symbolic manipulation combined with analysing how values flow to program statements is the essence of our approach. A human inspecting the program would most likely reason about units in a similar style. In the remainder of this paper we focus on formalisation and automation of these concepts.

3.3 Constant confusion

Now let us consider how to determine the units used for the vehicle's heading, as calculated on line 42. The value of the variable `heading` that may flow to this line is determined by either line 10 (the calibration reading) or line 42 itself (on the previous iteration of the loop). Assume that all these heading values are in units of U, the value of `rate_gyro` is in units of V, and the value of `PERIOD` is in units of W. The algebraic properties of units dictate that $U = V.W$.

Unfortunately, `PERIOD` is defined as a numeric constant, which we must always assume to be unitless! Foregoing this assumption would mask some measurement errors which we want to detect. For example, consider the expression for displacement used on line 32 (`speed*PERIOD + 0.5*a*PERIOD*PERIOD`). If constants were not assumed to be unitless, then we could infer *any* units we wanted for the constant 0.5 in this expression. If a unit error had actually been made (say velocity was in $\mathrm{ft.s}^{-1}$ but acceleration was in $\mathrm{m.s}^{-2}$) this would not be detected. Instead it would be inferred that 0.5 was in $\mathrm{ft.m}^{-1}$.

Returning to the calculation of the vehicle's heading, assuming `PERIOD` to be unitless would either cause the inference to fail or to produce unexpected instantiations for the symbolic units used. Therefore we must rely on the programmer to annotate the program with explicit unit coercions, as needed. To this end, we extend our source language with expressions of the form e **as** u, where e is an expression and u is a unit. In our example, we can continue if the programmer replaces the definition of `PERIOD` with:

```
static const double PERIOD = 0.01 as s;
```

This allows us to assume `PERIOD` is in units of 's' throughout our analysis.

In particular, such unit coercions are also necessary to flag any conversions between systems of measurement that the program may make; e.g., to convert measurements in metres to feet. Coercing a conversion factor to units of, say, $\mathrm{ft.m}^{-1}$ is a simple way of managing unit conversions.

However, we can relax this restriction for zero. Since zero acts as the additive identity it is always safe to assign any units we want to it. We say that zero is polymorphic [9]. As such, the initialisation of `speed` on line 12 and the comparison on line 24 do not require coercion.

3.4 Overview of inference technique

There are two main aspects to our approach: analysing which values flow to each program statement and inferring the units of measurement for those values. We use separate techniques to solve each of these two sub-problems.

The first step of the process is to transform the program into a functional form to make it easier to reason about. This transformation proceeds by converting the program to SSA form, which enables construction of a set of equations which relate program variables to each other. These equations are combined to form a let-expression in a simple λ-calculus representation. This translation is described in Sect. 5.

Once we have this program representation, we may apply well understood type inference techniques to analyse the units of measurement used throughout the program. We begin with a base type environment that associates (symbolic) units of measurement with each input variable. Then a type inference algorithm, extended with support for units of measurement, is applied to the functional form of the program. This will calculate a type for the program. The symbolic form of this type may contain type variables, as dictated by the initial type environment. The type system and inference algorithm we use are defined in Sect. 6. The algorithm is sound, but not complete.

4 The well-measuring property

We begin by defining the form which units of measurement can take and equality on this set.

Definition 1 (Unit of measurement) *Given a set of* base units $\mathcal{B}$ *and a set of (unit) variables* $\mathcal{V}$*, a unit of measurement* $u \in \mathcal{U}$ *has the form:*

$$u \quad ::= \quad b \mid \alpha \mid u_1.u_2 \mid u^n \mid \mathbf{1}$$

where $b \in \mathcal{B}$*,* $\alpha \in \mathcal{V}$*, and* $n \in \mathbb{Z}$*.*

Note that **1** is used to denote a unitless quantity.

Definition 2 (Unit equality) *The relation '=' on* $\mathcal{U}$ *is defined by the identities:*

$$\begin{array}{rclrcl} u & = & u & u^0 & = & \mathbf{1} \\ \mathbf{1}.u & = & u & u^n.u & = & u^{n+1} \\ u_1.u_2 & = & u_2.u_1 & (u_1.u_2).u_3 & = & u_1.(u_2.u_3) \end{array}$$

Our approach does not restrict each occurrence of a program variable to contain a value in the same units of measurement. We adopt a more liberal approach which focusses on values rather than variable names. This is necessary because a given program, even if it is well-typed, is not necessarily constrained to always use a particular variable to hold values in the same units. A variable x may be used to manipulate values in units of metres in one part of the program and units of kilograms in another part. Although we may criticise this programming style, our formalism must nevertheless allow for this possibility. We characterise this in the following definitions.

Definition 3 (Defines) *A statement* S defines *a program variable* x *iff.* S *is of the form* $x = E$ *for some expression* E*.*

Definition 4 (Reaching definitions) *A statement* S_1 *which defines a variable* x *is a* reaching definition *for statement* S_2 *iff. there is a (syntactically defined) execution path from* S_1 *to* S_2 *which does not contain a definition of* x*.*

We now define what it means for an individual program statement to be well-measuring.

Definition 5 (Well-measuring assignment) *Say* S *is of the form* $x = E$ *for some expression* E*. Then* S *is* well-measuring *iff. for all* S'*, such that* S *is a reaching definition of* S' *and* S' *uses* x*,* S *and* S' *"agree on" the units of* x*.*

Note that we have not yet defined what it means for statements to "agree on" the units of a variable. This is characterised fully in Sect. 6. Finally, we lift the well-measuring property to programs.

Definition 6 (Well-measuring program) *A program* P *is* well-measuring *iff. each assignment in* P *is well-measuring.*

This definition is syntactic. We do not attempt to define well-measuring programs in terms of their dynamic semantics, as any such definition would put the problem beyond the reach of automated inference. Instead, this definition is directly analogous to (though weaker than) usual definitions of well-typed programs. That is, there exist programs that compute in a unit-consistent manner but cannot be shown by our inference technique to be well-measuring. However, we conjecture that well-measuring programs cannot "go wrong", since all possible behaviours they can perform must be well-measuring.

5 Translating programs to a functional form

The first part of our analysis involves translating the program into a functional form. We achieve this by translating the program into SSA form and then reading off a set of equations relating program variables. These equations may then be used to form a let-expression, to which the unit inference algorithm described in Sect. 6 may be applied.

5.1 Translation to SSA form

SSA form is becoming an increasingly popular program representation in optimising compilers [13]. We use it here in a new way to enable us to reason about the program without being overwhelmed by inessential details of the program's execution. SSA form captures precisely the static information about the program that we are interested in to enable us to verify the well-measuring property.

A program is in SSA form iff. for each variable x there is at most one program statement that assigns x a value. When translating a program into SSA form, subscripting is usually used to differentiate occurrences of program variables. In order for programs to remain expressive in this form it is necessary to introduce applications of a special function ϕ at the points that more than one value for a variable may reach. A statement of the form $x_3 = \phi(x_1, x_2)$ miraculously assigns x_3 the value of either x_1 or x_2. Cytron, et al. [3] describe an efficient technique for translating a program into SSA form by examining dominance relationships among program statements.

Figure 2 shows our example program from Sect. 3.1 in SSA form. There are two sequences of assignments involving ϕ in this version of the program. Lines 36 and 37 "merge" the values flowing out of the two alternatives of the `if` statement, while lines 15–18 "merge" the values flowing into the loop from either the loop entry or iterative cases. (Also note that this version of the program incorporates the unit coercion in the definition of `PERIOD` discussed above.)

5.2 Translation to λ-terms

Once the program is in SSA form we may read off a set of (simultaneous) equations which define the relationships among program variables. The variant of the λ-calculus we use here includes recursive let-expressions, numeric constants, tuples, arithmetic expressions, and unit coercions, as defined below.

```
static const double PERIOD = 0.01 as s;
int eastings1, ..., northings1, ...;

main()
{
  eastings1 = eastings_calibration;
  northings1 = northings_calibration;
  heading1 = heading_calibration;
  mass1 = mass_calibration;
  speed1 = 0;

  while (1) {
    double a1, s1;

    eastings2 = φ(eastings1, eastings5);
    northings2 = φ(northings1,northings5);
    speed2 = φ(speed1, speed3);
    heading2 = φ(heading1, heading3);

    wait_for_clock_tick();

    a1 = accelerometer / mass1;

    if (gps_quality > 0) {
      eastings3 = gps_eastings;
      northings3 = gps_northings;
    } else {
      s1 = speed2 * PERIOD +
            0.5 * a1 * PERIOD * PERIOD;
      eastings4 = eastings2 +
                    s1 * cos(heading2);
      northings4 = northings2 +
                     s1 * sin(heading2);
    }

    eastings5 = φ(eastings3, eastings4);
    northings5 = φ(northings3,northings4);

    speed3 = speed2 + a1 * PERIOD;
    heading3 = heading2 +
                rate_gyro*PERIOD;

    display_eastings = eastings5;
    display_northings = northings5;
  }
}
```

Figure 2. Example program in SSA form

Definition 7 (λ-term) *Let $x \in \mathcal{X}$ denote a (program) variable and $c \in \mathbb{R}$ denote a constant. A λ-term e is of the form:*

$$\begin{array}{rcl} e & ::= & c \mid x \mid \lambda x.e \mid e_1\ e_2 \mid (e_1, ..., e_n) \mid \\ & & \mathsf{let}\ x_1 = e_1; ...; x_n = e_n\ \mathsf{in}\ e \mid \\ & & e_1 \oplus e_2 \mid e_1 * e_2 \mid e_1/e_2 \mid e_1 \sim e_2 \mid e\ \mathsf{as}\ u \end{array}$$

where $\oplus$ denotes an additive operator ('$+$' or '$-$') and $\sim$ denotes a relational operator ('$=$', '$<$', etc.).

Each assignment statement in the SSA form is translated into an equation in the obvious way. Additionally, the guard of each control statement is translated into equations with a fresh variable on the left-hand side. We may then write these equations in a (recursive) let-expression. The body of

```
let
  PERIOD = 0.01 as s
  eastings1 = eastings_calibration
  northings1 = northings_calibration
  heading1 = heading_calibration
  mass1 = mass_calibration
  speed1 = 0
  g = gps_quality > 0
  eastings2 = φ(eastings1, eastings5)
  northings2 = φ(northings1, northings5)
  speed2 = φ(speed1, speed3)
  heading2 = φ(heading1, heading3)
  a1 = accelerometer / mass1
  eastings3 = gps_eastings
  northings3 = gps_northings
  s1 = speed2*PERIOD + 0.5*a1*PERIOD*PERIOD
  eastings4 = eastings2 + s1*cos(heading2)
  northings4 = northings2 + s1*sin(heading2)
  eastings5 = φ(eastings3, eastings4)
  northings5 = φ(northings3, northings4)
  speed3 = speed2 + a1*PERIOD
  heading3 = heading2 + rate_gyro*PERIOD
  display_eastings = eastings5
  display_northings = northings5
in
  (display_eastings, display_northings)
```

Figure 3. Let-expression for example program

the let-expression can be any λ-term we want. In practice, we use tuples of those program variables we wish to focus on in our analysis. If we were interested in determining the units of measurement used at the outputs of our example program, we would write

```
(display_eastings, display_northings)
```

as the body of the let-expression, as shown in Fig. 3.

5.3 Procedures

So far we have discussed only simple single-procedure programs. To deal with procedures we need to analyse which, if any, global variables the procedure accesses. For each procedure we construct two sets of global variables: those assigned by the procedure and those used by the procedure. We then transform the procedure into an equivalent "pure" form by augmenting its return values and parameters with these variables. For example, if, in the example program, the vehicle's heading was updated by the following procedure:

```
update_heading() {
  heading = heading + rate_gyro * PERIOD;
}
```

we would rewrite this as:

```
update_heading(h, rg) {
  return h + rg * PERIOD;
}
```

In this case global variable `heading` was assigned by the original procedure and global variable `rate_gyro` was

$$
\begin{array}{llllll}
\text{(CON)} & \Gamma \vdash c : \forall\alpha.\alpha & \text{(ABS)} & \dfrac{\Gamma \cup \{x : u'\} \vdash e : u}{\Gamma \vdash \lambda x.e : u' \to u} & \text{(APP)} & \dfrac{\Gamma \vdash e_1 : u_2 \to u_1 \quad \Gamma \vdash e_2 : u_2}{\Gamma \vdash e_1\ e_2 : u_1} \\
\text{(AS)} & \Gamma \vdash e \ \mathsf{as}\ u : u & \text{(REL)} & \dfrac{\Gamma \vdash e_1 : u \quad \Gamma \vdash e_2 : u}{\Gamma \vdash e_1 \sim e_2 : \mathbf{1}} & \text{(PROD)} & \dfrac{\Gamma \vdash e_1 : u_1 \quad \Gamma \vdash e_2 : u_2}{\Gamma \vdash e_1 * e_2 : u_1.u_2} \\
\text{(VAR)} & \dfrac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma} & \text{(SUM)} & \dfrac{\Gamma \vdash e_1 : u \quad \Gamma \vdash e_2 : u}{\Gamma \vdash e_1 \oplus e_2 : u} & \text{(QUOT)} & \dfrac{\Gamma \vdash e_1 : u_1 \quad \Gamma \vdash e_2 : u_2}{\Gamma \vdash e_1/e_2 : u_1.u_2^{-1}} \\
\text{(INST)} & \dfrac{\Gamma \vdash e : \forall\alpha.\sigma}{\Gamma \vdash e : [u/\alpha]\sigma} & \text{(GEN)} & \dfrac{\Gamma \vdash e : \sigma \quad \alpha \notin fv(\Gamma)}{\Gamma \vdash e : \forall\alpha.\sigma} & \text{(TUP)} & \dfrac{\Gamma \vdash e_1 : u_1 \ldots \Gamma \vdash e_n : u_n}{\Gamma \vdash (e_1, \ldots, e_n) : u_1 \times \ldots \times u_n}
\end{array}
$$

$$
\text{(LET)} \quad \frac{\Gamma' \vdash e_1 : u_1 \ldots \Gamma' \vdash e_n : u_n \quad \Gamma' \vdash e : u \quad \Gamma' = \Gamma \cup \{x_1 : u_1, \ldots, x_n : u_n\}}{\Gamma \vdash \mathsf{let}\ x_1 = e_1; \ldots ;\ x_n = e_n\ \mathsf{in}\ e : u}
$$

Figure 4. Type system

used by it.

We would then transform the body of the procedure into the λ-calculus, wrap it in an abstraction, and create a let-equation for it:

```
let update_heading = λh.λrg.h + rg*PERIOD
```

The let-bindings produced in this way should be placed in an outer let-expression of the main program's let-expression. This allows procedures to be polymorphic. A call to this procedure in the original program is rewritten as:

```
heading =
  update_heading(heading, rate_gyro);
```

before the enclosing procedure is translated to SSA form.

6 Unit inference

Having translated the imperative program into a λ-term, we now assign units to that λ-term using a Hindley-Milner-style [12] type system extended with support for arithmetic expressions that obey the usual algebraic properties of units [2].

6.1 Type system and inference laws

The type system is polymorphic. Polymorphism is dealt with using the usual generalisation and instantiation laws. Generalisation of type variables requires the introduction of type schemes:

Definition 8 (Type scheme) *A* type scheme σ *is of the form:*

$$\sigma ::= u \mid \forall\alpha.\sigma$$

The type system is presented in Fig. 4. The laws of this type system relate typing judgements:

Definition 9 (Typing judgement) *A* typing judgement $\Gamma \vdash e : \sigma$ *states that the λ-term e is an expression in units of σ under the type environment Γ, where Γ is a function from (program) variables to type schemes.*

We use the following notational conventions: $fv(X)$ denotes the set of free type variables in term X, $[X/\alpha]Y$ denotes term Y with term X substituted for every free occurrence of type variable α, and $\Gamma(x)$ denotes the type scheme associated with the program variable x in the type environment Γ.

The laws of this type system cannot be applied deterministically since (GEN) and (INST) are inverses. We transform the type system into a type inference "algorithm" in the standard way: we confine generalisation to the body of a let-expression (let-bound variables used within a let-expression's equations remain monomorphic), and we instantiate type schemes when typing (program) variables. The type inference laws are shown in Fig. 5. The function gen, which quantifies those variables that are free in the term u but not in environment Γ, is defined as

$$
\begin{aligned}
gen(\Gamma, u) \quad = \quad & \forall\alpha_1 \ldots \alpha_n.u \\
& \text{where } \{\alpha_1, \ldots, \alpha_n\} = fv(u) \setminus fv(\Gamma).
\end{aligned}
$$

The inference laws shown in Fig. 5 also ensure that non-zero constants are treated as unitless, while zero remains polymorphic, as discussed in Sect. 3.3. With these restrictions, the inference system is incomplete with respect to the type system shown in Fig. 4, but remains sound. Programmer-supplied coercions are required to enable inference to proceed in the presence of some numeric constants.

For the most part these inference laws are completely standard. The obvious additions are the laws for arithmetic, coercions, and constants. Something less obvious lies in the definition of equality for units of measurement. Type inference systems are usually implemented using syntactic unification [16]. When dealing with units of measurement, however, types must be unified using a more general algorithm which can account for abelian group properties: AC1 unification [1].

$$\text{(ZERO)}\ \Gamma \vdash 0 : \alpha, \alpha \text{ fresh} \qquad \text{(CON}'\text{)}\ \frac{c \neq 0}{\Gamma \vdash c : \mathbf{1}} \qquad \text{(VAR}'\text{)}\ \frac{\Gamma(x) = \forall\bar{\alpha}.u \quad \bar{\beta} \text{ fresh}}{\Gamma \vdash x : [\bar{\beta}/\bar{\alpha}]u}$$

$$\text{(LET}'\text{)}\ \frac{\begin{array}{c}\Gamma' \vdash e_1 : u_1\ \ ...\ \ \Gamma' \vdash e_n : u_n\ \ \Gamma'' \vdash e : u\ \ \Gamma' = \Gamma \cup \{x_1 : u_1, ..., x_n : u_n\} \\ \Gamma'' = \Gamma \cup \{x_1 : gen(\Gamma, u_1), ..., x_n : gen(\Gamma, u_n)\}\end{array}}{\Gamma \vdash \mathsf{let}\ x_1 = e_1; ...\ ;\ x_n = e_n\ \mathsf{in}\ e : u}$$

Figure 5. Type inference laws (AS, TUP, APP, ABS, SUM, REL, PROD, and QUOT as before)

6.2 Using the inference laws

Before we can apply these inference laws we require an initial type environment. In general, this should consist of type schemes for any input variables, any library functions (e.g., `cos` and `sin`), and the ϕ function. The types chosen for input variables and library functions can be as specific, or as general as one wishes. The function ϕ on the other hand, has the type scheme $\forall\alpha.(\alpha \times \alpha) \rightarrow \alpha$. For our example program we have chosen to leave the types of the input variables in a symbolic form, and to give the trigonometric functions the type $\mathrm{rad} \rightarrow \mathbf{1}$.

Applying the inference laws to the let-expression in Fig. 3 we arrive at the following symbolic units for the two output variables:

```
(display_eastings, display_northings):
    U.s^2.V^-1 × U.s^2.V^-1
```

where V is the units of `mass_calibration` and U is the units of `accelerometer`. The only ground units appearing in this type ('s') come from the coercion in the definition of `PERIOD`. (We could have even used a variable in the coercion, and we would have arrived at a completely symbolic result.) This symbolic expression seems to say little about the units used in the program, but at the very least it tells us that the program calculates eastings and northings in the same units.

If we were provided with some more information about the program's inputs we could simplify these symbolic units further. For example, if we were told that the accelerometer provides readings in newtons (i.e., $\mathrm{kg.m.s}^{-2}$) and that the vehicle's mass is calibrated in kilograms, we could substitute these units for U and V respectively:

$$U.\mathrm{s}^2.V^{-1} = (\mathrm{kg.m.s}^{-2}).\mathrm{s}^2.(\mathrm{kg})^{-1} = \mathrm{m}$$

This dramatic simplification shows that merely supplying units for the input devices, which would normally be possible in any practical situation, allows us to get exactly the results we want.

On the other hand, if the vehicle's mass was calibrated in pounds (mass), we would arrive at the more startling result $\mathrm{kg.m.lb(m)}^{-1}$, which would alert the programmer to a likely error.

6.3 Polymorphism

The polymorphic type system defined above affords us significant power when inferring the units of measurement in the presence of functions. For example, in our example program in Fig. 1, the vehicle's speed (line 41) and heading (line 42) are both updated by adding their previous values to their rates of change multiplied by `PERIOD`; i.e., they are defined by integration over time. We could write a function to perform this integration for us:

```
integrate(int v, double dv_dt) {
  return v + dv_dt * PERIOD;
}
```

which is translated to the let-equation:

```
let integrate = λv.λdv_dt.v + dv_dt*PERIOD
in ...
```

The inference algorithm will assign the right-hand side of this equation the type $U \rightarrow U.\mathrm{s}^{-1} \rightarrow U$. When `integrate` is used in the body of its let-expression, however, this type is generalised to $\forall\alpha.\alpha \rightarrow \alpha.\mathrm{s}^{-1} \rightarrow \alpha$, allowing it to be used at different types.

In our example, we would have inner let-equations representing the updates of the vehicle's speed and heading:

```
speed3 = integrate speed2 a1
heading3 = integrate heading2 rate_gyro
```

If $\mathtt{speed}_2$ and $\mathtt{heading}_2$ are assumed to have the symbolic units V and W, respectively, then the `integrate` function would be instantiated to the types $V \rightarrow V.\mathrm{s}^{-1} \rightarrow V$ and $W \rightarrow W.\mathrm{s}^{-1} \rightarrow W$ in the respective equations.

7 Conclusions and future work

Programs that measure quantities from their physical environment must perform all their calculations in correct and consistent units of measurement. We have presented an analysis technique to help programmers verify their programs with respect to this requirement. Our approach is more sophisticated than earlier efforts because it combines a powerful type inference-based approach with a novel technique for reasoning about imperative programs in a functional style. The overall result is an analysis technique which allows flexibility in its application (through its

support for symbolic and polymorphic treatment of units) and promotes mathematical expressiveness in programs (by avoiding onerous annotation).

The technique we have presented fails to account for some programming language constructs. Aggregate data types such as structures and arrays could be accommodated by a straightforward extension of the type system presented in Sect. 6. Pointers, however, pose a more significant problem. We hope to apply alias analysis techniques [13] to provide a more general translation scheme than that presented in Sect. 5.

This research was conducted as part of an ongoing project investigating techniques for analysing legacy embedded programs. We plan to extend the work presented in this paper to make it applicable to machine code programs for which no high-level language representation is available.

We have constructed a simple Prolog-based prototype implementation of the inference technique discussed in Sect. 6. This prototype demonstrated the viability of our basic approach. A more complete implementation is planned.

8 Acknowledgements

This work was funded by Australian Research Council Discovery-Projects grant DP0449773, *Verified Emulation of Legacy Mission Computer Systems*.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[2] B. Chiswell and E. Grigg. *SI Units*. John Wiley and Sons Australasia, 1971.

[3] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] A. Dreiheller, M. Moerschbacher, and B. Mohr. PHYSCAL – Programming Pascal with physical units. *ACM SIGPLAN Notices*, 21(12):114–123, December 1986.

[5] N. Gehani. Ada's derived types and units of measure. *Software – Practice and Experience*, 15(6):555–569, June 1985.

[6] I. Hayes and B. Mahony. Using units of measurement in formal specifications. *Formal Aspects of Computing*, 7:329–347, 1995.

[7] R. House. A proposal for an extended form of type checking of expressions. *The Computer Journal*, 26(4):366–374, 1983.

[8] M. Karr and D. Loveman. Incorporation of units into programming languages. *Communications of the ACM*, 21(5):385–391, May 1978.

[9] A. Kennedy. Dimension types. In *Proceedings of the 5th European Symposium on Programming (ESOP'94)*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362. Springer-Verlag, April 1994.

[10] R. Männer. Strong typing and physical units. *ACM SIGPLAN Notices*, 21(3):11–20, March 1986.

[11] S. McElroy, I. Robbins, G. Jones, and D. Kinlyside. *Exploring GPS: A GPS User's Guide*. GPSCO, 1998.

[12] R. Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[13] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[14] P. Neumann. Letter from the Editor: Risks to the public. *ACM SIGSOFT Software Engineering Notes*, 10(3):3–16, July 1985.

[15] J. Oberg. Why the Mars probe went off course. *IEEE Spectrum*, 36(12):34–39, December 1999.

[16] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, January 1965.

[17] G. Rosu and F. Chen. Certifying measurement unit safety policy. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 304–309, Montreal, Canada, October 2003. IEEE Computer Society Press.

[18] M. Wand and P. O'Keefe. Automatic dimensional inference. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: in honor of J. Alan Robinson*, pages 479–486. MIT Press, 1991.