Getting started with Alpha and AlphaZ

Alpha is a polyhedral equational language, and AlphaZ is its “IDE:” tools to read, write, edit, and analyze Alpha Programs, and also to compile them to executable code.

Your first Alpha program (not hello world)

In this tutorial, which complements Section 1.1 from Foundations 1, we will walk through a simple Alpha program that solves the equation Lx=b where L is a lower triangular matrix with unit diagonals, b is a known vector and x is the unknown vector, using the forward substitution algorithm. With a little bit of algebraic reasoning, we derived a mathematical identity with x on the left hand side LHS. We now want to write it as an equational program in Alpha. You will see that it is surprisingly simple.

Step 1: Affine System and Parameters

Let's start from an empty alpha file “forward_sub“.ab, in which with “FS” is 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.

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 FS {N|N>0}
. // the system ends with a period, and this is a comment
```

Step 2: Variable Declarations

Alpha variables must be declared with a name, a data type, and a shape/size, which is a polyhedral domain. For this example, x and b are N vectors. And L is a lower triangular matrix with unit diagonals so its ***domain*** is a triangle whose side length is N-1 (the diagonal elements are not explicitly stored). Additionally, you will later use additional variables ‘’bp’’ (for “b-prime”) and ‘’Correct’’ to validate that the computation is correct. We must also specify whether variables are input/output or local, using appropriate keywords. So the declarations are as follows:

```
affine FS {N|N>0}
input // "given" is also a legal keyword for input
   float L {i,j|1<=j<i<=N};
   float b {j|1<=j<=N};
output // or "returns"
   float x {i|1<=i<=N};
// bool C {|}; // empty domain, i.e., a scalar variable
local
// float xp {j|1<=j<=N}; will be needed later
```

```
let
   // equation 1
   // equation 2
   // ...
.
```

The `let` keyword delimits the section where the equations will be specified.

Domains

As we saw above, polyhedral domains are written as { “index names” | “affine constraints using indices and parameters” }. Often we can use short-hand notation like “`a<b<c`” or “`(b,c)<0`”. Constraints can be intersected with “&&”.

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 Alpha domains is that the names of indices only have scope until the end of the declaration. Internally, all analysis/transformation/code generation tools only care about the dimension to which the constraint applies. For example, a domain `{i,j | 0<=i<j<N}` is equivalent to `{x,y | 0<=x<y<N }`.

Step 3: Equations

Now the only remaining step to complete the program is to write the equations, and this follows directly from the mathematical identity in Foundations 1. Alpha has a more elaborate syntax for writing more complex programs, but it is not needed here.

Equation for x

In this equation, `x` is on the left hand side (LHS) and implicitly declares `i` as the index name to be used on the RHS. Its scope is the entire equation. The name does not have to be the same as in the declaration of the variable. You could use `p` instead of `i` if you like.

```
x[i] = // some right hand side expression, RHSexpr;
```

Variable Expression

The simplest RHS is an expression as you would write in any standard language, using operators, and (array) variables and arrays. The array access function must be a linear/affine function of the LHS indices and the program parameters, e.g., `A[i,j]`, or `x[N-i]`, A variable by itself, 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` or even `X = A`).

Reduce Expressions

The definition of x involves a summation. Mathematically, a reduction is an operation that applies an associative operator to a set of values, such as summation (sum over a set of numbers). In Alpha, reduction operators must also be commutative and use the following syntax:

```
reduce(op, [index-list], expr);
```

Here op is the reduction operator (currently +, *, max, min, and, or are the ones supported);
index-list names the new indices used in the reduction. Their scope is just within the reduction expression
and expression is any Alpha expression and may use indices from `outside` the reduction, the new ones, and program parameters.

The mathematical expression \sum_j L_{i,j} x_j in our equation would be written as

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

Restrict Expressions

Notice that the bounds on the summation in Equation 1, from Foundations 1 run from 1 to `i-1`. Restrict expressions can be used to, literally, "restrict" the evaluation of a particular expression to a particular domain. The syntax of a restrict expression is:

```
restrict_domain : expr
```

The summation expression in Equation 2 therefore can be expressed as the following:

```
{| 1<=j<i } : L[i,j]*x[j]
```

Note that there is nothing to the left of the `|` because indices are already declared.

Case Expressions

Notice that in our equation, there is a condition on the `i` to be considered. This is expressed with CaseExpression in Alpha. It starts with the keyword "`case`", ends with keyword "`esac`", enclosing a list of "`;`"-delimited expressions, called "clauses". Often (but not always), each clause is a RestrictExpression. Thus the entire RHS of our equation is:

```
xp[i] = case
   {|i==1} : expr1;
   {|i>1}  : expr2;
esac;
```

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

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

See how close it is to the original equation.

Now, save your completed program as the file forward_sub.ab

Generating and Testing Alpha

Analyses, transformations, and code generation of Alpha 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 forward_sub program we wrote above.

```
# read program and store the internal representation in variable prog
prog = ReadAlphabets("./forward_sub.ab"); //AlphaZ uses an extended language
called Alphabets

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

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

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

# print out the program using the standard/show notation
Show(prog);

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

# generate codes
generateWriteC(prog, system, outDir);
# generateVerificationCode(prog, system, outDir);
generateWrapper(prog, system, outDir);
generateMakefile(prog, system, outDir);
```

Save this script with .cs extension, place the alpha 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 the basic code generator. Two code files are generated with the two commands `WriteC` and `Wrapper`.
`Wrapper` code is a wrapper around other generated codes that allocates/frees memory for input and output variables, and it also has different options for testing.

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 sized input data, “by hand” [Hint: you could start with some small well chosen L, pick a “solution” x, multiply them on paper, and give that as b to the program, expecting it to deliver your chosen x].

How to check more thoroughly

AlphaZ has a few commands that let you do this, but they require some gold standard, so the two options below are not really useful for your first program.

make verify

This only becomes useful if you have a known, correct implementation of the function. For this first tutorial, it does not make sense (it ends up comparing the outputs of two identical functions). Later, when you produce different versions using scheduling directives, it becomes useful to validate the correctness against the original equations, Compiles the code with another code named `xxx_verify.c` that defines another function with the same signature, `xxx_verify` (`xxx` is the system name). Users can provide different programs as `xxx_verify` to compare outputs.

make verify-rand

Like the above case, it only becomes useful if you have a known, correct implementation. For this first tutorial, it does not make sense. Same as verify, except the inputs are generated randomly.

How to really thoroughly check the very first equational program

It has to be done equationally!.

It idea is to write additional (simpler) equational programs that are easier to verify and trust and use them to build up the verification. For example, you could first write an Alpha program to multiply a (triangular or otherwise) matrix with a vector, and verify it. Let's assume that you have done this.

Then you can extend the FS system itself, to write an additional equation to compute the matrix-vector product of L and x, subtract b from it,=, v=giving a vector or error values. Now all we need to do is to check that some given threshold is greater than the absolute values of each all elements of the error (Alpha does not have an operator abs, so you can use a trick; `abs(x)` equals `max(x, -x)` of all x). We can return this in the Boolean variable C defined as follows.

```
error [i] =
    case
        {|i==1}: b[i] - x[i];
        {|i>1} : b[i] - x[i] - reduce(+, [j], L[i,j]*x[j]);
    esac;
C[] = reduce(and, [i], max(error[i], - error[i])<0.000000001);
      //you may also do a max reduction and/or use a different threshold
```

Note how we account for the (unit) diagonal elements of L (which are not explicitly part of the array L.

```
float abs (float);
```

But how to provide sufficiently large number of (random) inputs to the code if we don't use the verify option? The answer is to create a sufficiently large file of random inputs, and then use file redirection in linux to feed this file to `xxx_check` (make sure that the boolean output is either first or last, or even the only one).