Verifier

Bundle Setup

I have prepared a bundle including Eclipse itself:
http://www.cs.colostate.edu/AlphaZ/bundles/eclipse-verifier-bundle-linux64.tar.gz
The bundle is based on Eclipse Classic 3.7.1.

The above bundle contains all necessary plug-ins to use the following project:
http://www.cs.colostate.edu/AlphaZ/bundles/org.polymodel.verifier.IF.ada.tar

This verifier.IF.ada project contains examples to use the verifier. For trying out the verifier, I suggest that you download the bundle, and the above archived project, and then import the tar from `Import→General→Existing Projects Into Workspace`. The examples are from our ompVerify paper (http://www.springerlink.com/content/0gh74j23115861g6/).

http://www.cs.colostate.edu/AlphaZ/bundles/AlphaZ.tar

This is a jar file for the verifier. If run as a stand alone jar file, it verifies the legality of parallelization of some of the example programs. When used as a library, please specify the program representation according to the textual interface specified at the bottom of this page and call VerifierExample.verify

Source Access

The following plug-ins are installed as jar files under dropins directory in the above bundle. Sources to these projects are available in the GeCoS repository. Please follow the instructions in this website (https://gforge.inria.fr/scm/?group_id=510) to check out sources for these plug-ins. All plug-ins are located under trunk/polytools-emf, and are licensed under EPL.

- fr.irisa.cairn.eclipse.tom
- fr.irisa.cairn.jnimap.isl
- fr.irisa.cairn.model.integerlinearalgebra
- fr.irisa.cairn.model.polymodel
- fr.irisa.cairn.model.polymodel.ada
- fr.irisa.cairn.model.polymodel.isl
- fr.irisa.cairn.model.polymodel.prdg
- org.polymodel.verifier

Additional Plug-ins

The following plug-ins are required to use the verifier.

- EMF - Eclipse Modeling Framework SDK (2.7.1)
  - From Indigo update site under Modeling cateogry
- Xtext Antlr Runtime Feature (2.0.0)
  - From itemis update site (http://download.itemis.com/updates/milestones), Xtext Antlr-2.0.0M3 category

The bundle also contains subclipse for SVN access.

Using the Verifier

The provided interface takes the following information for each statement in a polyhedral region to verify:

- Polyhedral domain
- Schedule (as affine functions)
- Write access
- Set of read accesses
- Annotation for each dimension of the schedule specifying the type of the dimension.

The output of the verifier is an instance of VerifierOutput object, containing a flag to tell if the program was valid, and a list of messages from the verifier listing violations found in the program. The messages should contain sufficient information to give feedback to the user.

These input can easily be extracted from affine control loops (example later), and array dataflow analysis is performed using the provided information and then the resulting polyhedral reduced dependence graph (PRDG) is verified.

ISLSet and ISLMap

The verifier uses Integer Set Libarary (http://freshmeat.net/projects/isl/), and accepts textual representations for ISLSets and ISLMaps.

- ISLSets

```
[<parameter indices>]->{ [<indices>] : <list of constraints involving parameters and indices> }
```
where the constraints are delimited by any of one '&', '|', 'and', 'or'.

- ISLMaps

```
[<parameter indices>]->{ [<indices>] -> [<list of expressions involving parameters and indices>] }
```
where the expressions are delimited by ','.

Note that ISL accepts much more general syntax than shown above, since ISLMaps are relations, and not functions, but the above is sufficient to express inputs to the verifier.

Example

The example is for C programs with OpenMP. The verifier is applicable for other programming languages with parallel constructs that can be viewed as `doall` parallelization. In case of X10, if the only statement in the body of a loop is `async`, the loop can be considered as `doall` loop.

```
void matrix_multiply(int** A, int** B, int** C, int N) {
```

```
    int i,j,k;

    #pragma omp parallel for
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = 0; //S0
            for (k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j]; //S1
            }
        }
    }
}
```

Statement Domains

Each statement is surrounded by a set of loops with affine bounds. The polyhedral domain for each statement is simply the intersection of all loop bounds that surrounds a statement. There may be a `if` statement with affine constraints, such constraints from `if` statements are also intersected when computing the statement domain.

For the above matrix multiply, S0 is surrounded by `i,j` loops and thus has a square domain, where S1 has a cubic domain.

- `S0 : [N] -> { [i,j] : 0<=i<N & 0<=j<N }`
- `S1 : [N] -> { [i,j,k]:0<=i<N & 0<=j<N & 0<=k<N }`

Schedule

The input schedule given to the verifier is the sequential schedule, treating `doall` loops as sequential loops. `Doall` parallelism is later expressed as dimension types. The schedule is used to characterize the relative placement of the statements. In addition, we require that all statements are mapped to a common dimensional space via scheduling specification.

Although this is not a requirement, a common convention when applying polyhedral analyses to imperative programs is to use 2d+1 representation. For a loop with maximum depth d, additional d+1 dimensions (with constants) are used to specify textual ordering of the loops and statements.

For the above matrix multiply, the schedules are:

- `S0 : [N] -> { [i,j] -> [0,i,0,j,0,0,0] }`
- `S1 : [N] -> { [i,j,k]->[0,i,0,j,1,k,0] }`

Note that the 5th dimension is 0 for S0 and 1 for S1, and all preceding dimensions are identical. This specifies that S0 is textually before S1 at this dimension. Also note that the last two dimensions of the schedule for S0 are padded with 0s, to match the number of dimensions.

Accesses

Both reads and writes are specified as a pair of variable name and access function. Variable names corresponds to the arrays read/written in the program, and access function corresponds to the indexing into the arrays. The access functions are expressed as affine functions from the corresponding statement domains.

For the above example, `S0` has only one access:

- Write C : `[N] -> { [i,j] -> [i,j] }`

`S1` has total of 4 accesses:

- Write C : `[N] -> { [i,j,k] -> [i,j] }`
- Read C : `[N] -> { [i,j,k] -> [i,j] }`
- Read A : `[N] -> { [i,j,k] -> [i,k] }`
- Read B : `[N] -> { [i,j,k] -> [k,j] }`

Dimension Types

One last piece of information given to the verifier is the annotation on each dimension of the schedule. There are three types of dimensions:

- SEQUENTIAL : This dimension should be interpreted as time steps
- PARALLEL : This dimension should be interpreted as processor dimension
- ORDERING : This dimension only contains constants that expresses statement orderings

The parallel dimensions corresponds to loops that are parallelized by OpenMP `for` directive, or any other parallel construct for expressing `doall` parallelism.

private clause in OpenMP

When variables are declared as private in OpenMP, the variable is private to each iteration of the parallel loop. Therefore, it is expressed by adding additional dimensions to the memory accesses, indexed by the loop iterator of the parallel loop, so that each iteration of the parallel loop access different memory location.

In other words, the following code:

```
#pragma omp parallel for private(c)
for (i=0; i < N i++)
   c += ...
```

is translated as:

```
#pragma omp parallel for
for (i=0; i < N i++)
   c[i] += ...
```

during verification.

Interface

Textual

The following code is a possible interface based on arrays of Strings for testing purposes.

```
private static void matrix_multiply() {

    String[][] statements = new String[][] {
        new String[] {"S0", "[N] -> { [i,j] : 0<=i<N & 0<=j<N }",
                    "[N] -> { [i,j] -> [0,i,0,j,0,0,0] }"},
        new String[] {"S1", "[N] -> { [i,j,k] : 0<=i<N & 0<=j<N & 0<=k<N }",
                    "[N] -> { [i,j,k] -> [0,i,0,j,1,k,0] }"}
    };
    String[][] reads = new String[][] {
        new String[] {"S1", "A", "[N] -> { [i,j,k] -> [i,k] }"},
        new String[] {"S1", "B", "[N] -> { [i,j,k] -> [k,j] }"},
        new String[] {"S1", "C", "[N] -> { [i,j,k] -> [i,j] }"}
    };
    String[][] writes = new String[][] {
        new String[] {"S0", "C", "[N] -> { [i,j] -> [i,j] }"},
        new String[] {"S1", "C", "[N] -> { [i,j,k] -> [i,j] }"}
    };
    String[][] dims = new String[][] {
        //Legal 1D parallelization
        new String[] {"S0", "0,P,0,S,0,S,0"},
        new String[] {"S1", "0,P,0,S,0,S,0"},

        //illegal parallelization of k dimension
        //new String[] {"S0", "0,S,0,S,0,P,0"},
        //new String[] {"S1", "0,S,0,S,0,P,0"}
    };

    VerifierExample.verify(statements, reads, writes, dims);
}
```

Generic Interface

For a more general interface, the user must construct the following data structure, called `ADAInput` that represents the polyhedral region to analyze. In addition a map, Map<CandidateStatement, List<DIM_TYPE» is required for specifying the dimension types for each statement.

`fr.irisa.cairn.model.polymodel.ada.factory.ADAUserFactory` provides methods for constructing `ADAInput`.

```
   ADAInput
      |------variables  : List<Variable>
      |------statements : List<CandidateStatement>
```

```
   CandidateStatement
      |------ID       : String
      |------domain   : PolyhedralDomain
      |------schedule : AffineMapping
      |------write    : WriteAccess
      |------reads    : List<ReadAccess>

   Access
      |-----variable       : Variable
      |-----accessFunction : AffineMapping

   Variable
      |-----name : String

   ReadAccess->Access
   WriteAccess->Access
```

`ADAInput` is the input to Array Dataflow Analysis, which is a separate module, and thus dimension types are not part of this data structure. Once these two data structures are ready, the following two lines will invoke the verifier.

```
VerifierInput input = VerifierInput.build(adaInput, dimTypes);
VerifierOutput output = Verifier.verify(ISLDefaultFactory.INSTANCE,
input.prdg, input.schedules, input.memoryMaps, input.dimTypes);
```