COAL (COmplex Arithmetic Language) Reference Manual

March 09, 2009
COMS W4115
Eliot Scull (CVN)
e.scull@computer.org

1 Introduction

This manual describes the COAL language (COmplex Arithmetic Language) which is a "helper" language used to express complex arithmetic or algorithms expressively while allowing efficient interoperation with C. COAL is compiled to C which can then be compiled to native with any ANSI C compiler and linked to a C program.

COAL's motivation stems from the fact that expressing complex math in plain C code can be hard to maintain, debug, or understand. While moving to C++ or using Matlab can bring expressiveness to code using complex math, either of these options have high overhead in complexity and/or monetary cost. COAL is intended to be a lightweight computational language suitable for typical embedded systems or larger.

COAL has attributes of a functional language. COAL allows the definition of functions containing expressions that return the evaluation of expressions. Iteration is achieved through recursion and built-in map/reduce operators.

2 Syntax Notation

The context-free grammar notation used throughout this document is written in *italics* and is loosely based on the format used in the C Reference Manual in the appendix of the "C Programming Language", by Kernighan and Ritchie.

Throughout the sections in this document non-terminals of this grammar notation are suffixed with a number (i.e. *expresion1*, *expresion2*,…) This number has no significance for the grammar and is only used to describe the subordinate parts of a production.

3 Lexical Conventions

A program is defined across modules, where a single COAL file (.coal) corresponds to a module. There is only one pass of the compiler from a .coal file to .c and .h files with the same root name. For example `foo.coal` is compiled to `foo_coal.c` and `foo_coal.h`.

3.1 Tokens

User defined identifiers, keywords, numerical literals, comments, and operators comprise COAL's tokens. These are discussed below, except for operators which are described in section 5.

3.2 Comments

One line comments are supported by the pre-pending of # to characters that are to be ignored by the compiler:

```
This is not a comment # This is a comment
```

Characters to the left of the `#` are not ignored by the compiler. Characters to the right and including # are stripped off. If multi-line comments are desired, then the next line must contain a `#`.

3.3 Identifiers

Identifiers are a sequence of letters, digits, and "tick" marks ('). An identifier must start with a letter (upper or lower case). "tick" marks may only be used at the end of the identifier, but any number of "tick" marks may be used. An identifier can be of arbitrary length and is case sensitive:

```
     f (x) -> x^2 !
     f′(x) -> 2*x !
```

Identifiers are denoted with *identifier* in the grammar.

3.4 Keywords

There are four keywords: `if`, `then`, `else`, and `i`. The use of these keywords is described below.

3.5 Numerical Literals

All numerical literals are considered to be in the complex plane, where a number has a real and imaginary part. The following exemplify numerical constants:

```
97
97.0
1.0e2 - 2i
7e7i
```

The keyword `i` is used to signify an imaginary part of a complex number. There may only be one `i` in the imaginary term, and it must come at the end of the term. With the difference of the `i`, real and imaginary terms are lexically identical. `i` may not appear by itself which avoids possible namespace identifier clashes.

Real or imaginary literals before the `i` may be expressed using C's lexical convention for integers or floating point numbers. Although integers and floating point literals are considered to be different types in C, in COAL they are the same type, where 97 is equal to 97.0 which is equal to 9.7e1.

Numerical literals map to the `Number` type.

Real and imaginary literals are denoted by *real-literal* and *imaginary-literal* in the grammar.

4 Meaning of Identifiers

Identifiers refer to variables of numbers, arrays of numbers, or functions. The types of variables and parameters of functions used in a program do not need to be explicitly defined. Types are inferred from usage.

4.1 Variables of **`Number`**

Identifiers can be defined as variables of the `Number` type, which represent the real and imaginary parts of a complex number. The real and imaginary parts of `Number` each have the same numerical range as an IEEE-754 double floating point number.

Variables of `Number` are always passed by value. They have only automatic scope and so are either defined in a function or passed to or from a function.

```
K <- (3.4+9.1i) / 7.8i # K is defined and assigned
```

4.2 Arrays of **`Number`**

Variables may also represent arrays of `Number`. Arrays are referred to by handles and as such are passed around from or to functions by reference. Array references do not go out of scope when returned from a function and are garbage collected even though the identifier referring to the array will. Arrays may be passed into a function or created inside the scope of a function.

Arrays passed in from external C code calling COAL functions are immutable. Arrays created directly by the range operator within COAL code are immutable. Arrays created with the map operator within COAL are mutable.

```
 f(x) -> x[9] * 8!      # x is immutable when called from C
 x <- 1..100\2          # x is immutable 2,4,6…100
 x <- (n->n){1..100\2} # x is mutable 2,4,6…100
```

See range operator below (Section 5.2.3).

4.3 Functions

There are two kinds of functions, named and lambda (unnamed). Functions must always have a return value.

Named functions are always defined at module scope and cannot be defined within another function. A function can contain a single expression, returned after evaluation, or a semicolon separated series of expressions the last of which is returned after evaluation. Named functions are denoted in the grammar as such:

function-definition:
 identifier `(` *argument-list-opt* `)` `->` *expression* `!`

A named function may call itself recursively. Mutual recursion of two or more functions is not supported as COAL depends on a simple forward declaration model.

Named functions are defined in a global namespace and must be unique across all compiled COAL modules.

Examples:

```
constant(x) -> 3 !

sum(x,y) -> x + y !
```

```
stuff(x,n) ->
    a <- x[n]
  , b <- x[n-1]
  , a^b !
```

Only named functions are accessible from C (see section 6 on binding to C).

Lambdas (or unnamed functions) can only be defined within other functions and acquire the scope of the function in which they are defined. Lambdas must have at least one argument and must be invoked where they are defined using the function invocation operator (see 5.6). Lambdas are denoted in the grammar as such:

lambda-definition:
`(` *argument-list* `->` *expression* `! )`

Examples:

```
named(j,k) -> j + (n->n*k!)(j) !  # j + j*k

something(x) ->  (n->n*(h->h*2!)(n)!)(x) !   #  x*x*2
```

Identifiers for `Number` or array of `Number` may only defined within the scope of a named function or lambda and must not clash with each other, if different types, or with named function identifiers. Declarations for `Number` or array of `Number` identifiers are made either in the parameter list of a function definition or in an assignment expression. The following illustrates this scoping:

```
             # x    j    k  defined?
g(x) ->      # yes  no   no
  j <- x*x;  # yes  no   no
  k <- j*j;  # yes  yes  no
  k!         # yes  yes  yes
```

5 Expressions

There are no procedural statements in COAL but only expressions involving the types `Number` or array of `Number`. Every function must return a result which is the evaluation of an expression, all the way to the point at which a COAL function is invoked from C.

The type of an expression, that is whether it is `Number` or array of `Number`, is determined by type inference.

5.1 **`Number`** Expressions

Primary `Number` expressions consist of either a numerical literal (3.5), a variable of `Number` (4.1), or a grouping of a `Number` expression (5.3):

> *expression:*
> > *identifier*
> > *real-literal*
> > *imaginary-literal*
> > `(` *expression* `)`

Composite `Number` expressions can be formed by use of the following operators which take as operands primary `Number` expressions or other composite `Number` expressions.

5.1.1 Math Operators

The following table lists the math operators that work on `Number`. Operators are grouped at the same level of precedence. The bottom of the table has the highest level of precedence. All operators take and return the `Number` type.

Operators	Description	Associativity
`+ -`	Add and subtract	Left
`* /`	Multiple and divide	Left
`-`	Unary minus	Right
`^`	Exponentiation	Left

Math operators are denoted as follows in the grammar:

> *expression:*
> > *expression* + *expression*
> > *expression* − *expression*
> > *expression* * *expression*
> > *expression* / *expression*
> > *expression* ^ *expression*
> > − *expression*

5.1.2 Relational Operators

The following table lists the relational operators that work on `Number`. Operators are grouped at the same level of precedence. All relational operators have lower precedence than the math operators. The bottom of the table has the highest level of precedence. All operators take and return the `Number` type.

Operators	Description	Associativity
= <>	Equal and not equal	Left
< <= > >=	Less than (or equal) Greater than (or equal)	Left

Because there is no Boolean type, these operators return the value 1.0 for true and 0.0 for false. Because "greater than" and "less than" are not defined for complex numbers, the imaginary parts of operands for "less than (or equal)" and "greater than (or equal)" are ignored when these operators are used.

The "equal" operator is provided to be complete but it is not recommended to test the equality of two floating point numbers complex numbers. In addition to the "equal" operator, a built in function, `distance`, will be provided to test the proximity of two complex numbers (See 7).

Relational operators are denoted as follows in the grammar:

```
expression:
        expression < expression
        expression <= expression
        expression > expression
        expression >= expression
        expression = expression
        expression <> expression
```

5.1.3 Logical Operators

No specific logical operators are defined in COAL. However, logical operators can be substituted with math operators to get the same effect:

COAL	Meaning
`(a>b) * (d<e)`	(a>b) AND (d<e)
`(x=0) + (y=0)`	(x=0) OR (y=0)

For logical "not" functionality, a built-in function, `not`, is provided to invert logic.

5.1.4 `if then else`

Conditional expressions are formed with this syntax:

> *expression:*
>   `if` *expression1* `then` *expression2* `else` *expression3*

The imaginary part of *expression1* is ignored. If the absolute value of the real part of *expression1* is greater than or equal to .5, then *expression2* will be evaluated; if it's less than .5, then *expression3* will be evaluated.

The `else` keyword is not optional.

Conditional expressions may be nested:

```
if (a>b) then if (a>c) then 2i else -2i else (2 + 2i)
```

5.2 Array of `Number` Expressions

COAL supports arrays of `Number`. Like the type `Number`, expressions can be made of arrays or `Number` but over a different set of operators.

The primary array of `Number` expressions consist of reference to an array of `Number` (4.2), or a grouping of an array of `Number` expression (5.3):

> *expression:*
>   *identifier*
>   `(`*expression* `)`

5.2.1 Map Operator

A built-in operator is provided to transform one array to another array. This is the primary mechanism by which new arrays can be created in COAL. It has the following syntactical form:

> *expression:*
> > *identifier* { *expression* }
> > *lambda* { *expression* }

identifier refers to a named function definition and *lambda* an unnamed function definition. *expression* is an array of `Number` which is the array to be "mapped". The function defined for *identifier* or *lambda* must take exactly one argument which gets passed to it consecutively all of the elements of the array (*expression*) passed into the operator. A new array is returned that gets formed like this:

```
function(array[0]), function (array[2]), function (array [3]), …
```

Some examples:

```
(n->n!){10…100\10} # 10, 20, 30…
(x->x*x!){-10..-1} # 100, 81, …
```

5.2.2 Reduce Operator

A built-in operator is provided to transform an array into an expression, which could be a `Number` or another array of `Number`. It has the following form:

> *expression:*
> > *identifier* { *expression1, expression2* }
> > *lambda* { *expression1, expression2* }

identifier refers to a named function definition and *lambda* an unnamed function definition. *expression1* is an array of `Number` which is the array to be "reduced". *expression2* is the initial value used for the reduce operation. The function defined for *identifier* or *lambda* must take exactly two arguments which get passed to them consecutively all of the elements of the array (*expression1*) and the accumulated result that started with the initial value (*expression2*). A new array is returned that gets formed like this:

```
function(array[1], function (array[0], init_value)) …
```

Some examples:

```
    (a, sum-> sum + a!){1..10,0} # sum numbers 1 to 10

    # triple elements in array, in place
    (n, arr-> arr[n]<-arr[n]*3, arr!)(0..N(x), x)
```

5.2.3 Range Operator

The range operator generates immutable arrays containing real whole numbers. It has the syntactic form:

expression:
  expression1 `..` *expression2*
  expression1 `..` *expression2* `\` *expression3*

expression1, *expression2*, and *expression3* are of `Number` type. Operands for this operator are automatically rounded to the nearest integer, and imaginary parts set to 0. If literals are used for these operands, the compiler enforces that only integers (in the C style) are used.

expression1 is the starting value in the array and *expression2* the upper limit value of the array. *expression3* may optionally specify step increment. The default step value is 1.

The number of values, N, generated from this operator is:

N = (expression2 – expression1)/expression3 `- 1`

The returned values from this operator are:

expresion1 + 0**expression3*, *expresion1* + 1**expression3*, … *expresion1* + N**expression3*

If *expression1* > *expression2,* then *expression3* must be specified and be negative or an empty array is returned. If expression1 is equal to expression2, an empty array is returned.

The range operator acts as a seed into the map and reduce operators to index and create new arrays.

5.2.4 Array Indexer

> *expression:*
>
> > *expression1* `[` *expression2* `]`

The array index operator is used to access specific elements of an array. *expression1* is of type array of `Number`. *expression2* is of type `Number`. Similarly to the range operator, *expression2's* real part is rounded to nearest integer and imaginary part set to 0. Also, only integer literals may be used for *expression2* .

This operator returns a `Number` type. It is possible to assign a value to an array element using this operator if the array is mutable (5.4).

5.3 Grouping

The grouping operator is used, as in C, to force precedence for a sub-expression where the precedence would otherwise cause an expression to be evaluated differently.

It has the form:

> *expression:*
>
> > `(` *expression* `)`

expression inside the parentheses can be of type `Number` or array of `Number`.

5.4 Assignment Operator

The assignment operator is used to bind a `Number` or array of `Number` expression to an identifier:

It has the form:

> *expression:*
>
> > *identifier* `<-` *expression*

expression is either a *`Number`* or an array of `Number`. Within the resulting expression, *identifier* has no scope but rather must be used in conjunction with the sequence operator and referenced from a proceeding expression (5.5).

This result of this operation is *expression*, with a side-effect of defining *identifier* to be bound to *expression*. This operator is right associative.

5.5 Sequence Operator

The sequence operator is used to allow intermediate expressions, specifically assignments (5.4), to be evaluated before the last expression in the sequence which can incorporate the results of these intermediate expressions.

This operator has the form:

> *expression:*
>
> *expression1* ; *expression2*

expression1 would typically be an assignment and *expression2* is the resulting expression. Because of this behavior, any other sort of expression besides assignment doesn't usually make sense for *expression1*. However other sorts of expressions are allowed for the sake of debugging (8).

expression1 and *expression2* can be of different types.

Examples:

```
a<-3; b<-a; b^2      # 9
     |← a's  scope is live from here

x<-19; x<-x*2; x+1   # 35
      |←  first x live from here
              |←  second x live from here

101; 97; 17.2        # 17.2
```

Identifiers can only be redefined using the same expression type. For example, a `Number` cannot then be overridden to be an array of `Number` using the sequence operator:

```
x<-1..10; x<-x[3]; x # ERROR! 2nd x is different type.
```

5.6 Function Invocation

Invoked named functions result in an expression defined as follows:

> *expression:*
>   *identifier* (*invoke-argument-list-opt*)

Invoked lambda's result in an expression defined as follows:

> *expression:*
>   *lambda-definition* (*invoke-argument-list*)

Examples:

```
f(x) + g(x+y)

stuff(x, x^2, x^3) + 23

(a, b-> if (a>b) then a*b else a/b !)(4,5)

beg()..end()\step()

#recursion
fact(n) -> if (n=1) then 1 else (n * fact(n-1)) !
```

6 Binding to C

As described above, the COAL compiler will produce output consisting of .c and .h files which can then be integrated into a target C application. In the generated .h files will be C bound declarations of the COAL defined named functions. There will also be C bound types used to represent the COAL types, `Number` and array of `Number`, for return values and function arguments. For C programs to easily use COAL functions, conversion macros will also be provided to go back and forth between C and COAL types. Below is table summarizing these macros:

COAL C macro	Returns	Description
`TO_COAL_NUM(a, b)`	`CoalNumber`	Real part set to `a`, imaginary to `b`
`TO_DBL_RE(num)`	`C double`	Real part of CoalNumber
`TO_DBL_IM(num)`	`C double`	Imaginary part of CoalNumber
`TO_COAL_ARR_REAL(c_arr, N)`	`CoalArray`	`c_arr` points to a C `double` array containing only real elements. Actual size of c_`arr` is N * `sizeof(double)`.
`TO_COAL_ARR_CPLX(c_arr, N)`	`CoalArray`	`c_arr` points to a C `double` array containing only real and imaginary elements, interleaved (even offsets real). Actual size of c_`arr` is 2 * N * `sizeof(double)`.
`COAL_ARR_IDX(coal_arr, idx)`	`CoalNumber`	Access an element of `CoalArray`
`COAL_ARR_LEN(coal_arr)`	`C size_t`	Number of elements in `CoalArray`

A function called `CoalArrayFree` will also be provided for use by C clients so that storage behind `CoalArray`'s returned from a function can be freed.

Example:

in file square.coal ... `Square(x) -> x*x !`

in file square_coal.h ... `CoalNumber Square(CoalNumber number);`

in file main.c ...

```
include<stdio.h>
include"square_coal.h"

int main()
{
  CoalNumber num = Square(TO_COAL_NUM(2.0, 2.0));
  printf("re: %f, im: %f", TO_DBL_RE(num), TO_DBL_IM(num) );
}
```

7 Built-in Functions and Constants

Below is a list of built in functions to improve the usability of COAL.

Function	Return Type	Description
`N(`array of `Number)`	`Number`	Number of elements in array
`Re(Number)`	`Number`	Zero out imaginary part of argument.
`Im(Number)`	`Number`	Zero out real part of argument.
`distance(Number, Number)`	`Number`	Distance, as a real number, between two complex numbers
`not(Number)`	`Number`	If absolute value of real part of argument less than .5, returns 1; if greater than or equal to .5 returns 0.
`sin(Number)`	`Number`	Sine – argument in radians
`cos(Number)`	`Number`	Cosine – argument in radians
`tan(Number)`	`Number`	Tangent – argument in radians
`atan(Number)`	`Number`	Arctangent
`exp(Number)`	`Number`	Euler number to the power given by argument.

The constant PI will be defined in the global scope to represent 3.1415927…

8 Debug Mode

To minimize COAL's complexity, a formatted output facility will not provided. Instead, a debug mode will be settable on the compiler whereby operations and evaluations of intermediate expressions will be dumped to standard out for debug or test purposes.

9 Grammar

The following is the grammar for COAL. Terminals _identifier, real-literal,_ and _imaginary-literal_ are described above.

The _opt_ suffix on a non-terminal means zero or more of the non-terminal.

program: zero or more
    program function-definition

expression:
    identifier
    real-literal
    imaginary-literal
    expression + _expression_
    expression - _expression_
    expression * _expression_
    expression / _expression_
    expression ^ _expression_
    - _expression_
    expression < _expression_
    expression <= _expression_
    expression > _expression_
    expression >= _expression_
    expression = _expression_
    expression <> _expression_
    identifier <- _expression_
    expression [_expression_]
    identifier (_invoke-argument-list-opt_)
    lambda-definition (_invoke-argument-list_)
    identifier { _expression_ }
    identifier { _expression, expression_ }
    lambda { _expression_ }
    lambda { _expression, expression_ }
    (_expression_)
    expression `..` _expression_
    expression `..` _expression_ \ _expression_
    expression ; _expression_
    `if` _expression_ `then` _expression_ `else` _expression_

function-definition:
    identifier (_argument-list-opt_) -> _expression_ !

lambda-definition:
  `(` *argument-list* `->` *expression* `!` `)`

argument-list:
  identifier
  argument-list, *identifier*

invoke-argument-list:
  expression
  argument-list, *expression*