Fully Static Dimensional Analysis with C++

Zerksis D. Umrigar
447 Plaza Dr., 5-57, Vestal NY 13850
(607) 729-5919
Email: umrigar@cs.binghamton.edu

Abstract

Dimensional analysis is widely used by those involved in the physical sciences as a quick check on the dimensional sanity of formulas. Many programming errors can be prevented by performing similar checks automatically. Previous attempts at such dimensional analysis of computer programs either entailed some run-time overhead [CG88] [Hil88], or required the modification of an existing programming language to add such a capability [Geh77] [Hou83]. This paper describes how an existing language C++ [Str91] can be used to write programs which can be guaranteed to be dimensionally correct **before** run-time. Unfortunately, the method depends on a language construct which violates the language definition. The paper argues that the rule forbidding the construct be removed from the language definition, or that the forbidden construct be allowed as an extension by most compilers. In fact, a popular C++ compiler does accept this construct. This compiler was used to implement a general package for dimensional analysis. The capabilities of the package are briefly illustrated by means of an example. Ultimately, the method should have negligible run-time costs.

1 Introduction

There are two aspects to the dimensional analysis of computer programs:

1. Verifying the dimensional integrity of expressions. This prevents nonsense like adding mass to time.

2. Provide automatic scaling between quantities. For example, if a quantity in inches is added to a quantity in centimeters, one of the quantities needs to be scaled appropriately before the addition can be performed.

The proposed method provides (1). It circumvents the need for (2), by representing all quantities internally using only a single reference unit (specified by the programmer) for each dimension.[1]

The method allows static checking of the dimensional consistency of expressions involving the usual arithmetic and relational operators with negligible run-time overhead (in theory). It is also possible to specify dimensioned literals using arbitrary units, as well as do I/O of dimensioned quantities with full static checking. The term *static checking* means that the checking is done before run-time: usually at compile-time, but sometimes at link-time.

The sequel assumes that the reader has some familiarity with C++.

2 The Basic Idea

The method makes extensive use of C++ templates to track the dimensions of quantities at compile-time. The basic idea is straight-forward and is illustrated by three simple C++ program fragments. The first fragment declares a class `Dim` which represents dimensioned quantities in a dimensional space characterized by the three dimensions *mass*, *length* and *time*.

```
template<int M, int L, int T> class Dim {
  double val;                    //Actual value.
public:
  Dim(double v= 0) { val= v; }  //Constructor.
  double value() { return val; }//Accessor.
};
```

When the `Dim` class is used for a particular dimension, the template arguments `M`, `L` and `T` provided should correspond to the exponents of mass, length

[1] In this paper, we distinguish between the terms "dimension" and "unit". Informally, a *dimension* corresponds to the quantity which is measured, whereas a *unit* is a standard measure of the quantity, relative to which all other measurements are done. For example, a *gram* is a unit for the *mass* dimension.

and time in that dimension. Using this class it is easy to enforce dimensional consistency. For example, to enforce the restriction that only quantities having identical dimensions can be added together, we can overload the + operator to work on Dims as follows:

```
template<int M, int L, int T>
Dim<M, L, T> operator+(Dim<M, L, T> &d1,
                       Dim<M, L, T> &d2) {
  return Dim<M, L, T>(d1.value() + d2.value());
}
```

Since operator+ is defined only for Dims with the same template arguments (corresponding to the same dimensional exponents), the C++ compiler will enforce the restriction that only quantities having identical dimensions can be added together.

Multiplying two dimensioned quantities together produces a dimensioned quantity whose dimensional exponents are the sum of the dimensional exponents of the operands. Again, using template arguments for the dimensional exponents, we can overload the * operator to work on Dims as follows:

```
template<int M1, int L1, int T1,
         int M2, int L2, int T2>
Dim<M1 + M2, L1 + L2, T1 + T2>
  operator*(Dim<M1, L1, T1> &d1,
            Dim<M2, L2, T2> &d2) {
  return Dim<M1 + M2, L1 + L2, T1 + T2>
           (d1.value() * d2.value());
}
```

The above examples summarize the essential ideas behind the proposed method. The rest is mere syntactic sugar (though essential to developing a usable system).

3 But is it C++?

Unfortunately, the two function templates used in the last section are not allowed by the C++ Annotated Reference Manual (ARM) [ES90]. On pg. 347 of the ARM there is a categorical rule:

> All *template-arg*s for a function template must be `type-argument`s.

We violate this rule, since our template arguments are `int`s. We can avoid the problem for the operator+ function by simply redefining operator+ as a member function of the Dim class. However, in the absence of nested template declarations (the ARM forbids them), we cannot declare operator* to be a member function as there does not appear to be any way to bind the template arguments of the second operand. We are left with no choice but to use a function template to define operator*. This makes our programs violate the language definition and unacceptable to at least one existing compiler (Borland's BCC 4.0). Fortunately, another existing compiler (the Free Software Foundation's G++ [Sta93]) accepts such function templates.

The justification presented by the ARM for the problematic rule is that template arguments "must be deduced from actual arguments in calls of the template function." The rule appears to be too strong for its justification; in fact the weaker rule on pg. 346 of the ARM

> Every *template-arg* specified in the *template-arg-list* must be used in the argument types of a function template.

appears to be sufficient, and eliminates our problem.

Another problem with forbidding function templates with non-type template arguments, is that it appears impossible to define a non-inline function which is a friend of a class which uses non-type template arguments. Also, since there are no restrictions on the use of non-type template arguments within member functions, permitting their use in function templates should not add greatly to the complexity of a compiler. Hence, even if it is not possible to allow such function templates in the language definition, compiler writers could permit their use as a language extension.

4 An Example

Using G++, it was possible to implement a general dimensional analysis package Dim.h which incorporates the ideas discussed in section 2. Some of the capabilities of this package are illustrated by the following program which uses C++ to simulate a block sliding down a surface. The program is based on similar examples presented in [CG88] and [Hil88].

Figure 1 shows the declarations in the program. Lines 5 and 6 give definitions used by the package: N_DIM is the number of dimensions, and DIM_VAL_TYPE describes the default underlying type for dimensioned quantities. The package is included on line 9, and defines several macros which are used subsequently. On lines 17–20 we use one of those macros to declare

```
01 #include <iostream.h>
02
03 #include <math.h>
04
05 #define N_DIM 3
06 #define DIM_VAL_TYPE double
07 //#define DIM_NO_CHECK
08
09 #include "Dim.h"
10
11 //Name each dimensional axis.
12 #define MASS_DIM     0
13 #define LENGTH_DIM   1
14 #define TIME_DIM     2
15
16 //Declare units.
17 UNIT(Gm, MASS_DIM, 1.0, "gm");
18 UNIT(Cm, LENGTH_DIM, 1.0, "cm");
19 UNIT(Feet, LENGTH_DIM, 2.54*12, "feet");
20 UNIT(Sec, TIME_DIM, 1.0, "sec");
21
22 typedef DIM( 1,  0,  0) Mass;
23 typedef DIM( 0,  1,  0) Length;
24 typedef DIM( 0,  0,  1) Time;
25 typedef DIM( 0,  1, -2) Acceleration;
26 typedef DIM( 1,  0, -1) Friction;
27 typedef DIM( 0,  1, -1) Velocity;
28 typedef DIM( 0,  0,  0) Number;
```

Figure 1: Program Declarations

units along each dimensional axis with conversion factors (to the chosen reference units) and print-names. From the given conversion factors, we can deduce that `Gm` is the chosen reference unit along `MASS_DIM`, `Cm` along `LENGTH_DIM` and `Sec` along `TIME_DIM`. Using `UNIT(`*Name*`, ... )` also creates a scaling object called *Name* and a units object called *Name*`_`. Specifically, the given declarations create scaling objects `Gm`, `Cm`, `Feet` and `Sec` and unit objects `Gm_`, `Cm_`, `Feet_` and `Sec_`. The use of these objects will be described later. Finally we `typedef` mnemonic names for several dimensioned types.

Figure 2 shows the program code. We first define an auxiliary function `height()`. Note that the multiplication of a dimensioned length by the scalar `0.5` is accepted by the system. In the main program which follows, we first declare dimensioned constants and variables on lines 6–14. Note that the scaling objects `Gm`, `Cm` and `Sec` created by the `UNIT` macro are used to create dimensioned literals in a very natural way. The compiler will signal an error if the dimension of the initializing expression conflicts with the dimension of the variable being declared.

On lines 16–20 we output prompts and input dimensioned quantities. Scaling objects are used in a manner similar to manipulators [Str91] to allow dimen-

```
01 static Length height(Length x) {
02   return 0.5 * x;
03 }
04
05 int main() {
06   const Acceleration g= 980.7 * Cm/(Sec*Sec);
07   const Mass blockMass= 1000 * Gm;
08   const Length deltaX= 0.01 * Cm;
09   const Friction friction= 20 * Gm/Sec;
10   const Number a= 1.0;
11   Length x, y;       //Initialized to 0.
12   const Time deltaT= 0.1*Sec;
13   Velocity v;
14   Time maxTime;
15
16   cout << "Enter initial velocity (cm/sec): "
17        << flush;
18   cin >> Cm/Sec >> v;
19   cout << "Enter time limit (sec): " << flush;
20   cin >> Sec >> maxTime;
21
22   for (Time t= 0*Sec; t < maxTime; t+= deltaT) {
23     Number slope= (height(x + deltaX) - y)/deltaX;
24     Number cosAngle=
25       1.0/sqrt(1 + +(slope * slope));
26     x+= v * deltaT * cosAngle;
27     y= height(x);
28     v+= deltaT * (g * slope * cosAngle -
29                   v * friction / blockMass);
30     cout << "At T= " << Sec_ << t << " ";
31     cout << "X= " << Feet_ << x << "; ";
32     cout << " Y= " << Feet_ << y << "; ";
33     cout << " and V= " << Feet_/Sec_ << v << "; ";
34     cout << '\n';
35   }
36
37   return 0;
38 }
```

Figure 2: Program Code

sional validation of the input statements at compile time. These input manipulators also serve to scale the input values from the specified units to their chosen reference units.

Lines 22–29 illustrate the very natural expressions possible due to operator overloading. If any of the expressions is dimensionally inconsistent, a compile-time error occurs. Line 25 uses the unary `+` operator to extract the actual value of a (dimensionless) `DIM` expression (as in [Hil88]). Lines 30–34 illustrate the use of the unit objects (created by the `UNIT` macro) as output manipulators to output the print-names of the units for a quantity along with its magnitude. The unit objects in the output statements also perform automatic scaling of the length dimension from its internal representation in `Cm` to its output in `Feet`. Once again, a compile-time error results if the dimension of the constructed unit object does not match the

dimension of the quantity being output.

The implementation and capabilities of the `Dim.h` package will be more fully described elsewhere. For now, we simply list some of its capabilities:

- Allows up to 10 dimensions.
- Partial support for using arbitrary underlying types for dimensioned quantities.[2] Hence it is possible to have dimensioned `structs` (which may be convenient for representing a vector using polar coordinates).
- Dimensional checking (and any consequent overhead) can be turned off completely by defining a single preprocessor symbol (see the comment on line 7 in Figure 1). With this option, most template classes and the entire `DIM` class disappear. Operations on dimensioned quantities operate directly on the underlying type.
- The use of arbitrary rationals as dimensional exponents. Dimensional expressions are allowed to be raised to **constant** rational powers, albeit with a relatively clumsy syntax.[3]

5 Analysis of the Method

The advantages of the proposed method are as follows:

- Full analysis of dimensional correctness **before** run-time.
- No run-time space overhead for dimensional data, unlike the method proposed in [CG88]. Space is used for storing conversion factors and print-names, but similar amounts of space would be used by any method which converted between units or printed them out.
- Once compiler technology for processing inline template functions is sufficiently developed, there should not be any substantial time or code-space overhead. This follows from the statement on pg. 342 of the ARM that the C++ template mechanism "allows close to optimal run-time performance through macro expansion of definitions and inlining of function calls."
- The method almost fits into an existing language and is accepted by an existing compiler. This is in contrast to methods like those in [Geh77] or [Hou83] which require changes to existing languages.
- Higher programmer productivity, since programmers need not spend their time tracking down dimensional errors at run-time.

Some of the disadvantages of the method, along with possible solutions are:

- All quantities having a particular dimension use the same internal unit. Hence the programmer does not have sufficient control over the precision of dimensional quantities, which may lead to an accumulation of floating point error. This may be the most serious disadvantage of the method. It may be possible to extend the method to allow multiple reference units along each dimension. In the interim, a possible workaround would be for the programmer to assign the same quantity to multiple dimensions having internal units of different precision, and manually program operations involving these dimensions.
- The binary arithmetic operations require that their operands be dimensioned quantities with *identical* underlying types. This can be inconvenient when dimensioned quantities contain multiple underlying types. For example, addition of a dimensioned `int` to a dimensioned `double` is not supported.
- No direct support for derived units. However, it is always possible for the programmer to include definitions like:

  ```
  #define Newton (Kg*Meter/(Sec*Sec))
  #define Newton_ (Kg_*Meter_/(Sec_*Sec_))
  ```

  and to provide output functions to handle such derived units.
- The set of dimensions required by the program must be known at compile-time.
- It is not possible to represent relationships between units having different origins (like that between the Celsius and Fahrenheit temperature scales). This should not be a problem as such situations are relatively uncommon.
- An increase in compilation time (by over a factor of 3 for the example in section 4). However, it is still much easier for a compiler rather than a human to find errors.

[2] Full support is not provided because of an internal bug in G++.

[3] It has not been possible to debug this feature fully, as G++ does not currently implement template argument expressions involving `*` and `/`.

- In some C++ environments, a few dimensional errors may not be detected until link-time when definitions are generated for template functions. This may be rather inconvenient.
- Errors are not reported in terms of dimensional violations, but in terms of missing function definitions. The error messages often contain instantiated template arguments and even mangled function names. It may be hard for a programmer to relate an error message to a dimensional error. Tools could be developed to convert concrete error messages to more abstract dimensional error messages.
- Some C++ environments expect help from the programmer in determining which template instantiations are required. The package provides the programmer with a macro to declare instances of all the (hidden and visible) template classes used by it, but the programmer still needs to determine which instances need to be declared.
- The code produced by the one compiler (G++) known to accept the package is currently of poor quality with no inline expansion of template functions. Hopefully, this problem should disappear over time, for the reasons mentioned above. In the interim, a program should first be compiled with full dimensional checking. Once it is free of dimensional errors, it can be recompiled with dimensional checking turned off (as discussed in the previous section). However, this recompilation is rather inconvenient and some programmers may sometimes choose to omit the first compilation step.

6 Conclusions

It should be realized that it is possible to perform compile-time dimensional analysis in any language which has name-equivalence of structures. The programmer needs to define a unique structure for each dimension which may be required by the program at run-time, and also define functions implementing all the required operations over all the dimensions. This approach suffers from poor syntax, run-time overhead due to function-call overhead and inordinate tedium for the programmer. C++ provides a happy combination of operator overloading (which provides readable syntax), inline functions (which eliminate function-call overhead) and templates (which replace the programmer's tedium with that of the compiler) It is the coming together of these features which makes static dimensional analysis practical.

Note

The dimensional analysis package mentioned above runs under G++ version 2.5.8. It can be obtained gratis by those who are interested, by sending an email request to the author.

Acknowledgments

The essential ideas behind this paper were conceived while the author was with the computer science department at the State University of New York at Binghamton.

References

[CG88] Robert F. Cmelik and Narain H. Gehani. Dimensional Analysis with C++. *IEEE Software*, pages 21–27, May 1988.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, New York, 1990.

[Geh77] Narain Gehani. Units of Measure as a Data Attribute. *Computer Languages*, 2(3):93–111, 1977.

[Hil88] Paul N. Hilfinger. An Ada Package for Dimensional Analysis. *ACM Transactions on Programming Languages and Systems*, 10(2):189–203, April 1988.

[Hou83] R. T. House. A Proposal for an Extended Form of Type Checking of Expressions. *Computer Journal*, 26(4):366–374, November 1983.

[Sta93] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, 675 Massachusetts Av., Cambridge, MA 02139, October 1993.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, New York, second edition, 1991.