DRAFT IN PROGRESS

Developing Robust Software in Applied Mathematics

Dr Mark Tarver
dr.mtarver@ukonline.co.uk
Lambda Associates
www.lambdassociates.org

An apprentice carpenter may want only a hammer and saw, but a master craftsman employs many precision tools. Computer programming likewise requires sophisticated tools to cope with the complexity of real applications, and only practice with these tools will build skill in their use.

Robert L. Kruse, Data Structures and Program Design

One of the hallmarks of an advanced civilisation is the complexity of its technology. Just over one hundred years ago, in 1903, the first powered flight was made by Orville Wright at Kittyhawk. The Wright aeroplane was assembled by two brothers armed with a background in making bicycles and a rudimentary grasp of the principles of aerodynamics. The *Flyer I* cost less than a thousand dollars to construct.. It had a wingspan of 40 feet (12 m), weighed 750 pounds (340 kg), and sported a 12 hp (9 kW), 170 pound (77 kg) engine.

One hundred years later the American air force unveiled the F-22 Raptor stealth fighter at a unit cost of over 300 million dollars. Using advanced metallurgy, including composite materials, the Raptor uses Raytheon processors rated at 10.5 billion instructions per second to process its radar information and the programs controlling the flight and radar total 1,7 million lines of code. Whereas the Flyer was largely the product of two men, and understandable to anybody with a grasp of nineteenth century technology, the Raptor is a product of mass cooperation and its C21 complexity is completely beyond the grasp of any single human mind.

The technological gap between these two pieces of engineering is tribute to the progress in science and engineering that took place during the last century. But the very complexity of our creations has placed undue strain on our traditional methods of organizing our communication and integrating the efforts of our engineers necessary to create products like the Raptor. These processes are often stretched to breaking point and when they do break in the context of a major engineering initiative, the results can be embarrassing, expensive and potentially even fatal. Once such example was the NASA Mars Lander. Dr Edward Weiler, NASA's Associate Administrator for Space Science, recorded the reason for the $120 million crash of the Mars Lander in 1999.

> “People sometimes make errors. The problem here was not the error, it was the failure of NASA's systems engineering, and the checks and balances in our processes to detect the error. That's why we lost the spacecraft."

The peer review preliminary found that one software team had used English units (i.e. inches, feet and pounds) while the other used metric units for a key spacecraft operation. This information was critical to the maneuvers required to place the spacecraft in the proper Mars orbit.[1] Apart from the loss of money, the blow to NASA’s prestige was severe.

One Reason Why Traditional Engineering Project Often Result in Buggy Software

Modern science and engineering is now massively dependent on computing resources to the extent that hardly any significant scientific or engineering project can be conducted without the

[1] http://marsprogram.jpl.nasa.gov/msp98/news/mco990930.html

use of computers to perform the maths. The part of maths that we are concerned with here is of course applied mathematics. We define applied mathematics as the application of mathematics to the calculation of quantity. By quantity we understand any form of measure – whether of mass, energy, work, length or time.

Since applied mathematics is concerned with measure, we might expect that computer models that use applied mathematics would explicitly represent those measures. In fact, this is not generally true. In current computer science, computations involving applied mathematics are not different from computations of pure mathematics. In both cases, numerical computations are involved, and it is the *interpretation* given to these computations that demarcate pure from applied studies. This interpretation is typically not part of the program itself, but exists in the mind of the developer. If she is particularly thorough, then it may exist informally as documentation to the program; but this documentation is incidental to the compilation and execution of the program itself.

This rather loose and haphazard arrangement has significant dangers. The dangers arise when the informal interpretation does not match the numerical computation. This of course is precisely what happened in the case of the Mars Lander. Since the interpretation is informal and not part of the program itself, a failure of this kind will pass unnoticed by the computer.

Tackling the Problem

Existing approaches to the problem emphasise project management approaches to develop reliable software. However though organizing and reshuffling the human component may have its uses, a more promising approach is to minimize human error by formalizing the hitherto informal interpretations attached to the parameters of the computation. Just as applied mathematics originally advanced in speed and reliability by removing the responsibility for calculation from slow and error-prone human beings, so it is logical to look to computers to tackle the problem of guaranteeing the reliability of engineering projects. Such a guarantee must rely on the formalisation of the hitherto informal and loosely worded descriptions of what the program is supposed to be calculating. Such a formalisation can be either **explicit** or **implicit**.

The explicit approach relies on explicitly attaching the interpretation to the parameters in the program. Hence instead of writing 9.8 and commenting that this figure is to be read in pounds, it would be possible to explicitly write [9.8 lbs]. The interpretation of any applied mathematics program would then be immediately recognisable to the computer. In the philosophy of mathematics, objects that mix units of measure with numbers are referred to as **impure numbers**. Hence the explicit approach requires that impure numbers be treated as structures distinct from numbers.

However there are two compelling arguments against the explicit approach.

The first is that programmers in science and engineering do not operate with impure numbers as objects different in kind from numbers. Since impure numbers, under the explicit approach, are lists and not numbers, two impure numbers cannot be multiplied together nor can any impure number be multiplied by any kind of number. Not a single mathematical operation, however humble, could be applied to an impure number without recoding that operation. Using impure numbers with explicit measures means that all math libraries must be redefined.

Second, even if the libraries could be rewritten, the resulting operations would be less efficient since every arithmetic operation would require extracting the numerical component from a structure, performing the operation, and then re-embedding the result back into a new structure.

Ergo the explicit approach is not a feasible solution.

In contrast the implicit approach does not divorce impure numbers from numbers. *This approach follows the orthodox view that impure numbers are just numbers put to a certain use. The difference is that the implicit approach requires that the use is codified and formalised in the type theory of impure numbers.* It then becomes the responsibility of the computer running the type checker to ensure that the program is free from certain basic coding errors common to engineering applications. These errors range from basic blunders involving inappropriate metrics to conversion errors of various kinds and the misrepresentation of mathematical equations within stretches of code.

The implicit approach is promising, but the requirements placed on the type checker go well the resources of what current programming languages can deliver; including those considered state of the art such as ML, Haskell and OCaml. To implement this approach comprehensively, a type notation is required capable of expressing the complex logical and numerical relationships that exist within the type theory of applied mathematics.

The sequent calculus notation of *Qi*, derived from the foundational work of the logician Gerhard Gentzen, is a good notation for constructing a specification of these requirements. It has the advantage of being based on a clear and powerful notation used widely in theoretical computer science, and, through *Qi*'s efficient compiler technology, of delivering fast and efficient type checkers from what is, in effect, an executable high-level specification of a type theory.

A Type Theory of Metrics

What does this theory look like? It is a formal theory of **metrics**. Metrics are what we use to measure quantity. There are an indefinitely large number of metrics corresponding to the conventional metrics for measuring distance, mass, energy, time and many other aspects of the physical world. The theory begins with a statement about metrics.

1. Objects grouped under metrics are numbers.

This is a fundamental claim. This immediately allows the connection of impure numbers to any mathematical operation and opens the way for the free application of functions from pure mathematics to applied mathematics. We use *Qi* sequent notation to express these relationships.

```
(datatype metric
______________________
(subtype metric number);

(subtype B A); X : B;
__________________
 X  : A;)
```

Lets concentrate on metrics for distance. There are many different conventions for measuring distances; the European metric system and the Imperial are but two. Let us take one metric from each; inches and centimeters.

```
(datatype distance

_________________
centimeters : metric;

______________
inches : metric;)
```

We can add distances under different metrics to produce numbers.

```
(define add-distances
   {inches --> centimeters --> number}
    X Y -> (+ X Y))
```

But we cannot add inches to centimeters to produce inches.

```
(define add-distances
   {inches --> centimeters --> inches}
    X Y -> (+ X Y))
```

This is right. We should not be able to add inches to centimeters and gain anything more than a number. This was precisely the error of the software in the Mars Lander and already this error is blocked.

However our system is too severe, we cannot add inches to inches and produce inches.

```
(define  add-inches
   {inches --> inches --> inches}
    X Y -> (+ X Y))
```

This leads to the second claim.

2. A numeric operation applied to objects grouped under the same metric gives an object of the same metric.

We want to say that two objects that belong to the same metric can added or subtracted together and the result will be an object belonging to the same metric as the inputs. For instance, it should be possible to double the distance in miles between two points to calculate a round trip and this doubling should produce an impure number of the same type as the input i.e. a quantity in miles.

The following rules state this claim for 1 and 2-place functions.

```
Op : (number --> number);
A : metric; N : A;
(Op N) : A;
```

```
 Op : (number --> number --> number); A : metric; M : A; N : A;
(Op M N) : A;
```

Metric Conversion

Metric conversion is a basic operation. It occurs whenever we want to convert one metric into another; dollars to pounds, inches to centimeters, joules to calories are examples. Often this involves multiplication or division by a constant. Inches to centimeters is a good example. Inches convert to centimeters under the formula C centimeters = (C * 2.54) inches. In sequent calculus we write

```
I : inches;
(* I 2.54) : centimeters;
```

The inverse conversion is trivially derivable

```
C : centimeters;
(/ C 2.54) : inches;
```

But if the inverse conversion is trivially derivable then it should be the responsibility of the computer to infer it. By adopting this approach, we avoid having to write an inverse for every conversion of this kind. We know

I : inches; (* I 2.54) : centimeters;	if and only if	C : centimeters; (/ C 2.54) : inches;

In fact the nature of the metric (inches, centimeters etc) and the value of the constant 2.54 is irrelevant here. We can assert for all metrics A, B and for any constant K.

I : B; (* I K) : A;	if and only if	C : A; (/ C K) : B;

Hence

I : B; (* I K) : A;	and C : A;	implies	(/ C K) : B;

Written in sequent calculus we are affirming that where I is an arbitrary term and A and B are metrics, the following rule holds

```
C : A;
I : B >> (* I K) : A;
(/ C K) : B;
```

In *Qi* this is easily transcribed **(gensym "&&")** generates an arbitrary term. The following script shows the

```
(2+) (datatype simple-metric-conversion

   let I (gensym "&&")
   C : A;
   I : B >> (* I K) : A;
   A : metric; B : metric;
   ---------------------------
    (/ C K) : B;)


 (3+) (define inches_to_centimeters
          {inches --> centimeters}
           I -> (* I 2.54))
inches_to_centimeters : (inches --> centimeters)

(4+) (define centimeters_to_inches
          {centimeters --> inches}
          C -> (/ C 2))
centimeters_to_inches : (centimeters --> inches)
```

The simple-metric-conversion rule removes the necessity for providing inverses for every conversion rule driven by a simple equation of the form x = Ky where K is a constant.

But not every conversion rule takes such as simple form. The conversion of centigrade to fahrenheit is driven by an equation, but not one of that precise form. The equation is $x^{o}F = (x^{o}C * 5/9) + 32$.

Though it is possible to orient this equation to derive the inverse conversion using a similar pattern of reasoning, the utility of such a rule is far less than the type rule relating multiplication and division. It is highly unlikely that the rule will find any employment beyond centigrade to fahrenheit conversions. A general solution to the problem would allow the computer to infer the inverse by examination of the equations driving the transformation. This is an advanced problem but not intractable given achievements in computer algebra and the resources of *Qi*.

Derived Metrics as Ratios

Derived metrics are metrics that express relations between other metrics. A simple example is speed which is the ratio between distance covered and time taken. In English the word 'per' is used to denote such ratios. Type theoretically, per is a 2-place type operator receiving metrics as operands.

3. *The ratio of any two metrics is a new metric*

The type rule is

```
(datatype per

          A : metric;
          B : metric;
          X : A; Y : B;
          ____________
          (/ X Y) : (per A B);

           (per X Y) : metric;)
```

We assume seconds as a metric.

```
(datatype time
  ________________
  seconds : metric; )
```

This allows the immediate definition of inches-per-second.

```
(define inches-per-second
   {inches --> seconds --> (per inches seconds)}
    I S -> (/ I S))
```

If we multiply 5.6 inches per second by 4 seconds we get a figure in inches. Also if we divide 5.6 inches per second into 4 inches we get seconds. This is generally true of derived metrics built using per and the reasoning is encapsulated in two rules.

```
X : (per A B); Y : B;
_________________
(* X Y) : A;

Y : (per A B); X : A;
_________________
(/ X Y) : B;
```

Derived Metrics as Powers

A basic operation over impure numbers is the raising a measure to a higher power. Examples are centimeters to square centimeters, square centimeters to cubic centimeters and so on. Typically such numbers are derived by the multiplication of two impure numbers that use identical metrics of distance.

3 inches * 4 inches = 12 square inches

4. The multiplication of two objects under identical metrics produces an object under a metric of a higher power.

Here we hit a problem. Suppose we assert

3 inches * 4 inches = 3 inches + 3 inches + 3 inches + 3 inches

On the basis of the previous rule that inches plus inches gives inches we derive

12 square inches = 3 inches * 4 inches = 3 inches + 3 inches + 3 inches + 3 inches = 12 inches

which is absurd.

Accordingly we must reject the identity

3 inches * 4 inches = 3 inches + 3 inches + 3 inches + 3 inches

and assert instead that although 3 * 4 inches = 3 inches + 3 inches + 3 inches + 3 inches = 12 inches, 3 inches * 4 inches raises the metric to a higher power. Conventionally this higher power is represented by terms such as 'squared' or 'cubed'. For our purposes we use 'square' as an operator for raising the power of a metric. Hence '9 : (square (square inches))' means for us the same as '9 cubic inches'.

```
(datatype cubic

X : (square (square A));
==================
X : (cubic A);)
```

The rule for squaring is

```
(datatype square

  A : metric; X : A; Y : A;
  =================
  (* X Y) : (square A);
  ..........
```

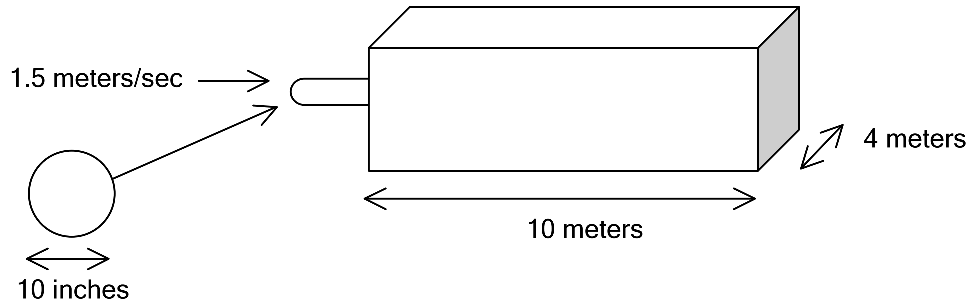
To lower the power of a metric, we divide a higher power of that metric by a lower power.

```
  ...........
   A : metric; X : (square A); Y : A;
  ______________________________
  (/ X Y) : A;)
```

A Concrete Problem

A properly formulated working type theory of metrics has a potential to make a significant contribution to the reliability of engineering and science programs. Let us take as an example the following problem.

A pipe with a diameter of 10 inches pumps water at a rate of 1.5 meters/sec into a tank which has sides 10 meters long by 4 meters wide. How long does it take to fill the tank to a depth of 3 meters?



This is a good example involving several metrics and two different measuring systems. To implement a program to compute the answer requires first designing an interface to receive the inputs. These inputs must carry with them the metrics which are attached to them if the program is to be checked as type secure.

We cannot simply input numbers (as in **(input+ : number)**) because although objects that fall under metrics are numbers, not all numbers fall under a metric. A special function **input-metric** is constructed to do the job. It carries two arguments; the first is a metric and the second is a message which is displayed when the function is called. The function waits for the user to enter a number and returns that number under the metric.

The program begins by requiring the user to enter the parameters of the problem.

```
(define solve-problem
  {A --> seconds}
   _ -> (compute-solution (input-metric inches “Enter bore of pipe”)
                          (input-metric (per meters seconds) “Enter flow rate”)
                          (input-metric meters “Enter length of tank”)
                          (input-metric (square meters) “Enter width of tank”)
                          (input-metric meters “Enter desired depth of water”)))
```

The program that computes the solution performs the following operations.

1. It calculates the radius of the bore in meters.
2. It uses this figure to calculate the area of the bore in square meters.
3. It uses the area figure to calculate the volume flow in cubic meters per second .
4. It calculates the volume of the tank in cubic meters.
5. It divides the volume flow by the volume to get a figure in seconds.

The task of the type system is to check that the computation is type secure with respect to its use of metrics. Here is the program.

```
(define compute-solution
   {inches --> (per meters seconds) --> meters --> meters --> meters --> seconds}
    Bore Flow Length Width Depth
     -> (mlet Bore-Meters-Radius (/ (/ (* Bore 2.54) 100) 2)
               Bore-Area (* (* Bore-Meters-Radius Bore-Meters-Radius) (value *pi*))
               Volume-per-sec (* Flow Bore-Area)
               Tank-Volume (* Depth (* Length Width))
               Time (/ Tank-Volume Volume-per-sec)
               Time))

(set *pi* 3.142)
```

The type theory for this program is given below.

CODETO BE COMPLETED

Conclusion

Orthodox programs in science and engineering manipulate figures which correspond to physical quantities. However orthodox programming is simply 'number crunching' without safeguards to guarantee that the computation models a physically tenable process. This arises because the interpretations attached to the parameters of the program are typically held either (a) within the brain of the programmer or (b) written in an informal and *ad hoc* fashion in the supporting documentation. No formal guarantee exists that the program is representing any process consistent with the equations that are supposed to drive its development.

This draft has described a small but essential fragment of the type theory of applied mathematics. The construction of an adequate theory for applied mathematics is not a small project. But by carefully codifying the type theory of applied mathematics, it is possible to automate the verification of the physical credentials of the program with respect to the underlying science. This approach opens the way for a new generation of reliable engineering programs.