In this tutorial, we write an Alphabets program, starting from a mathematical equation for LU decomposition. Then we will generate code to execute the alphabets program, and test the generated code for correctness.

The equation for LU Decomposition, derived from first principles using simple algebra in Foundations (pg.3), is as follows: <latex>

```
  $U_{i,j}=\begin{cases}
```

1=i\le j & A_{i,j}
1<i\le j & A_{i,j}-\sum_{k=1}^{i-1}L_{i,k}U_{k,j} \end{cases}

L_{i,j}=\begin{cases} 1=j\le i & \frac{A_{i,j}}{U_{j,j}}
1<i\le j & \frac{1}{U_{j,j}}(A_{i,j}-\sum_{k=1}^{j-1}L_{i,k}U_{k,j}) \end{cases}$ </latex>

[Temp note: in the last case of L, the condition is “1 < j ⇐ i”]

Writing Alphabets

Step 1 : Affine System and Parameters

Let's start from an empty alphabets file, with LUD as the name of the system, and a positive integer N as its parameter. A system (Affine System) takes its name from system of affine recurrence equations, and represents a block of computation. An Alphabets program may contain multiple systems.

Caveat: Remember the phrase, “It's not a bug, it's a feature”? Well, in a tutorial, a feature is called a “learning opportunity.”

Parameters are runtime constants represented with some symbol in the code. In this example, parameter N will be used to define the size of the matrices, which is not known until runtime.

```
affine LUD {N|N>0}
.
```

Step 2 : Variable Declarations

In most cases, a computation uses some inputs and produces outputs. Such variables must be declared with a name, a data type, and a shape/size. In Alphabets, the shape/size is represented with polyhedral domains. For this example, the A matrix is given, and we are computing two triangular matrices L and U. A is an NxN square matrix. The declaration for A looks as follows:

```
float A {i,j|1<=(i,j)<=N}; //starting from 1 to be consistent with the
equation in the notes
```

Similarly, L is a lower triangular matrix of size N (with unit diagonals, implicit) and U is an upper triangular matrix of size N. The declarations should look like the following:

```
//  The convention is that i is the vertical axis going down, and j is the
horizontal axis
float L {i,j|1<i<=N && 1<=j<i};  // Note that the diagonal elements of L are
not explicitly declared
float U {i,j|1<=j<=N && 1<=i<=j};
```

Now these variable declarations need to be placed at appropriate places to specify whether they are input/output/local. `given` is the keyword for input, `returns` is the keyword for output, and `using` is the keyword for local variables.

```
affine LUD {N|N>0}
given
   float A {i,j|1<=(i,j)<=N};
returns
   float L {i,j|1<=j<i<=N};
   float U {i,j|1<=i<=j<=N};
.
```

Domains

Polyhedral domains are represented as { “index names” | “affine constraints using indices and parameters” }, where constraints can be intersected with “&&”. Sometimes constraints can be expressed with short-hand notation like “a<b<c” or “(b,c)<0”.

Unions of such domains can be expressed as “{ a,b | constraints on a and b } || { c,d | constraints on c and d }”. One important point about Alphabets domains is that the names given to indices are only for textual representation. Internally, all analysis/transformation/code generation tools only care about which dimension the constraint applies to. For example, a domain { i,j | 0⇐i<j<N } is equivalent to { x,y | 0⇐x<y<N }, because `i` and x are both names given to the first dimension, and `j` and y are names given to the second dimension.

Step 3 : Equations

Now the only remaining step before a complete Alphabets program is writing the equations. After a little experience, the connection from mathematical equations (of a certain form) to Alphabets equations should become increasingly clear. There are two slightly different syntactic conventions for writing equations, one is called the “Show syntax” and the other is called “AShow syntax”. Show syntax is closer to the internal representation of Alphabets programs, and is more expressive when writing complex programs. AShow syntax uses “array notation” so that it is easier for people used to imperative programs.

We will first write the equation for U in AShow syntax, and then move on to Show as we write the equation for L.

Equation for U (AShow syntax)

In this equation, U is on the left hand side, and the right hand side should define U for each point in

the declared domain of the U variable. In AShow syntax, the names for indices used appear on the LHS of the equations.

For this example, the following LHS for U gives `i, j` as the names for the first and second dimensions to be used when writing the expressions in the RHS. These names do not have to match the names used in variable declaration. You could use `x,y` instead of `i,j` if desired.

```
U[i,j] = RHSexpr;
```

Case Expression

The first thing you notice in the definition of U in the mathematical equation is the branch based on values of `i` and `j`. This branching is expressed with CaseExpression in Alphabets. A CaseExpression starts with the keyword "`case`", ends with keyword "`esac`", and has list of "`;`"-delimited expressions, called "clauses" as its children. Often (but not always), each child of a case is a RestrictExpression (whose syntax is "domain : expr"), which restricts the domain to the specified domain.

Using the above expressions, the branching of the definition of U is as follows :

```
U[i,j] = case
   {|1==i} : expr1;
   {|1<i} : expr2;
esac;
```

Note that because index names are already declared in the context (equation LHS), there is nothing to the left of the `|` in the AShow syntax.

Variable Expression

Moving on to the definitions in each case, the first case is <latex>$A_{i,j}$</latex>. This is written as `A[i,j]` in AShow syntax, similar to accessing an array. A variable without a square bracket, is treated either as a scalar variable (as in, `X[i,j] = 0`) or as an access with the identity dependence function, (i.e., `X[i] = A[i]` would be the same as `X[i] = A`).

Reductions

The last piece missing before completing the definition of U is the summation in the second branch. Mathematically, a reduction is an operation that applies an associative-comutative operator (in general, the operator may only be associative, but In Alphabets, we have only associative-comutative operators) to a set of values, such as summation (sum over a set of numbers).

Reductions are expressed with the following syntax :

```
reduce(operator, projection, expr);
```

operator: operator to be applied (+, *, max, min, and, or)
expr: Any Alphabets expression. "Slices" of the result of evaluating this expression are combined

using the reduction operator.
projection: The projection is an affine function with the same syntax as a dependence (as explained below). By its very nature, a reduction **reduces** the number of dimensions of the expression, so the rxpr has more dimensions than the result. Which values contribute to which result is specified by projection, a many-to-one affine function.

In the mathematical equation, summation with one new index k is used. For each value of `k,` the expression `L[i,k]*U[k,j]` is computed and added up to produce the result `U[i,j]`. Thus, the projection function is `(i,j,k → i,j)`. (from the three dimensional space indexed by `i,j,k`, all values computed at `[i,j,k]` are used to compute `U[i,j]` in the two dimensional space indexed by `i,j` – i.e., the k is 'projected out')

When the projection function is canonic (e.g., `(i,j,k→i,j)`), then the projection function can be replaced with a simpler syntax (AShow syntax for reductions) that specifies the names of new dimensions surrounded by square brackets. For example, the projection `(i,j,x,y→i,j)` can be expressed as `[x,y]`.

Using the above, summation in the original equation can be written as the following Alphabets fragment.

```
reduce(+, [k], L[i,k]*U[k,j]);
```

Putting all this together, the final equation for U is:

```
U[i,j] = case
   {|1==i} : A[i,j];
   {|1<i} : A[i,j] - reduce(+, [k], L[i,k]*U[k,j]);
esac;
```

This is exactly like the original equation Caveat

Equation for L (Show syntax)

Now we will write the equation for L, but this time in Show syntax. Unlike the AShow syntax, Show syntax does not rely on the context for naming of indices. Index names can be different in every (sub)expression if it makes sense to do so. Because of this, the LHS does not have square brackets, all we need is the variable name.

```
L = RHSexpr; //Show syntax
```

Case Expression

CaseExpression and RestrictExpression are same as AShow syntax. However, since index names are no longer deduced from the context where they occur, they must be explicitly named everywhere. While this may seem cumbersome, it allows expressions to have compositional semantics. In our

example, the index names used in the domain of RestrictExpression have to be made explicit. The branch in the definition of L becomes the following Alphabets :

```
L = case
   {i,j|1==j} : expr1;
   {i,j|1<i} : expr2;
esac;
```

Dependence Expression

In the array notation in AShow syntax, a DependenceExpression was implicit: just add expressions within square brackets to access variables). In the Show syntax `DependenceExpression` is used to explicitly specify which value of a variable is required for a computation. The syntax of `DependenceExpression` is "(affine_function)@expr", where `affine_function` is of the form `(list_of_indices → list_of_affine_expressions)`. For example, the dependence `(i,j→i-1,i+j)@A` means that at index point `(i,j)` this computation evaluates to the value of A at index point `(i-1,i+j)`.

The child of `DependenceExpression` can be any Alphabets expression, possibly another DependenceExpression. For example, `(i,j→i,j,i+j,0)@(a,b,c,d→a,c-a)@A` is a perfectly legal Alphabets expression.

Reductions

Reductions in Show syntax are exactly like in the AShow syntax, except that the projection function is specified in the dependence syntax. This is all you need in order to write the rest in of the equation in Show syntax.

```
L = case
   {i,j|1==j} : (A / (i,j->j,j)@U);
   {i,j|1<i} : (A - reduce(+, (i,j,k->i,j),
(i,j,k->i,k)@L*(i,j,k->k,j)@U))/(i,j->i,i)@U;
esac;
```

Final Alphabets Program

Combine all of the above, and you will get the Alphabets program for LU decomposition. Don't forget the keyword `through` before equations the period at the end (since our example has no local variables). Notice how we can mix and match Show and AShow syntax within the program, but each equation must obviously, be consistent.

```
affine LUD {N|N>0}
given
   float A {i,j|1<=(i,j)<=N};
```

```
returns
   float L {i,j|1<i<=N && 1<=j<i};
   float U {i,j|1<=j<=N && 1<=i<=j};
through
   U[i,j] = case
      {|1==i} : A[i,j];
      {|1<i} : A[i,j] - reduce(+, [k], L[i,k]*U[k,j]);
   esac;
   L = case
      {i,j|1==j} : A / (i,j->j,j)@U;
      {i,j|1<j} : (A - reduce(+, (i,j,k->i,j),
(i,j,k->i,k)@L*(i,j,k->k,j)@U))/(i,j->j,j)@U;
   esac;
.
```

Generating and Testing Alphabets

Analyses, transformations, and code generation of Alphabets programs are performed using the AlphaZ system. The normal interface for using AlphaZ is the scripting interface called compiler scripts. Given below is an example script for that does several things using the LUD program we wrote above.

```
# read program and store the internal representation in variable prog
prog = ReadAlphabets("./LUD.ab");

# store string (corresponding to system name) to variable system
system = "LUD";

# store output directory name to variable outDir
outDir = "./test-out/"+system;

# print out the program using Show syntax
Show(prog);

# print out the program using AShow syntax
AShow(prog);

# prints out the AST of the program (commented out)
#PrintAST(prog);

# generate codes (this is demand-driven, memoized code)
generateWriteC(prog, system, outDir);
generateWrapper(prog, system, outDir);
generateMakefile(prog, system, outDir);
```

Save this script with .cs extension, place the alphabets file in the same directory as the script, and then right click on the editor and select "`Run As → Compiler Script`" to run the script.

If you get some error message, try looking at the first line of the error messages to find out what it is about. Common problems are:

- not having the file in the right place (you will see `FileNotFoundException` in this case)
- system name does not match the any of the system in the program (error message should say system `xxx` does not exist)

Code Generators

In this tutorial, we use two basic code generators, without going into too much detail. The two types of codes generated are `WriteC` and `Wrapper`.
`WriteC` code may not be efficient, but it can be generated without any additional specification beyond the program.
`Wrapper` code is a wrapper around other generated codes that allocates/frees memory for input and output variables, and it also have different options for testing.

Note:Current implementation of the Wrapper prints out the bounding box of the domain of the output variable.

generateMakefile produces a Makefile that should compile the generated codes. You can make with different options.

Compiling and Executing the Generated Code

Congratulations!! You are nearly at the end. Now, you will actually make and execute the code (in a separate terminal window).

make

compiles the code and produces an executable `xxx` (where `xxx` is the system name) that executes the program with default input that is 1 everywhere. Compiling with this option does not test very much, but it will test if it compiles and runs and produces no errors.

make check

Compiles the code and produces an executable `xxx.check` (`xxx` is the system name) that prompts the user for all values of input variables. After executing, it prints out all values of the output variables. This option should be used for testing small to mid-sized input data.

make verify

Compiles the code with another code named `xxx_verify.c` that defines a function `xxx_verify` (`xxx` is the system name). Users can provide different program as `xxx_verify` to compare outputs.

make verify-rand

Same as verify, except the inputs are generated randomly.

OOPS WHAT HAPPENED

You will see that when you execute the code, ***it will produce an error***. You may be able to easily fix the error in your Alpha program and regenerate correctly executing C code, or you may want a bit of help. In either case, we would like to know. Please email Sanjay.Rajopadhye@colostate.edu with the error message that is produced.