Sparse Constraints Design Specifications

Alan LaMielle, Michelle Strout

August 4, 2010

1 Summary

This document describes the design of a library for representing a subset of presburger arithmetic. It supports a single level of existential quantification, no universal quantifiers, and adds support for arbitrarily nested uninterpreted function symbols (UFSs) in constraint expressions.

The Omega library and the Integer Set Library (ISL) are two libraries that seek to represent similar mathematical entities. Omega has limited support for UFSs and ISL supports arbitrary nesting of existential quantifiers.

1.1 General Approach

The general approach for representing sets and relations is based on the standard approach taken by many polyhedral libraries: matrices of coefficients where each row represents a single constraint and each column represents a dimension of the polyhedron. Sparse constraints differs in its requirement to support UFSs. For this feature, we allocate an additional column in the coefficient vectors for each *instance* of a UFS. For example, the expressions $f(i)$ and $f(j)$ are different instances of the UFS f and thus will each have a distinct column in the coefficient vector.

2 Classes and Interfaces

SparseConstraints:

- SymTable (NEED ptr to ST)
- Set of Conjuncts (NOT ptrs to conjuncts)

Subclasses of SparseConstraints:

- Set (e.g., $\{[a,b] : 0 \leq a \leq 10 \wedge 20 \leq b \leq n\}$). Have Set take a list of strings in the constructor. It will use the list size to determine the arity and create an array of STEs that big. The constructor will then look through the strings creating either ConstVal STEs or putting the strings into a set to pass to the Symbol table constructor. After the symbol table constructor is done, the Set constructor will have to loop through the list of strings again to get pointers to the appropriate TupleVar STEs.

  - Contains arity of the tuple variables

- Relation (e.g., $\{[a] \rightarrow [b] : 0 \leq a \leq 10 \wedge 20 \leq b \leq n\}$). The Relation will take two lists of strings in the constructor, one for the input tuple and one for the output tuple.

  - Contains arity of the input tuple variables and output tuple variables

SymTable

- Fields:

  - int num_symconsts;
  - int num_tuplevars;
  - int num_existvars;
  - int num_ufcalls;
  - map<string,STE*> mSymToSTE;
  - map<pair<TypeEnum,int>,STE*> mTypeColToSTE;

- Methods:

  - SymTable(list<string> tuplevars, set<string> symconsts): Constructor will create STEs with type and column info for tuplevars and symconsts. Useful for testing only. Might want to remove due to old functionality.
  - SymTable(set<string> symconsts, set<string> tuplevars, set<string> existentials, int numUFCalls): Constructor will create STEs with type and column info for tuplevars, existentials, and symconsts and will allow up to numUFCalls to be inserted into the symbol table.
  - SymTable(SymTable&): Copy constructor will fail assertion upon calling.
  - STE * lookup(string sym); // need this to lookup variables in constraints. If doing a lookup on unknown string will create an existential variable STE and return ptr to that.
  - STE * lookup(string fname, list<ExpVec> params); // Use funcExpString() to index into map. Create a new STE for UFCall if one doesn't already exist.
  - STE * lookup(TypeEnum symtype, int col); // this is for accessing STE while in ExpVec, for printing, anything else?
  - // No inserts on purpose. We will either be creating a new STE for a existential var or for a funcExp the first time we look them up.

  - void incrNumUFCalls(); // increment the number of uninterpreted function calls
  - private string funcExpString(string fname, list<ExpVec> params); // Create a string by concatenating the fname with the toString() results for the param ExpVec representations.
  - constructExpVec();
    Create an empty ExpVec, useful for testing only.
  - constructExpVec(string var, int coeff);
    Create an ExpVec where the given var has the given coefficient.
  - constructExpVec(int const_coeff);
    Create an ExpVec the given constant coefficient.
  - constructExpVec(string fname, list<ExpVec> params, int coeff);
    Create an ExpVec where the given function call has the given coefficient.

STE base class:

- String id;
- TypeEnum symtype;
- int col;

STE Subclasses:

- TupleVar
- ExistVar
- SymConst
- UFCall(string fname, list<ExpVec> paramIDs)

Conjunction:

- Array of tuple variables
- Set of Equality
- Set of Inequality

Equality

- ExpVec

Inequality

- ExpVec

ExpVec

- Public constructor. Decided to go with a public constructor to make testing easier and avoid making the SymbolTable a friend. In general, should still use factory methods in the SymbolTable to construct an ExpVec.
- 4 integer vectors, constructor should take size for each one
- vector<int> symconsts_coeffs;
- vector<int> tuplevars_coeffs;
- vector<int> existvars_coeffs;
- vector<int> ufcalls_coeffs;
- int const_coeff;
- void add(int addVal); // Add addVal to const entry in ExpVec.
- operator+(ExpVec const & other); // add other ExpVec to this ExpVec
- string toString(); //returns the ExpVec vector sizes
- string vectorStrings(); //returns the vectors inside the ExpVec as strings

Notes: Everything except SymTable and maybe SparseConstraints is going to need a copy constructor.

3 Procedures

Building sparse constraints from an AST will be done in two passes (i.e. two visitors):

3.1 BuildSymTableVisitor

BuildSymTableVisitor (list<string> symbolics) will visit an AST to create the SymbolTable for a SparseConstraints object.

- The SymbolTable constructor will have to create SymConst STE instances for each symbolic and insert them in the appropriate hash tables.
- The BuildSymTableVisitor constructor will have to keep track of the symbolics list and upon visiting the tuple variables create a set of them and THEN call the SymbolTable constructor.
- SymTable * getSymTable();

This visitor will call the SymTable constructor and do lookup calls for all variable identifiers in expressions and count count the function calls (using the incrNumUFCalls() method).

3.2 BuildSparseConstraints

BuildSparseConstraints (SymTable * ST): This visitor is going to call the lookup for UFCalls in the ST upon visiting the FuncExp nodes in the AST and it will create a SparseConstraints instance that it will add constraints to when visiting equality and inequalities. Each expression node in AST should be mapped to an ExpVec in this visitor.

- map<Node*,ExpVec> mNodeToExpVec;
- When visiting expressions will be building ExpVec instances and mapping nodes in the AST to the ExpVec instances.
- visitUnion: Just do a union operation. (Alan and MMS needs to do ST merges first)
- visitNormExp: Just add all ExpVec for included expressions and add in constant.
- visitVarExp:
  - mNodeToExpVec[node] = ST.constructExpVec(varname, coeff)
- visitFuncExp:
  - mNodeToExpVec[node] = ST.constructExpVec(name, list of child ExpVec, coeff)
- outEquality:

  ExpVec vec = mNodeToExpVec.find(node.getExpression())
  - EqConjSet.insert(SREquality(vec))
- outInequality:
  - ExpVec vec = mNodeToExpVec.find(node.getExpression())
  - IeqConjSet.insert(SRInequality(vec))
- outConjunction:
  - create conjunction out of the EqConjSet and the IeqConjSet and place itself into a set

3.3 SparseConstraints::normalize()

- Ensure that any variable tuple elements have a unique variable name.
- Project out a variable: Alan will figure out this and get it working before we do the compose operation.

3.4 SparseConstraints output

We would like to output a set or relation object

- to an omega string so as to match input or
- to a dot file as shown for the example `{[0,x,x,y] :  x < 10 and y = f(x,g(x,1)) and i > x + y }` in Figure 1. (NOTE: the text .dot file is in the git repository under doc/).

3.5 SymbolTable::mergeWith(SymbolTable & other)

Anytime we do an operation that combines two SparseConstraint (i.e., Set or Relation) instances in some way (i.e., union, compose, apply, etc.), we need to combine their symbol tables. The approach will be to add unique symbols from the `other` SymbolTable to the `this` SymbolTable. In the process, we should generate an object called an STMap that can help us convert constraints using the `other` symbol table to constraints using the new symbol table.

3.6 Union

- Combine symbol tables
- Iterate over 2 sets of input conjuncts
- Convert all old expression vectors from old ST to new ST for each new conjunct in new SparseConstraints object.

3.7 Compose

Can't do this one until we can project out existential variables.

- Combine symbol tables
- Outer product between conjuncts of input SparseConstraints
- Convert all old expression vectors from old ST to new ST for each new conjunct in new SparseConstraints object.
- Add equality constraints constraints that set "inner" tuple variables equal to each other

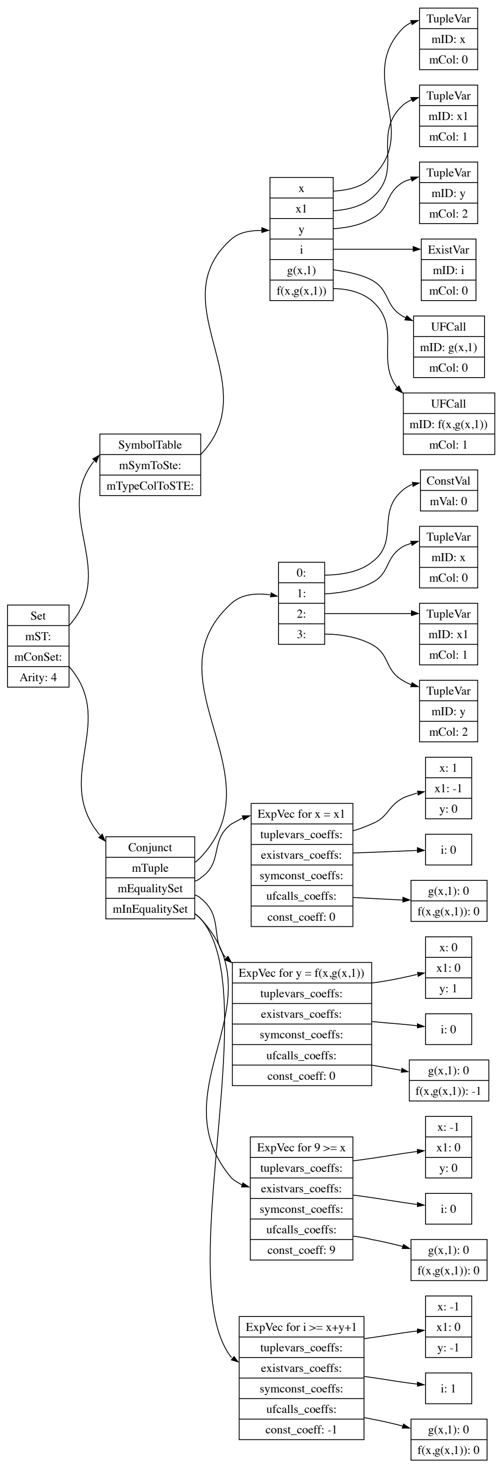

Figure 1: SparseConstraints object for $\{[0, x, x, y] \mid (x < 10) \wedge (y = f(x, g(x, 1))) \wedge (i > x + y)\}$.