Automatic Unit Checking in Neuron Simulations

Scott Urnikis and Epaminondas Rosa
Illinois State University
(Dated: April 23, 2019)

ABSTRACT

The utility of a language, be it natural language, mathematical notation, or programming language, is to express thought to some audience. For example, with natural language, the audience is other people that we wish to share our thoughts with. We may define new vocabulary in order to extend the language and make communication easier. As long as these linguistic extensions are communicated and understood, the expressiveness of the language increases and the thoughts that we want to communicate become easier to state in the newly extended language.

As another example, we may consider the units used in physics as a linguistic extension to basic mathematical notation. These units allow us to annotate numbers with information about the physical significance of those quantities. Our interest, then, is to see whether we can create linguistic extensions to computer programming languages that mirror the units found in the manual computations done by practitioners of physics. Specifically, using this proposed method, we attempt to verify the physical meaning of the quantities found using numerical methods in simulations of neurons using the Hodgkin Huxley model. We use C++ as an implementation language because it has faculties that allow us to extend the language so that the compiler can understand and enforce our unit annotations. It is our hope that being able to express information about units in our source code will help eliminate the same class of errors that are found by utilizing units in the manual computations.

I. MOTIVATION

In order to simulate a neuron using the Hodgkin Huxley model for a single neuron, we must translate that set of first order differential equations into a format that is executable. As an example, the change in voltage with respect to time is represented by the equation

$$C\dot{V} = I - \overline{g_{\mathrm{K}}} n^4 (V - E_{\mathrm{K}}) - \overline{g_{\mathrm{Na}}} m^3 h (V - E_{\mathrm{Na}}) - g_{\mathrm{L}} (V - E_{\mathrm{L}}),$$

where V is the voltage across the membrane of the neuron, I is the electric current being applied to the current, g_K, g_{Na}, and g_L are the conductances for the ion channels in the cell membrane, E_K, E_{Na}, and E_L are voltage sources determined from differing concentrations of the ions of interest, and n, n, and h are dimensionless quantities that are related to the activation of the ion channels. The values of n, m, and h are also determined by time derivatives that are dependent on the voltage, but we omit their definitions here for brevity. We find that the values of V, n, m, and h are variable over time, and all other values are approximated as being constant over time.

By noting that $\dot{V}$ is a function of V, n, m, and h, we may write a direct translation in C++. We make an assumption that any free variables in the expression below are well defined in the program this function resides in.

```
double
dV_wrt_dt(double V,
          double n,
          double m,
          double h)
{
  return
   (I - g_K*pow(n, 4)*(V - E_K)
      - g_Na*pow(m, 3)*h*(V - E_Na)
      - g_L*(V - E_L))
   / C;
}
```

Unfortunately, this definition of $\dot{V}$ does not contain any information about the dimension of the operands with respect to the Hodgkin Huxley model. We consider this omission of physical information to be wasteful, precisely because this information already exists in the model that we are simulating. Instead of discarding it, we may create new types using the faculties provided to the user by The C++ Programming Language so that are able to encapsulate the information that could not currently be expressed. Although the following techniques are not unique to C++, we find it to be an appropriate implementation programming language due to its popularity in the scientific computing community as well as its sufficiently featureful type system.

II. DEFINITION OF THE QUANTITY TYPE

In order to store physical information into the simulation program, we must first define a family of types that are capable of representing all of the relevent units used in the Hodgkin Huxley model. To simplify this problem, we will only consider the set of units defined by the International System of Units. By noting that there are seven base units defined this standard, we may define the `Quantity` type as a parametric type that depends on a tuple of length seven that contains information about the exponent of the base unit. Conceptually, we may

consider this as a function that takes the tuple as an argument and creates a one-to-one mapping to a unique type from the universe $\mathscr{U}$, which can be read as the type of types.

$$Quantity : \mathbb{Z}^7 \to \mathscr{U}$$

Here we assign base units to the element positions inside this tuple of length seven. Let m, kg, s, K, A, mol, and cd be integers. Then, a unit may be represented by the type

$$Quantity\,(m, kg, s, K, A, mol, cd)$$

where m, kg, s, K, A, mol, and cd represent the exponent of the unit.

We find that integers are adequate for describing the dimensionality of the Hodgkin Huxely model. Not all operations are able to be represented within this framework, however. For full generality, one would use a tuple of rationals instead of integers. We omit this functionality for simplicity of implementation. A scheme for representing units with rational exponents can be found in the report "Applied Template Metaprogramming in SIUNITS: the Library of Unit-Based Computation" written by Walter E. Brown.

In C++, we may declare a `Quantity` family of types in the following way:

```
template <int m, int kg, int s, int K,
          int A, int mol, int cd>
class Quantity {
  Quantity() :value{0.0} {}
  Quantity(double value) :value{value} {}
  double value;
};
```

By giving the `Quantity` class a member named `value` of type `double`, we can begin do define operations on quantities so that the properties of a `Quantity` closely mirror that of a real number.

To make this more concrete, we may define types now that can be used to represent units. In C++, we are able to define new names for existing types by using the `using` keyword.

```
using meter   = Quantity<1, 0, 0, 0, 0, 0, 0>;
using second  = Quantity<0, 0, 1, 0, 0, 0, 0>;
using Newton  = Quantity<1, 1, -2, 0, 0, 0, 0>;
```

III. OPERATIONS ON QUANTITIES

The usefulness of this technique becomes more apparant once functions that operate on `Quantity` objects are defined. When an expression is manually computed in the physical sciences, the resulting values from the computed expression has a unit that is determined by the operator and the units of the operands. These rules are consistent and can be easily implemented so that `Quantity` objects can be operated on in the same way.

A. Addition

Addition between two `Quantity` objects is only defined when both of the operands have the same unit. We may define addition in the following way.

```
template <int m, int kg, int s, int K,
          int A, int mol, int cd>
Quantity<m, kg, s, K, A, mol, cd>
operator+(Quantity<m, kg, s, K,
                   A, mol, cd> op1,
          Quantity<m, kg, s, K,
                   A, mol, cd> op2)
{
  return Quantity<m, kg, s, K, A, mol, cd>{
    op1.value + op2.value
  };
}
```

We can see that the resulting value from the addition of two `Quantity` objects is a `Quantity` object of the same type.

B. Multiplication

Multiplication between two `Quantity` objects is defined for all operands, regardless of the unit of the operand.

```
template <int m1, int kg1, int s1, int K1,
          int A1, int mol1, int cd1,
          int m2, int kg2, int s2, int K2,
          int A2, int mol2, int cd2 >
Quantity<m1 + m2, kg1 + kg2, s1 + s2,
         K1 + K2, A1 + A2,
         mol1 + mol2, cd1 + cd2>
operator*(Quantity<m1, kg1, s1, K1,
                   A1, mol1, cd1> q1,
          Quantity<m2, kg2, s2, K2
                   A2, mol2, cd2> q2)
{
  return Quantity<m1 + m2, kg1 + kg2,
                  s1 + s2, K1 + K2,
                  A1 + A2, mol1 + mol2,
                  cd1 + cd2> {
     q1.value * q2.value
  };
}
```

Now that these operations are defined, it is easy to imagine how subtraction, multiplication, and exponentiation, could be defined. The implementations discussed thus far first appeared in the textbook "Scientific and Engineering C++" by Barton and Nackman. The following are very useful extensions to the previous design.

C. Exponentiation

Exponentiation of quantities follows the same rules as repeated multiplication. These operations are not provided by the programming language itself, but instead are typically provided in the standard library. Because of this, it would be useful to be able to interface with the already existing functions provided for performing mathematical computations. In this light, exponentiation is a special case of this general problem of interfacing with previously existing libraries, but we consider it separately here because exponentiation can be used on any `Quantity` to produce a new `Quantity` that represents a value with a different unit from the operand. As a concrete example, consider that we must model the behavior exhibited in the following expression.

$$(2 \cdot \text{meter})^2 = 2^2 \cdot \text{meter}^2 = 4 \cdot \text{meter}^2.$$

Here we find that a `Quantity` of unit meter[1], when exponentiated with a power of 2, will result in a new `Quantity` with a unit meter[2]. The general rule is that the elements of the tuple that contains the exponents of the base units of the `Quantity` being operated on is multiplied elementwise by the power used in the exponentiation operation and then used as the tuple for the new `Quantity`.

To provide the user with flexibility, a function object is provided that accepts a type that is the class of another function object that the user may provide for implementing the exponentiation operation on objects of type `double`. The definition of this mechanism is provided below.

```
template <typename P, int e>
class CreatePowerFunction {
private:
  P function;
public:
  template <int m, int kg, int s, int K,
            int A, int mol, int cd>
  Quantity<m * e, kg * e, s * e, K * e,
           A * e, mol * e, cd * e>
  operator() (Quantity< m, kg, s, K,
                        A, mol, cd> base)
  {
    return function(base.value, e);
  }
 };
```

The `CreatePowerFunction` class accepts a type as an argument and an exponent value. Internally, it instantiates a function object of that class and implements a function that uses that object in its definition. Furthermore, the `CreatePowerFunction` class also accpets an integer, and encodes that information into the return type of the function so that the `Quantity` that is returned by the function has the correct unit.

It is important to note that if we had access to dependent types, we could simply make the return type of the exponentiation function depend on the exponent passed to it, and represent this function as a function of arity two. Instead, we must make different exponentiation functions for each exponent that we would like to use in this operation, and generate arity one functions that accept only the base of the exponentiation operation. We describe a family of functions that represent the different possible integer exponents available.

In order to create objects from the `CreatePowerFunction` class, we must first wrap the function we would like to use to implement exponentiation in a class so that we can associate some type with a specific function. Ideally, this association would provide the compiler with enough information to inline the definition of the `operator()` function where it is used.

```
class Power {
public:
  double
  operator()(double base,
             int exponent)
  {
    /* pow from the math.h header */
    return pow(base,
               static_cast<double>(
                 exponent));
  }
};
```

Then, using this new type `Power`, we may create objects `pow3` and `pow4` for exponentiating `Quantity` objects to the third and fourth power, respectively.

```
CreatePowerFunction<Power, 3> pow3;
CreatePowerFunction<Power, 4> pow4;
```

The line between object and function is blurred here; technically, `pow3` and `pow4` are instances of the class `CreatePowerFunction` that use the `Power` class to create a function `operator()` that is callable with the same syntax as if `pow3` and `pow4` were instead simple fuctions.

The strategy that we have used to implement exponentiation could be used with similar affect for implementing the root. Because this functionality would require access to rational elements in the tuple that we use to represent the unit of a `Quantity`, we omit it for simplicity of implementation. We are allowed this freedom because the Hodgkin Huxley model does not require the use of the root operation.

D. Dimensional Coercion

Upon close inspection of the Hodgkin Huxley model, we find that some of the expressions will not unit check properly. Specifically, this anomaly appears in the equations that describe the rate of change over time of the

variables n, m, and h. We note that the form of these differential equations are very similar to each other. Because of these similarities, we will only discuss the specifics of the equation for describing the change in n over time here.

$$\dot{n} = \alpha(V)(1-n) - \beta(V)n$$

By citing the Hodgkin Huxley model, we find that the α and β terms are defined as functions of voltage.

$$\alpha(V) = 0.01\frac{10-V}{e^{10-\frac{V}{10}}}$$
$$\beta(V) = 0.125\mathrm{n}e^{\frac{-V}{80}}$$

Because n is a dimensionless value, the unit of $\dot{n}$ should be second^{-1}. The expression that defines $\dot{n}$, however, is not something that has a well defined unit. Because of this, we should be able to coerce the dimensionality of the expression to be that of second^{-1}. To facilitate this, we must introduce a new operation that takes a value, either a `Quantity` or a `double`, and returns the same numeric value but stored in a new `Quantity`. Here, we define the function `CoerceTo` as a function that takes an expression of a generic type and returns a value of a different type.

```
template <typename T, typename E>
constexpr T CoerceTo(E expr);
```

Then, we specialize this operation only for cases where the argument to the function is a `Quantity` object or a `double` value.

```
template <typename T,
          int m, int kg, int s, int K,
          int A, int mol, int cd>
constexpr T
CoerceTo
(Quantity<m, kg, s, K, A, mol, cd> expr)
{
  return T{expr.value};
}

template <typename T>
constexpr T CoerceTo(double value)
{
  return T{value};
}
```

E. Interfacing With Other Libraries

In the Hodgkin Huxley model, the exponential function is used in the definitions of $\dot{n}$, $\dot{m}$, and $\dot{h}$. We would like to be able to extend the exponential function found in other libraries so that it can operate naturally on dimensionless `Quantity` objects. We do this by first defining a short hand notation for the type of the dimensionless `Quantity`.

```
using dimless =
  Quantity<0, 0, 0, 0, 0, 0, 0>;
```

Then, using the same strategy that we used for defining exponentiation on `Quantity` objects, we define a new class called `CreateNumericFunction` that takes a type that describes how to instantiate function objects that represent the operation that we would like to extend to understand how to operate on dimensionless `Quantity` objects.

```
template <typename M>
class CreateNumericFunction {
private:
  M function;
public:
  dimless operator() (dimless arg)
  {
    return dimless{function(arg.value)};
  }
};
```

Then, for our case, we must create a class that defines how to instantiate function objects that wrap the `exp` function found in the C++ standard library.

```
class Exp {
public:
  double operator()(double exponent)
  {
    /* exp from the math.h header */
    return exp(exponent);
  }
};
```

Finally, we may use these abstractions to instantiate a function object that mimics the behavior of the exponential function from the standard library, but differs in that it operates on dimensionless `Quantity` objects.

```
CreateNumericFunction<Exp> expd;
```

F. Composing Units

The current method for defining a new type is to construct a tuple of seven integers and pass that tuple to the `Quantity` class for template instantiation. This works very well for common types that have been named, such as the Newton.

```
using Newton =
  Quantity<1, 1, -2, 0, 0, 0, 0>;
```

For implementing types that do not have a shorthand name, it may be more expressive to be able to describe the type in terms of the arithmetic operations that are used to compose it. As an example, the unit Volt $\cdot$ second^{-1} does not have a common name, and so it may be easier to refer to this unit as the quotient unit of

Volt and second. To this end, we define metafunctions that accept `Quantity` types and return a type that represents the indicated unit. We define three metafunctions below for covering the possible use cases. We use the `decltype` keyword to obtain the type of the requested arithmetic expression. Because of our previous operator overloading, these definitions are quite succinct.

```
template <typename T, typename R>
struct QuotientUnit {
  T op1;
  R op2;
  using type = decltype(op1 / op2);
};

template <typename T, typename R>
struct ProductUnit {
  T op1;
  R op2;
  using type = decltype(op1 * op2);
};

template <typename T>
struct Inverse {
  T op;
  using type = decltype(1.0 / op);
};
```

As a concrete example of how these metafunctions could be used, we define the Newton type by composing units instead of specifying the tuple argument manually.

```
using second =
  Quotient<0, 0, 1, 0, 0, 0, 0>;
using meter =
  Quotient<1, 0, 0, 0, 0, 0, 0>;
using kilogram =
  Quotient<0, 1, 0, 0, 0, 0, 0>;

using Newton = QuotientUnit<
  ProductUnit<meter, kilogram>::type,
  ProductUnit<second, second>::type
>::type;
```

IV. EXAMPLE OF USAGE

As an example of these abstractions in use, we may now define the rate of change in voltage and the rate of change of the value of n in terms of `Quantity` objects that demonstrate the physical significance of the values used in the definitions from the Hodgkin Huxley model. Because of the way that types are enforced and checked in C++, unit cancellation will be enforced. If the types of these expressions do not simplify to a type that represents Volt $\cdot$ second^{-1} and second^{-1} respectively, then the program will not be able to be created due to compile time errors during the type checking phase of the compilation process.

```
QuotientUnit<Volt, second>::type
dV_wrt_dt(Volt V,
          dimless n,
          dimless m,
          dimless h)
{
  return (I - g_K*pow4(n)*(V - E_K)
            - g_Na*pow3(m)*h*(V - E_Na)
            - g_L*(V - E_L))
          / C;
}

InverseUnit<second>::type
alpha_n(Volt V)
{
  return CoerceTo<
    InverseUnit<second>::type>(
      0.01*(10.0 - V)/
      (expd(CoerceTo<dimless>(
         (10.0-V)/10.0)
      ) - 1.0)
    );
}

InverseUnit<second>::type
beta_n(Volt V)
{
  return CoerceTo<
    InverseUnit<second>::type>(
      0.125 *
      expd(CoerceTo<dimless>(-V/80.0))
    );
}

InverseUnit<second>::type
dn_wrt_dt(Volt V,
          dimless n,
          dimless m,
          dimless h)
{
  return alpha_n(V) *
         (1.0 - n) - beta_n(V)*n;
}
```

There are two operations above that have not been fully defined thus far. We will explain what these two operations are and define the operations here. The first operation that we will discuss is the least controversial of the two, which is scalar multiplication being denoted as the multiplication of a `double` value and a `Quantity` object. We can extend multiplication such that these evocations are well defined.

```
template <int m, int kg, int s, int K,
          int A, int mol, int cd>
Quantity<m, kg, s, K, A, mol, cd>
operator*(double op1,
          Quantity<R, m, kg, s, K,
                   A, mol, cd> op2)
```

```
{
  return Quantity<R, m, kg, s,
                  K, A, mol, cd> {
    op1 * op2.value
  };
}
```

Unfortunately, there is no easy way to express in C++ that this operation is commutative, so we would have to define a new definition to allow for multiplication expressed in the order of having the `Quantity` object first and the `double` value second. We omit this definition for brevity.

The second operation that was shown above but not defined is much more controversial than scalar multiplication. For ergonomics, our implementation allows for addition between `double` values and `Quantity` objects. This is not so shocking when one adopts the philosophy that a `double` and a dimensionless `Quantity` are separate concepts and should not be conflated. This helps the ergonomics of expressing dimensionally invalid but semantically valid arithmetic expressions when needed, such as in the definition of `alpha_n`.

V. CONCLUSIONS

In an attempt to measure the effectiveness of this technique, we will count the number of possible implementations that can be produced by changing the operands to other variables that exist in the context of execution. We will do this by using the formula

$$N_{\text{impl.}} = \prod_{operands} n_{operand}$$

where N is the number of possible implementations that are deemed valid by the type checker, and $n_{operand}$ is the number of variables that have the type of the current operand. Although the strategy for creating different implementations to count is fairly arbitrary, we find that it expresses the narrowing of possible erroneous implementations nicely. We will compare the implementation of the rate of change of voltage over time in the two different implementation styles; the first using `double` values, and the second using `Quantity` objects.

```
double const I    =   10.0;
double const C    =   1.0;
double const g_Na =   120.0;
double const g_K  =   36.0;
double const g_L  =   0.3;
double const E_Na =   120.0;
double const E_K  =  -12.0;
double const E_L  =   10.6;

double
dV_wrt_dt(double V,
          double n,
          double m,
          double h)
{
  return (I - g_K*pow(n, 4)*(V - E_K)
    - g_Na*pow(m, 3)*h*(V - E_Na)
    - g_L*(V - E_L))
    / C;
}
```

The number of possible implementations that could be generated by changing the operands to other values that exist in the context is computed in the following way.

$$\begin{aligned} N_{\text{double impl.}} &= \prod_{operands} n_{operand} \\ &= 12^{14} \\ &= 1283918464548864 \end{aligned}$$

Now, by writing the same logic using `Quantity` objects in a context where the constants are defined with their unit information, we may produce a function like the definition here.

```
Ampere const I     =  10.0;
Farad const C      =  1.0;
Siemens const g_Na =  120.0;
Siemens const g_K  =  36.0;
Siemens const g_L  =  0.3;
Volt const E_Na    =  120.0;
Volt const E_K     = -12.0;
Volt const E_L     =  10.6;

QuotientType<Volt, second>::type
dV_wrt_dt(Volt V,
          dimless n,
          dimless m,
          dimless h)
{
  return (I - g_K*pow4(n)*(V - E_K)
            - g_Na*pow3(m)*h*(V - E_Na)
            - g_L*(V - E_L))
            / C;
}
```

In this implementation, the number of possible implementations that could be generated by changing the operands to other values that exist in the context is found to be substantially smaller than the previous count.

$$\begin{aligned} N_{\text{Quantity impl.}} &= \prod_{operands} n_{operand} \\ &= 1^1 \cdot 3^6 \cdot 4^6 \cdot 1^1 \\ &= 2985984 \end{aligned}$$

The difference between $N_{\text{double impl.}}$ and $N_{\text{Quantity impl.}}$ is found to be 1283918461562880, a remarkably large

number. We may interpret this number to be the amount of erroneous implementations that are possible in the first implementation using `double` values but are impossible in the second implementation using `Quantity` objects. Many of the implementations counted are trivially incorrect, such as the return statement

```
return (I - I*pow4(I)*(I - I)
           - I*pow3(I)*I*(I - I)
           - I*(I - I))
           / I;
```

and so the large size of the number should not be considered an indication of direct utility to the programmer. Instead, the difference should be interpreted as the successful narrowing of the amount of erroneous but possible implementations. The utility comes from the realization that many erroneous typographic errors in the source code of the simulation fall into the set of implementations described by the difference between $N_{\text{double impl.}}$ and $N_{\text{Quantity impl.}}$. In these cases, the use of `Quantity` objects is invaluable for quickly and decisively finding a certain class of errors and potentially preventing the accidental use of a erroneous implementation for generating scientific data.