Summary and Descriptions of Operations on Sets and Relations in IEGen

Alan LaMielle
Colorado State University

I. Friday October 23, 2008 Meeting Topics

- J&L progress report
- Pointer update/array alignment
- Codegen progress

II. Thursday October 2, 2008 Meeting Topics

- Time
- Operations writeup
- Big Picture
- Inequality simplification

III. Introduction

IEGEN is a tool for generating irregular computational kernels optimized using the Inspector/Executor strategy. The user provides a description of the computational kernel and the desired compiletime and runtime program transformations. IEGen then produces code that implements the corresponding inspector and executor for this computation.

IV. Description of formula data structure

IEGen is built upon and utilizes the mathematical concept of Presburger formulas. However, we have introduced one unique addition to this well-known concept: the use of uninterpreted function symbols to represent indirect array accesses. Few libraries have been written that work with Presburger formulas in this capacity–with full logic support for manipulating formulas that include uninterpreted functions. The closest library is Omega. However, Omega's support for uninterpreted function symbols is lacking. Some operations support and work with uninterpreted functions, some don't, and some produce incorrect results. Therefore, we were forced to develop our own implementation of Presburger formulas that includes correct manipulation of uninterpreted function symbols.

The following are some of the features included in IEGen related to working with Presburger formulas:

- A data structure for representing arbitrary formulas
- A parser for converting textual string descriptions of a formula to the equivalent in-memory data structure
- String generation support for producing a human-readable version of an in-memory formula
- Various operations for manipulating and combining formulas
- A variety of simplification routines that convert one formula to an equivalent, but simplified, version

We support the following aspects of presburger formulas:

- Sets
- Relations
- AND
- OR
- One implicit level of existential quantifiers
- Equality
- Inequality
- Uninterpreted function symbols
- Symbolic Variables

The data structure for representing formulas consists of the following classes:

- Formula: The base class for both Set and Relation. This contains any common functionality between these two classes.
- Set: A Set represents a set of integer tuples restricted by the given constraints. A Set consists of the disjunction of a collection of PresSets.
- Relation: A Relation abstractly represents some mapping from input tuples to output tuples restricted by the given constraints. A Relation consists of the disjunction of a collection of PresRelations.
- PresForm: The base class for both PresSet and PresRelation. This contains any common functionality between these two classes.
- PresSet: Represents a Presburger set of the form '{ [tuple variables] : constraints }'.
- PresRelation: Represents a Presburger relation of the form '{ [input tuple variables] $\rightarrow$ [output tuple variables] : constraints }'.
- VarTuple: Contains the collection of variables that represent a PresSet's set tuple variables or a PresRelation's input or output tuple variables.
- Conjunction: Contains a collection of Constraints that are, as it is a conjunction, ANDed together.
- Constraint: Represents one of two types of constraints: Equality or Inequality. This contains any common functionality between these two classes.
- Equality: Represents an equality constraint of the form: NormExp $= 0$.
- Inequality: Represents an inequality constraint of the form: NormExp $>= 0$.
- Expression: The base class for all expressions. This contains any common functionality between all expression types.

- VarExp: Represents a variable expression of the form: $c_v * v$ where c_v is an integer constant and v is the variable name.
- FuncExp Represents a function expression of the form: $c_f * f(a_1, a_2, \ldots, a_n)$ where c_f is an integer constant, f is the function name, and the n a_i terms are the arguments to the functions defined as NormExps.
- NormExp: Represents an expression of the form: $c_{v_1} * v_1 + c_{v_2} * v_2 + \ldots + c_{v_n} * v_n + c_{f_1} * f_1 + c_{f_2} * f_2 + \ldots + c_{f_m} * f_m + c = 0$ where the n v_i are variables (VarExps), the c_{v_i} are integer constants, the m f_i are functions (FuncExps), the c_{f_i} are integer constants, and c is an integer constant. This is essentially an affine expression with the addition that functions may also be included in the terms.

V. Description of High Level Operations

Arity querying: We are able to query a Set or Relation for its arity. For Sets, it is simply the set tuple's arity. For Relations, there is an arity method that returns a 2-tuple of (input_arity,output_arity). There are also individual methods for both the input and output arity. There is one comprehensive test for many tuple sizes for the Set and Relation arity methods.

Union: Implemented in both Set and Relation. No limitations exist for this operation as it is quite simple. There are four tests for each of the Set and Relation union implementations.

Apply: Implemented in Set. We are able to apply a Relation to a Set. This operation is limited in the naming/renaming aspects. If we apply a Relation to a Set with conflicting names, issues may aries. Apply has 8 test cases implemented.

Compose: Implemented in Relation. We are able to compose two Relations. This operation has the same limitations as apply in terms of renaming. Compose has 8 test cases implemented.

Inverse: Implemented in Relation. We are able to take the inverse of a Relation. This operation has no limitations. This operation has 2 test cases written.

Project Out: Implemented in Set We are able to project out a given tuple variable from a set. Currently we are only looking at inequality constraints, not equality constraints. Currently we are only looking at the first set in a union of sets.

Codegen: We can generate code from a Set.

Some sub operations that the main operations use:

arity methods depend on arity methods of PresSet/PresRelation (not tested directly, methods checked for)

inverse $\rightarrow$ invert_dict

apply/compose $\rightarrow$ _combine_pres_formulas,_get_rename_dict,_get_unrename_dict

VI. Description of Simplification Operations

Simplification steps:

- Merge terms
- Remove zero coefficients
- Remove empty constraints
- remove duplicate constraints (needed, but unimplemented)
- remove symbolics
- remove free variable equalities
- remove free variable inequalities (partially implemented)
- remove duplicate formulas