Dimensional Analysis and Partial Evaluation

by

Xianbin Long

A Thesis

Present to the Faculty of Graduate Studies

in partial fulfilment of the requirements

for the degree of

Master of Science

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba

0-612-23390-1

DIMENSIONAL ANALYSIS AND PARTIAL EVALUATION

BY

XIANBIN LONG

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree

of

MASTER OF SCIENCE

Abstract

Incorporating units of measure into a programming language is beneficial for dimensional analysis and error detection. One of the methods of doing this is to use the abstraction facilities of an existing programming language. In this thesis, I present the design of a units package for manipulating units of measure along with numerical values for the Safer_C language and discuss methods of using partial evaluation to improve the performance of the units package. This work generalizes and enhances previous work and analysis and applies the new analysis to the Safer_C programming language. In particular, it examines the use of partial evaluation to perform dimensional analysis. This work consists of three parts: 1) A survey of methods for incorporating units of measure into programming languages and research on partial evaluation for this purpose. 2) A presentation of a design for a units package and discussion of several important features of the package namely: dynamic dimensions, computation of rational powers, handling precision, and handling temperature computations. 3) A presentation of a technique of partial evaluation to achieve good performance for dimensional analysis. With partial evaluation, the units checking and computation can be done at compile time. The results of experiments show that my design of the units package is correct and that existing partial evaluation technology can be used to improve the efficiency of dimensional analysis.

Acknowledgments

I would like to thank Dr. D. Salomon for suggesting this topic of study to me, for his invaluable counseling, extensive guidance, and unhesitating willingness to review my manuscript throughout the evolution and culmination of this investigation. I would also like to thank professor Dereck Meek for his precious time in reading my thesis and his many valuable comments. Finally, I would like to thank professor George C. Tabisz of the Department of Physics for providing helpful comments and serving on my examining committee.

Contents

List of Figures and Table

Chapter 1

Introduction

1.1. The Problem

Dimensional analysis plays an important role in scientific investigation. In many fields, such as physics and engineering, it is common practice to associate units of measure with variables and constants and to carry these units along with computations. Scientific equations are built not on abstractions but on measurements of actual phenomena. It is quite possible for an equation to be mathematically correct and yet be scientific nonsense. Dimensional analysis is the study of measurement and its influence on scientific relationships. The techniques of dimensional analysis are widely used in science to derive theoretical relationships.

Unfortunately, most programming languages such as Pascal, C, C++, FORTRAN, and PL/I do not deal with units of measure. When engineers use computers to solve their problems, they have to do the dimensional analysis manually. It has become clear that programs in high-level languages should in some way provide the mechanisms needed to support dimensional analysis.

1.2. Related Work

1.2.1 Units in Programming Languages

The earliest language which allowed units is the ATLAS language (Abbreviated Test Language for All Systems) [Atl82]. ATLAS allows only a limited set of units and a limited language for constructing combinations of units. The ATLAS language is intended to be used for the writing of test programs for Units Under Test (UUTs), so that these programs can operate on various makes and models of Automatic Test Equipment (ATE).

Units of measure in general high-level languages are discussed by Gehani [Geh77]. Gehani describes dimensional analysis for a more general high-level language, Pascal. Gehani proposes the inclusion of the units of the quantity being represented as an additional attribute in high-level programming languages. He argues that all or most of the additional processing required by the units attribute can be done at compile time.

House [Hou83] gives a critique of Gehani's work and proposes a method which can be completely implemented at compile time. He discusses language syntax issues and efficient implementation.

In dimensional analysis, an important aspect is units conversion. Karr and Loveman [KL78] propose the incorporation of units into programming languages; they discuss the relationship between units conversion and linear algebra, dimensional analysis, and language syntax issues. Novak [Nov95] presents efficient algorithms for converting units of measurement from a given form to a desired form. For saving space and increasing efficiency, Novak discusses the representation of the dimensions. He packs a vector of eight integers into bit fields within an integer word, and implements units for the GLISP language.

Gehani, Karr & Loveman, and House, all require that the language definition be changed to support dimensional analysis. Gehani [Geh85] and Hilfinger ([Hil83] and [Hil88]) describe methods for using Ada's abstraction facilities to use an existing programming language to support dimensional analysis. Hilfinger describes methods for including units with numeric data using Ada packages, and discusses modifications of Ada compilers that would be required to make the use of these packages efficient [Hil88]. There are several other packages which are discussed in [CG88], [Cun92], and [Umr94]. Cmelik and Gehani gives a package for handling units of measure in C++ using classes. Cunis discusses a package for handling units of measure in Lisp. Umrigar gives a package for handling units of measure in C++ using templates. All these packages have some disadvantages, which we will discuss in chapter 2.

1.2.2 Partial Evaluation

Partial evaluation is a program optimization whereby as much as possible of the computation specified in the program is carried out before the program is translated to machine language. Any computation possible on literal constants or relatively stable input data supplied by the programmer is carried out, and the results are propagated through the program. The resulting simplification of the final program can lead to a substantial speed improvement. Partial evaluation has been the subject of a rapidly increasing amount of activity over the past decade due to recent advances both in theory and practice [BEJ88], [CD93], and [JGS93].

Partial evaluation has been successfully applied to declarative languages, such as Scheme and Prolog. Recent years have seen a growth in the study of partial evaluation in imperative programs [Cha90], [Mey91], [And93a], [WL95], and [KKZG95]. Anderson [And93a], [And93b] describes a partial evaluator for a substantial subset of C. Kleinrubatscher, Kriegshaber, Zochling, and Gluck [KKZG95] describe a partial evaluator for a substantial subset of Fortran 77.

Baier, Gluck, and Zochling [BGZ94] investigate the application of partial evaluation to numerically-oriented computation and engineering applications. Salomon [Sal96] uses partial evaluation to support many important language features and implements a partial evaluator for Safer_C.

Despite the successful application of partial evaluation to many fields, few attempts have been made to study the partial evaluation of dimensional analysis [Hil88].

1.3. Research Objectives

In this thesis, I study dimensional analysis in Safer_C and combine it with partial evaluation to improve the efficiency of programs which use units of measure. I generalize and extend previous units package and analysis to the Safer_C programming language. I investigate some problems which are relevant to the features of the dimensional analysis package, the system, and partial evaluation. For decreasing the size of the implementation, I make maximum use of the existing features of Safer_C. For example, I use the partial evaluator which exists in Safer_C. I also suggest improvements to the partial evaluation techniques that would speed up dimensional analysis, with the goal that any such improvements would enrich all uses of the language, not just dimensional analysis.

1.3.1 The Method

Basically, there are two methods of supporting dimensional analysis.

a) The programming language itself could support dimensional analysis as a feature. This method requires changes to the language intended solely for supporting dimensional analysis. It may be impossible to change existing programming languages to meet this goal.

b) Use the abstraction facilities of an existing language to construct a units package to support dimensional analysis. Method b) has several advantages. First, we do not need to change the source language specifically for dimensional analysis. The standard version of the language still can be used. Second, the user does not need to learn a new language for doing dimensional analysis. Finally it is easier to implement. Therefore I intend to use method b) for dimensional analysis in Safer_C.

Although a predefined units package for dimensional analysis has several benefits, the units checking would normally have to be done at run time instead of compile time. That means that the execution of programs which make use of dimensional analysis would be slow. For tackling this problem, I propose the use of partial evaluation.

There are several reasons for choosing partial evaluation to speed up execution. First, when units are declared along with variables or constants, the units are static, since we know the units at compile-time. Thus we can completely deal with these static quantities at compile time. Second, in Safer_C the partial evaluator already exists. Therefore this makes the work much easier. We only need to use and perhaps expand the existing partial evaluator to deal with different units components. Finally, Safer_C has the same computational power at translation time as at run time. Thus we can do any necessary computation at compile-time rather than at run time.

1.3.2 Structure of the System

The system which supports units of measure in Safer_C basically consists of a parser, a partial evaluator, and a units package. My work principally involves the partial evaluator and the units package. It includes the following parts:

1) Designing the features of dimensional analysis which support units of measure to be provided by the system. The features of dimensional analysis directly indicate how powerful the system is.

2) Designing a convenient notation for specifying units of measure. This is important because if users feel uncomfortable with the notation they will not use the system.

3) Discovering under which circumstances, and which parts of dimensional analysis can be carried out during partial evaluation.

4) Implementing a demonstrational units package which supports dimensional analysis.

Whenever the users want to use units, they simply declare the units of each variable and manipulate these units using ordinary operations. The system will do the units checking.

From the user's point of view the system should have the following characteristics [Hil88]:

(1) It must be possible to declare each variable, constant, and parameter to have a particular unit of measure and to perform the ordinary arithmetic operations between quantities having the declared units of measure.

(2) There should be some provision for handling conversion of commensurable units.

(3) There should be compile-time checking for dimensional consistency.

The system can be represented by the following diagram

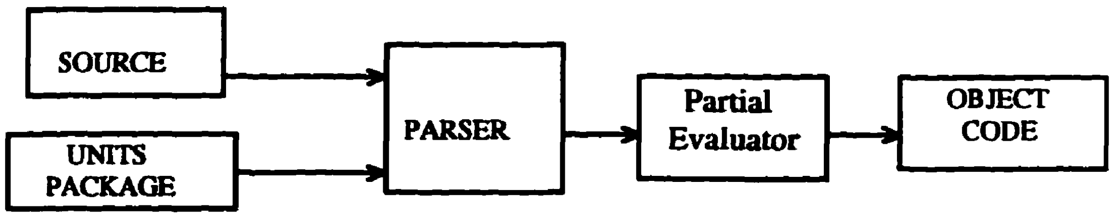

Figure 1.1 System Diagram

The source is the user's code which includes some operations using units of measure. The units package is constructed by using Safer_C abstraction facilities, such as operator overloading, and parameterized constructor. The source and the units package are sent to the parser. According to the source and units package, the parser generates the intermediate code, which is a parse tree. Then the parser tree is sent to the partial evaluator (PE). According to the annotated evaluation time, the partial evaluator performs the partial evaluation. The partial evaluator does the units consistency check and units operations. By partial evaluation, the manipulation of units of measure can be eliminated from the object code as much as possible. Therefore we can get a faster running program.

1.4. Perceived Benefit

The work described here has several benefits. First of all, it offers dimensional analysis. This is important to scientists and engineers as it gives a check on the correcmess of their formulas. The checking is similar to that traditionally performed by scientist on their own hand calculations. In addition, dimensional analysis can accurately perform the conversion between dimensional quantities in different systems of units. With this feature the users would need to declare only the units of quantities and the system then would automatically manipulate the units and do the dimensional analysis. As a brief example, a programmer could code the following program to compute a speed:

```
units_package()
<<main>> :: func() void
block
        Speed :: Doubleu(0.0, km/hour)
        Time :: Doubleu(5, hour)
        Distance :: Doubleu(600.0, km)
        Speed := Distance/Time
        Printu (Speed)
end
```

When executing the program it will printout: 120(km/hour).

Second, using the information provided by units, more errors can be detected. Third, thanks to partial evaluation the system would be efficient. Incorporating units of measure into a programming language would require a lot of space and computation. If a program runs slowly as a result of the incorporation of units into the program then fewer programmers would want to use the feature of dimensional analysis. Thus the work described here is an important step towards putting a programming language which supports units of measure into practical use. Finally, the system demonstrates the application of some important language features and partial evaluation.

1.5. Thesis Structure

My thesis consists of the following chapters:

1. Introduction
2. Related Work — Survey the related work on units of measure in programming languages and partial evaluation in imperative languages.
3. Background Knowledge — Discuss some background knowledge which is relevant to dimensional analysis and partial evaluation.
4. Design — Present a design of the features of dimensional analysis and a convenient notation.
5. PE units package in Safer_C — Study partial evaluation and a units package which can efficiently support units of measure.
6. Implementation — Construct a prototype of a units package in C++.
7. Conclusion and future work — Present conclusions, and suggestions for future work.

Chapter 2

Related Work

In this chapter, I survey the existing work on dimensional analysis in programming languages and on partial evaluation that is directly related to my work.

2.1 Introduction

Programming languages have improved a lot since the earliest high-level languages appeared in the 1950s. Many new features have been added to programming languages since then but the design of high-level languages has not yet been perfected. Incorporating units of measure into programming language is an interesting research area of programming language design. Such a feature is called dimensional analysis. Units of measure play an important role not only in scientific investigation but also in our every day life. Unfortunately, most programming languages such as C, C++, Fortran, and PL/I do not deal with units of measure. It has become clear that programs in a high-level language should in some way provide physical and mechanical units-e.g. volts, hertz, kgm, dyne, etc. From the point of view of programming languages, explicit mention of units can not only enhance readability of programs but also increase the ability of a programming language to correct errors of inconsistent units. Research into dimensional analysis in programming languages over the past twenty years has achieved many results. Two methods have been adopted to incorporate units of measure into programming languages. One is to modify an existing language to directly incorporate units of measure into the syntax and semantics of the language. The other is to use the existing high-level features provided by a programming language to implement dimensional analysis. The following are the main design problems which we should consider when we incorporate units into a programming language.

1. How to represent the units themselves in the source code.
2. How to deal with conversion between two commensurate units.
3. How to provide compile time consistency checking.
4. How to permit efficient implementation.
5. How to provide precision control.

2.2 Units in Programming Languages

2.2.1 Atlas

The earliest programming language which allowed units of measure was ATLAS language (Abbreviated Test Language for All Systems) [Atl82]. ATLAS was developed originally for avionics applications under the auspices of Aeronautical Radio, Inc. (ARINC) and under the direction of the Airlines Electronic Engineering Committee (AEEC), which approved the original version on October 10, 1968. ATLAS was approved by the United States Department of Defense as an interim standard language for automatic test equipment (1976).

ATLAS is a standard abbreviated English language used in the preparation and documentation of test procedures which can be implemented either manually or with automatic or semi-automatic test equipment. The ATLAS language is intended to be used for the writing of test programs which describe test procedures for a wide class of Units Under Test (UUTs), so that these programs can operate on various makes and models of Automatic Test Equipment (ATE).

Since ATLAS is specific for testing, ATLAS allows only a limited set of units of measure and a limited language for constructing combinations of units.

2.2.2 The Early Work of Gehani

In 1977, Gehani [Geh77] discussed the units of measure in the general high-level programming language Pascal. Gehani's arguments can be summarized as follows. The basic function of a computer program is manipulating data. An important attribute of a datum is its type. In a high-level typed language, the type of a variable determines the range of values which the variable can have and the set of operations that are defined for variables of that type. Using type information we can enhance the error detection capabilities of a compiler. A compiler should report an error if an operator is applied to an incompatible operand; for example, if a pointer variable and a float variable are added together. Similarly, a units error should be detected when incompatible units are combined; for example when a value with units of speed is assigned to a variable with units of volume. Gehani proposes the inclusion of the units of the quantity being represented as an additional attribute in high-level programming languages. Using the notation proposed by Gehani we can write down following program segment:

```
var T: real UNITS(*);
   V: real UNITS (METER = 3);          {METER =3 means V has unit m^3 }
   W: real UNITS (KILOGRAM);
      ...
   T := V + W;
      ....
```

where UNITS(*) means that the temporary variable T can be used to hold a value having different units, V is a real and has units attribute meter = 3, and W is a real and has units attribute kilogram. In the statement T := V + W, the compiler should produce an error message since V and W have different UNITS attribute. Gehani gives a detailed discussion of the notation for the units attribute in Pascal, the computation on the new data (which has value, type attribute, and units attribute), conversion, and implementation. For efficient performance, Gehani claims that checking for consistency of units can all be done at compile time if the following restrictions are made:

(1) Expressions with units may be exponentiated only to constant or compile time determinable values.

(2) The expressions representing the exponents in the units attribute declaration may be constants or compile-time determinable values.

(3) Variables with the attribute UNITS(*) are not allowed to be assigned values with different units depending upon certain conditions (and therefore program flow).

For example, if T has the attribute UNITS(*), then the statement:

if e then T := e_1 else T := e_2

should not be permitted if e_1 and e_2 have different units.

2.2.3 The Work of House

House [Hou83] gives a critique of Gehani's work. The main objection by House is that Gehani's implementation scheme is not capable of performing the type of units checking required of it. House gave an example which satisfies all the conditions given by Gehani but cannot check consistency errors. The example given by House is:

```
Program faulty;
var q: real UNITS(*);
   m: real UNITS(KG);
   a: real UNITS(M, SEC = -2);
   f: real UNITS(M, KG, SEC = -2);
function ratio (x : real UNITS(*); var y : real UNITS(*)) : real UNITS(*);
    begin
       q := a;
       ratio := x/y
    end;
procedure x(function fun(m : real UNITS(*); var n : real UNITS(*)) : real UNITS(*);
    begin
       a := fun(f, q);
    end;
begin
    q := m;
    x(ratio)
end
```

The above program is intended to calculate the ratio of a force to a mass. The result should be an acceleration (i.e. has units (M, SEC = –2)). By a circuitous route, the ratio of the variables *f* and *m* is computed by function *ratio*. However, the value of *y* is changed by the statement *q* := *a*. Thus the units of *ratio* that we get are (KG = 1). This error cannot be detected in *ratio* alone, since we do not know what are the actual parameters. We can not detect any error in the procedure *x* either, since in the procedure *x* it involved function *fun* which is the function ratio, but we do not know that yet. The source which causes the problem is that "if the two parameters should bear some given relation to each other, there is no syntactic mechanism for specifying so". Thus House proposes a method in which the relationship between parameters and return value can be specified. Using this mechanism the consistency checking can be completely implemented at compile time. He also discusses language syntax issues and efficient implementation.

2.2.4 The Work of Karr and Loveman

In dimensional analysis, another important aspect is units conversion. Karr and Loveman [KL78] propose the incorporation of units into programming languages and give a very interesting method for commensurate units checking and units conversion using linear algebra. To discuss the main idea of the method, first let us give the concept of units being commensurate. We say that "two quantities are *commensurate* if one is a constant multiple of the other." For example *feet = 12 inches*, thus *feet* and *inches* are commensurate. The basic idea of the conversion method discussed by Loveman is as follows:

If A and B are commensurate then we have

$$A = C^*B, \tag{2.1}$$

where A and B are dimensional quantities, and C is a constant. From (2.1) we have

$$A/B = C^*1. \tag{2.2}$$

The formula (2.2) means that if we want to know whether A and B are commensurate we only need to check whether A/B is commensurate with 1 or whether A/B is a constant under condition (2.1). The question of determining whether a quantity is commensurate with 1 can be answered using pure linear algebra. To make this connection, we will apply the log to each of the equations describing commensurateness.

Let U be a set which consists of all the units (base or derived) that are used. We may assume that there are n units. Let D_1, D_2 be dimensional quantities and $u', u'' \in$ U be the units of D_1, D_2, respectively, and q_1, q_2 be the measurement of D_1, D_2 respectively. If D_1, D_2 are commensurate then (2.2) will be true. Taking log on both sides in (2.2) and using e to denote q_1 / q_2 we get

$$\log u' - \log u'' - \log C = -\log e. \tag{2.3}$$

Suppose that we have equations which describe commensurateness as follows:

$$u_i = c_l \, u_j \ , \tag{2.4}$$

where c_l is a real number, $l = 1, 2, ..., k$; u_i, $u_j \in$ U, $i, j = 1, 2, ... n$. The k denotes the number of conversion relations.

Using a similar method taking log on (2.4) we get

$$\log u_i - \log u_j = \log c_l. \qquad (2.5)$$

Combining (2.3) and (2.5) we get following systems of linear equations:

$$\begin{cases} \log u_i \; - \; \log u_j \; = \; \log c_l \\ \log u' \; - \; \log u'' \; - \; \log C \; = \; -\log e \end{cases} \qquad (2.6)$$

where $i, j = 1, 2, \ldots$ n. $l = 1, 2, \ldots, k$.

Let A be the coefficient matrix of (2.6) (that is a k+1 rows by n+1 columns coefficient matrix),

$$\mathrm{X} = (x_1, \; x_2, \; \ldots \; x_n, \; x_{n+1})^{\mathrm{T}} = (\log u_1, \; \log u_2, \; \ldots \; \log u_n, \; \log C)^{\mathrm{T}},$$

and

$$\mathrm{B} = (b_1, \; b_2, \; \ldots \; b_k, \; b_{k+1})^{\mathrm{T}} = (\log c_1, \; \log c_2, \; \ldots \; \log c_k, \; -\log e)^{\mathrm{T}}$$

then (2.6) can be written as

$$\mathrm{AX} = \mathrm{B}. \qquad (2.7)$$

Therefore, if we think of X as unknown then the question of whether a quantity is commensurate with 1, when translated into linear algebra terms, becomes a question of whether the systems of linear equations (2.7) has a solution. To solve the system of linear equations (2.7), we can use some method given in any linear algebra book. For example, we can use row operations on the coefficient matrix A to give the row-echelon form. Then we could get the solution.

In general, (2.7) may not have a solution. If d = |A| ≠ 0 then system AX = B has a unique solution. Notice that the matrix is not square but in our case we care only about the solution of log *C*. Thus more specifically, we need only to consider whether we can add multiples of the rows of matrix A to the last row of A in such a way that all entries, except possibly the last two entries, are zero. If we can find the value of log *C* then we get the value of *C*. Thus we know that D_1 / D_2 is commensurate with 1 and $D_1 / D_2 = C * 1$. Let us consider an example which is given in [KL78].

Example 2.1. Finding the radius in inches of a circle whose area is one acre. Using the formula $r = \sqrt{acre / \pi}$, we want to know if $\sqrt{acre / \pi}$ is commensurate with inches, in other words, if $\frac{\sqrt{acre / \pi}}{inches}$ is commensurate with 1. Let the units order be (*acre, sec, grams, inches, feet*). Then the X will be (log *acre*, log *sec*, log *grams*, log *inches*, log *feet, log C*). The vector for $\frac{\sqrt{acre / \pi}}{inches} = C$ is:

$$(\frac{1}{2} \;\; 0 \;\; 0 \;\; -1 \;\; 0 \;\; -1 \;\; -\log\frac{1}{\sqrt{\pi}})$$

If we give the conversion relations $1\,feet = 12\ inches$ and $1\ acre = 43560\ feet^2$ then we have the following coefficient matrix:

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & -2 & 0 & \log 43560 \\ 0 & 0 & 0 & -1 & 1 & 0 & \log 12 \\ \frac{1}{2} & 0 & 0 & -1 & 0 & -1 & -\log\frac{1}{\sqrt{\pi}} \end{pmatrix}.$$

Using row operation we have

$$\begin{pmatrix} 1 & 0 & 0 & 0 & -2 & 0 & \log 43560 \\ 0 & 0 & 0 & -1 & 1 & 0 & \log 12 \\ 0 & 0 & 0 & 0 & 0 & -1 & -\log 12\frac{\sqrt{43560}}{\sqrt{\pi}} \end{pmatrix}.$$

Thus $\log C = \log 12\sqrt{\frac{43560}{\pi}}$ and the $C = 12\sqrt{\frac{43560}{\pi}} \approx 1413$. Therefore the $\sqrt{acre / \pi}$ is commensurate with inches and the number of inches of the desired radius is $12\sqrt{\frac{43560}{\pi}} \approx$ 1413.

Notice that:

1. Since $a * \log x = \log x^a$, $\log x + \log y = \log (x * y)$ we do not need to actually compute the logarithms in the last columns. Thus in the row operations, if addition is required multiplication is done, and if the multiplication is required then exponentiation is done. We can even remove the log notation from the matrix.

2. We do not need to store the column which indicates the variable log C, since in this column only last entry is non zero, therefore the row operations will not affect its value.

The method discussed above is a very interesting method as it makes a connection between units commensuration and pure linear algebra. In this method, we have to solve a system of linear equations. Normally for solving systems of linear equations the cost is expensive both for space and time. When the number of units and the conversion relations become large, the matrix A will become large. Thus another conversion method is introduced by Novak [Nov95].

2.2.5 The Work of Novak

Novak presents an efficient algorithm for converting units of measure from a given form to a desired form. The method given by Novak is based on some standard units system. For example, the standard units system is SI system. Let

$$u_1 = c * u_2,$$

where u_1 is a units, u_2 is a base units in SI system, and c is a real number. Then the c is called *conversion factor* of u_1. With conversion factor then we can convert units of measure from a given form to a desired form.

Let D_1, D_2 be dimensional quantities and $u', u'' \in$ U be the units of D_1, D_2, respectively, and q_1, q_2 be the measurement of D_1, D_2 respectively. If the conversion factor of u' is f_1, the conversion factor of u'' is f_2, and q_{si} is the equivalent quantity of D_1, D_2 in SI system i.e.

$$q_1 * f_1 = q_{si} = q_2 * f_2,$$

then we can convert q_1 in units u' to q_2 in units u''.

$$q_2 = q_1 * f_1 / f_2.$$

Novak's method saved the space which would be required by Loveman's method. Novak also discusses the representation of the dimensions for saving space and for efficiency. He packs a vector of eight integers into bit fields within an integer word, and implemented the use of units in the GLISP language.

There is a common characteristic in the work of Gehani, Karr & Loveman, and House. That is they all require that the language definition be changed to support dimensional analysis. Changing a language is not an easy task. There are many problems we need to consider. For example, we need to change the compiler, and we need to consider whether it is easily accepted by users. There is another way to introduce units of measure into programming language. That is using a high-level language's abstraction facilities to let the programming language support dimensional analysis. Here when we say high-level languages, we mean imperative languages. In some languages such as LISP, to define a units package is relatively easier than in an imperative language because LISP is a functional language. Basically, you can define anything you want in LISP. In an imperative language, if the language has no such feature then it is very hard for you to define the feature using the language itself.

An early discussion of using a high-level language's abstraction facilities to let a programming language supports dimensional analysis can be found in Hilfinger's book "Abstraction Mechanisms and Language Design" [Hil83]. Although later, Gehani [Geh85] and Hilfinger [Hil88] give another Ada package to support dimensional analysis respectively their emphasis is different. To reduce errors resulting from the inconsistent usage of objects we can use many methods, such as derived types and units of measure.

2.2.6 The Later Work of Gehani

One of the benefits of incorporating units of measure into a programming language is helping in detecting errors. In some languages such as Ada, we can use derived type to help detect errors. In [Geh85], Gehani examined and analyzed the idea of using derived types and units of measure to specify additional information in Ada. This can be described as follows:

Let x, y be two FLOAT variables. Normally, we can do any computation on x and y. If x, y has some practical meaning, for example x denotes a price and y denotes a weight, then assigning x to y or adding x, y together is not correct. This error can not be detected automatically. In Ada, we can use derived type to solve this kind of problem. A derived type introduces a new type which is identical to an existing type except that it is logically distinct. Using derived type we can declare PRICE and WEIGHT as follows:

```
type PRICE is new FLOAT
type WEIGHT is new FLOAT
```

Here PRICE and WEIGHT are two new types. They both have the same range of values, say FLOAT, but logically they are different type. Mistaken use of variable of type PRICE for those of type WEIGHT can be detected automatically. Thus if we declare

x : PRICE

y : WEIGHT

then assigning x to y or adding x, y together would violate the typing rule and this violation would be detected during compilation. The result obtained by Gehani is that the units of measure approach is better than the derived types approach to specify additional information. Gehani uses the method of units of measure to define an Ada package to implement units in Ada.

2.2.7 The Work of Hilfinger

Although Hilfinger [Hil88] describes methods for including units with numeric data using Ada packages, he emphasizes the modifications of the Ada compilers that would be required to make the use of these packages efficiently. We will give a more detailed discussion about Hilfinger's work in the section 2.4.

2.2.8 Other Work

There are several other packages such as [CG88], [Cun92], and [Umr94]. In [CG88], Cmelik and Gehani use class and operator overloading to give a package for handling units of measure in C++. But in these packages the dimensional checks have to be done at run time. Umrigar also gives a package for handling units of measurement in C++. Umrigar's method makes use of C++ templates to track the dimensions of quantities at compile-time.

Although Umrigar's package can check some dimensional correctness before run-time there are some drawbacks in his method.

(1) The method only handle integer demension.

(2) Because all quantities having a particular dimension use the same internal units the programmer does not have sufficient control over the precision of dimensional quantities, which may lead to an accumulation of floating-point error.

(3) Errors are not reported in terms of dimensional violations but rather in terms of type errors.

(4) Since the dimensions are directly incorporated into the template type placehoder, the dynamic dimensions are not allowed.

From the work of Cmelik, Gehani, and Umrigar we can see that the higher the facilities you use the less control over the process you get.

We have mentioned that when we incorporate units of measure into a programming language (more precisely into an imperative language) an important aspect that we should consider is compile time consistency checking. Why do we need this? The motivation is that we want to incorporate units of measure into programming languages, and we also want to "compile away" any computational overhead associated with handling dimensional information at run time. In [Cun92], Cunis gives a different view. Cunis discusses a package for handling units of measure in Lisp. Most LISP systems are interactive interpreters. The users interact with the LISP interpreter by typing in function invocations. The LISP system then interpreters them and prints out the result. Therefore Cunis argues in favour of actually incorporating units of measure information with numeric data objects in a dynamic and interactive programming environment.

2.3 Safer_C

In this thesis our goal is to add dimensional analysis to Safer_C. Thus in this section we give a brief introduction to Safer_C. Safer_C is a new language developed by Salomon [Sal95] in the Department of Computer Science at University of Manitoba. Safer_C is a modern descendant of the C language. The popular C language is over 20 years old. The C language has many syntactic deficiencies that lead to common programming errors. Some of these errors can persist in a program until run time. The primary object in the design of Safer_C is to produce a language that is more error-resistant than C without sacrificing any expressiveness or computational power. Safer_C is semantically identical to C, but has most of the syntactic deficiencies eliminated by using modern conventions. Safer_C is a unified name of Safer_C/1 and Safer_C/2. Safer_C/1 and Safer_C/2 correspond to C and C++. Safer_C/2 will be equivalent to C++ in expressive power, but with less of the awkward syntax baggage that C++ inherited from C.

A simple Safer_C program is given here as a sample:

```
Safer_C version 3.1
stdio_h()
<<doit>> :: func(x, y :: int) int
block
        sum :: int
        sum := x + y
        return sum
end
<<main>> :: func () void
block
        printf("The sum is: %d", doit(2, 3))
end
```

2.4 Partial Evaluation

To understand what partial evaluation is and what research has been done on it is very important as we will use partial evaluation to improve the performance of our units package. Partial evaluation is a program optimization technique. It provides a unifying paradigm for a broad range of work in program optimization, interpretation, and compilation. Partial evaluation can improve the efficiency of programs by exploiting known information about the input of a program. Partial evaluation has been the subject of a rapidly increasing amount of activity over the past decade due to recent advances both in theory and practice ([BEJ88], [CD93], and [JGS93]). A more detailed discussion of partial evaluation will be given in Chapter 3. Here we only outline the basic research which has been done on partial evaluation.

Partial evaluation has been successfully applied to declarative languages, such as Scheme and Prolog. In recent years there is a growth in the study of partial evaluation in imperative programs [Cha90], [Mey91], [And93a], [WL95], and [KKZG95]. Anderson ([And93a], [And93b]) describes a partial evaluator for a substantial subset of Kleinrubatscher, Kriegshaber, Zochling, and Gluck [KKZG95] describes a partial evaluator for a substantial subset of Fortran 77. Baier, Gluck, and Zochling [BGZ94] investigate the application of partial evaluation to numerically-oriented computation and engineering applications.

Salomon [Sal96] uses partial evaluation to support many important language features and implements a partial evaluator for Safer_C. The motivation of using partial evaluation in Safer_C is to replace the functionality of a preprocessor. The greatest obstacle to the modernization of C that was encountered is its preprocessor phase. Since preprocessors are used to change source text, the machine translation of C programs into a new version or a different form can be blocked by even tame preprocessor statements. Sometimes the actual C program that is being manipulated cannot be known until specific values are assumed for some of the preprocessor variables, and then only the program generated by those specific values can be manipulated, not the general form of the program. Since the existence of a preprocessor phase impedes even simple source-to-source code manipulation, it was decided that the preprocessor should be replaced early in the evolution of Safer_C so that future translation with language evolution would be easy. The Safer_C translator can be described in figure 2.1. We will discuss some detail partial evaluation technique used in Safer_C in Chapter 5.

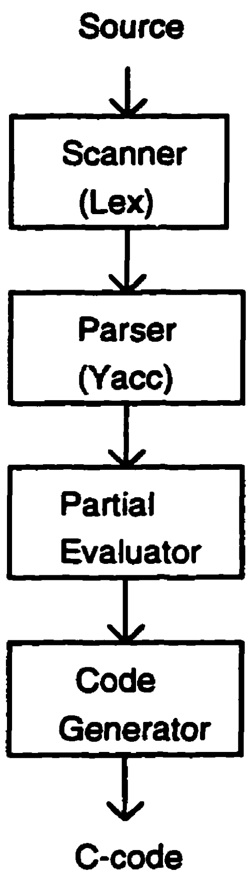

Figure 2.1 Safer_C translator

Despite the successful application of partial evaluation to many fields, few attempts have been made to study the partial evaluation of dimensional analysis. In [Hil88], Hilfinger describes an Ada package to support dimensional analysis and argues for good compilers to efficiently execute the package. In his paper, Hilfinger proposes using a variant record to define a UNITS. A dimensional quantity is defined by

```
type QUANT($D_0$, $D_1$, $D_2$, $D_3$: INTEGER := 0) is
        record V : FLOAT;
        end record;
```

Hilfinger proposes that the compiler split the type QUANT into two parts QUANT_descrim_type and QUANT_value_type:

```
type QUANT_discrim_type is
record $D_0$, $D_1$, $D_2$, $D_3$: INTEGER := 0 ; end record;
type QUANT_value_type is
record v: FLOAT; end record.
```

Then he extends constant folding [ASU86] to composite objects to handle the units package efficiently. Using constant folding

```
        X := X + DELTA_X
```

will become

```
        RETURN_value := (V ⇒ $X_0$.V + $Y_0$.V );
        X := RETURN_value;
```

Note: Hilfinger suggests that a compiler could implement the type QUANT as two types. From this implementation we got the separation of the static part and the dynamic part of a record. Therefore Hilfinger further suggests that the compiler apply an optimization technique such as value propagation, peephole optimization [ASU86], or an expansion of the inline functions to achieve efficient performance of the units package.

In Hilfinger's Ada package, we found the following disadvantages:

1) Because Ada does not allow overloading of assignments, some uses of the Ada units package are not natural.

2) Hilfinger discusses constant folding only for some simple cases.

   a) Hilfinger discusses only how to get the compiler implement QUANT and use the implementation to improve the efficiency of units checking. Using partial evaluation we can deal with more general cases.

   b) When we expand the units package to handle more complex cases such as rational powers, the simple constant folding is not enough to handle units computation. Also from the point of view of partial evaluation it is unreasonable to expect a compiler to execute static statements since the compiler lacks binding-time information.

3) In the Ada units package, since discriminants are used when we declare dimensional variables we have to specify discriminants. Thus dynamic units are not allowed in the Ada units package.

Chapter 3

Dimensional Analysis and Partial Evaluation

3.1 Units System

In this section, I present a short review of the units system which one might learn in a beginning physics course [Mor69]. To ensure accuracy and reproducibility of a physical quantity, it is first necessary to define units in which the physical quantity is measured. In general, a unit is a basic physical quantity by which other physical quantities can be expressed. There are numerous physical quantities but not all of them are independent. Many physical quantities can be described in terms of a small set of fundamental quantities. For example, velocity can be described in terms of length and time. In mechanics, only three fundamental quantities are used. They are length (L), mass (M), and time (T), or length (L), force (F), and time (T).

We call the units for the three fundamental quantities *basic units*. A unit which is described in terms of fundamental units is called a *derived unit*. The complete set of basic and derived units that are used to represent all quantities is called a *system of units*. According to the fundamental quantities that are used, we can have the following six units systems.

a) Mass-based systems: Mass-based system are also called *absolute systems*. In these units systems, the fundamental quantities are length, mass, and time. They are

1) International System (SI): In the International System, the three fundamental quantities length, mass, and time are the meter (*m*), the kilogram mass (*kg*), and the second (*sec*), respectively. Actually, SI has four other basic units for other branches of physics. They are those for temperature (the degree Kelvin), electric current (the ampere), luminous intensity (the candela), and the amount of a substance (the mole). The meter, the kilogram, the second, together with the kelvin, the ampere, the candela, and the mole form the seven basic SI units.

2) Centimeter-gram-second System (*cgs*): In this system, the three fundamental quantities length, mass, and time are the centimeter (*cm*), gram mass (*g*), and the second (*sec*), respectively.

3) British Mass System (*fps*): In British Mass System, the three fundamental quantities length, mass, and time are the foot (*ft*), the pound mass (*pm*), and the second (*sec*), respectively.

b) Force-based systems: Force-based systems are also called *gravitational systems*. In these units systems, the fundamental quantities are length, force, and time. In a similar fashion to mass-based systems, we have *cgs*, *mks*, and *fps* system in force-based system since the length and time units in each system are same. The only difference is the change from mass to force. But the mass and force can be expressed in terms of each other. The force-based systems are

1) Meter-kilogram-second Force System (*mks*): In mks system, the three fundamental quantities length, force, and time are the meter (*m*), the kilogram force (*kgf*), and the second (*sec*), respectively.

2) Centimeter-gram-second System (*cgs*): In this System, the three fundamental quantities length, force, and time are the centimeter (*cm*), gram force (*gf*), and the second (*sec*), respectively.

3) American/British Engineering Force System: In this System, the three fundamental quantities length, force, and time are the foot (*ft*), the pound force (*pf*), and the second (*sec*), respectively.

Let us summarize these systems in a table

	Mass-based system			Force-based system		
	cgs	mks	fps	cgs	mks	eng.
Length	cm	m	ft	cm	m	ft
Mass	gm	kgm	pm	(*)	(*)	slug
Force	dyne	newton	poundal	gf	kgf	pf
Time	sec	sec	sec	sec	sec	sec

Table 3.1 Units Systems

Because in ***absolute systems*** there are no basic units for force, the ***dyne***, ***newton***, and ***poundal*** are derived units. Similarly, in ***gravitational systems***, there is no basic unit for ***mass***. The ***slug*** is a derived unit. No name has been assigned to the *cgs* force-based *mass*. It is rigorously defined as the mass of a body that accelerates one centimeter per second per second (1 $cm/\sec^2$) when acted upon by a force of one gram force (1 *gf*). Similarly, No name has been assigned to the ***mks*** force-based ***mass***, but it is rigorously defined as the mass of a body that accelerates one meter per second per second (1 $m/\sec^2$) when acted upon by a force of one kilogram force (1 *gf*). For the conversion of these units we have

$1\ m = 3.28\, ft,\qquad 1\ pm = 0.4536\ kgm,\qquad 1 dyne = gm{*}cm/\sec^2,$

$1\ newton = 100000\ dynes,\qquad 1\ poundal = 1.38{*}10000\ dynes,$

$1\ kgm = 1000\ gm,\qquad 1\ slug = 32.2\ pm,\qquad 1\ pf = 1\ slug{*}1\ ft/\sec^2,$

$1\ gf = 1\ gm{*}981\ cm/\sec^2,\qquad 1\ kgf = 1\ kgm{*}9.81\ m/\sec^2.$

If a quantity can be expressed as a multiple of another then we call these two quantities commensurable. From above we see that feet and meters are commensurable.

Among the six systems, the three most commonly used units systems are the *cgs* absolute system, the SI absolute system, and the American/British system. Besides the six systems we may have some other systems depending on the specific field of application such as goldsmiths. In addition there are some measurement systems used only in particular countries. In this paper, however, we are concerned principally with the SI system.

3.2 Dimensional Analysis

In physical science, physical quantities are dimensional quantities. They are represented by a measure, and its units. The measure is a magnitude of the quantity and the units is a physical meaning of the quantity. There are a large variety of the definition of dimensions in physics. There are many books that discuss dimensional analysis [Foc53], [Pan64], [Isa75], and [Tay74]. In this thesis, we use dimensions to indicate that we are concerned here only with the nature of the quantity and not with its measure in any particular units. For example, whether a distance is measured in units of feet or meters, it is a distance. We say its dimension is length(L). Similarly, we say that the dimensions of area are $(length)^2$ or the dimensions of velocity are length/time. We will often use brackets [ ] to denote the dimensions. Thus, in this notation, the dimensions of velocity are written as $[v] = LT^{-1}$

Suppose $a, b, c \in Q$, where Q is the set of rational numbers. If we choose length(L), mass (M), and time (T) as the basic quantities, then for a given quantity x, its dimensional representation is given by $[x] = L^a M^b T^c$ or $[x] = (a, b, c)$. Using this notation a dimensional quantity $\bar{x}$ is written as $\bar{x} = (x, a, b, c)$. If we choose a force-based system then the dimensional representation is $[x] = L^a F^b T^c$.

There is an important property of dimensions of physical quantities. The dimensions of physical quantities can be manipulated algebraically and we can interpret the results to provide a great deal of information about the physical processes involved in the situations considered. ***Dimensional analysis*** is the study of the nature of the relationship between the various quantities which are involved in a physical problem.

Dimensional analysis plays an important role in physics and engineering. The main benefits of dimensional analysis that are pointed out by the majority of authors are: (i) to derive theoretical relationships; (ii) to check the correctness of the equations involved in the description of the phenomenon under investigation; (iii) to reduce the number of relevant dimensional variables to a smaller number of dimensionless variables; and (iv) to serve as a basis for model laws. Sophisticated study of dimensional analysis can be found in [Mar95] and [Kay93]. Perhaps the simplest application of dimensional analysis is to provide a means of checking the dimensional correctness in a mathematical solution of a physical problem.

In general, let α and β be dimensional expressions. We define a *term* as

i) α, or

ii) the product of α and β, or

iii) the quotient of α and β.

In a given units system, let $f(x_1, x_2, \ldots, x_n) = 0$ be a derived numerical relationship between the measures of the various quantities. Using dimensional analysis to check the equation is based on the principle of dimensionally homogeneity, which implies that:

> All the *terms* in the equation must be expressible as the same combination of dimensions. The exponents and arguments of transcendental functions must have a dimension of 1.

For example, if a car starts from rest and moves with constant acceleration a, then the distance traveled by a car in time t can be expressed as $d = \frac{1}{2}at^2$. Let us check the validity of this expression from a dimensional analysis approach. The quantity d on the left side has the dimensions of length. In order for the equation to be dimensionally correct, the quantity on the right side must also have the dimensions of length. On the right side the acceleration a has dimensions L/T^2, and t^2 has dimensions T^2. Thus, the dimension form of the equation $d = \frac{1}{2}at^2$ is

$$L = \frac{L}{T^2} \cdot T^2 = L.$$

Here the units of time have been canceled out. Therefore $d = \frac{1}{2}at^2$ has dimensional homogeneity.

There is a special quantity, we call a dimensionless quantity. A dimensionless quantity has dimensions $L^0M^0T^0 = (0,0,0)$. For example, π is a dimensionless quantity.

3.3 Partial Evaluation

Since we will use partial evaluation as a tool to improve the performance of our units package we will give an introduction of partial evaluation in this section.

3.3.1 The Principle of Partial Evaluation

Partial evaluation is a source to source program transformation technique for specializing programs with respect to part of their input [CD93]. The translator which completes these tasks is called the *partial evaluator*, *mix*, or *program specializer*. The following diagram illustrates the process of partial evaluation.

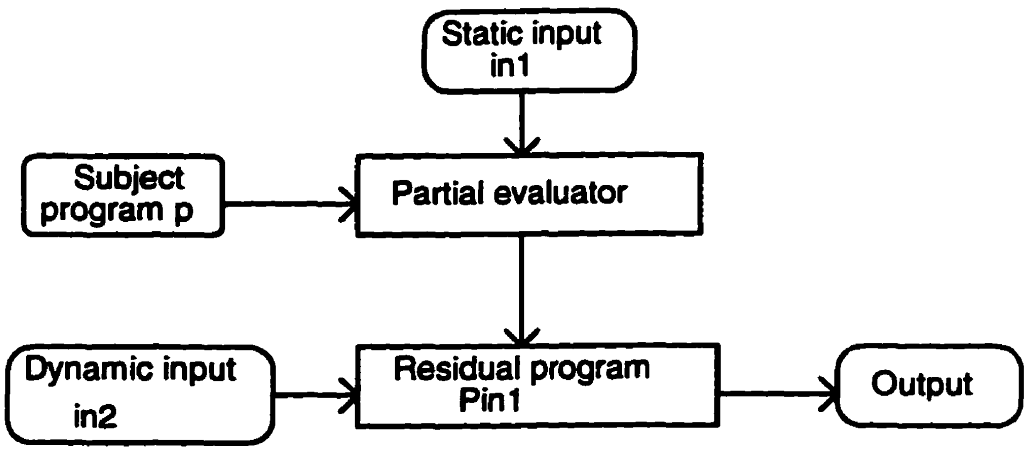

Figure 3.1 Partial Evaluation Diagram

In general, a program has many inputs called in_1 , in_2 , ..., in_n . If the program is correct and all the inputs are known then by executing the program with the input, we can get the output (result). Usually the program is written to be general purpose and some input may not be known. We can classify these inputs as static inputs and dynamic inputs. Static inputs are those inputs whose values we know or whose values can be determined at program specialization time. Whereas the values of dynamic inputs are unknown or their values cannot be determined at program specialization time. Such a static/dynamic classification is called a *division*. The process which computes the division of all program variables given a division of the input variables is called *binding-time analysis* (BTA). The idea of partial evaluation is to execute those calculation of a program that depend only on its static input, generating code (the residual program) for those calculations that depend on the as yet unavailable dynamic inputs.

Formally, using the notation in [JGS93] we have following definition:

Let L-programs denote the set of syntactically correct programs in language L. The meaning of program $p \in$ L-programs is denoted by

$$\|p\|_L : \text{input} \rightarrow \text{output}.$$

The result of running the program p on some input data d is denoted by

$$\|p\|_L\, d = \text{result or } \|p\|_L\ [d] = \text{result}.$$

We will use L to denote the implementation language, S to denote the source language, and T to denote the target language.

Definition 3.1: Let p be an L-program taking as input a two-element list, and let $d_1 \in D$, where D is an input set (since partial evaluation accepts both programs and data as input, we assume that both p and d are drawn from a common set D). Then an L-program r is a *residual program* for p with respect to d_1 iff $\|p\|_L [d_1, d_2] = \|r\|_L\, d_2$, for all $d_2 \in D$

Definition 3.2: An L-program *mix* is a *partial evaluator* iff for every p, $d_1 \in D$, the program

$$\| p \|_S [d_1, d_2] = \| \| mix \|_L [p, d_1] \|_T d_2, \text{ for } d_2 \in D.$$

Example: Consider following function power() which computes base to the n-th power (written in Safer_C).

```
<<power>> :: func(base, n :: int) int
block
    pow :: int
    for (pow := 1; n > 0; n--)
        pow *:= base
    endfor
    return pow
end
```

If n is equal to 3 and suppose that *base* is dynamic input. Then the partial evaluation will output following residual program.

```
<<power_3>> :: func(base :: int) int
block
    pow :: int := 1
    pow *:= base
    pow *:= base
    pow *:= base
    return pow
end
```

This residual program is longer than original program, but actually it is more efficient than the original one. We can use traditional optimization methods in a good optimizing C, or C++ compiler to further optimize the residual program *.

There are two kinds of partial evaluations: online partial evaluation and offline partial evaluation.

In offline partial evaluation, the specialization is divided into two stages. The first stage is the preprocessing stage which annotates all the variables used in the object program. We call this stage the *binding time analysis* (BTA). The second stage is the specialization stage which generates the residual program according to the biding time analysis. In this stage the specialization depends on only the binding time not the values of variables.

In online partial evaluation, there is no preprocessing stage. During the specialization, the values of variables are considered. A detail discussion of online, offline partial evaluation can be found in [JGS93], or [WCRS91].

* Note that the output of Safer_C is a C or C++ program.

3.3.2 Partial Evaluation In Safer_C

Partial evaluation in Safer_C [Sol96] is different from normal partial evaluation. For convenience, we denote the partial evaluation used in Safer_C as PES. PES has features of both online and offline partial evaluation. The difference between PES and offline partial evaluation is the first stage in the partial evaluation. In PES, the programmer annotates each variable with an evaluation time. Safer_C as C is intended for use by professional programmers. Thus emphasis is placed on programmers being able to predict what computations will be done at compile time, and being able to control when computations will be performed. The difference between PES and online partial evaluation is that PES has annotation, but online PE does not. Therefore in PES the programmer has more control than in online PE. We will discuss more about how partial evaluation is done in Safer_C in Chapter 5.

Chapter 4

Design

In this chapter, we design the features of a dimensional analysis package. Our goal is to let the computer do the dimensional analysis for the user. From the user's point of view, the main features of the system are ease of declaration of the units of measure, evaluation of the ordinary arithmetic operations between quantities that have units of measure, automatic checking of violation of dimensional consistency, and automatic handling of the conversion of commensurable units. In 4.1, we first give an example to show how programming languages use units. Then in 4.2, 4.3, 4.4, and 4.5, we discuss several important features of the units package. In 4.6, we discuss units and units computation. In 4.7, we discuss consistency and the dim_quant computation. In 4.8 we give the formal definition of the declaration of dimensional quantities. Finally, in 4.9 we describe the basic structure of the units package.

4.1 An Example

In this section we give an example to show how units are used in programming languages. In dimensional analysis, the first thing is to get the units information from the user. Variables, constants, and parameters are not just numerical quantities; they also have units, which the user has to declare. In our system, we require that the user first give the base units. Then the user can define the derived units, the variables, and describe the algorithm for the problem. The system then performs the algorithm, does the units conversion, and units computation according to the base units which are given by the users. For declaring the fundamental units which will be used in a computation the users can just write the following

km :: units := "kilometer"

hour :: units := "hours"

With these fundamental units, the user can then declare a variable which has units consisting of fundamental units. For example, we can declare units *foot* as:

foot ::= units("*foot*",30.48**cm*).

Let us consider a simple example to see how the user can declare variables that have units of measure.

Example 4.1. A car travels at a constant speed of 60 *km/hour*. Assume that it takes 3.5 *hours* for the car to travel from location *A* to *B*. Find the distance from *A* to *B*. For solving the problem, the user may write the following Safer_C main program.

```
<<main>>::func ( ) void
block
  Speed:: doubleu(0.0, km/hour)
  Time :: doubleu(3.5, hour)
  Distance :: doubleu(0.0, km)
  printf("Please input speed in km/hour:")
  scanf (Speed.v)
  Distance := Speed*Time
  printf("The distance is:")
  printu(Distance)
end
```

In this program, doubleu is a new data type which can be used to declare a variable of type double-precision float that also has units of measure. The function printu is an overloaded function which is used to output the quantity which has units of measure.

If we run the program, it will print

```
Please input speed in km/hour:
```

If the user types 60 *then the output will be

The distance is 210 (*km*).

4.2 Dynamic Dimensions

In the normal case, we will specify the particular dimensions of a dimensional quantity. But in some cases we may want the dimensions of a dimensional quantity are dynamically changeable. These cases are

(1) Dynamic input

In some applications, each time a program starts to run, it may require the dynamic input of some dimensional quantities. In this case, if we can dynamically input dimensions then we do not need to change and recompile the program. Since we are mainly using partial evaluation to improve efficiency, the dynamic input of units is not treated in this thesis.

(2) Temporary variables

Temporary variables are often used in programs. Whenever the user wants a dimensional variable to hold values having different units they can use a temporary

* In this particular example, since the input function scanf is used, the user could not enter any other units.

variable. Sometimes, in the computation of an expression we also need temporary variables to hold the intermediate or final result.

(3) Function parameters

Functions are the basic building blocks of a programming language. They are one of the most important features of programming languages. If a function is to use arguments, it must declare formal parameters that accept the values of the arguments. Normally when you declare formal parameters you have to indicate their types. Similarly in dimensional analysis, when you declare formal parameters, you should indicate the units of the parameters. However, if we are not allowed to change the units dynamically then the function will be too restrictive. It would, therefore, be a valuable feature to allow formal parameters to hold values having different units.

In our package, cases (2) and (3) are allowed.

4.3 Conversion

In this section we will discuss some basic ideas about conversion. The details of how to convert dimensional quantities will be discussed in section 4.7.2.

The reasons we need conversions are:

(1) To check consistency.

(2) To compute dimensional quantities.

(3) To convert traditional or engineering units to SI units.

Normally, engineers perform units computation according their own tradition, then convert the result to the SI system. For example the units of the thermal conductivity coefficient are Kcal$(m \cdot h \cdot ^\circ\mathrm{C})^{-1}$ in the engineering units system, and the units of the thermal conductivity coefficient are W/(m·°C) in the SI system.

It is often the case that in a problem the units we are concerned with are not in the given units system. We need to convert them into the same units system. Even with the units which are in the same units system, we may still need to do some conversion. For example, to compute 2 *cm* + 3 *m* we need to convert 2 *cm* to 0.2 *m* then compute 0.2 *m* + 3 *m*, to compute 1 *dyne**2 *gm* we need convert 1 *dyne* to *gm***cm*/sec^2 then compute (1 *gm***cm*/sec^2)*(2 *gm*).

(4) To facilitate international trade and technical exchange.

Let x and y be dimensional quantities. And x can be converted to y. To convert x to y we need to know the relation between their units. For example, x has units u_1, y has units u_2 and the relation between u_1 and u_2 is

$$u_1 = au_2,$$

where a is a real number. The a is called the *conversion factor* of u_1.

Using the relation between u_1 and u_2 or their conversion factors, we can convert the measurement between x and y. If m_1 is the measurement of x and m_2 is the measurement of y, then the conversion between the measurement of x and y can be given by a function f

$$m_1 = f(m_2).$$

The function f which is commonly used in physics is

$$m_1 = am_2 + b, \qquad (4.1)$$

where a, b are real numbers.

When we deal with conversion, we have two possible strategies:

1) All dimensional quantities are represented internally using only the chosen base units. In this method, derived units can be used, but internally they are represented in base units. For example, in the *cgs* system, 3 *inches* will be represented as 7.62 *cm*.

2) The second method is to allow that derived units be used in the intermediate computation. In this method if we want to compute 2 *inches* + 3 *inches* then the answer is 5 *inches*. In the computation there is no conversion. Of course if we work in the *cgs* system we will ultimately convert 5 *inches* to 12.7 *cm*.

Each of these methods has some advantages and disadvantages which are discussed in the following sections (4.4).

4.4 Precision Control

We mentioned that there are two strategies which could be used in conversion. a) All dimensional quantities are represented internally using only chosen base units. b) Derived units are used in the intermediate computations.

The advantage of the first method is that it is easy to implement. The disadvantage of this method is that the programmer does not have sufficient control over the precision of dimensional quantities, which may lead to an accumulation of floating point error. The discussion of error propagation can be found in some numerical analysis books [Atk89]. Let x_T and y_T be no error numbers, x_A and y_A be the approximation of x_T and y_T i.e.

$$x_T = x_A + \varepsilon, \quad y_T = y_A + \eta,$$

where ε and η are errors. The *relative error* in x_A is denoted as:

$$\text{Rel}\,(x_A) = (x_T - x_A)\,/\; x_T\,.$$

In general, we have following results [Atk89]:

$$\text{Rel}\,(x_A\, y_A) \approx \text{Rel}\,(x_A) + \text{Rel}\,(y_A),$$

$$\text{Rel}\,(x_A / y_A) \approx \text{Rel}\,(x_A) - \text{Rel}\,(y_A),$$

$$\text{Rel}\,(x_A \pm y_A) \approx \text{Rel}\,(x_A) \pm \text{Rel}\,(y_A).$$

A process of computation can be described as follows:

1. Get some initial value,
2. Execute some algorithm to manipulate data,
3. Get results.

In a simple computations, if some of the initial values have a small error the accumulated error would not be significant. However, in some computations, especially if large systems or complex arithmetic are involved, the errors of the initial values are crucial. In this case, The error of initial values may generate totally wrong results.

Generally speaking, in units computation we cannot avoid conversion (which may cause some rounding error) but in some cases we can reduce the error as much as possible. For example:

1) Suppose that all the units we use are in a units system A and we want the results to be expressed in another units system B. In this case we can do the computation in units system A first and then convert the results into the units system B. This way is better than first of converting all the quantities in units system A into units system B and then performing the computations in units system B. Although ultimately the result of a computation will be converted to the required base units system the intermediate computation will not produce any rounding error caused by the initial conversion error.

2) If there are mixed units in a computation then we cannot avoid conversion. In this case, allowing the use of derived units is better than not allowing the use of derived units. There are two advantages: one is that there would be no rounding error since there would be no conversion; another is that the user could control the order of a computation. For example, in the expression $E_1 + \cdots + E_n$ the user could group the E_i $(1<i<n)$ into sets with identical units, and simply add these expressions together. There is no conversion necessary. The user can also write $E_i + E_j$ as $E_j + E_i$. The order of computation here may matter since it may mean different conversions are applied. For the first expression the system may convert E_j to E_i. For the second expression the system may convert E_i to E_j. If we use only the first method then it does not matter in which order the expression is written. We will get the same results.

4.5 Rational Powers

We know that if x is a dimensional quantity then the dimensions of x are

$$[x] = L^a M^b T^c = (a, b, c).$$

In the existing papers, a, b, and c can be only integers. This is not enough for practical work. For example in magnetism, the force between two poles can be written as

$$f = mm'/(\mu r^2),$$

where m and m' are the magnetic strength of the two poles, r is the distance between the two poles, $1/\mu$ is a constant of proportionality and depends on the medium and the units chosen. Now let us consider the dimensions of m. If $m = m'$ and disregard the dimensions of μ then

$$f = \frac{m^2}{r^2}$$

whence $$m = \sqrt{r^2 f}.$$

Force has the dimensions given by $[f] = LMT^{-2}$.

Also

$$[r^2] = L^2.$$

Therefore

$$[m] = \mathrm{L}^{\frac{3}{2}}\mathrm{M}^{\frac{1}{2}}\mathrm{T}^{-1} = (\frac{3}{2},\ \frac{1}{2},\ -1).$$

To express such rational numbers we may declare a, b, and c as float or double. Therefore we will have dimensions such as $L^{\frac{3}{2}}M^{\frac{1}{2}}T^{-1} = (1.5,\ 0.5,\ -1)$. A rational number represented in the form of a floating-point value has two drawbacks. First, it is hard to read. Second, it may have some error (for example the value 1/3 can not be stored exactly as a floating-point value). To make the dimension more readable and eliminate error, we may use a rational algorithm to manipulate the rational notation of a rational number. This cannot be done at compile time in any existing dimensional packages.

Let $x = a/b$ and $y = c/d$ then

$$x + y = (a * d + c * b) / (b*d),\ x - y = (a * d - c * b) / (b * d),$$

$$x * y = (a * c) / (b * d),\ x / y = (a * d) / (b * c),\ \text{and}\ x^r = a^r / b^r,$$

where r is a rational number.

After some computations we may use the division algorithm to simplify a rational number.

Definition 4.1 An integer x divides an integer y if there exists an integer q such that

$$q \cdot x = y.$$

We use $x \mid y$ to denote x divides y. When $x \mid y$, we say that y is a multiple of x and x is a divisor of y.

Definition 4.2 Let a and b be integers, $gcd(a, b) = \max\{c\colon c|a,\ c|b\}$. The function $gcd(a, b)$ is called *greatest-common-divisor* (*gcd*).

Let $x = a/b$ be a rational number. If $gcd(a, b) = 1$ then x is in a non-reducible form. If $gcd(a, b) = c \neq 1$ then x can be simplified as $x = (a/c)/(b/c)$.

Finding the greatest-common-divisor is based on the important algorithm (Euclid's algorithm). Using Euclid's algorithm we can easily find the greatest-common-divisor. A complet discussion of Euclid's algorithm can be found in [Knu81].

Example 4.2 Let a = 525, b = 231. According to the division algorithm we have

$$525 = 231*2 + 63$$

$$231 = 63*3 + 42$$

$$63 = 42*1 + 21$$

$$42 = 21*2$$

Therefore *gcd* (525, 231) = 21.

4.6 Units and Units Computation

In dimensional analysis, units are the fundamental components. To perform dimensional analysis we need to have the units information. The main information we should have is:

1) The name of the units such as *inches*, *kilograms*, and *dollars*. The units' name can be expressed as a string.
2) The conversion factor which is used to convert between dimensional quantities. To store the conversion factor we need only one floating point or double precision floating point variable.
3) The dimension of a dimensional quantity. There are two kinds of units: basic units and derived units. In chapter 3, we have seen that a dimension of given dimensional quantity x can be denoted as $[x] = (a, b, c)$, where a, b, and c are rational numbers. We have decided to use rational notation to express a, b, and c. Therefore we should use a two-dimensional array to store the dimensions.
4) The functions which are used to manipulate the units. Normally there are two kinds of functions: the functions which are used to get information about a unit, and the functions which are used to manipulate the units. The functions will be introduced later.

All the information given above is about units. Therefore we may use the following structure to represent units.

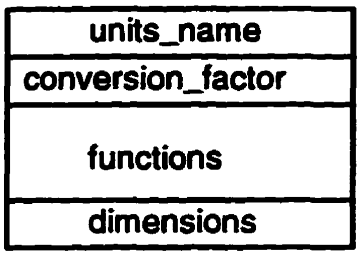

Figure 4.1 Units Diagram

Note the dimensions component in the diagram is a two-dimensional array used to hold the power of each dimension. The frame can be easily expressed by a class or a structure in an object-oriented programming language. Thus if u denotes a units class then we can use *u.name* to denote the field units_name, *u.cf* to denote the field conversion-factor, and *u.d* to denote the dimension of u. Let us consider some examples, the units *inch* and *m/sec* can be stored as

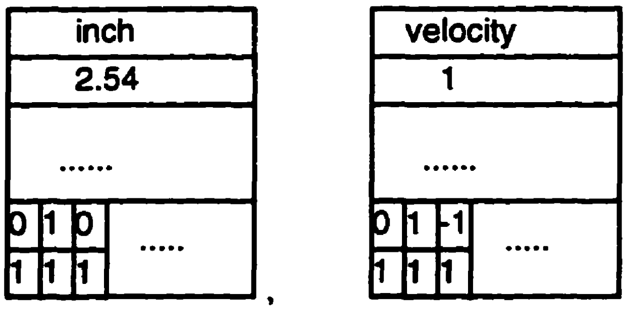

Figure 4.2 Units inch

Note here, in the frame *inch*, the dimension is stored in the form of *cm*. The units exponent is stored as a rational number. Since the exponent of *cm* is 1 it is stored 1/1.

Using this representation of units, we then can discuss computation between two units. Let u be a unit (base or derived unit), r be a rational number, u_1 and u_2 be base units or derived units. The basic computations on units are $u_1{}^r$, $u_1 * u_2$, and u_1/u_2 which are called ***composite units***. To compute ***composite units***, there are two components which we need to calculate.

The first component which is needed is the conversion factor. We have mentioned that we allow our system to automatically perform the conversion of commensurable units. For example, suppose $x = 3.5$ *cm*, $y = 2$ *in* we want to compute $x + y$. The users would not need to convert the units. They would only need to give the relation 1 *in* = 2.54 *cm*. Then the system will automatically do the conversion and perform $x + y$. To do the conversion an important thing to know is the conversion factor of the units. We will attach a conversion factor to each unit. A fundamental unit of the chosen standard units system has a conversion factor of 1. The derived units have a conversion factor which is given by a declared relation. For example, if we declare 1 *in* = 2.54 *cm* then we set the conversion factor for *in* to be 2.54. The conversion factors of ***composite units*** are formed as follows.

For $u = u_1{}^r$, the conversion factor is $(u_1.cf)^r$.

For $u = u_1 * u_2$, the conversion factor is $u_1.cf * u_2.cf$.

For $u = u_1/u_2$, the conversion factor is $u_1.cf / u_2.cf$.

Note here that for the result units of computation, there is no units name assigned. Because this is an intermediate result we do not need the name. The only thing which we care about is the conversion factor and the dimension.

The second component which needs to be computed is the dimension which is computed as follows:

1) For u_1^r, The dimension is $f(r * u_1.d)$,

2) For $u_1 * u_2$, The dimension is $f(u_1.d + u_2.d)$,

3) For u_1/u_2, The dimension is $f(u_1.d - u_2.d)$,

where f is a function (a Euclid's algorithm) which simplifies the resulting dimension.

For example, if the base units system is the *cgs* system, x denotes the unit *sec* and y denotes the unit *inch* then $x.cf = 1$, $[x] = (0, 0, 1/1)$ and $y.cf = 2.54$, $[y] = (1/1, 0, 0)$. Therefor $(y/x).cf = 1/2.54 = 0.3937$, $[y/x] = (1/1, 0, -1/1)$.

Note that in our method we require that if we declare a derived unit which has relevance to other units then the relevant units have to be previously declared. This requirement is reasonable since when we declared the derived units the declaration involved some computation. For example if we declare x :: units("*inch*", 2.54**cm*) then the system will want to compute 2.54**cm*. If the *cm* is not declared then the system will complain that *cm* is not declared.

4.7 Consistency and Dim_Quant Computation

Having discussed units, let us consider how to check the consistency of dimensional quantities, how to convert dimensional quantities which have different units, and how to perform arithmetic operations between dimensional quantities,

4.7.1 Consistency Check

In a statically typed programming language, each variable has a type. Using the type information we can detect the type errors. Similarly, when we consider computing the quantities which have units of measure, the system should detect units inconsistency errors. For example, we cannot add two quantities which have different units of measure. If t denotes the time in ***second***, s denotes the area in m^2, and v denotes the velocity in ***km/hour***, then the following statements are not correct:

```
if (t < s) printf("Time\n")
    else printf("Area")
```

```
endif
v := s / t
```

In the condition statement, we cannot compare t with s as they have different units. In the assignment statement, the dimensions of left side and right side are incommensurable.

4.7.1.1 The Consistency Check Rules

In our system, a consistency check is based on the principle of dimensional homogeneity. Let x and y be dimensional quantities, and r and p be rational numbers. Then the check is done as follows:

1) $x + y$, or $x - y$ if and only if x, y are dimensionally homogeneous.
2) $x := y$ if and only if x, y are dimensionally homogeneous.
3) For function invocations, if the formal parameters and actual parameters have dimensions then they must be dimensionally homogeneous.
4) x, y are comparable if and only if x and y are dimensionally homogeneous.
5) Exponents and arguments of the transcendental functions (*sin*, *cos*, *log* etc.) must be dimensionless. It is possible to allow degrees used in these functions.

4.7.1.2 Iteration

The assignment operator should be discussed in more detailed. Let $x := y$ be an assignment statement in a program. According to the rule, the system will give an error indication if the x and y are not compatible. In some cases, we may want the assignment to be forcibly performed. Why would we want this? The reason is that sometimes we want to use assignment during iterative multiplication or division. Such iteration is a an essential feature of many important algorithms. We do not want lose this important feature because of the introduction of units. For example, to compute the *base* to the n'th power we may use following iteration (Notice that the *base* is a dimensional quantity):

```
<<power>> :: func(base, n :: int) int
block
        pow :: int
        for (pow := 1; n > 0; n--)
                pow := pow * base
        endfor
        return pow
end
```

In the assignment *pow* := *pow* * *base*, obviously the left side and the right side have different dimensions. Thus according to our dimension checking rule, the system will reject the assignment. Therefore we need a special way to force the assignment to be performed. There are many ways to do this.

1) Use a special "=" operator.

2) Use cast. In C/C++ programming language, if x is an integer and y is a float then we can use () to cast x as follows

(float)$x := y$.

Thus a natural way is to use a cast operator. However, in iteration such as $x = x * y$, the cast cannot be directly used since the lvalue of x will be used in the right operand. If a cast used directly then after casting the units in left operand x are changed and the units of x in the right side will also be changed. Also in cast we have to indicate that what type we want cast. In dimensional analysis, to indicate the units which we want to cast to we need to do the computation. To overcome the difficulties we use the overloaded operator () and introduce another *temp* variable to do the cast as follows:

```
temp()
temp := x * y
x()
x := temp
```

where () is an overloaded operator which changes a variable accept any dimensions.

4) For some special iteration such as finding the exponentiation we may use following two methods:

a) Introduce an exponentiation operator **.

The exponentiation operator is provided by many languages such as Ada, Fortran, and Algol but no exponentiation operator is provided by C or C++ programming language because these languages were intended principally for systems rather than scientific programming. We feel that a language intended for scientific computation should provided an exponentiation.

b) Directly use the operator *:=.

In our package, for exponentiation we use *:= operator.

4.7.2 Conversion of Dimensional Quantities

To perform a computation involving dimensional quantities, a conversion may be involved. In the following sections, we first discuss how to convert dimensional quantities which do not involve temperature. For temperature conversion, we need special attention; see section 4.7.2.2.

4.7.2.1 Conversion Measurements

In this section we discuss how to convert the measurements of dimensional quantities. The main conversions that need to be considered are:

A) Conversions for expressions.

Using conversion factors we can convert any measurement in some units to a measurement in the desired units if their units are commensurable. Let x be a dimensional quantity, x_q be the measurement of x, x_u be the units of x, y be the equivalent dimensional quantity in the unit y_u, and z be their equivalent dimensional quantity in the standard system. We have :

$$x_q * x_u.cf = z_q = y_q * y_u.cf.$$

Thus if we want to convert y to x then

$$x_q = (y_q * y_u.cf)/x_u.cf.$$

Therefore we can compute $x \circ y$ as follows (where the symbol $\circ$ denotes an arithmetic operator, assignment operator, or relational operator).

1) Check if the dimensions of x and y are the same. If they are same then the units of x and y are commensurable otherwise they are not commensurable.
2) If the units of x and y are commensurable then convert the measurement of y to the measurement of x and perform $x \circ y$

   i.e.

$$(x \circ y)_q = x_q \circ (y_q * \frac{y_u.cf}{x_u.cf}),$$

where $x \circ y$ has units of x.

If unit x_u belongs to base system then $x_u.cf = 1$. Thus

$$(x \circ y)_q = x_q \circ (y_q * y_u.cf)$$

B) Conversion for function calls

In general, we do not require that the user indicate the units of the formal parameters. However, if the formal parameters have their units specified then the actual parameters should be converted to the units of the formal parameters. For function calls, the consistency checking and conversion should be done at partial evaluation time.

4.7.2.2 Conversion of Temperature

In many cases, the conversion functions for measurements involve only one constant factor such as the case $b = 0$ in formula (4.1). In this case it does not matter what conversions are done. We can always compare the results of computations in different systems. In some cases, however, the conversion is somewhat more complicated, as, for example the conversion between degrees Celsius and Fahrenheit. In this case, we cannot do the conversion arbitrarily during computation since we cannot compare the computed result in different systems.

Let us consider the example of computation t_c / t_f, where t_c = 3°C denotes a temperature in degrees Celsius, t_f = 41°F denotes the temperature in degrees Fahrenheit. The conversion function between degrees Celsius and Fahrenheit is

$$t_f = f(t_c) = 1.8t_c + 32.$$

If the computation is done in Celsius then we have

$$3°C / 41°F = 3°C / 5°C = 0.6.$$

If the computation is done in Fahrenheit then we have

$$3°C / 41°F = 37.4°F / 41°F = 0.912.$$

The reason why the results are different is that the results are in different scale systems. There are two differences between the two scale systems. First, the origins are different. In Celsius, the freezing point of water is 0, but in Fahrenheit the freezing point is 32. Second, the size of degrees is different. We use C° to indicate the size of degrees in Celsius. Similarly, we use F° to indicate the size of degrees in Fahrenheit. The size of one degree between Celsius and Fahrenheit satisfies the relation

$$5\ C° = 9\ F°.$$

To make sure the correct computation is performed between quantities which belong to different temperature-scale systems we could use the following strategy. Before doing any computations, we convert all the temperature quantities into the same scale system (For example Kelvin). Then we perform the computations with no conversion on temperatures. The conversion between different temperatures can be done by means of a function call.

For example, in a computation if there is a quantity which is Celsius degrees and the temperature unit used in the base system is Kelvin degrees, we then can define the function c_to_k which converts Celsius degrees to Kelvin degrees as follows:

```
<<c_to_k >> :: func ($t_c$ :: float)
block
        return ($t_c$ + 273)
end
```

Using this function then we can say x = doubleu(c_to_k(23.4), K_D), where K_D denote degree in Kelvin.

We have seen that there is a distinction between indicating the size of the degree or temperature interval, C°, and the temperature °C. Since there are 100C° corresponding to 180F° the size of one F° must be 5/9 times the size of one C°, or

$$5C^\circ = 9F^\circ.$$

Notice also that the size of one degree C° is same as the size of one degree K° i.e.

$$C^\circ = K^\circ.$$

Since the conversions between C°, F°, and K°only involves one conversion factor, we could treat the size of degree (C°, F°, or K°) as normal units such as *cm*, *in*, and *m*.

Let us consider an example in which the size of the degree is used.

Example 4.3 An aluminum plate at 68.5°F has an 8.00-inch-diameter hole in it. What is the diameter of the hole when the plate is heated to a temperature of 150°F? [Mor69].

Solution: To solve the problem, we may apply following equation

$$D^2 = D_0^2(1 + 2\,\alpha\,\Delta t),$$

where the α is the linear expansion coefficient of aluminum that is 23.8 * 10^{-6} / C°, D_0 and D are respectively the initial and expanded diameters of the hole, and Δt is the temperature difference. Here we have

$$\alpha = 23.8 * 10^{-6} / C° = 13.2 * 10^{-6} / F°, \qquad (C° = 1.8F°)$$

$$\Delta t = (150 - 68.5)F°,$$

and

$$D^2 = 8.0\text{ in}*8.0\text{ in}*[1 + 2*13.2*10^{-6}*\Delta t / F°]$$

$$D = 8.01\text{ in.}$$

4.7.2.3 Alternative Techniques

It has been suggested by professor Meek* that a dimensional analysis system should forbid the programmer from coding some operations on units with an arbitrary origin, such as temperatures and dates. For instance, the system should forbid the addition of 21°C and 15°C or of 1970AD and 1990AD. Such a restriction would, however, prohibit simple computations such as finding average temperatures, $(T_1 + T_2)/2$, or interpolating dates, $(3 * D_1 + 5 * D_2)/8$. The question of how computations on units with arbitrary origins should be restricted is not at all simple, and is beyond the scope of this thesis. Meek himself has shown that recognizing the equivalence of such formulae as x-y and $\frac{x^2 - y^2}{x + y}$ is beyond the capabilities of pure dimensional analysis.

4.7.3 Computation Involving Dimensional Quantities

Now let us consider computations involving dimensional quantities. From the discussion given above, we have seen that the system should have the ability to perform ordinary arithmetic operations between quantities that have units of measure. Velocity multiplied by time should give us distance. Kilograms plus kilograms should give kilograms.

* Dereck Meek, in personal communications.

Let x and y be dimensional quantities, and r be a rational number. The operation $x \circ y$ can be computed as follows:

(1) If $\circ$ denotes "+" or "–":

```
If $x$ and $y$ are commensurable
    then {
        a) $temp_q = x_q \circ (y_q * \frac{y_u.cf}{x_u.cf})$
        b) return $temp_q$, $x_u$
    }
  else (report error in +, –).
```

(2) If $\circ$ denotes "*" or "/":

```
If $x$ and $y$ are commensurable
  then {
        a) $temp_q = x_q \circ (y_q * \frac{y_u.cf}{x_u.cf})$
        b) $temp_u = x_u \circ x_u$
    }
    else {
        a) $temp_q = x_q \circ y_q$
        b) $temp_u = x_u \circ y_u$
    }
return $temp_q$, $temp_u$
```

For example, if $x = 3$ *in*, $y = 2$ *m*, and the base system is *cgs* then x has the units *in* with conversion factor 2.54. y has units *m* with conversion factor 100. Thus

$$x{*}y = 236.22\ in^2,$$

where $temp_q = 236.22$, $temp_u.cf = 6.4516$, and $temp_u.d = cm^2$.

Note that each unit has a conversion factor, therefore it does not matter what units x and y have; the computation of units is automatically done in the standard units system.

3) For the operation x^r:

```
    $temp_q = x_q^r$
    $temp_u = x_u^r$
    return $temp_q$, $temp_u$
```

4) For the operation $r{*}x$, where r has no units:

```
    $temp_q = r{*}x_q$
    $temp_u = x_u$
    return $temp_q$, $temp_u$
```

Since r has no units, the units of the result are the same as the units of x.

5) For the assignment $x := y$:

```
    if ($x_u.d == 0$)          !! i.e. $x$ is dimensionless
        then $x = y$
```

```
else if $x$ and $y$ are commensurable
        then $\{x_q = x_q \circ (y_q * \frac{y_u.cf}{x_u.cf}),\ x_u = y_u\}$
        else error("Assignment :=")
```

(in our package we should implement all the operators including –:=, *:=, +:=, ++, --, and relational operators which are defined in Safer_C)

4.8 Notation Design

In the dimensional analysis package, the units are declared by member functions. The syntax for declaring units is given by the following context-free grammar rules. Note that These grammar rules show how to use units of measure, they are not actually part of the Safer_C grammar. In particular **units** and **doubleu** are type names not reserved words.

units →	*base_units* :: **units** := "*units_name*"
	| *derived_units* :: **units** ("*units_name*", *c_units*)
r_number →	*rational_number*
c_units →	*units_t* * *units_t* | *units_t* / *units_t* | *units_t* ^ *r_number*
	| *float_number* * *units_t*
units_t →	*base_units* | *derived_units*

To declare a dimensional quantity we use the following syntax:

d_quant → *identifier* :: **doubleu** (*double_number*, *c_units*).

Example 4.4 Sample use of units:

cm :: *units* := "*cm*"

sec :: *units* := "*sec*"

in :: *units*("*inch*", *2.54*cm*)

x :: *doubleu* (*23.4*, *cm/sec*)

4.9 Package Design

The units package is designed for Safer_C to support dimensional analysis. In our method, we could use structures or classes to represent the units quantities, use parameterized types to initialize the objects, use operator overloading to perform the computation between quantities with units, and use partial evaluation to perform the dimensional consistency check and units computation at compile time. Basically, in the dimensional analysis package there are two kinds of classes: the units class and the doubleu class. The units class is used to express basic units and derived units. The doubleu class is used to express all the double precision quantities which have units. Similarly we can define floatu to express float quantities which have units. In our system, there is a difference between pure quantities (that means no units) and dimensionless quantities since pure quantities and dimensionless quantities are of different data types.

The structure of the units class is as follows. We define the maximum dimension (MaxDim) as 7 because the international system only has seven fundamental quantities. We can freely define MaxDim according to the specific use. In the units class, the u_name is the printable name of the unit and u_exp is a two-dimensional integer array used to store the exponents of the units. The field u_factor is the conversion factor.

For example, if we use a class to denote *inch* then in the class *inch* the u_name is "inch", the u_factor is 2.54, and the u_exp is *cm*.

```
MaxDim :: const int := 7
:: class units{
    u_name :: -> char
    u_exp :: [2][MaxDim] int
    u_factor :: float

public:
!! Constructors ->char
<<units>> :: func(->char) void
<<units>> :: func(->char, units) void
!! Destructor
<<~units>> :: func() void
.....
}

:: class doubleu{
   v :: double
   u :: units
public:
!! Constructors
   <<doubleu>> :: func () void
   <<doubleu>> :: func(double; units) void
  ....
}
```

By default the constructor for doubleu will initialize the dimension of object doubleu as zero. Using operator overloading we can define operators to perform computations on doubleu as follows

```
<<op ∘>> :: func(ref a :: doubleu; ref b :: doubleu) doubleu
block
         temp = a ∘ b        (we need to fill the body which is given in 4.7.3)
         return temp
end
```

When we perform operations such as multiplication, addition and subtraction etc. we need to check dimensional consistency. The compatible function is defined as follows:

```
<<compatible>> :: func(ref a :: units, ref b :: units) int
block
  i :: int
  for(i = 0; i < MaxDim; i++)
      if(a.u_exp[0][i] =/= b.u_exp[0][i] and
         a.u_exp[1][i] =/= b.u_exp[1][i])
           return 0
```

```
        endif
    endfor
    return 1
end
```

Chapter 5

Partial Evaluation for Dimensional Analysis in Safer_C

We have seen that partial evaluation is a program specialization technique which computes the static part of the program and generates a residual program for the dynamic part. In this chapter, we will investigate how partial evaluation can be applied to dimensional analysis. In particular, we are interested in using, the techniques for the Safer_C language. The aim is to use partial evaluation to improve the efficiency of a dimensional analysis package. The discussion focuses on partial evaluation of static structures and pointers. An important technique that can be used for the partial evaluation of a dimensional analysis package is presented.

5.1 Evaluation Annotation in Safer_C

The evaluation time plays an important role in Safer_C. For example, *i* can be a translation time integer, *sum* can be a run time float, etc. In Safer_C, several features that are important for partial evaluation are present:

1) Symbols can be annotated at their declaration with a designation of their evaluation time.
2) The evaluation time of symbols is propagated through a program to determine the evaluation time of expressions, and control structures.
3) Control structures can be additionally annotated with an evaluation time to assist the compiler, or to clarify the programmer's intentions.
4) Declaration are treated as compile-time "executable" statements.

More details about Safer_C can be found in [Sal96].

In this chapter, we will consider the terms *static* or *known* equivalent for translation time data, and the terms *dynamic* or *unknown* equivalent for run-time data. Pointers are an exceptional case which is discussed in section 5.5.

5.2 Partial Evaluation of Structures

In dimensional analysis, a dimensional quantity has two parts: measure and units. The computation between dimensional quantities has two parts: computation on their measure and computation on their units. In many cases, the measure of a dimensional quantity is dynamic but its units are static. Therefore the question is whether we can remove the static computation part and leave only the dynamic computation part at run-time. If we can do this then we will get a fast run-time program. A dimensional quantity can be expressed by a structure. If we treat the whole structure as single entity then we will lose the static information. To do the partial evaluation, we need to discover the static information. In the following, we show how this can be done by annotating different evaluation times for different fields and using a splitting technique to discover the static information.

A structure is a heterogeneous aggregate of data elements. For example we can declare a structure as

```
s :: struct {
        i, j :: int
        k, l :: double
        }
```

In a structure, if some fields are static and some fields are dynamic then the structure is called a *partially static structure* or *mixed structure*. In Safer_C we need to annotate a structure with an evaluation. If all the fields of a structure are static then we can annotate the structure as a tran-time structure. If all the fields of a structure are dynamic then we can annotate the structure as a run-time structure. However how should a partially static structure be annotated? We cannot annotate it as tran-time since there are some fields that are dynamic. If we annotate it as being run-time then the information in the static fields will be lost. The proper choice is to annotate mixed structure with a mixed-evaluation time.

Conceptually, we say that if some fields of a structure are translation-time fields and others are run-time fields then the structure has an evaluation time of both run-time and translation time denoted as (T ×···× T), where T is the evaluation time of structure's fields. For example, a structure {*x* :: tran int; *y* :: float} could have evaluation time $(t \times r)$ meaning that *x* is a translation time field but *y* is a run time field.

Normally, a tran-time variable exists only at compile-time. After compile-time the tran-time variable will be removed. To do partial evaluation for partially static structures we could use a splitting technique. This is because if we define

```
s :: struct {
        i, j :: tran int
        k, l :: run int
       }.
```

then the definition of *s* would be equivalent to the following two definitions:

```
        st :: tran struct {i, j :: int}
```

and

```
        sr :: run struct {k, l :: int}
```

The object *st* would exist only at translation-time, and the object *sr* would exist only at run-time. Their treatment would be same as for other purely tran-time or run-time objects. Therefore we can split a structure *s* into two structures: *st* which contains the dynamic fields and *sr* which contains the static fields. All the accesses to the dynamic fields of *s* will be changed to accesses to the corresponding fields of *st*. All the accesses to the static fields of *s* will be changed to accesses to the corresponding fields of *sr*.

By splitting we separate the static part and dynamic part of a partial static structure. Thus we can use normal partial evaluation to perform relevant operations on the static fields.

5.3 Compact Representation of Units

By splitting we can also save some space. For example, the powers of units that are encountered in practice are not very large. One can assume that they are between -128 and 127 [Hil88]. To store the units we need seven 8-bit bytes since in SI there are seven base units. For handling rational powers, we add seven more 8-bit bytes. The units then can be compacted into fourteen 8-bit bytes. Although the space used by several units are not large when we use an array of dimensional quantities, the space used for units will become noticeable. In many cases, all the units used by the dimensional quantities in an array are the same. In these cases, there is no reason to allocate space for the units of each dimensional quantity. For example,

```
struct dim_quant {
                  v :: double
                  u :: struct units
                 }
y :: [0..100] struct dim_quant
```

By using the spitting technique we will get

```
yv :: [0..100] double
yu :: [0..100] struct units
```

If all the units are the same, we do not need to save all the same units in an array *yu*. We can compact *yu* into one variable *cyu* :: struct units.

5.4 Function Specialization

In our dimensional analysis package, all operations are overloaded operator functions, therefore the main problem to be considered in using partial evaluation for dimensional analysis is the specialization of the functions for structures with some tran-time members.

5.4.1 Partial Evaluation of Function in Safer_C

Because the basic structure of Safer_C is the same as that of C and C++, a Safer_C program can be seen as a set of modules. Safer_C's main structural component is the function. All Safer_C programs consist of one or more functions. Therefore partial evaluation of a Safer_C program means the specialization of Safer_C functions. Safer_C provides three kinds of partial evaluation of functions [Sal96].

1. Replacement by Result.

   This kind of partial evaluation is done if:

   a) The values of all of the actual arguments and external variables accessed by the function are known at translation-time.

   b) Either the source code for the function is available at translation-time, or the object code for the function is available and a dynamic loader is provided to the translator.

   c) The function has no side effects.

2. In-Line Expansion.

   This kind of partial evaluation is done if:

   a) The source code for the function is available at translation-time.

   b) The function is declared to have translation-time evaluation.

3. Function Specialization.

   This kind of partial evaluation is done if:

   a) The source code for the function is available at translation-time.

   b) The function is declared to have run-time evaluation.

   c) Some of the formal parameters of the function are declared to have translation-time evaluation.

5.4.2 Partial Evaluation of Functions with Structured Parameters*

When a function has structures as its parameters we can also split the parameters. Consider a structure type with mixed-time fields such as:

```
MixedTime :: type := struct {i, j :: tran int
                             k, l :: run int
                            }
```

Such a type declaration would be the same as declaring two types that are always used together:

```
MixedTimet :: type := tran struct {i, j :: int}

MixedTimer :: type := run struct {k, l :: int}
```

With such a mixed-time type declaration, the declaration of object *s* given in section 5.2 would be the same as the following declaration:

```
        s :: MixedTime
```

Consider also a function ***mtfunv*** that accepts a parameter of type MixedTime and returns type void:

```
<<mtfunv>> :: func (p :: MixedTime) void
```

* In other partial evaluation systems, a partially static structure is not split but specialized for its static fields [And93a].

Such a function declaration would be the same as the function declaration

<<*mtfunv*>> :: func (*pt* :: MixedTimet; *pr* :: MixedTimer) void

Since the formal parameter *pt* is a translation-time value, each invocation of function *mtfunv* would be changed into an invocation of a newly created function *mtfunv_pt* that has been specialized for the value of *pt* at each invocation. This is the same treatment that is currently given to functions with tran-time formal parameters. Thus the invocation of a function with a parameter of a mixed-time struct type is the same as the invocation of a function with two corresponding struct parameters, each with the tran and run parts respectively of the original parameter.

When a function returns a result of a mixed-time struct type we can use following technique to deal with. In this technique, every function returning a partially static result is split into two functions, one returning the static part, and one returning the dynamic part. The static part depends only on static arguments and thus can be fully computed at specialization time. For example:

<<*fun_mt*>> :: tran func (*i* :: MixedTime) MixedTime

The function call:

S := *func_mt* (*j*)

would be the same as the two calls:

St := *fun_mt_t* (*jt*) !! Compute tran-time part of *S*

Sr := *fun_mt_r* (*jr*) !! Compute run-time part of *S*

Since the tran-time part of S could not depend on any run-time parts of j, the specialized function *fun_mt_t* could be created to compute those tran-time parts using only the tran-time parts of j. The function *fun_mt_r* would be a version of *fun_mt*, that has been specialized for the particular values for the tran-time parts of j and accepts the run-time parts as arguments.

5.4.3. Partial Evaluation for Overloaded Operators

For normal arithmetic operations and function calls on structures, there is no problem with splitting a structure. However when we consider function calls, especially an overloaded operator function, and pass structures to the function, the situation is somewhat more complicated. One of our goals is to use partial evaluation for dimensional analysis. In dimensional analysis, the functions mainly used are overloaded operator functions. Thus to apply partial evaluation to dimensional analysis there are three new features which we should deal with. One, the dimensional quantities are partially static objects. Two, the function calls require that whole structures be passed to the function and require the return of whole structures. Third, the operators have precedence. Let's consider an example *:

Example 5.1 To compute $z = x + y$. we can define operator + as follows:

```
MixedTime:: type := struct A {i, j :: int
                              k, l :: tran float
                             } x, y, z
```

* In this and later examples, if the evaluation time is omitted it defaults to "run".

```
<<op +>> :: tran func( a :: MixedTime, b :: MixedTime) MixedTime
block
        temp :: MixedTime
        ....              !! Perform a+b
        return temp
end
```

From the example we see that the function calls and function returns both use the whole structure. By using the evaluation-time splitting technique discussed above, the dynamic part and static part are separated. Thus normal partial evaluation techniques can be applied. Notice that after partial evaluation the overloaded operator function has been specialized. We cannot use the operator notation (such as + and *) anymore.

For example, in example 5.1, if we split struct *x*, *y*, *z* as follows

xij{*i*, *j* :: int}, *yij*{*i*, *j* :: int}, and *zij*{*i*, *j* :: int}

xkl{*k*, *l* :: tran float}, *ykl*{*k*, *l* :: tran float}, and *zkl*{*k*, *l* :: tran float}

and $z = x + y$ be split as

zij := *xij* + *yij*

zkl := *xkl* + *ykl*.

then we cannot find the operator + for *xij*, *yij* and operator + for *zkl*, *ykl* since the operator function + has been specialized.

To satisfy the requirements of overloading operators and to do partial evaluation we use the following technique.

In an arithmetic expression, whenever an overloaded operator is executed we introduce a new local variable to store the result of the residual function. If an operator has higher precedence it will be executed earlier. Using this method the overloaded operator is replaced by a residual function. The precedence of the operator is solved by introducing a new local variable.

Using the techniques discussed above, we can describe partial evaluation processing as having two phases as follows:

• The pre-processing phase (splitting stage)

In this phase, we split Safer_C's structures according to their evaluation-time based on the techniques which we discussed above. The parameters are split. All structure variables are split. All the assignment statements are replaced by two statements. All the accesses of structure members are replaced according to the split type structures.

• Partial evaluation phase

During this stage we do normal partial evaluation i.e. calculate, remove the static parts and generate residual code for the dynamic parts. Note that after partial evaluation overloaded operators are replaced by residual functions. The precedence of operators can be resolved by the introduction of new local variables if necessary.

5.4.4 Examples

Example 5.2 Consider the following program:

```
<<main>> :: func() void
block
        Doubleu :: type := struct {
                                    v :: run double
                                    u :: tran units
                                  }
        s, x, y, z :: Doubleu
        s := x + y*z
        printu (s)
end
```

To perform partial evaluation on the program we first split the program as follows:

```
<<main>> :: func() void
block
        Doubleut :: type := tran struct {u:: units}
        Doubleur :: type := run struct {v :: double}
        st, xt, yt, zt, new1t, new2t :: Doubleut
        sr, xr, yr, zr, new1r, new2r :: Doubleur

        new1t := f1t(yt, zt)
        new1r := f1r(yr, zr)
        new2t := f2t(xt, new1t)
        new2r := f2r(xr, new1r)
        st := f3t(new2t)
        sr := f3r(new2r)
        printu2(sr, st)
end
```

where *f1r* is the residual function of the operator* with respect to *yt* and *zt*. *f2r* is the residual function of operator+ with respect to the static parts *xt* and *new1t*. *f3r* is the residual function of operator:= with respect to the static parts *st* and *new2t*. *f1t*, *f2t*, and *f3t* are obtained by removing all the dynamic parts from operator*, operator+, and operator:= respectively.

If operator+ is defined as

```
<<op +>> :: tran func( a :: Doubleu, b :: Doubleu) Doubleu
block
        temp :: Doubleu
        x :: tran double
        x := compatible(a.u, b.u)
        if(x)
                temp.v := a.v+x*b.v
                temp.u := a.u
                return temp
        else error ("Operator +")
        endif
end
```

where *error* is a tran-time function which generates a compile-time error message,

then *f2t* will be

```
<<f2t>> :: tran func( at :: Doubleut, bt :: Doubleut) Doubleut
block
        tempt :: Doubleut
        x :: tran double
        x := compatible(a.u, b.u)
        if(x)
                tempt.u := at.u
                return tempt
                else error ("Operator +")
                endif
        end
```

and *f2r* will be (if the units is compatible and *x* is 2.5):

```
    <<f2r>> :: func( ar :: Doubleur, br :: Doubleur) Doubleur
    block
        tempr :: Doubleur
        tempr.v := ar.v+2.5*br.v
        return tempr
    end
```

Note that:

1) The above discussion shows the logical process of partial evaluation. It is possible to obtain *f2t* and *f2r* at same time. For example, during partial evaluation *f2t* and *f2r* can be obtained at the same time. This is because *f2r* is a residual program. Therefore when we derive the residual program we have already computed the result of *f2t*.

2) A postprocessing phase can remove singleton structs such as struct {v :: double}.

3) For efficiency consideration we could use inline functions instead function calls.

Suppose $yt = zt = cm$, $xr = cm^2$. After partial evaluation, we have

```
        <<mainr>> :: func() void
        block
            sr, xr, yr, zr, new1r, new2r :: double
            new1r := yr * zr
            new2r := xr + new1r
            sr := new2r
            printu_cm^2 (sr)
        end
```

This program can be further optimized via traditional optimization. Thus eventually we will get *sr* := *xr* + *yr* * *zr*. Notice that in fact the residual function is a C++ program. For ease of understanding we still use Safer_C notation.

Example 5.3 The function which computes *base* to the *n*'th is defined as follows:

```
<<power>> :: tran func(base :: doubleu, n :: tran int) doubleu
   block
       i :: tran int
       pow :: doubleu(1.0, u :: tran units)
       for (i := 1; n > 0; n--)
              pow  *:= base
       endfor
       return pow
    end
```

If the *base* has units *cm* after function invocation and *power* is to be specialized with respect to the units of *base* and *n* = 3 then after specialization the residual program will be:

```
<<power_3_cm_r>> :: func(baser :: double) double
   block
       powr :: double := 1.0
       powr *:= baser
       powr  *:= baser
       powr  *:= baser
       return powr
   end
```

The function *power_3_cm_t* which computes static part will has value cm^3.

5.5 Pointers

Pointers are one of C's and C++'s strongest features. Pointers are closely related to arrays. We can use both pointer arithmetic and array indexing to access array elements. We can use the *ref* operator to get the address of its operand and use a function pointer to call a function. Perhaps the most important use of pointers is to dynamically allocate memory. Actually the only way to refer to heap-allocated objects is via pointer variables. On the one hand, pointers are very important in C and C++. They give you tremendous power and are necessary for many programs. On the other hand, nothing will get you into more trouble than a wild pointer! Pointers are very hard to control since pointers can points to any thing. We agree that without detailed information about pointers, the annotation of pointers must be overly conservative [And93b]. In this section, we discuss only some basic aspects of pointers in our partial evaluation for dimensional analysis.

5.5.1 Annotation of Pointers

In this section we will discuss what is the meaning of partially evaluating a pointer and what is the meaning of the evaluation time of a pointer. In the following paragraphs, we first discuss what is the meaning of translation-time pointers and run-time pointers. We will, then discuss pointer splitting technique for dimensional analysis.

In Safer_C, since, a variable annotated as a translation time variable will exist only at translation time, we will assume that all tran-time pointers point at purely tran-time objects, and all run-time pointers point at purely run-time objects.

The evaluation-time of a pointer can be denoted by *T. For example, if a pointer *p* points to the structure *A*, then *p* has evaluation-time *T where T is the evaluation-time of *A*.

If a pointer *p* is declared as a run-time pointer then during partial evaluation all the computation associated with pointer *p* will be suspended.

If a pointer is a translation-time pointer then during partial evaluation all the operations on the pointer can be done at compile time.

5.5.2 Pointers to Mixed Structures

A pointer *p* may point to a mixed structure. Since we use splitting technique to split a mixed structure, therefore if a pointer points to a mixed structure then we would use similar method to deal with pointers. For example if a pointer *p* is declared as

```
p :: -> struct{
            i, j :: tran int
            k, l :: float
            }
```

The above declaration of *p* would be equivalent to the following declarations:

pt :: -> struct{*i*, *j* :: tran int}

and

pr :: -> struct{*k*, *l* :: run float}

The pointer *pt* would exist only at translation time, and the pointer *sr* would exist only at run time. Their treatment would be the same as for other purely tran-time or run-time pointers.

5.5.3 Functions with Pointers

Sometimes we may want to use pointers as the parameters of a function. The meaning of specialization with respect to a pointer is given as follows [And93a],

Suppose a function foo(*p* :: tran -> int) has a translation-time formal parameter of pointer type, and is to be specialized due to a call foo(*e*) giving the residual function foo'(). The specialization must be with respect to both the address (of *e*) and the indirection, that is , the content of all the locations that *p* legally can point to when the actual parameter expression is *e*. For example, if *e* is *a* where *a* is a translation-time array int *a*[10], then *p* can refer to *a*[0], *a*[1], ..., *a*[9]. After partial evaluation all the operations on the pointer disappear and the objects pointed to by the pointer and all the indirection are absorbed.

For a function call which has pointers as its parameters and the pointers point at mixed time structures the considerations are similar with the case where mixed structures are passed to a function. Consider following example, if we have a function:

```
    <<fun_mt>> :: tran func (p :: -> MixedTime) MixedTime
```

Then function call:

```
    S := fun_mt(p)
```

would be the same as the two calls:

```
    St := fun_mt_t(pt)     !! Compute tran-time part of S
    Sr := fun_mt_r(pr)     !! Compute run-time part of S
```

Since the tran-time part of *S* could not depend on any run-time parts of *p*, the specialized function fun_mt_t could be created to compute those tran-time parts using only the tran-time parts of *p*. The function fun_mt_r would be a version of fun_mt, that was specialized for the particular values for the tran-time parts of *p* and accepted the run-time parts as an argument.

Using the splitting technique, pointers are split, all dereferencing of original pointers are replaced by split pointers. For example, let dim_quant be a structure {*v* :: double, *u* :: tran units}. According to our method the dim_quant will be split as dim_quantv {*v* :: double} and dim_quantu {*u* :: tran units}. If we declare *p* :: -> struct dim_quant then during partial evaluation *p* will be replaced by *pr* :: -> dim_quantv and *pt* :: -> dim_quantu. Since *v* is run-time variable and *u* is tran-time variable *pr* will be a run-time pointer and *pt* is a tran-time pointer. *p->v* will be replaced by *pr->v* and *p->u* will be replaced by dim_quantu. Notice that *pr* is a run-time pointer. Thus we do not evaluate *pr*, but replace *p->v* by *pr->v*. Since *pt* is a tran-time pointer we replace *p->u* by *pt->u* which can then be evaluated as dim_quantu. If there is a function call *f(p* :: -> dim_quant) we will get *f(pr*, tran *pt)*. Furthermore, since *pt* is translation variable after partial evaluation, we will get *fr(pr)*.

We already annotated a pointer as either a translation time or a run time pointer, and if a pointer points to a mixed structure we use the splitting technique to split the pointer according to the tran-time and run-time fields of the structure. Therefore in our method we do not need global analysis of pointers.

5.6 Summary

From the above discussion we see that partial evaluation can be used to improve the execution of a dimensional analysis package. During partial evaluation most of the static units consistency checking and computation can be removed, and only the dynamic computations are left for run-time. To ensure that partial evaluation can be done, we have to have the source code of our units package available.

In a dimensional analysis package, there are two kinds of structures. One is the structure *units* another is the structure *doubleu*. For the operators on the structure *units*, all the fields are static and satisfy the conditions of replacement by result. Thus we can use replacing by result to specialize the operators on *units*. For example we only need to annotate operator* as follows:

```
units :: type := tran struct {
                                    u_name :: [0..20] char
                                    u_factor :: double
                                    u_exp :: [1..2][0..7] int
                                  }

<<op *>> :: tran func(a :: tran units, b :: tran units) units
block
          temp :: tran units
          ....      !! units computation part.
          return temp
end
```

For the operators on *doubleu*, since the operands are partially static we will use the method discussed above to specialize the operator functions on *doubleu*.

Chapter 6

Implementation

Safer_C is an ongoing project in the Department of Computer Science at University of Manitoba. Our units package is one of the new features of Safer_C. In our package, we used some features which are still in their developmental stages and thus we cannot fully implement the dimensional analysis feature in Safer_C as of yet. Nevertheless I have performed some experiments in C++ to demonstrate that my design is correct. In this chapter, I discuss some implementation problems which are mainly based on the experiments in C++.

6.1 Representation of Units

In chapter 4 and 5, we discussed our package in its general form. We showed that the units can be expressed as either a structure or a class. Using different data structures will cause the implementation to be different. Since our experiments are done in C++ on a UNIX system we will use a class to express the units.

6.1.1 Parameterized Constructors

An important feature that we used to express our units is parameterized constructors. In C++ it is possible to pass arguments to constructor functions. Typically, these arguments are used to help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. For example, if we have *demo_2p* class which has two parameters as follows:

```
class demo_2p{
    int a, b;
public:
    demo_2p(int i, int j) { a = i; b = j;}
};
```

then the statement

```
demo_2p  myobject(2, 3)
```

will create an object called *myobject* and pass the arguments 2 and 3 to the *i* and *j* parameters of *demo_2p*( ). There is another way to define the object *myobject*. That is

```
demo_2p  myobject = demo_2p (2, 3).
```

When a constructor function has only one parameter, then there is a third way to pass an initial value to that constructor. For example,

```
class demo_1p{
      int a;
public:
      demo_1p(int j) { a = j;}
      int get_a() { return a;}
};

main(){
      demo_1p myob = 99;       //pass 99 to j
      cout << myob.get_a();
      return 0;
}
```

As this example shows, in cases where the constructor takes only one argument, you can simply use the normal initialization form. The C++ compiler will automatically assign the value on the right of = to the constructor's parameter. Using this feature we can easily declare our units as

```
units cm = "cm";
units gm = "gm".
```

We found that this form is more intuitive than the first methods. Thus we use this form in our units package.

6.1.2 The Representation of Dimensions

In this experiment, since we are concentrating on our methodology we do not compress the representation of dimensions (base units). In SI, there are 7 base units and we permit the use of fractions as units powers thus we implement dimensions as a two-dimensional array denoted by u_exp[2][7]. Notice that although we use 7, the standard base units system is not constrained to the SI system.

Because different base units represent different dimensions we need to represent the *base units* in different classes. Also, for reporting the units name in printouts we use an array name[7] to store the name of the declared base units, and use a static variable *dim* to record how many *base units* we have used. Thus if we define units *cm* = "cm", then we have *dim* = 1 and u_exp is

1	0	0	0	0	0	0
1	1	1	1	1	1	1

Figure 6.1 Base units *cm*

If we define another *base unit* such as units *gm* = "gm" then we have *dim* = 2 and u_exp is

0	1	0	0	0	0	0
1	1	1	1	1	1	1

.

Figure 6.2 Base units *gm*

If needed, we can cut off the space used for base units. For example we can define the base units as class bunits:

```
        class bunits{
                        char u_name[20];
                        int u_factor;
                        int u_dim;
                }
```

where u_dim = 0, 1, ..., 6. Since in base units the u_factor is always equal to 1, therefore the u_factor can be omitted from the class bunits.

6.1.3 The Units Member Functions

Theoretically, all the functions (member or friends) can be used by the user. However, only constructors and overloaded operators are used by the user. The rest are used by the package for internal communication and debugging. The main functions can be specified as one of three kinds. The first kind is used to define the units, second one is used to get information on the units, and the third one is used to do computation on the units. For the first kind, since we allow the user to define dimensionless variables and derived units, we have overloaded the constructor function. The operations between units are overloaded operators. Operator overloading is similar to function overloading. An operator function can be either a member or a nonmember of the class that it will operate on. In our package we use friend functions. Since a friend is not a member of the class, it does not have a *this* pointer. Therefore, the operands of an overloaded friend operator function are passed explicitly.

6.2 Operations between Units

Normally, we may think that the operations, addition, subtraction, multiplication, and division between numbers, are very simple. However, in dimensional analysis, we need not only to consider the measure but also the units. For example, conventionally zero multiplied by any number is zero. In dimensional analysis, zero can have units. The definition of zero in our package is defined as:

A dimensional quantity is called zero if its value is zero and it is dimensionless.

For identifying the difference between dimensioned zero and real zero, we sometimes call the real zero *pure zero.*

Some algorithms on dimensional quantities are given as follows.

●**Units multiplication**

Let x and y be dimensional quantities. Then the units of x and y can be expressed as

$$x.\mathrm{u} = d_0^{x_0} d_1^{x_1} d_2^{x_2} d_3^{x_3} d_4^{x_4} d_5^{x_5} d_6^{x_6},$$

$$y.\mathrm{u} = d_0^{y_0} d_1^{y_1} d_2^{y_2} d_3^{y_3} d_4^{y_4} d_5^{y_5} d_6^{y_6}.$$

Therefore $x.\mathrm{u}*y.\mathrm{u} = \prod_{i=0}^{7} d_i^{x_i + y_i}$.

Let $x_i = b/a$, $y_i = d/c$ then $x_i + y_i = b/a + d/c = (b*c + a*d)/(a*c)$, where a, b, c, and d are integers. Let $x_i + y_i = e/f$, where e, f are integers and are initialized to pure zero. Then $e = b*c + a*d$, $f = a*c$. In pseudo-code, the computation is performed as follows

```
Begin
   for $i$ := 0 to 6 do
      if $b = 0$ and $d \neq 0$ then
            $e/f := d/c$;    !! Means $e := d, f := c$
      else if $b \neq 0$ and $d = 0$ then
            $e/f := b/a$;
      else if $b \neq 0$ and $d \neq 0$ then
            $e/f := (b*c + a*d)/(a*c)$;
```

```
        return $e / f$;
    endfor;
End
```

• Units power

Normally, we would like to use ** or ^ to denote the exponentiation operator. However, we could not use them because ** is not an operator in Safer_C. Only existing operators can be overloaded to handle new types. Allowing the definition of new operators is a new feature which is still under development for Safer_C. We cannot use the operator ^ either since ^ has the wrong precedence for exponentiation. Operator precedence cannot be changed in Safer_C.

In [CG88], Gehani chose the subscript operator [] for exponentiation. Technically, the parameter does not have to be of type integer, but an operator[]( ) function is typically used to provide array subscription, and as such, an integer value is generally used. In our package we will compute rational powers. Therefore the operator [] cannot be used. We adopt the convention of using a function call to deal with exponentiation. Thus to compute x^r we use *power*(x, r), where r is a rational.

• Compatible Units

The function *compatible* is used to determine whether two units are dimensionally consistent. The argument to the function is two units. Since we have expressed all units in a certain units system, we can check whether two units are same, or consistent. Even though the units of dimensional quantities are consistent the operations on the dimensional quantities may not be performed directly. To do the operation on dimensional quantities *a* and *b* which have consistent units we may have to convert their measurement. In each conversion we need to compute *b.cf / a.cf*. If we do conversion then the consistency check must already be done. Therefore we do not need to put the computation *b.cf / a.cf* in each overloaded operator. We can put the computation in the consistency checking function. In this way we can shorten our units package.

Let *a*, *b* be two units, then the algorithm for the function *compatible*(*a*, *b*) is described as follows:

```
Begin
    if $a$ and $b$ are same
        then return $b.cf / a.cf$
    else return 0;
End
```

• Computation between dimensional quantities

To do the computations between dimensional quantities the main consideration is whether the dimensional quantities are compatible. If they are compatible then we may need to do a conversion. In the following, we take addition as an example to demonstrate the algorithm. Let *a* and *b* be dimensional quantities. The algorithm is as follows:

```
Begin
    $x$ = compatible($a.u$, $b.u$);
    if $x$ =/= 0 then
```

```
            {       1. conversion;
                    2. addition;
                    3. return value;
            }
        else error("$a$ and $b$ are not compatible")
    End
```

The list of the main functions used in our dimensional-analysis package are given in Appendix B.

6.3 Print Out Dim_Qunt

To print out a dimensional quantity we need print out two parts: one is the measure, another is the units. We will use the following form for printing a dimensional quantity:

<value>(u_name1^<r>, u_name2^<r>, ..., u_namen^<r>),

where <value> is the measure, <r> is the rational power of unit.

For example, if $x = 45$ *m/sec* then in our prototype it will be printed out as:

45 (m^1, *sec*^(−1)).

If a dimensional quantity x has a specified units name then we can simply print out its value $x.v$ and its units name. For example, if $x = 3$ *in*, $y = 5$ *in*, z is declared as dimensional quantity which has units *in*, and $z = x + y$ then the output of z will be

z.v(*z.u.u_name*) = 5 (*in*).

If a dimensional quantity x has dynamic units then the output units normally will be given in the user defined base units system, that is *x.v*x.u.cf* (base units). For example, if a user specifies that the base units system is *cgs*, $x = 3$ *in*, $y = 5$ *in*, z is a doubleu, and $z = x * y$ then the output is:

96.774 (cm^2).

Note that it is possible to give a more readable printout form such as 45 m/sec.

6.4 Examples

We have run some test programs to test our units package. The examples are taken from different sources. The examples, programs, and running results are given in appendix C.

Chapter 7

Conclusion

This chapter will summarize the main contributions of the research work presented in this thesis. Further, it will discuss possible future work in three aspects:

- Developing and improving a complete units package.
- Establishing a units conversion library.
- Doing partial evaluation for Object-Oriented Programming Languages such as C++.

7.1 Summary

Dimensional analysis plays an important role in the mechanical and physical sciences and some other areas. Some researchers have used the abstraction facilities of high-level language to let programming languages support dimensional analysis [Hil88], [Geh85], and [Umr94]. These previous attempts have some of the following drawbacks:

1. They entail some run-time overhead.
2. They require the substantial modification of an existing programming language.
3. The dimension can take only integer value.
4. Users do not have much precision control over the computation.
5. They do not allow dynamic dimensions.

All these problems are very important in practice. In this thesis, I have designed a units package that uses partial evaluation to eliminate the run-time overhead. This package can handle rational powers, some precision control, error reporting, dynamic dimension and iteration, and fully compile-time checks and computations for static units. I have described the splitting algorithm for Safer_C to efficiently perform dimensional analysis.

The experiments show that our package design is correct. Using the splitting algorithm, we can do the entire computation for static units at compile-time. This result demonstrates that existing partial-evaluation technology can be used to improve the efficiency of dimensional analysis.

Notice that the package does not require any change to the existing Safer_C language. The package is not constrained to a particular standard base units system. This package can be used for any units system.

7.2 Future Work

Some experience was gained from this experiment including the discovery that providing a units package to support dimensional analysis is not as easy as we first thought. Based on current work, further research will aim at three aspects: Developing a complete units package, establishing a units conversion library, and extending partial evaluation for Object-Oriented programming languages.

7.2.1 Developing a Complete Package

At the present stage, I have implemented only the main body of the units package (or called a prototype). To develop a more practical units package there is more work that needs to be done. This work includes:

1. Adding more functions such as *:=, +:=, ++, -- etc.
2. Improving the package such as reducing memory usage and handling more complex practical problems.
3. Trying out more complex examples to test the packages.

7.2.2 Establishing a Units Conversion Library

From the test examples we see that in a dimensional analysis program there is quite a big section used to define units relations. If these units relations can be put into a units package as a library module then it would be very convenient for users. To put units relations into the units package, the big problem is how to deal with multiple relations such as 1 *metre* = 100 *cm*, 1 *metre* = 3.28 *foot*. Although House [Hou83] gave a critique of Gehani's work [Geh77], some questions are very important and need to be considered. For example, 1) In practice, sometimes we may use 1 *metre* = 100 *cm*, or 1 *metre* = 3.28 *foot*. This should be handled in a units package as we expect. 2) If we give the relations, 1 *kilometre* = *3280 foot*, 1 *metre* = 3.28 *foot*, later when *foot* are encountered which formula should be used? If these problems can be solved then we can add all the commensurate units into our units package. Thus when the users use the package they do not need to define the commensurate units. The package will automatically do the conversion according to the relations in its library.

7.2.3 Extending Partial Evaluation for OO-Language

From this research, we developed another concept that is extending existing partial evaluation for Object-Oriented programming language such as C++. There are two main problems that we may encounter. The first is that partial evaluation for imperative languages is still in its research stages. The second problem comes from the advanced features of object-oriented languages. The significant features of C++ in this regard are classes, which have private parts, inheritance which has protected parts, operator overloading, and polymorphism. In this thesis, I discussed only how to deal with operator overloading. It seems that there are other promising areas in which to handle partial evaluation for object-oriented programming languages such as C++.

Appendix A. Safer_C Declaration Grammar

```
declaration =            name-list "::" property-specifier [":=" initializer] EOS
                         | name-list "::" "type" ":=" type-expression EOS
                         | "::" struct-or-union-specifier EOS
                         | "::" enum-specifier EOS

name-list =              identifier | name-list "," identifier

property-specifier =     type-expression
                         | storage-class-specifier type-expression

storage-class-specifier =
                         "auto" | "register" | "static" | "extern"

type-expression =        [type-qualifier] type-specifier
                         | "[0.." [constant-expression] "]" type-expression
                         | "func" "(" [var-len-parm-list] ")" type-expression
                         | [type-qualifier] "->" type-expression

type-specifier =         "void" | "char" | "int" | "float"
                         | struct-or-union-specifier
                         | enum-specifier
                         | type-identifier
                         | type-modifier type-specifier

type-modifier =          "short" | "long" | "signed" | "unsigned"

type-qualifier =         "const" | "volatile" | "volatile" "const"

struct-or-union-specifier =
                         struct-or-union [identifier] "{" struct-declaration-list "}"
                         | struct-or-union identifier
```

struct-or-union = "struct" | "union"

var-len-parm-list = parameter-proto-list | parameter-proto-list EOS "..."

parameter-proto-list = parameter-protos | parameter-proto-list EOS parameter-protos

parameter-protos = formal-parameters | "::" property-specifier

formal-parameters = name-list "::" property-specifier

function_definition = "<<" designator ">>" "::" property_specifier EOS block_or_body

designator = tt_identifier | "op" opsign "LIKE" oper | "op" opsign

opsign = tt_identifier | oper

oper = "+" | "-" | "*" | "/" | ":=" | "*:=" | ...

Appendix B. Main Functions of Units Package

This package is written in C++. The purpose of the package is to illustrate that the design idea of a dimensional analysis package for Safer_C is correct.

```
#include <iostream.h>
#include <string.h>
#include <math.h>

#define U_NDIMS 7
char *name[U_NDIMS];
char dname[20];
int tempdim[2][U_NDIMS];

___________ DEFINE UNITS CLASS  _____________

struct units{
 char u_name[20];
 double u_factor;
 int u_exp[2][U_NDIMS];
 static int dim;

public:
 units();
 units(char *string);
 units(char *a, units b);
 ~units() { }

 void name_of()   {strcpy(dname, u_name);}
 double factor_of() {return u_factor;}
 void u_exp_of();
 void printu_exp();

 friend int gcd(int a, int b);
```

```
  friend units power(units a, int num, int den);
  friend units power(units a, float b);

  friend units operator*(units&, units&),
               operator/(units&, units&),
               operator*(double a, units b),

  friend double compatible(units a, units b);
};

___________ UNITS CONSTRUCTOR 1 ____________

units::units()
{ int i;
  strcpy(u_name, "");
  u_factor=1;
  for (i=0; i<U_NDIMS; i++)
    {u_exp[0][i]=0; u_exp[1][i]=1;};
}

___________ UNITS CONSTRUCTOR 2 ____________

units::units(char *string)
{ int i;
  name[dim]=string;
  strcpy(u_name, string);
  u_factor=1;
  for(i=0; i<U_NDIMS; i++)
    if(i==dim) {u_exp[0][i]=1;
                u_exp[1][i]=1;
               }
    else { u_exp[0][i]=0;
           u_exp[1][i]=1;
         };
  dim++;
}

___________ UNITS CONSTRUCTOR 3 ___________

units::units(char *a, units b)
{ int i;
  strcpy(u_name, a);
  u_factor=b.u_factor;
  for(i=0; i<U_NDIMS; i++)
    {u_exp[0][i]=b.u_exp[0][i];
     u_exp[1][i]=b.u_exp[1][i];
    };
}

_______ COMPUTE UNITS POWER ______

units power(units a, int num, int den)
{ units temp;
  int i=0,t2=0;
  temp.u_factor=pow(a.factor_of(), (float)num/den);
  if(num == 0)
    for(i=0;  i<U_NDIMS; i++)
      {if(a.u_exp[0][i] !=0)
          { temp.u_exp[0][i]=1;
            temp.u_exp[1][i]=1;
          }
      }
  else
    for(i=0;i<U_NDIMS;i++)
       {if(a.u_exp[0][i] !=0)
           {temp.u_exp[0][i]=a.u_exp[0][i]*num;
            temp.u_exp[1][i]=a.u_exp[1][i]*den;
             t2=gcd(temp.u_exp[0][i], temp.u_exp[1][i]);
             temp.u_exp[0][i]=temp.u_exp[0][i]/t2;
            temp.u_exp[1][i]=temp.u_exp[1][i]/t2;
           }
       }
return temp;
}
```

________ COMPUTE UNITS POWER ________

```
units power(units a, float b)
{
  units temp;
  int i=0,j=0,k=0,l=0,t=0,t2=0;
  float x, z;
  temp.u_factor=pow(a.factor_of(), b);
  j=(int)b;
  x=b-j;
  k=1;
  for(i=0;i<8;i++)
     {k=k*10;
      l=(int)(x*k);
      z=(float)l/k;
      if(z== x)
         break;
     }
  t=gcd(l,k);
  if(l == 0 && j == 0)
     for(i=0;  i<U_NDIMS; i++)
        {if(a.u_exp[0][i] !=0)
            {temp.u_exp[0][i]=0;
             temp.u_exp[1][i]=1;
            }
        }
  else
     for(i=0;i<U_NDIMS;i++)
        {if(a.u_exp[0][i] !=0)
            {temp.u_exp[0][i]=a.u_exp[0][i]*(j*(k/t)+l/t);
             temp.u_exp[1][i]=a.u_exp[1][i]*(k/t);
             t2=gcd(temp.u_exp[0][i], temp.u_exp[1][i]);
             temp.u_exp[0][i]=temp.u_exp[0][i]/t2;
            temp.u_exp[1][i]=temp.u_exp[1][i]/t2;
           }
       }
return temp;
}
```

____________ COMPUTE GCD ____________

```
int gcd(int a, int b)
{
  int x;
  while (b!=0)
    {x=a%b; a=b; b=x;};
  if(a>0)
       return a;
  else return -a;
}
```

____________ UNITS * ____________

```
units operator*(units&a, units&b){
 units temp;
 int i;
 temp.u_factor=a.u_factor*b.u_factor;
 for(i=0; i<U_NDIMS; i++)
      {if(a.u_exp[0][i] ==0 && b.u_exp[0][i] !=0)
             {temp.u_exp[0][i]=b.u_exp[0][i];
              temp.u_exp[1][i]=b.u_exp[1][i];
             }
       else if (b.u_exp[0][i] ==0 && a.u_exp[0][i] !=0)
                    {temp.u_exp[0][i]=a.u_exp[0][i];
                     temp.u_exp[1][i]=a.u_exp[1][i];
                    }
             else if(a.u_exp[0][i] !=0 && b.u_exp[0][i] !=0)
                                {
    temp.u_exp[0][i]=a.u_exp[0][i]*b.u_exp[1][i]+a.u_exp[1][i]*b.u_exp[0][i];
    temp.u_exp[1][i]=a.u_exp[1][i]*b.u_exp[1][i];
    temp.u_exp[0][i]=temp.u_exp[0][i]/gcd(temp.u_exp[0][i], temp.u_exp[1][i]);
    temp.u_exp[1][i]=temp.u_exp[1][i]/gcd(temp.u_exp[0][i], temp.u_exp[1][i]);
                                }
          }
 return temp;
}
```

____________ D, U * ____________

```
units operator*(double a, units b){
  units temp;
  int i, flag=0;
  for(i=0; i<U_NDIMS; i++)
    if(b.u_exp[0][i] !=0)
         { flag=1; break;}
  if(flag)
    {
    temp.u_factor=b.u_factor*a;
```

```
      for(i=0; i<U_NDIMS; i++)
         {temp.u_exp[0][i]=b.u_exp[0][i];
          temp.u_exp[1][i]=b.u_exp[1][i];
         };
     }
  else temp=b;
  return temp;
}
```

___________ UNITS COMPANTIBLE __________

```
double compatible(units a, units b){
 int i,flag=1;
 for(i=0; i<U_NDIMS; i++)
   if(a.u_exp[0][i] != b.u_exp[0][i] || a.u_exp[1][i] != b.u_exp[1][i])
       {flag=0;break;}
 if(flag)
     return b.factor_of()/a.factor_of();
 else return 0;
}
```

_____________ DEFINE DOUBLEU CLASS __________

```
struct doubleu {
 double v;
 units u;

public:
 doubleu();
 doubleu(double a, units b);
 doubleu(double a);

 double value_of() {return v;}
 void u_name() {u.name_of();};
 double factor_of() { return u.factor_of();}
 void units_of();

 void change_units(units uu) {u=uu;}

 doubleu operator=(doubleu&);
 doubleu operator+=(doubleu&);
 doubleu operator-=(doubleu&);
 doubleu operator*=(doubleu&);

 doubleu operator()(units a);

 friend void printu(doubleu&);
 friend doubleu power(doubleu a, double b);
 friend doubleu power(doubleu a, int n);

 friend double todouble(doubleu&);
 friend double sin(doubleu&),
               cos(doubleu&),
               tan(doubleu&),
 friend doubleu operator-(doubleu&),
            operator+(doubleu,doubleu),
            operator*(doubleu,doubleu),
            operator*(double,doubleu),
            operator/(doubleu,doubleu),

 friend int operator<(doubleu, doubleu),
            operator>(doubleu,doubleu),
            operator==(doubleu,doubleu),
            operator!=(doubleu,doubleu);

 friend double toduble(doubleu);
};

__________ OVERLOADED OPERATOR = __________

doubleu doubleu::operator=(doubleu& a){
 int i;
 int flag;
 double z;
 u.u_exp_of();
 for(i=0;i<U_NDIMS; i++)
   if(tempdim[0][i] == 0)
         flag=0;
   else {flag=1; break;}
 if(flag != 1)
   { v=a.v;
     u=a.u;
   }
 else {
         double x=compatible(u, a.u);
         if (x) {
                 z=a.factor_of();
                 z=z/this->factor_of();
                 v=a.v*z;
                return *this;
               }
         else cout<<"uerror(=) dimension is not correct";
     }
}


___________ COMPUTE THE POWER OF UQ ___________

doubleu power(doubleu a, double b)
{
  doubleu temp;
  int i=0,j=0,k=0,l=0,t=0, numerator=0, denominator=0;
  float x, z;
  if(b != 0)
    {
     j=(int)b;
     x=b-j;
     k=1;
```

```
      for(i=0;i<8;i++)
         {
          k=k*10;
          l=(int)(x*k);
          z=(float)l/k;
          if(z== x)
             break;
         }
      t=gcd(l,k);
      numerator=j*(k/t)+l/t;
      denominator=k/t;
      if(a.value_of()<0 && numerator%2 && !(denominator%2))
          {cout<<"negative number's sqrt";
           cout<<"\n";}
      else
          {temp.v=pow(a.v, b);
           temp.u=power(a.u, numerator, denominator);
          }
     }
  else
      {temp.v=1;
       temp.u=power(a.u, 0, 0);
      }
   return temp;
}
```

________ COMPUTE THE POWER OF UQ __________

```
doubleu power(doubleu a, int num, int den)
{ doubleu temp;
  if(a.value_of()<0 && num%2 && !(den%2))
    {cout<<"negative number's sqrt";
     cout<<"\n"; return temp;}
  else
    {temp.v=pow(a.v, (float)num/den);
     temp.u=power(a.u, num, den);
     return temp;
    }
}
```

_________ COMPUTE THE POWER OF UQ ________

```
doubleu power(doubleu a, int n)
{ int i;
  doubleu temp=doubleu(1.0);
  for (i=1; n; n--)
     temp*= a;
  return temp;
}
```

__________ UQ CONDITION < ___________

```
int operator<(doubleu a, doubleu b)
{ double x;
```

```
  x=compatible(a.u, b.u);
  if(x)
     {
      x=b.v*x;
      if(a.v < x)
           return 1;
      else return 0;
     }
  else cout<<"uerror(<)";
}
```

________ CONVERT A DIMENSIONLESS QUANT TO DOUBLE ______

```
double todouble(doubleu&b)
{
 int i, flag;
 double a;
 b.units_of();
 for(i=0;i<U_NDIMS; i++)
   if(tempdim[0][i] == 0)
          flag=1;
   else {flag=0; break;}
 if(flag == 1)
   { a=b.v;
     return a;
   }
 else cout<<"uerror(=): A units quantity assign to a double."<<"\n";
}
```

____________ COMPUTE SIN ______________

```
double sin(doubleu&b)
{
 int i, flag;
 double a;
 b.units_of();
 for(i=0;i<U_NDIMS; i++)
   if(tempdim[0][i] == 0)
          flag=1;
   else {flag=0; break;}
 if(flag == 1)
   { a=sin(b.v);
     return a;
   }
 else cout<<"uerror(=): A units quantity assign to a double."<<"\n";
}
```

_________ ASSIGNMENT *= ________

```
doubleu doubleu::operator+=(doubleu &a){
   doubleu temp;
   temp=*this*a;
   v=temp.v;
```

```
    u=temp.u;
    return *this;
}

____________ QU + ____________

doubleu operator+(doubleu a, doubleu b){
  doubleu temp;
  double x=compatible(a.u, b.u);
  if (x)
            {temp.v=a.v+x*b.v;
             temp.u=a.u;
             return temp;
            }
  else cout<<"uerror(+) dimesion error"<< "\n";
}

___________ QU * ________________

doubleu operator*(doubleu a, doubleu b){
  doubleu temp;
  double x;
  x=compatible(a.u, b.u);
  if(x)
            {
             temp.v=a.v*x*b.v;
             temp.u=a.u*a.u;
            }
  else { temp.v=a.v*b.v;
         temp.u=a.u*b.u;
       }
  return temp;
}
```

Appendix C. Examples

Example 1. If a body is projected horizontally with a velocity of 80.0 ft/sec from the top of the tower which is 160 ft high. Find the time of flight to reach the ground.

Solution: Using formula $\frac{a}{2}t^2 + v_0t - s = 0$. Where s=160ft, $v_0 = 0$,; $a = g = \frac{32ft}{\sec^2}$. The standard answer is t=3.16sec. (This example is token from University Physics [Mor69]. page 44-46. Using the English gravitational system.)

The program is given as below:

```
main()
{
 units ft="ft";
 units pound="pound";
 units sec="sec";

 doubleu s=doubleu(-160.0, ft);
 doubleu v0=doubleu(0, ft/sec);
 doubleu g=doubleu(32.0, ft/power(sec, 2));
 doubleu t=doubleu(0, sec);
 doubleu a=g/2;
 doubleu b=v0;
 doubleu c=s;
 doubleu y;
 doubleu zero=doubleu(0,power(ft/sec, 2));

y=power(b,2)-4*a*c;
 if(y>=zero)
   {
   t=(-b+power(y, 0.5))/(2*a);
   cout<<"The answer is:";
   printu(t);
   }
 else cout<<"There are complex solution."<<"\n";
 cout<<"\n";
cout<<"End of example 1. "<<"\n";
}
```

The output is:

```
The answer is:3.16228(sec=1).
```

End of example 1.

Example 2. What is the speed of a transverse wave traveling along a cord that has a linear density of $2.5*10^{-3}$ pounds-mass/foot and is under a tension of 15.0 pounds force?

Solution: Using formula v=sqrt(F/mu), where F=15.0pf, mu= $(2.5/32.0)*10^{-3}$ slug/ft. The standard answer is v=438 ft/sec. (This example is token from University Physics [Mor69]. page 460. Using the gcs system.)

The program is:

```
main()
{
 units gm="gm";
 units cm="cm";
 units sec="sec";
 units in=("in", 2.54*cm);
 units ft=("ft", 12*in);
 units kgm=("kgm", 1000*gm);
 units pm=("pm", 0.454*kgm);
 units slug=("slug", 32.3*pm);
 units pf=("pf", slug*ft/power(sec, 2));

 doubleu f=doubleu(15.0, pf);
 doubleu mu=doubleu(2.5/32.0, pow(10, -3)*slug/ft);
 doubleu v;

v=power(f/mu, 0.5);
 cout<<"The time is=";
 printu(v);
 cout<<"\n";
cout<<"End of example 2. "<<"\n";
}
```

The output is:

```
The speed is=13355.7(cm=1, sec=-1)

End of example 2.
```

Example 3. The example is same as example 2 but we directly use ft, slugs, and pf as computation units. Thus we could get more accurate results than example 2.

The program is:

```
main()
{
 units ft="ft";
 units sec="sec";
 units slug="slug";
 units pf=("pf", slug*ft/power(sec, 2));

 doubleu f=doubleu(15.0, pf);
 doubleu mu=doubleu(2.5/32.0, pow(10, -3)*slug/ft);
 doubleu v;

 v=power(f/mu, 0.5);
 cout<<"The speed is=";
 printu(v);
 cout<<"\n";
cout<<"End of example 3. "<<"\n";
}
```

The output is:

```
The speed is=438.178(ft=1, sec=-1)

End of example 3.
```

Example 4. A stone is projected from the surface of a flat field with a speed of 20m/s at an angle 53.1 degree above the horizontal. Find the stone's velocity and position at any instant.

Solution: Using formula:

```
        v0x=v0*cos(angle);  v0=20m/s, angle=53.1 degree.
        v0y=v0*sin(angle);
        vx=v0x;
        vy=v0y-g*t;
        v=sqrt(vx^2+vy^2);
        x=v0x*t;
        y=v0y*t-g*t^2/2;
        angle=atan(vy/vx)*180/3.14;
```

(The example is based on a similar example on pp 42-43 in [Til79].)

The program is:

```
main()
```

```
{
 units m="m";
 units kg="kg";
 units sec="sec";

 doubleu v0=doubleu(20, m/sec);
 doubleu v0x=doubleu(0, m/sec);
 doubleu v0y=doubleu(0, m/sec);
 doubleu vx=doubleu(0, m/sec);
 doubleu vy=doubleu(0, m/sec);
 doubleu v=doubleu(0, m/sec);
 doubleu t_max=doubleu(3.5, sec);
 doubleu t=doubleu(0, sec);
 doubleu delta_t=doubleu(.2, sec);
 doubleu g=doubleu(9.807, m/power(sec,2));
 doubleu y=doubleu(0, m);
 doubleu zero=doubleu(-0.00001, m);
 doubleu x;
 double angle=0;

 angle=0.926769817;      //0.926769817=53.1 degree.
 v0x=v0*cos(angle);
 v0y=v0*sin(angle);
 while (y>zero)  //In C 0.0>0.0 is true thus we put zero=-.00001.
   {
    x=v0x*t;
    y=v0y*t-g*power(t, 2)/2.0;
    vx=v0x;
    vy=v0y-g*t;
    v=power(power(vx,2)+power(vy, 2), 1/2.0);
    angle=atan(todouble(vy/vx));
    cout<<"The t, x, y, v are:"<<"\n";
    cout<<"____________________"<<"\n";
    cout<<"t= ";
    printu(t);
    cout<<"\n"<<"x= ";
    printu(x);
    cout<<"\n"<<"y= ";
    printu(y);
    cout<<"\n"<<"v= ";
    printu(v);
    cout<<"\n"<<"angle= ";
    cout<<angle*180/3.14;
    cout<<"\n";
    t=t+delta_t;
   };
cout<<"End of example 4. "<<"\n";
}
```

The output is:

```
The t, x, y, v are:
__________________
```

```
t= 0(sec=1)
x= 0(m=1)
y= 0(m=1)
v= 20(m=1, sec=-1)
angle= 53.1269

The t, x, y, v are:
____________________
t= 0.2(sec=1)
x= 2.40168(m=1)
y= 3.0026(m=1)
v= 18.4691(m=1, sec=-1)
angle= 49.4692

The t, x, y, v are:
____________________
t= 0.4(sec=1)
x= 4.80336(m=1)
y= 5.61292(m=1)
v= 17.0267(m=1, sec=-1)
angle= 45.1716

The t, x, y, v are:
____________________
t= 0.6(sec=1)
x= 7.20504(m=1)
y= 7.83096(m=1)
v= 15.6972(m=1, sec=-1)
angle= 40.1133


The t, x, y, v are:
____________________
t= 0.8(sec=1)
x= 9.60672(m=1)
y= 9.65671(m=1)
v= 14.5118(m=1, sec=-1)
angle= 34.1755

The t, x, y, v are:
____________________
t= 1(sec=1)
x= 12.0084(m=1)
y= 11.0902(m=1)
v= 13.5084(m=1, sec=-1)
angle= 27.2712

The t, x, y, v are:
____________________
t= 1.2(sec=1)
x= 14.4101(m=1)
y= 12.1314(m=1)
v= 12.7301(m=1, sec=-1)
angle= 19.3948
```

```
The t, x, y, v are:
____________________
t= 1.4(sec=1)
x= 16.8118(m=1)
y= 12.7803(m=1)
v= 12.2199(m=1, sec=-1)
angle= 10.6818

The t, x, y, v are:
_____________________
t= 1.6(sec=1)
x= 19.2134(m=1)
y= 13.0369(m=1)
v= 12.0122(m=1, sec=-1)
angle= 1.44371

The t, x, y, v are:
____________________
t= 1.8(sec=1)
x= 21.6151(m=1)
y= 12.9013(m=1)
v= 12.1224(m=1, sec=-1)
angle= -7.86936

The t, x, y, v are:
_____________________
t= 2(sec=1)
x= 24.0168(m=1)
y= 12.3734(m=1)
v= 12.5423(m=1, sec=-1)
angle= -16.7856

The t, x, y, v are:
____________________
t= 2.2(sec=1)
x= 26.4185(m=1)
y= 11.4532(m=1)
v= 13.2423(m=1, sec=-1)
angle= -24.9424



The t, x, y, v are:
_____________________
t= 2.4(sec=1)
x= 28.8202(m=1)
y= 10.1407(m=1)
v= 14.181(m=1, sec=-1)
angle= -32.1514

The t, x, y, v are:
```

```
________________________
t= 2.6(sec=1)
x= 31.2219(m=1)
y= 8.43594(m=1)
v= 15.3146(m=1, sec=-1)
angle= -38.3806

The t, x, y, v are:
________________________
t= 2.8(sec=1)
x= 33.6235(m=1)
y= 6.3389(m=1)
v= 16.6033(m=1, sec=-1)
angle= -43.6983

The t, x, y, v are:
________________________
t= 3(sec=1)
x= 36.0252(m=1)
y= 3.84958(m=1)
v= 18.0137(m=1, sec=-1)
angle= -48.2173

The t, x, y, v are:
________________________
t= 3.2(sec=1)
x= 38.4269(m=1)
y= 0.967978(m=1)
v= 19.5196(m=1, sec=-1)
angle= -52.0601

The t, x, y, v are:
________________________
t= 3.4(sec=1)
x= 40.8286(m=1)
y= -2.3059(m=1)
v= 21.1004(m=1, sec=-1)
angle= -55.3401

End of example 4.
```

References

[And93a] Lars Ole Andersen, Partial Evaluation for the C Language, in *Partial Evaluation and Automatic Program Generation*, pp. 229-258, 1993.

[And93b] Lars Ole Andersen, Binding-time analysis and the taming of C pointers, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 48-58, 1993.

[ASU86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading Mass., Addison-Wesley Publishing Company, 1986.

[Atk89] Kendall E. Atkinson, *An Introduction to Numerical Analysis*, John Wiley & Sons Inc. 1989.

[Atl82] *IEEE Standard C/ATLAS*, IEEE Standard 716-1982.

[BEJ88] D.Bjorner, A.P. Ershov, and N.D. Jones, *Partial Evaluation and Mixed Computation*, North-Holland: Amsterdam 1988.

[BGZ94] Romana Baier, Robert Gluck, and Robert Zochling, Partial evaluation of numerical programs in Fortran, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 119-132, 1994.

[CD93] C. Consel and O. Danvy, Tutorial notes on partial evaluation. *Conference Record of the Twentieth Symposium on Principles of Programming Languages*, Charlston, South Carolina. pp. 493-501, ACM Press, 1993.

[CG88] Robert F. Cmelik and Narain H. Gehani, Dimensional analysis with C++, *IEEE Software*, pp. 21-27, May 1988.

[Cha90] David R. Chase, Analysis of pointers and structures, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. White Plains*, New York, June 20-22, pp. 296-310, 1990.

[Cun92] R.Cunis, A Package for handling units of measure in Lisp, *ACM Lisp Pointers*, Vol. 5, No. 2, 1992.

[Foc53] C.M. Focken, *Dimensional Methods and Their Applications*, Edward Arnold, London, 1953.

[Geh77] Narain Gehani, Units of Measure as a data attribute, *Computer Languages*, Vol. 2, No 3, pp. 93-111,1977.

[Geh85] Narain Gehani, Ada's derived type and units of measure, *Softw. Pract. Exper.* 15, 6, pp. 555-569, 1985.

[Hil83] Paul N. Hilfinger, *Abstraction Mechanisms and Language design*, ACM Distinguished Dissertations., MIT Press, Cambridge, Mass. 1983.

[Hil88] Paul N. Hilfinger, An Ada package for dimensional analysis, *ACM Transactions on Programming Languages and Systems*, Vol. 10, No. 2, pp. 189-203, 1988.

[Hou83] R.T. House, A proposal for an extended form of type checking of expressions, *Computer Journal*, Vol. 26, No. 4, pp. 366-374, 1983.

[Isa75] E. de St Q. Isaacson and M. de St Q. Isaacson, *Dimensional Methods in Engineering and Physics*, Edward Arnold, London, 1975.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall International Series in Computer Science. Prentice Hall: New York, London, Toronto, 1993.

[Kay93] A.I. Kayssi, A methodology for the construction of accurate timing macromodels for digital circuits, *Ph.D. Thesis*, University of Michigan, 1993.

[KKZG95] Paul Kleinrubatscher, Albert Kriegshaber, Robert Zochling, and Robert Gluck, Fortran program specialization, *ACM SIGPLAN Notice*, Volume 30, No.4, pp. 61-70, 1995.

[KL78] Michael Karr and David B. Loveman III, Incorporation of units into programming languages, *Communications of the ACM*, Vol. 21, No.5, pp. 385-391, 1978.

[Knu81] Donald E. Knuth, *The Art of Computer Programming*, Addison-Wesley Publishing Company. Reading, Massachusetts, 1981.

[Mar95] S.A. Marinov, Reversed dimensional analysis, *Report*, Department of Mechanical and Industrial Engineering, The University of Manitoba, 1995.

[Mey91] Uwe Meyer, Techniques for partial evaluation of imperative languages, *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM'91 New Haven, SIGPLAN Notices* Vol. 26, No. 9, pp. 94-105, 1991.

[Mor69] Joseph Morgan, *Introduction to University Physics*, Allyn and Bacon, Inc., Boston, 1969.

[Nov95] Gordon S. Novak, Conversion of units of measurement, *IEEE Transactions on software engineering*. Vol. 21, No. 8, pp. 651-661, 1995.

[Pan64] R.C. Pankhurst, *Dimensional Analysis and Scale Factors*, Chapman & Hall, London, 1964.

[Sal95] D.J. Salomon, Safer_C: Syntactically improving the C language for error resistance, *Technical Report* 95/07, Department of Computer Science, University of Manitoba, 1995.

[Sal96] D.J. Salomon, Using partial evaluation in support of portability, reusability, and maintainability, *6th International Conference on Compiler Construction* CC'96, Linkoping Sweden, April 24-26, LNCS 1060, Springier, pp. 208-222, 1996.

[Tay74] Edward S. Taylor, *Dimensional Analysis for Engineers*, Clarendon Press, Oxford University Press, Ely House, London, 1974.

[Til79] Donald E. Tilley, *Contemporary College Physics*, The Benjamin/ Cummings Publishing Company, Inc. 2727 Sand Road, Menlo Park, California, 1979.

[Umr94] Z. D. Umrigar, Fully static dimensional analysis with C++, *SIGPLAN Notices*, Vol. 29, Iss:9, pp. 135-139, 1994.

[WCRS91] D. Weise, R. Conybeare, E. Ruf, and S. Seligman, Automatic online partial evaluation, *FPCA'91, 5th International Conference on Functional Programming Languages and Computer Architecture, number 523 in Lecture Notes in Computer Science*, pp. 165-191, 1991.

[WL95] Robert P. Wilson and Monica S. Lam, Efficient context-sensitive pointer analysis for C programs, *ACM SIGPLAN Notices, Conference on Programming Language Design and Implementation*. La Jolla, CA, June 18-21, pp. 1-12,1995.