

# MapIR Design

Alan LaMielle

August 4, 2010

## 1 Overview

MapIR stands for Mapping Intermediate Representation. This IR is used in the sparse polyhedral model to represent a computation. The computation in question is first converted to the MapIR, various transformations are applied to the MapIR, and code is generated that implements the transformed computation.

## 2 MapIR Specification

The following information is maintained by the MapIR to represent a given computation:

- symbolic constants
- data arrays
- index arrays
- statements
- access relations for each statement
- data dependences

Each of these entities is associated with a name that is unique for the whole MapIR data structure. For example, if a statement has the name 'S0', no other symbolic constant, data array, etc. can have that same name.

The implementation of the MapIR is based on maps of entity name to the object with that name. The MapIR class itself is a very simple container class:

**MapIR:**

- `set<string> names;` //Contains all used names in the MapIR instance
- `map<string,Symbolic> symbolics`
- `map<string,DataArray> data_arrays`
- `map<string,IndexArray> index_arrays`

- `map<string,ERSpec> er_specs`
- `map<string,Statement> statements`
- `map<string>DataDependence> data_deps`
- `add_{symbolic,data_array,index_array,er_spec,statement,data_dep}`: Each of these methods will accept the appropriate object type corresponding to its name, check that that object's name isn't used in the MapIR already, and add it to the appropriate map field. For entities that refer to others (such as the `data_array` field of `AccessRelation`), the MapIR should validate that the referenced entity already exists in the MapIR.
- `get_{symbolics,data_arrays,index_arrays,er_specs,statements,data_deps}`: Each of these methods will return a collection of the appropriate object type.
- `get_{symbolic,data_array,index_array,er_spec,statement,data_dep}(string name)`: Each of these methods will return the appropriate object associated with the given name.

Each entity that the MapIR contains will have its own class that derives from a base class:

**MapIREntity (abstract class, cannot instantiate):**

- `name`
- `string get_name()`: returns the name of this object

The subclasses of `MapIREntity` all add a certain number of fields. These fields should all be private and have only getter methods for the fields. Once the object is created there is rarely a reason to update the fields (therefore no setters are necessary except in a few circumstances).

**Symbolic: MapIREntity:**

- `lower_bound`: an integer representing the lower bound of the symbolic
- `upper_bound`: an integer representing the upper bound of the symbolic

**DataArray: MapIREntity:**

- `Set bounds`: a `Set` representing the bounds of the data array. This method should check that the bounds contain only a single conjunction as data arrays must be a continuous sequence of elements.

**ERSpec: MapIREntity:**

- `Set input_bounds`: a `Set` representing the input bounds of the ER this `ERSpec` represents
- `Set output_bounds`: a `Set` representing the output bounds of the ER this `ERSpec` represents

- Relation relation: a Relation representing the relation for the ER this ERSpec represents

**IndexArray: ERSpec:**

- No fields in addition to an ERSpec, the relation is never used, so it can be created and initialized as the empty relation  $\{\{\}->\{\}\}$

**Statement: MapIREntity:**

- string text: the statement's text in the program.
- Set iter\_space: the iteration space of the statement
- Relation scatter: the scattering function for the statement, input tuple variables are just the iteration space iterators, output tuple variables are the iterators interleaved with constant values. For example: iter\_space= $\{[i,j]: 0 \leq i,j \leq 10\}$  scatter= $\{[i,j]->[0,i,1,j,0]\}$
- map<string,AccessRelation> access\_relations: mapping of access relation name to AccessRelation object
- get\_access\_relations(): returns a collection of access relations that the statement instance contains
- get\_access\_relation(string ar\_name): returns the access relation of the given name

**AccessRelation: MapIREntity:**

- string data\_array: name of the data array being accessed
- Relation iter\_to\_data: Relation mapping iterators to positions in the data array that are accessed

**DataDepdence: MapIREntity:**

- Relation dep\_rel: a Relation that represents the data dependence relation

### 3 Testing and Development

To test the MapIR data structure, we'll develop tests for each component class (DataArray, Statement, etc.) first. Then test the MapIR data structure itself. The development workflow should be:

- Create stub classes and methods for the Symbolic, DataArray, ERSpec, IndexArray, Statement, AccessRelation, DataDepdence and MapIR classes. Do not implement any actual logic for the methods or constructors. This is so that we can write tests that compile but do not pass.

- Write tests that test the Symbolic, DataArray, ERSpec, IndexArray, Statement, AccessRelation, DataDependence, and MapIR classes.
- Implement the methods that are being tested so that the tests now pass.

As a more specific example, consider the AccessRelation class. This class will have two public methods: `get_data_array()` and `get_iter_to_data()`. Write a test that creates an AccessRelation with a test name, test data\_array name, and a test Relation for the iter\_to\_data field. The test will then ask for the name, data array name, and iter\_to\_data relation and ensure that they are equal to the given information.

Error handling: The test cases should test situations such as adding an access relation to a statement that doesn't exist. For example, calling `mapir.add_access_relation("ar1","S3")` when statement S3 does not exist. We will raise an exception (such as `MapIRException?`) in this situation. The test should verify that an exception is actually raised and fail if this is not the case.