Dimensionality in Programming – Literature summary

Charles Care

January 2005

1 Strong Typing and Physical Units [Män86]

This paper draws on the importance of syntax contraints in programming language design. Männer draws attention to the fact that although strong typing is a powerful way of "forcing compilation checks" it is "not as powerful as the [method] used in all physical calculations" where quantities are associated with units.

Männer also draws attention to the distinction between physical dimension and physical units. That is that a particular physical quantitiy always has a unique dimension, but can be represented by a number of units. For example, a distance has dimension *length* but can be expressed as a quantity with various *units* (e.g. 1 mile, 1.609 kilometers or 1609 metres.)

The relationship between dimension and typing is then discussed. The author shows that it is possible to define types for the different dimensions required. The following example is given:

```
TYPE LENGTH = INTEGER;
     TIME   = INTEGER;
```

All this really does is to provide 'friendly' names for two partitions of the `INTEGER` class. Quantities defined using such a type system can be treated in two ways. Firstly (Männer claims that most PASCAL compilers do this), the two types are treated as compatable (since they both have the same underlying type). If they are treated as compatable then the compiler provides no protection against a length being added to a time (which is physically meaningless). Secondly, a compiler might treat the two types as incompatable (Männer claims this is what would happen in Ada). As incompatible quantities there is protection against length being added to time; however, multiplication and division are also not allowed (because the types are incompatible) and this is a problem because quantities of different dimension can be combined in this way.

The implementation suggested by Männer is an extension to PASCAL and allows numeric literals to be given a unit. Units are defined by the programmer using the keyword UNIT.

```
UNIT g;        (* mass    *)
     cm;       (* length  *)
     sec;      (* time    *)
     cents;    (* money   *)
```

From defined units it is also possible to declare a unit as a derivation.

```
UNIT k  = 1000;                (* scale factor    *)
     kg = k * g;               (* mass            *)
     m  = 100 * cm;            (* length          *)
     N  = kg * m/(sec * sec);  (* force           *)
```

It is interesting how Männer defines the kilogram in terms of `k` and `g`. It would be a more elegant system if the meaning of the SI prefixes (i.e. m-, c-, k-, G- etc.) were defined independent

of any particular dimension. Otherwise a separate definition will be required for each type of unit, adding unnecessary code.

Although there is no discussion about polymorphic types the proposed system does appear to allow numeric types to be mixed in the conventional ways of imperitive programming. In the example program *Energy Loss* there is the statement:

```
I := 11.5 eV * ChargeTarget;
```

where `ChargeTarget` is a dimensionless *integer* and 11.5 eV has type *real*. This illustrates an important difference between *type* which is associated with machine representation and *dimension* which is associated with meaning. A quantity's dimension is independent of it's machine representation. Type checking prevents run-time problems assocated with the way the computer processes information, but dimension checking will help protect against the computer performing calculations that are not meaningful.

2 Implementation of Physical Units [Bal87]

This is an account of a simple implementation of physical units as a "minimal extension of a commerical Pascal compiler" and was produced for a commercial application. The implementation does not provide for user defined types and actually was only required to support the five base dimensions of Charge[1] , Length, Mass, Temperature, and Time. These are essentially the base dimensions of the SI system for physical units but with the Mole and the Candela ommitted.

The dimension of a value is represented by an array of five integers. Any dimension supported by the system can be written in terms of the five base dimension and these numbers correspond to the various powers of these. For example, velocity is represented ms^{-1} (or in terms of all the system's base dimensions $C^0m^1kg^0K^0s^{-1}$) this is then represented by the computer as `[0,1,0,0,-1]`.

Baldwin uses the idea of using normalised values and attributes this to [DMM86]. This means that all values of a quantity will be represented by the system in terms of its base unit. (E.g. all values of electrical current are "normalized to Ampere".)

For input values are allowed to have scaling prefixes on their units. These are implemented by Baldwin in the lexer of the system. For example $1.2mV$ is translated into $1.2 \times 10^{-3}V$.

3 Programming Pascal with Physical Units [DMM86]

The authors claim that programming with units can improve "readability and reliabity of software".

The paper gives a list of necessary ad desirable features of a language that incorporates phyical units. Attention is drawn to units being a higher level than types and are referred to as a "data attribute" units can be assigned to constants, types and variables. Although dimension is seen as something that could be attached to both integers and reals, it is suggested that for simplicity they could be attached just to reals. The proposal is to use the SI standard base units and to reserve the standard prefixes so that defining the meaning of "kilo" automatically gives meaning to km, kg, kA etc. This is their main disagreement frrom manner.

The physical language allows the introduction of new base units and the derivation of units. Also provided is a mechanism for restricting the way that a unit can be used. A unit is given one of the following three 'sorts'.

Short No scaling is allowed (e.g. you dont have milli-minutes)

Single Can not be used in further derivations

[1] In the SI System, Current is given as a base dimension instead of Charge. In the SI system Charge would have the base unit As as defined by: $Charge = Current \times Time$ In this system, Charge is a base dimension and so Current would have the base unit Cs^{-1}

Long No restriction

A detailed account is given of how WRITE statements are used to deliver units with various scaling and precion with or wothout their units.

Physical was written as a preprocessor not a compiler. a compiler would allow run-time checks and allow input and output routines to be integrated. Future work is outined in three areas:

1. To produce a complete compiler
2. To manage units with arbitary fixed points (e.g. Celcius and Fareneheit) and logarithic units (e.g. the dB). Although it is acknowledged that there is probably limited use for such an extension the writers feel that "allowing arbitary and not only multiplicative combinations between units could be an interesting extension."
3. To investigate whether such a system could perform error correction.

4 A Compact Representation of Units [Ham96]

A summary of a type calculus is given.

Based on arrays of exponent – references work by Novak on GLISP. Although [Bal87] is referenced by Hamilton as an example of a Pascal extension, no direct reference is made to the fact that the two implementations are practically identical. Hamilton differs from Baldwin in that the implementation is designed to be compact and so there are heavier restrictions on the range of values the various element of the array can take.

Emphasis is on the representstion being compact and unambiguous. The system is based heavilly on the SI system wih te inroduction of SI units. Extensibility is provided withing the SI unit space only. Money has the same dimension as number of sugars and rate of change of population is measured in Hz.

More importantly the system doesn't allow for scae factors which makes it difficult to support applications where lengths for instance are given in both feet and metres. Other writers have seen the protection such a construction offers as central to their motivations. However the emphasis on communication means that everything is converted back to SI base units before tranmission.

5 Incorporation of Units into Programming Languages [KDBL78]

The motivation for this work was the "design of a language for computer-controlled test and diagostic equipment" where having notations for physical units would be useful.

Karr and Loveman refer to the ATLAS test language which provided some support for expressing physical units. They also point to a paper by Cheatham as the "earliest mention of the idea".

In an attempt to capture what units actualy are, the writers see units as an "aspect of mode". Here the real numbers are partitioned by their dimension.

One issue drawn attention to is that although an equation might *commensurate* – that is have the same dimension on both sides. Its units might not be the same. Karr and Loveman have a vision that the system could provide necessary scaling so that 1km + 1mile is a valid formula.

Finally the authors point out the units that would be difficult to apply to their proposal. These are Celsius and Fahrenheit as well as AD for years. Although support for these is not generally required in engineering computing, it is pointed out that "they and other even more pathological 'units', such as the *decibel*, can be elegently handled by the 'pouches' mechanism". Pouches are a programming language construct invented by Cleaveland and are also mentioned by [SG80].

GO THROUGH THE VARIOUS STRENGTHS AND WEAKNESSES HE DESCRIBES AND RELATE THEM TO OTHER WRITING.

6 Rule-based Analysis of Dimensional Safety [CRV]

Discusses two examples of NASA failures due to quantities being provided in the wrong units. It is claimed that dimensional analysis tools would have avoided these failures.

The writers mention a C++ library developed for NASA that incorporates "hundreds of classes representing typical units ... together with appropriate methods to replace the arithmetic operators". They are critical of such a system and feel that the library "limit[s] the class of allowable... programs to an unnecceptably low level."

The feeling of the authors is that the concepts of programming with units hasn't been accepted by "mainstream programmers" and that this is because all of the extensions proposed so far not very light-weight.

The system described in the paper is an augmentation of BC – a GNU calculator language. It uses Maude?

7 Using Units of Measurement in Formal Specification [HM94]

8 Units of Measure as a Data Attribute [Geh77]

Gehani proposes an extension to Pascal that supports units called PASCAL/U. Each expression valid in the language will have two attribute – a *type* and its *units*.

8.1 Unit attributes of expressions

Gehani's propsals for representation of dimension are similar to those of [Bal87, Ham96] where a dimension is given by specifying the composite powers of bas dimension. The units attribute of an expression is define as a set of m ordered pairs, where m is a the number of base dimensions defined in the system. Each ordered pair contains an identifier of a base dimension and its corresponding power.

In Gehani's notation $\mathscr{A}(e) \cdot UNITS = [\langle d_1, x_1\rangle, \langle d_2, x_2\rangle, ..., \langle d_m, x_m\rangle]$ corresponds to the expression e having units $d_1^{x_1} d_2^{x_2} ... d_m^{x_m}$

One notable difference between Gehani's notation and other notations is that zero-th power dimensions are left out. e.g. $\mathscr{A}(sales) \cdot UNITS = [\langle dollar, 1\rangle]$ does not try to list all the other $m - 1$ dimensions with a power of zero.

Using this notation, Gehani defines what are legal combinations of various units in arithmetical operations.

8.2 Relating the theory to the PASCAL language

Gehani decides not to incorporate the UNITS attribute into the type attribute because of three main reasons.

1. Units of measure reflect physical characteristics while types reflect means of storage.
2. Different types require different amounts machine storage whereas different units do not.
3. Operand types affect the code generated whereas units just show whether an operation is legal (although units might provide some scaling factors).

Three representations are considered, but The syntax chosen is based on the ordered-pairs notation. Only non-zero dimensions need to be explicitly defined, and the syntax allows the exponent of a dimension to be omitted if it is 1.

```
const LIGHTSPEED = 3.0E8 UNITS (CM, SEC = -1);
var ROCKETSPEED: real UNITS (CM, SEC = -1);
```

Gehani notes that it might be desirable to have an equivilence between 'feet' and 'foot' in a dimensional programming language PASCAL/U doesn't support this.

Although PASCAL doesn't support dynamic type changing because of improved performance and compile-time checks, Gehani decides that units should be dynamically modifable. This provides the ability to have temporary variables that can hold data of any dimension.

```
var T: real UNITS(*);
```

This links with Kennedy's work on dimension and polymorphism [Ken]. The following procedure is Gehani's version of dimension polymorphism.

```
procedure SWAP (var X, Y : real UNITS(*));
var T: real UNITS(*);
begin  T := X;
X := Y;
Y := T end
```

It is important to note that although the definition of `SWAP` would allow the procedure to be called with arguments with different units, such a computation would not be allowed by the interal computation where `X` and `Y` need to have the same dimension.

It is however, possible to set the units of a variable based on the units of another.

```
var T : integer UNITSOF (A);
```

Numeric literals have no dimension so that

```
PROFIT := SALES - 1000;
```

Must be written as

```
PROFIT := SALES - 1000 UNITS(DOLLARS)
```

or

```
PROFIT := SALES - 1000 UNITSOF(SALES)
```

8.3 Units and dimension

Gehani's PASCAL/U does "unitwise" checking and not dimensional analysis

Hence the following is not allowed.

```
var M : real UNITS(MILES);
    K : real UNITS(KILOMETERS);

M=K;
```

However, Gehani does suggest that conversion could be done with a conversion factor that had the unit $km \cdot miles^{-1}$ e.g.

```
K = 1.6 UNITS(KILOMETERS,MILES = -1) * M;
```

The syntax of this is improved using the `counits` keyword.

```
UNITS(MILES) = 1.6 UNITS (KILOMETER);
```

9 A Proposal for an Extended Form of Type Checking of Expressions [Hou83]

Starts with a discussion about the problems with using types to represent units and then summarises Gehani's proposals for unit checking [Geh77]. In these proposals Gehani separates the notions of storage type and units.

However, House raises a number of issues with Gehani's PASCAL/U. One problem appears to be that in PASCAL/U "all names of units are essentially unrelated." There is no way of defining a new unit name – e.g. speed.

For House, the inclusion of the `UNIT(*)` construction leads to an unnacceptable restriction on the amount of compilation checks available.

House then proposes his implementation. He draws a disctinction between 'dimension' and 'unit' and makes use of the SI system with base and derived units. It is pointed out that there is no need to include both dimension and units in the language.

```
dim
  metre, kg, sec;
  volume:metre**3;
  unitspeed:metre/sec;
  newton:metre*kg*sec**(-2);

...

type
  speed = real dim unitspeed;
  time = real dim sec;
```

10 Software support for physical quantities [Hal02]

This paper wishes to push the support for physical quantities in programs further than previous literature. Hall emphasises that "units and measurment scales are closely related."

11 Meaning and Syntactic Redundancy [Cle75]

Cleaveland introduces the concept of pouches to provide subsets of existing types. The examples of pouches given show how they can be used to manage dimension.

```
POUCH years INT age, birth, death;
POUCH people INT population,workers;
...
population := workers + 5 people;
```

This is effectively a way of using a type system to restrict cross-dimensional addition. However, Cleaveland does take it further to show how pouches could model multiplicative combinations of dimensions.

> *Many users will probably want to say more by saying what kind of relationships exist between pouches. For example a variable in pouch x when mulitplies by a variable in pouch y produces a variable in pouch z, such as in the case of the formula "pay = rate * time".*
>
> [Cle75]

One difficulty with the system proposed is that there is no way of giving a pouch a name other than its definition. Therefore write the quanity $1.5ms^{-1}$ will require the input[2]: `1.5 meters•second^-1`

Pouches are more general than just dimension and have their own algebra which forms an Abelian group. Dimensionality is therefore just one instance of how pouches could be used in a programmind language.

Cleaveland suggests an implementation where each element of the group can be represented by an n-tuple (where n is the number of base pouches) of integers. Each integer corresponds to the exponent of the base pouch. Although it's not referred to in the paper, this could be extended to a tuple of reals should non-integer exponents be required. The problem with this representation is that the number of base pouches need to be determined to decide how these tuples should be. Cleaveland provides no suggestion how this might be done. Also it is unclear how the user would be restricted from creating a new base pouch whose tuple is not linearly-independent from the tuples of the existing base pouches.

Cleaveland points out that "generic routines cannot be written for integer for any pouch , since the modes of formal and actual parameters must agree." This difficulty is got around by introducing the "hidden pouch" that "hides the pouch of the actual operator". In this provision, Cleaveland's work is not greatly different from the `UNITS(*)` construct in [Geh77], which was criticised by [Hou83].

12 side issue

Why not have different unit spaces i.e dimension money = SI[0,0,0,0,0,0,0] 10 U[money] + 13 U[money] + money¡13 SI[0,0,0,0,0,0,0]¿

dimensionless quantiites might be added to others using the ? unit. this allows literals to be used for incrementing e.g. in solving an equation for lengths 1m to 10m one would want to do

```
for(x=1 m; x <= 10 m; x = x + 1 ?) {...}
```

WHAT IS SUCH A SYSTEM TRYING TO ACHEIVE?

References

[Bal87] Geoff Baldwin. Implementation of physical units. *SIGPLAN Not.*, 22(8), 1987.

[Cle75] J. C. Cleaveland. Meaning and syntactic redundancy. pages 115–124. Institut de Recherche D'Informatique et D'Automatic, 1975.

[CRV] Feng Chen, Gigore Rosu, and Ran Prasad Venkatensan. Rule-based analysis of dimensional safety. http://fsl.cs.uiuc.edu/ grosu/.

[DMM86] A Dreiheller, B Mohr, and M Moerschbacher. Programming pascal with physical units. *SIGPLAN Not.*, 21(12):114–123, 1986.

[Geh77] Narain Gehani. Units of measure as a data attribute. *Computer Languages*, pages 93–111, 1977.

[Ham96] Bruce Hamilton. A compact representation of units. Technical report, Measurement Systems Department, Hewlett-Packard Laboratories, 1996.

[HM94] Ian J. Hayes and Brendan P. Mahony. Using units of measurement in formal specifications. Technical report, Software Verification Research Centre, University of Queensland, Brisbane, Australia, 1994.

[2]Cleaveand doesn't give us an ascii representation of his Algol 68-style syntax but leaves these dots and superscripts.

[Hou83] R. T. House. A proposal for an extended form of type checking of expressions. *The Computer Journal*, pages 366–374, 1983.

[KDBL78] Michael Karr and III David B. Loveman. Incorporation of units into programming languages. *Communications of the ACM*, 21(5):385–391, 1978.

[Ken] Andrew Kennedy. Programming languages and dimensions. Technical report.

[Män86] R. Männer. Strong typing and physical units. *SIGPLAN Not.*, 21(3):11–20, 1986.

[SG80] H.T. Smith and T.R. Green, editors. *Human Interaction With Computers*. Academic Press, London, 1980.