Proposal for COAL

(COmplex Arithmetic Language)

February 8, 2009
Eliot Scull (CVN)
e.scull@computer.org

1 Why Use COAL?

High-level languages like C or Java are typically expressive enough to allow mathematical computations and algorithms be clearly and intuitively coded. However, when it comes to computations involving complex numbers, this is not always the case, and relatively simple computations can turn out to look rather arcane when coded. This can become an issue when trying to bring up new code or maintain already released production code that uses complex math.

There are several well established ways for expressing complex math more naturally in code. Languages like C++ provide operator overloading and rich support for custom types – but they are not native to C++ and have to be defined. Environments like Matlab or Scilab could also be used but have overhead associated with them which may not be appropriate for some applications, like embedded systems.

A special purpose language, COAL (COmplex Arithmetic Language), is proposed to supplement high-level languages when doing computations in the complex plane. Math and algorithms involving the complex plane can be expressed in COAL and called by code written in a mainstream high level language.

2 How It Works

COAL could be made to work with any high level language. However, for this scope of this project, COAL source code (see next section for examples), will be compiled to intermediate C code. This C code will then be compiled to native .o's which will then be callable from any language that can link to C binaries.

Since COAL will be used to supplement mainstream languages, COAL will not have extensive support for I/O. However a method for outputting debug messages will be available (see below).

3 Language Description

The following is a high level description of COAL's features.

3.1 Types

A variable of any of the three types below can be declared. Variables are declared implicitly and their types inferred by context.

`Number` – This type is used to represent numbers in the complex plane.

```
     x = 3.1 + 4.9i;  y = 17;  z = 9.12i;
```

`...Number` – This is a sequence of numbers. Sequences of numbers are collections of `Number'`s that can be iterated over. Instances of this type will be referenced counted so that they can be created in one scope and passed to another without copying. Sequence size can be set ahead of time or as the sequence is initialized (less efficient). Sequences will be mutable unless passed in externally from C.

```
     q = 1..300 step .5;  # 600 elements 1, 1.5, ...

     p = ...256; # 256 elements init'd to 0

     r[2] = 3.1 - 8.1i;  # one element (so far)
```

`Function` – Functions can be named or unnamed and are provided for functional decomposition of code. The last expression of a function is evaluated and returned to the caller without the use of a "return" statement.

```
foo (x, y)
{
    x + y(3);
}

# bar returns the value 3 + (3 * 8) = 27
bar()
{
    foo (3, (z => z * 8));
}
```

3.2 Flow Control

`if (expr)… else …` – General purpose conditional statement.

```
if ( a>b ) { a; } else { b; }
```

`while (expr)…` – General purpose iteration.

```
i = 0;
while (i < 10)
{
     i = i + 1;
}
```

`for iter in ...Number` – Iteration over sequences.

```
x = ...256;
for n in 0..(x.N-1) # iterate over temp sequence to index into x
{
    x[n] = n * 2.0;
}

sum = 0.0;
for j in x # iterate over x directly
{
    sum += j;
}
```

3.3 Sample Code

```
main()
{
   # generate signal - could also be measured data
   input = ...1000;
   for n in input.N-1..0  # iterate either direction
       input[n] = sin(100.0*2.0*PI*n/input.N)
                  + .2 * cos(200.0*2.0*PI*n/input.N);

   # calculate 2nd harmonic distortion
   fund = DFT_one_bin(100, input);
   sec = DFT_one_bin(200, input);

   distortion = mag(sec)/mag(fund);

   # Output debug message
   ""2nd harmonic distortion is $(distortion)\n"";

   if (distortion > .3) -1 else 0;
}
```

(Continued)

```
mag(c)
{
   sqrt(c.re^2 + c.im^2);
}

sum(range, what_to_sum)
{
   res = zero;
   for n in range
   { res = res + what_to_sum(n); }
   res;
}

DFT_one_bin(k, x)
{
   sum(
         0..x.N-1
       , n => x[n] * exp( (-2.0*PI*i / x.N) * k*n )
   );
}
```