Set and Relation Manipulation for the Sparse Polyhedral Framework

Michelle Mills Strout, Geri Georg, and Catherine Olschanowsky

Colorado State University
{mstrout,georg,cathie}@cs.colostate.edu

Abstract. The Sparse Polyhedral Framework (SPF) extends the Polyhedral Model by using the uninterpreted function call abstraction for the compile-time specification of run-time reordering transformations such as loop and data reordering and sparse tiling approaches that schedule irregular sets of iteration across loops. The Polyhedral Model represents sets of iteration points in imperfectly nested loops with unions of polyhedral and represents loop transformations with affine functions applied to such polyhedra sets. Existing tools such as ISL, Cloog, and Omega manipulate polyhedral sets and affine functions, however the ability to represent the sets and functions where some of the constraints include uninterpreted function calls such as those needed in the SPF is non-existant or severely restricted. This paper presents algorithms for manipulating sets and relations with uninterpreted function symbols to enable the Sparse Polyhedral Framework. The algorithms have been implemented in an open source, C++ library called IEGenLib (The Inspector/Executor Generator Library).

1 Introduction

Particle simulations, irregular mesh based applications, and sparse matrix computations are difficult to parallelize and optimize with a compiler due to indirect memory accesses such as `x[k-1][col[p]]` Saltz et al. [1, 2] pioneered inspector/executor strategies for creating parallel communication schedules for such computations at run time. An *inspector/executor* strategy involves generating inspector and executor code at compile time. At runtime an *inspector* traverses index arrays to determine how loop iterations are accessing data, create communication and/or computation schedules, and/or reorder data. An *executor* is the transformed version of the original code. The executor re-uses the schedules and/or reordered data created by the inspector multiple times.

In the late 90s and early 2000s, researchers developed additional inspector/executor strategies to detect fully parallel loops at runtime [3], expose wavefront parallelism [4], improve data locality [5–9], improve the locality in irregular producer/consumer parallelism [10, 11], and schedule sparse tiles across loops so as to expose a level of course-grain parallelism with improved temporal locality [12–14]. The Sparse Polyhedral Framework (SPF) research [15, 16, 13] seeks to provide a compilation framework for automating the application of inspector /executor strategies and their many possible compositions.

H. Kasahara and K. Kimura (Eds.): LCPC 2012, LNCS 7760, pp. 61–75, 2013.

```
for (k=1; k<=m; k++) {
  for (p=0; p<nz; p++) {
    x[k][row[p]] += a[p]*x[k-1][col[p]];
} }
```

Fig. 1. Matrix powers kernel where the matrix is stored in coordinate storage (COO). The Matrix Powers kernel computes a set of vectors $\{A^0\boldsymbol{x}, A^1\boldsymbol{x}, ..., A^m\boldsymbol{x}\}$. This loop is performing k sparse matrix vector products.

```
for (t=0; t<Nt; t++) {
  for (k=1; k<=m; k++) {
    for (i=0; i<N; i++) {
      for (p=0; p<nz; p++) {
        if (sigma[row[p]]==i && tile(k,i)==t))
          x[k][sigma[row[p]]]
            += a[p]*x[k-1][sigma[col[p]]];
} } } }
```

Fig. 2. The transformed matrix powers kernel after the second dimension of `x` has been reordered and a full sparse tiling has been performed. Note that further optimizations are done to remove the conditional from the inner loop and remove double indirections, but such optimizations are not within the scope of this paper.

Transformation frameworks such as the polyhedral framework [17–22] enable the specification and exploration of a space of possible *compile-time* reordering transformations for static control parts [23]. *Static control parts (SCoP)* require that the loop bounds and array accesses in the loops being transformed be affine functions of the loop iterators and variables that do not change in the loops.

A portion of the matrix powers, $A^m\boldsymbol{x}$, kernel in Figure 1 falls within the polyhedral model, specifically the iteration space that contains all integer tuples $[k, p]$ within the specified loop bounds. However, the indirect memory accesses `x[k-1][col[p]]` and `x[k][row[p]]` do not fall directly within the polyhedral model. In previous work, the polyhedral model has been extended to handle indirect memory references by using uninterpreted function calls to represent such memory accesses and using this information to make data dependence analysis more precise [24], approximate data dependences in spite of indirect memory references [25, 26], and handle while loops [22].

A problem arose when the Sparse Polyhedral Framework [15, 16] extended the polyhedral framework further by using uninterpreted function calls to represent run-time reordering transformations. We are aware of only one loop transformation tool that attempts to deal with uninterpreted function calls: omega [27] and a newer version of omega called omega+ [28, 29]. Omega uses uninterpreted function calls to aid in the precision of data dependence analysis.

However, Omega does not use uninterpreted function calls to represent run-time reordering transformations and, therefore, manipulations of the intermediate representations for the computation and for the transformations are not precise enough. For example, the conversion of the memory access `x[k][row[p]]` in Figure 1 to `x[k][sigma[row[p]]]` in Figure 2 is not possible with omega.

Transforming code with the SPF requires composing relations, inverting relations, and applying relations to sets when both the relation(s) and the set can have uninterpreted function call constraints as well as affine constraints. In this paper, we make the following contributions:

- Practical algorithms for performing compositions and applying relations to sets based on the relations typically used in the SPF.
- An open source library that makes the algorithm implementation available for general use.

Section 2 reviews the Sparse Polyhedral Framework terminology revolving around sets and relations and the compose and apply operations. Section 3 presents the compose and apply algorithms. Section 4 presents the IEGenLib software package available at `http://www.cs.colostate.edu/hpc/PIES` that implements the presented algorithms. In Section 5 we conclude.

2 The Sparse Polyhedral Framework (SPF)

Within a polyhedral transformation framework such as Omega [20], Pluto [30], Orio [31], Chill [32, 29], AlphaZ [33], or POET [34], the intermediate representation includes an iteration space set to represent all of the iterations in the loop, a function that maps each iteration in the loop to an array index for each array access, data dependence relations between iterations in the loop, and some representation of the statements themselves.

We originally introduced the sparse polyhedral framework in [15] where it was described as a compile-time framework for composing run-time reordering transformations. In this section, we provide a basic introduction to the SPF: how to represent computations in the SPF, how to transform these computations, and describe the problem of projecting out existential variables that arises in this context.

2.1 Sets and Relations in SPF

Sets and relations are the fundamental building blocks for the SPF. Data and iteration spaces are represented with sets, and access functions, data dependences, and transformations are represented with relations. Sets are specified as $s = \{[x_1, \ldots, x_d] : c_1 \wedge \ldots \wedge c_p\}$, where each x_i is an integer tuple variable/iterator and each c_j is a constraint. The *arity* of the set is the dimensionality of the tuples, which for the above is d.

The constraints in a set are equalities and inequalities. Each equality and inequality is a summation expression containing terms with constant coefficients,

where the terms can be tuple variables x_i, symbolic constants, or uninterpreted function calls. A symbolic constant represents a constant value that does not change during the computation, but may not be known until runtime. An uninterpreted function call $f(p_1, p_2, ..., p_3)$ is a function, therefore, $\boldsymbol{p} = \boldsymbol{q}$ implies that $f(\boldsymbol{p}) = f(\boldsymbol{q})$, however the actual output values are not known until compile time. We also allow the actual parameters p_v passed to any uninterpreted function symbol to be affine expressions of the tuple variables, symbolic constants, free variables, or uninterpreted function symbols, whereas in omega [24] uninterpreted function calls are not allowed as parameters to other uninterpreted function calls. We represent the iteration space I in Figure 1 as a set with only affine constraints, $I = \{[k, p] \mid 1 \leq k < N_k \wedge 0 \leq p < nz\}$.

A relation represents a set of integer tuple pairs, where the first tuple in the pair is called the input tuple (often the relation is a function) and the second tuple in the pair is called the output tuple. Relations are specified as $r = \{[x_1, \ldots, x_m] \rightarrow [y_1, \ldots, y_n] : c_1 \wedge \ldots \wedge c_p\}$, where each x_i is an input tuple variable in $\mathbb{Z}$, each y_j is an output tuple variable in $\mathbb{Z}$, and each c_v is a constraint. The constraints of a relation follow the same restrictions as set constraints and additionally the relation needs to include equalities that make the relation a function or the inverse of a function (see Section 3 for more details).

It is possible to represent the array access functions in Figure 1 (`A1: x[k-1][col[p]]` and `A2: x[k][row[p]]`) as follows:

$$A1_{I \rightarrow X} = \{[k, p] \rightarrow [v, w] \mid v = k - 1 \wedge w = col(p)\}$$
$$A2_{I \rightarrow X} = \{[k, p] \rightarrow [v, w] \mid v = k \wedge w = row(p)\}.$$

As a notational convenience we subscript the names of abstract relations to indicate which sets are the domain and range of the relation. For example, the array access function $A1_{I \rightarrow X}$ has the iteration space set I as its domain and data space set X as its range.

2.2 Transforming Iteration and Data Spaces

The SPF uses relations to represent transformation functions for iteration and data spaces. Given sets that express iteration and data spaces, relations that specify how an iteration space accesses data spaces (access functions), and relations that represent dependences between iteration points (data dependence relations), we can express how data and/or iteration reordering transformations affect these entities by performing certain set and relation operations.

For the matrix powers kernel computation $A^k\boldsymbol{x}$ in Figure 1, assume we plan to reorder the rows and columns of the sparse matrix by reordering the rows of the `x` array to improve the data locality [35]. This *run-time data reordering transformation* can be specified as follows:

$$R_{X \rightarrow X'} = \{[k, i] \rightarrow [k, i'] \mid i' = \sigma(i)\},$$

where $\sigma()$ is an uninterpreted function that represents the permutation for the data that will be created by a heuristic in the inspector at runtime.

The data reordering transformation affects the data space for the array x, therefore, any access functions that target the data space X need to be modified. We use relation composition to compute the new access function:

$$A1_{I\to X'} = R_{X\to X'} \circ A1_{I\to X} = \{[k,p] \to [v,w] \mid v = k-1 \wedge w = \sigma(col(p))\}.$$

An *iteration-reordering* transformation is expressed as a mapping between the original iteration space and the transformed iteration space. The new execution order is given by the lexicographic order of the iterations in I'. In the example, we transform Figure 1 to Figure 2 using full-sparse tiling, a run-time reordering transformation [13] (also equivalent to the "implicit sequential algorithm" in [14]) that provides task graph asynchronous parallelism [36]. The tile() function aggregates iteration points into atomically executable groups of computation.

$$T_{I\to I'} = \{[k,p] \to [t,k,i,p] \mid t = tile(k,i) \wedge i = \sigma(row(p)) \\ \wedge 1 \le t < N_t \wedge 0 \le i < N_r\}.$$

This requires modifying the access functions $A1_{I'\to X'} = A1_{I\to X'} \circ T_{I'\to I} = A1_{I\to X'} \circ T^{-1}_{I\to I'}$ and $A2_{I'\to X'} = A2_{I\to X'} \circ T_{I'\to I} = A2_{I\to X'} \circ T^{-1}_{I\to I'}$, and transforming the iteration space $I' = T_{I\to I'}(I)$. Given the transformed access functions, scheduling functions, and dependences, we can specify further run-time reordering transformations (RTRTs).

2.3 Necessary Set and Relation Operations

Modifying the iteration space and access functions to reflect the impact of run-time reordering transformations requires the following set of operations:

- relation inverse $r = r_1^{-1} = (\boldsymbol{x} \to \boldsymbol{y} \in r) \iff (\boldsymbol{y} \to \boldsymbol{x} \in r_1)$,
- relation composition
  $r = r_2 \circ r_1 = (\boldsymbol{x} \to \boldsymbol{y} \in r) \iff (\exists \boldsymbol{z} \mid \boldsymbol{x} \to \boldsymbol{z} \in r_1 \wedge \boldsymbol{z} \to \boldsymbol{y} \in r_2)$,
- and applying a relation to a set
  $s = r_1(s_1) = (\boldsymbol{x} \in s) \iff (\exists \boldsymbol{z} \mid \boldsymbol{z} \in s_1 \wedge \boldsymbol{z} \to \boldsymbol{x} \in r_1)$.

2.4 The Problem: Implementing Compose and Apply Is Difficult

The inverse operation can easily be implemented by swapping the input and output tuple variables in a relation. However, implementing relation composition and applying a relation to a set is difficult due to the existential variables (i.e. the vector $\boldsymbol{z}$ in Section 2.3) introduced while computing both. These existential variables need to be projected out of the resulting set or relation so that the remaining constraints only involve tuple variables, symbolic constants, and uninterpreted function calls.

When all of the constraints are affine, then each conjunct is a polyhedron. It is possible to use integer versions of Fourier Motzkin [27, 37] to project out any existential variables. The Omega library and calculator [27] enable the expression

of constraints with uninterpreted function calls, but it has two key limitations in terms of manipulating uninterpreted function calls. One limitation is that the arguments to an uninterpreted function have to be a prefix of the input or output tuples. Therefore, the following input and output occurs (the example uses omega+, which was built on omega and has similar behavior with respect to uninterpreted function calls):

```
Omega+ and CodeGen+ v2.2.3 (built on 08/15/2012)
Copyright (C) 1994-2000 the Omega Project Team
Copyright (C) 2005-2011 Chun Chen
>>> symbolic col(1);
>>> A1_I_to_X := { [k,p] -> [k,w] : w=col(p) };
arguments to function must be prefix of input or output tuple ...
```

Even when working around this constraint by using a prefix of the input or output tuple as input to the uninterpreted function call, when a compose or apply operation results in an existential variable that is the parameter to an uninterpreted function call, the UNKNOWN term is included within the conjunct thus making the resulting set or relation lose its precision.

```
>>> symbolic col(2),row(2);
>>> A1_I_to_X := { [k,p] -> [k,w] : w=col(k,p) };
>>> symbolic sigma(2);
>>> R_X_to_X' := {[k,i] -> [k,i'] : i'=sigma(k,i)};
>>> R_X_to_X' compose A1_I_to_X;
{[k,p] -> [k,i] : UNKNOWN}
```

Since in the SPF we are representing computation with iteration spaces and access functions, this level of precision loss is problematic.

Previously, we developed heuristics for eliminating existential variables involved in uninterpreted function call constraints [16]. The heuristics involved solving for existential variables and then substituting the resulting expression in an attempt to remove such existential variables from the constraints. The approach we present in Section 3 is much simpler to explain and prove correct, but is more restrictive in the kinds of relations handled.

3 Algorithms for Implementing Compose and Apply

In the Sparse Polyhedral Framework (SPF), relations and sets have certain characteristics because of what they represent and how the relations are used. A relation can represent (1) a function mapping an iteration point to a memory location integer tuple (access function), (2) the mapping of an iteration point for a statement to a shared iteration space that represent space and lexicographical time (scheduling/scattering function [38]), or (3) a transformation function mapping each iteration (or data point) to a new shared iteration space (or data layout). For (1), (2), and (3), the output tuple is a function of the input tuple.

Based on the above uses of relations in SPF, a relation in SPF is either a function $\{\boldsymbol{x} \rightarrow \boldsymbol{y} \mid \boldsymbol{y} = F(\boldsymbol{x}) \wedge C\}$ or the inverse of a function $\{\boldsymbol{x} \rightarrow \boldsymbol{y} \mid \boldsymbol{x} = G(\boldsymbol{y}) \wedge C\}$ such that $\boldsymbol{x}$ is the input tuple, $\boldsymbol{y}$ is the output tuple, F and G are affine or uninterpreted functions, and C is a set of constraints involving equalities, inequalities, linear arithmetic, and uninterpreted function calls. We can use this information to develop algorithms for relation composition and applying a relation to a set.

This section shows that there are closed form solutions for composing relations and applying a relation to a set that do not involve existential variables when the relations satisfy certain assumptions. The algorithms can be implemented directly by using the closed form solution provided in each theorem and implementing a routine that solves for one set of tuple variables with respect to another set and provides substitution for a set of tuple variables.

3.1 Relation Composition Theorems

Our algorithms for implementing relation composition requires that either both relations must be functions or both relations must be the inverse of a function. By making this assumption, the relation resulting from a composition will be either a function and/or the inverse of a function.

Theorem 1 (Case 1: Both Relations are Functions). *Let $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{v}$, and $\boldsymbol{z}$ be integer tuples where $|\boldsymbol{y}| = |\boldsymbol{v}|$, $F1()$ and $F2()$ be either affine or uninterpreted functions, and $C1$ and $C2$ be sets of constraints involving equalities, inequalities, linear arithmetic, and uninterpreted function calls in*

$$\{\boldsymbol{v} \rightarrow \boldsymbol{z} \mid \boldsymbol{z} = F1(\boldsymbol{v}) \wedge C1\} \circ \{\boldsymbol{x} \rightarrow \boldsymbol{y} \mid \boldsymbol{y} = F2(\boldsymbol{x}) \wedge C2\}.$$

The result of the composition is $\{\boldsymbol{x} \rightarrow \boldsymbol{z} \mid \exists \boldsymbol{y}, \boldsymbol{v} \mid \boldsymbol{y} = \boldsymbol{v} \wedge \boldsymbol{z} = F1(\boldsymbol{v}) \wedge C1 \wedge \boldsymbol{y} = F2(\boldsymbol{x}) \wedge C2\}$, which is equivalent to

$$\{\boldsymbol{x} \rightarrow \boldsymbol{z} \mid \boldsymbol{z} = F1(F2(\boldsymbol{x})) \wedge C1[\boldsymbol{v}/F2(\boldsymbol{x})] \wedge C2[\boldsymbol{y}/F2(\boldsymbol{x})]\}$$

where $C1[\boldsymbol{v}/F2(\boldsymbol{x})]$ indicates that $\boldsymbol{v}$ should be replaced with $F2(\boldsymbol{x})$ in the set of constraints $C1$.

Proof
Starting from $\{\boldsymbol{x} \rightarrow \boldsymbol{z} \mid \exists \boldsymbol{y}, \boldsymbol{v} \mid \boldsymbol{y} = \boldsymbol{v} \wedge \boldsymbol{z} = F1(\boldsymbol{v}) \wedge C1 \wedge \boldsymbol{y} = F2(\boldsymbol{x}) \wedge C2\}$, we first substitute $\boldsymbol{y}$ with $\boldsymbol{v}$ to obtain $\{\boldsymbol{x} \rightarrow \boldsymbol{z} \mid \exists \boldsymbol{v}$ s.t. $\wedge\, \boldsymbol{z} = F1(\boldsymbol{v}) \wedge C1 \wedge \boldsymbol{v} = F2(\boldsymbol{x}) \wedge C2[\boldsymbol{y}/\boldsymbol{v}]\}$. Then we substitute $\boldsymbol{v}$ with $F2(\boldsymbol{x})$ to obtain the forward equivalence $\{\boldsymbol{x} \rightarrow \boldsymbol{z} \mid \boldsymbol{z} = F1(F2(\boldsymbol{x})) \wedge C1[\boldsymbol{v}/F2(\boldsymbol{x})] \wedge C2[\boldsymbol{y}/F2(\boldsymbol{x})]\}$. The backward direction of the equivalence requires performing the reverse substitutions in the reverse order where instead of removing existential variables we are introducing them. ∎

From the running example, both the access relation $A1_{I \rightarrow X'}$ and the transformation $T_{I \rightarrow I'}$ are functions. Therefore, to compute the effect of the transformation on A1 ($A1_{I' \rightarrow X'} = A1_{I \rightarrow X'} \circ T_{I' \rightarrow I}$), we can use Theorem 1. For $A1_{I \rightarrow X'}$, the output

tuple variables are a function of the input tuple variables: $[v, w] = F1([k, p]) = [k-1, \sigma(col(p))]$. For $T_{I' \to I}$, we have the following function: $[k, p] = F2([t, k, i, p]) = [k, p]$. Therefore the result of the composition is

$$A1_{I' \to X'} = \{[t, k, i, p] \to [v, w] \mid v = k - 1 \wedge w = \sigma(col(p))\}.$$

Theorem 2 (Case 2: The Inverses of both Relations are Functions). *Let $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{v}$, and $\boldsymbol{z}$ be integer tuples where $|\boldsymbol{y}| = |\boldsymbol{v}|$, $G1()$ and $G2()$ be either affine or uninterpreted functions, and $C1$ and $C2$ be sets of constraints involving equalities, inequalities, linear arithmetic, and uninterpreted functions in*

$$\{\boldsymbol{v} \to \boldsymbol{z} \mid \boldsymbol{v} = G1(\boldsymbol{z}) \wedge C1\} \circ \{\boldsymbol{x} \to \boldsymbol{y} \mid \boldsymbol{x} = G2(\boldsymbol{y}) \wedge C2\}.$$

The result of the Case 2 composition is $\{\boldsymbol{x} \to \boldsymbol{z} \mid \exists \boldsymbol{y}, \boldsymbol{v}$ s.t. $\boldsymbol{y} = \boldsymbol{v} \wedge \boldsymbol{v} = G1(\boldsymbol{z}) \wedge C1 \wedge \boldsymbol{x} = G2(\boldsymbol{y}) \wedge C2\}$, which is equivalent to

$$\{\boldsymbol{x} \to \boldsymbol{z} \mid \boldsymbol{x} = G2(G1(\boldsymbol{z})) \wedge C1[\boldsymbol{v}/G1(\boldsymbol{z})] \wedge C2[\boldsymbol{y}/G1(\boldsymbol{z})]\}.$$

Proof. As with Theorem 1, we can perform substitutions to show the equivalence. For Theorem 2, we substitute $\boldsymbol{y}$ with $\boldsymbol{v}$ and then substitute $\boldsymbol{v}$ with $G1(\boldsymbol{z})$. ∎

3.2 Relation Application to Set Theorem

For applying a relation to a set, the relation must be the inverse of a function. This is necessary because the existential variables resulting from the application are replaced by functions of the output tuple variables. The below theorem shows why this is the case.

Theorem 3 (Relation to Set Application). *Let $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ be integer tuples where $|\boldsymbol{x}| = |\boldsymbol{z}|$, $G()$ be either an affine or uninterpreted function, and C and D be sets of constraints involving equalities, inequalities, linear arithmetic, and uninterpreted function calls in*

$$\{\boldsymbol{x} \to \boldsymbol{y} \mid \boldsymbol{x} = G(\boldsymbol{y}) \wedge C\}(\{\boldsymbol{z} \mid D\}).$$

The result of applying the relation to the set is $\{\boldsymbol{y} \mid \exists \boldsymbol{x}, \boldsymbol{z} \mid \boldsymbol{z} = \boldsymbol{x} \wedge \boldsymbol{x} = G(\boldsymbol{y}) \wedge C \wedge D\}$, which is equivalent to

$$\{\boldsymbol{y} \mid C[\boldsymbol{x}/G(\boldsymbol{y})] \wedge D[\boldsymbol{z}/G(\boldsymbol{y})]\}.$$

Proof. As with Theorem 1, we can perform substitutions to show the equivalence. For Theorem 3, we substitute $\boldsymbol{z}$ with $\boldsymbol{x}$ and then substitute $\boldsymbol{x}$ with $G(\boldsymbol{y})$. ∎

4 The Inspector/Executor Generator Library

The Inspector/Executor Generator Library (IEGenLib) enables the programmatic manipulation of sets and relations with constraints involving affine expressions where terms can be uninterpreted function calls for the use in specifying run-time reordering transformations. This section provides an overview of typical IEGenLib usage and functionality. Release 1 of the IEGenLib along with a user manual and API documentation can be found at `http://www.cs.colostate.edu/hpc/PIES`.

The IEGenLib is similar to the Omega library [27] in that IEGenLib provides a C++ API for manipulating sets and relations with inequality and equality constraints. The main differences are that IEGenLib enables uninterpreted function calls to have any affine expressions as arguments including those with uninterpreted function calls, and IEGenLib maintains more detail when performing relation to set application and relation composition when the constraints involved include uninterpreted function calls.

IEGenLib Release 1 has fewer features than the current Omega library and new versions of that library such as Omega+ [28]. For example, the IEGenLib calculator **iegenlib_calc** does not generate code at this time. Additionally the IEGenLib calculator and library provide a subet of set and relation operations. IEGenLib does provide the following operations: composition of two relations, applying a relation to a set, union, and relation inverse.

4.1 Typical Usage of IEGenLib

The IEGenLib ships with three convenient interfaces: the IEGenLib API available through a C++ library, the IEGenLib calculator (a sample program using

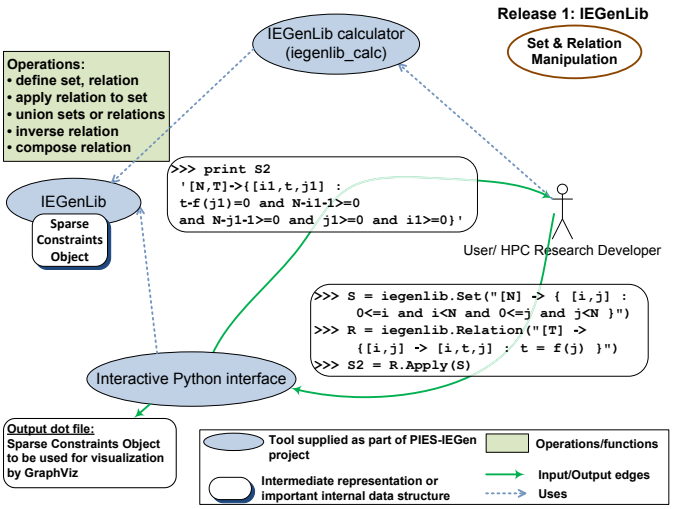

Fig. 3. Shows how the iegenlib is typically used

Table 1. Set and Relation Operations

Operation	**Notation**	**Semantics** `Syntax using Python Bindings`
constant Apply	$s = r_1(s_1)$	$(x \in s) \iff (\exists y \text{ s.t. } y \in s_1 \wedge y \rightarrow x \in r_1)$ `Iprime = T_I_to_Iprime.Apply( I )`
Union	$s = s_1 \cup s_2$ $r = r_1 \cup r_2$	$(x \in s) \iff (x \in s_1 \vee x \in s_2)$ $(x \rightarrow y \in r) \iff (x \rightarrow y \in r_1 \vee x \rightarrow y \in r_2)$
Inverse	$r = r_1^{-1}$	$(x \rightarrow y \in r) \iff (y \rightarrow x \in r_1)$ `T_I_to_I = T_I_to_Iprime.Inverse()`
Compose	$r = r_2 \circ r_1$	$(x \rightarrow y \in r) \iff (\exists z \text{ s.t. } x \rightarrow z \in r_1 \wedge z \rightarrow y \in r_2)$ `A1_I_to_Xprime = R_X_to_Xprime.Compose(A1_I_to_X)`

the library that enables interactive experimentation), and the interactive Python interface (i.e. python bindings). Section 4.2 provides an overview of the IEGenLib API and underlying class structure. The IEGenLib calculator and Python interface are each supplied to allow users quick access to the IEGenLib capabilities.

The IEGenLib calculator (iegenlib_calc) is a C++ program written using the IEGenLib. It is both useful as a standalone tool and the source code is provided as an example of how to use the library API.

The interactive Python interface is automatically created using SWIG. After the application of SWIG it is possible to access the C++ library directly from Python scripts and the interactive Python interface. All of the examples in the following section are written using the Python syntax. Figure 3 shows the usage relationship between the three interfaces and gives a brief example of using the Python interface.

4.2 Class Structure of IEGenLib

This section gives an overview of both the programmatic interface exposed by the IEGenLib and the class structure that supports the given interface. The interface is designed to be easily accessible and at the same time enable advanced users direct access to the internal structures.

The primary function of the IEGenLib is to provide a programmatic interface for the manipulation of sets and relations, therefore, the primary high-level objects are exposed as two classes, Set and Relation (each in the iegenlib namespace). Sets and Relations are each instantiated using a constructor that takes a string as a parameter. As an example, instantiating the Relations used in the example in Section 1 is done as follows.

$A1_{I \rightarrow X} = \{[k,p] \rightarrow [v,w] \mid v = k-1 \wedge w = col(p)\}$

```
# Python code to represent access function for x[k-1][col[p]]
import iegenlib
A1_I_to_X = Relation("{[k,p] → [v,w] : v=k-1 && w=col(p)}")
```

$A2_{I \to X} = \{[k,p] \to [v,w] \mid v = k \wedge w = row(p)\}$

```
# Python code to represent access function for x[k][row[p]]
A2_I_to_X = Relation("{[k,p] → [v,w] : v=k && w=row(p) }")
```

$T_{I \to I'} = \{[k,p] \to [t,k,i,p] \mid t = tile(k,i) \wedge i = sigma(row(p))$
$\wedge 0 \leq t < N_t \wedge 0 \leq i < N_r\}$

```
# Python code to represent sparse tiling transformation
T_I_to_Iprime = Relation("{[k,p] → [t,k,i,p] : t=tile(k,i)
&& i=sigma(row(p)) && 0 ≤ t && t < N_t && 0 ≤ i && i < N_r }")
```

Table 1 lists the high-level operations available for the Set and Relation classes: apply, union, inverse, and compose. The table shows the syntax used to use these functions through the Python bindings. The examples in the table use the objects that result from the above construction examples.

An internal class structure supports the Set and Relation class operations. The class structure is centralized around Expressions. Expressions (class name Exp) consist of at least one Term. A Term can fall into one of four categories. First, a Term may be an integer constant, in that case it is implemented using the Term class directly. In the other three cases a Term may be coefficient multiplied by a variable (VarTerm), a coefficient multiplied by a tuple variable (TupleVarTerm),

```
Relation R_X_to_Xprime(2,2);
Conjunction *c = new Conjunction(2+2);
c->setTupleElem(0,"k");
c->setTupleElem(1,"i");
c->setTupleElem(2,"k");
c->setTupleElem(3,"i'");

// Create the expression
Exp* exp = new Exp();
exp->addTerm(new VarTerm("i'"));
std::list<Exp*> *args = new std::list<Exp*>;
Exp *arg0 = new Exp();
arg0->addTerm(new VarTerm("i"));
args->push_back(arg0);
exp->addTerm(new UFCallTerm(-1, "sigma", args));

// add the equality to the conjunction
c->addEquality(exp);

// add the conjunction to the relation
R_X_to_Xprime.addConjunction(c);
```

Fig. 4. Building the Relation in Figure 5 manually

```
Relation R_X_to_Xprime =
        Relation("{[k,i] -> [k,i'] : i' = sigma(i) }")
```

Fig. 5. Building the Relation in Figure 4 using the parser

or a coefficient multiplied by an uninterpreted function call (UFCallTerm). A UFCallTerm contains a list of parameters that are instances of the Exp class.

While it is possible to utilize the IEGenLib to create Sets and Relations using the class structure directly a parser is included in the library that allows for much more simple construction. A built-in parser enables constructors in the Set and Relation classes that accept a string. The string can use the Omega or ISL syntax. The parser does all of the internal work to build the appropriate underlying structure that represents the Set or Relation desired. Figures 4 and 5 demonstrate the significant reduction in user code size that results from using this feature.

Another helpful capability of the IEGenLIb is that each class implements a function that writes a representation of that object to dot. Dot is a syntax for creating "hierarchical" or layered drawings of directed graphs. Tools such as

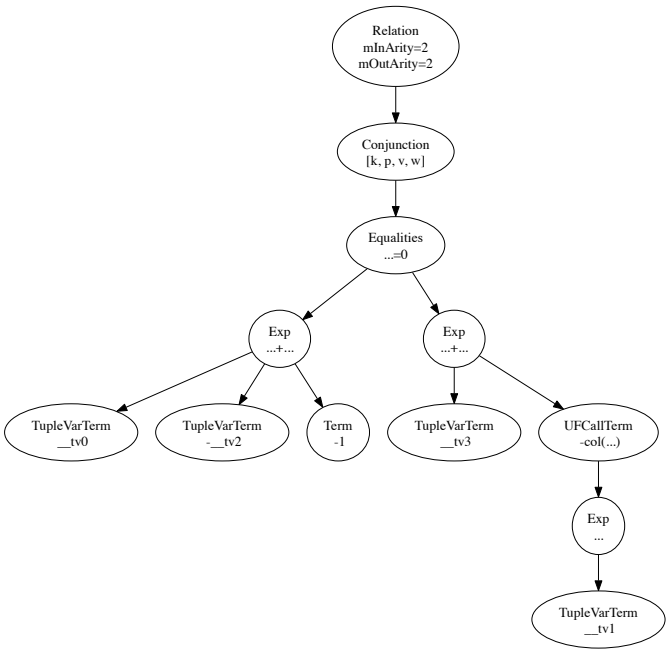

Fig. 6. The dot visualization for the relation $A1_{I \to X}$

Graphviz create images using the dot files as input. The visual representations of sets and relations is a quick way to understand the underlying structure for a specific Set or Relation. Figure 6 shows an example taken from the introduction.

5 Conclusions

This work is another step in automating the process of generating inspector/executor code. We present algorithms for composing relations and applying relations to sets, when the relation(s) and set involved in those operations include affine constraints and constraints involving uninterpreted function calls. The IEGenLib software package implements the presented algorithms. This paper also shows how a user of IEGenLib can specify and perform relation composition and the application of a relation to a set.

Acknowledgements. We would like to thank previous participants of the IEGenLib project for their implementation work including Brendan Sheridan, Mark Heim, Mohammed Al-Refai, Nicholas Wes Jeannette, and Ian Craig. We would especially like to thank Alan LaMielle who developed an earlier prototype of IEGenLib and Joseph Strout who made significant programming contributions to the current version. This project is supported by a Department of Energy Early Career Grant DE-SC0003956, by a National Science Foundation CAREER grant CCF 0746693, and by the Department of Energy CACHE Institute grant DE-SC04030.

References

1. Mirchandaney, R., Saltz, J.H., Smith, R.M., Nico, D.M., Crowley, K.: Principles of runtime support for parallel processors. In: ICS 1988: Proceedings of the 2nd International Conference on Supercomputing, pp. 140–152. ACM, New York (1988)
2. Saltz, J., Chang, C., Edjlali, G., Hwang, Y.S., Moon, B., Ponnusamy, R., Sharma, S., Sussman, A., Uysal, M., Agrawal, G., Das, R., Havlak, P.: Programming irregular applications: Runtime support, compilation and tools. Advances in Computers 45, 105–153 (1997)
3. Rauchwerger, L.: Run-time parallelization: Its time has come. Parallel Computing 24, 527–556 (1998)
4. Rauchwerger, L., Amato, N.M., Padua, D.A.: A scalable method for run-time loop parallelization. International Journal of Parallel Programming 23, 537–576 (1995)
5. Ding, C., Kennedy, K.: Improving cache performance in dynamic applications through data and computation reorganization at run time. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 229–241. ACM, New York (1999)
6. Mitchell, N., Carter, L., Ferrante, J.: Localizing non-affine array references. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 192–202. IEEE Computer Society, Los Alamitos (1999)

7. Im, E.J.: Optimizing the Performance of Sparse Matrix-Vector Multiply. Ph.d. thesis, University of California, Berkeley (2000)
8. Mellor-Crummey, J., Whalley, D., Kennedy, K.: Improving memory hierarchy performance for irregular applications using data and computation reorderings. International Journal of Parallel Programming 29, 217–247 (2001)
9. Pingali, V.K., McKee, S.A., Hsieh, W.C., Carter, J.B.: Restructuring computations for temporal data cache locality. International Journal of Parallel Programming 31, 305–338 (2003)
10. Basumallik, A., Eigenmann, R.: Optimizing irregular shared-memory applications for distributed-memory systems. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 119–128. ACM Press, New York (2006)
11. Ravishankar, M., Eisenlohr, J., Pouchet, L.N., Ramanujam, J., Rountev, A., Sadayappan, P.: Code generation for parallel execution of a class of irregular loops on distributed memory systems. In: The International Conference for High Performance Computing, Networking, Storage, and Analysis, SC (2012)
12. Douglas, C.C., Hu, J., Kowarschik, M., Rüde, U., Weiß., C.: Cache Optimization for Structured and Unstructured Grid Multigrid. Electronic Transaction on Numerical Analysis, 21–40 (2000)
13. Strout, M.M., Carter, L., Ferrante, J., Kreaseck, B.: Sparse tiling for stationary iterative methods. International Journal of High Performance Computing Applications 18, 95–114 (2004)
14. Mohiyuddin, M., Hoemmen, M., Demmel, J., Yelick, K.: Minimizing communication in sparse matrix solvers. In: Supercomputing, ACM, New York (2009)
15. Strout, M.M., Carter, L., Ferrante, J.: Compile-time composition of run-time data and iteration reorderings. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, New York (2003)
16. LaMielle, A., Strout, M.M.: Enabling code generation within the sparse polyhedral framework. Technical report, Technical Report CS-10-102 Colorado State University (2010)
17. Wolf, M.E., Lam, M.S.: Loop transformation theory and an algorithm to maximize parallelism. IEEE Transactions on Parallel and Distributed Systems 2, 452–471 (1991)
18. Feautrier, P.: Some efficient solutions to the affine scheduling problem. part II. multidimensional time. International Journal of Parallel Programming 21, 389–420 (1992)
19. Sarkar, V., Thekkath, R.: A general framework for iteration-reordering loop transformations. In: Fraser, C.W. (ed.) Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 175–187. ACM, New York (1992)
20. Kelly, W., Pugh, W.: A unifying framework for iteration reordering transformations. Technical Report CS-TR-3430, University of Maryland, College Park (1995)
21. Cohen, A., Donadio, S., Garzaran, M.J., Herrmann, C., Kiselyov, O., Padua, D.: In search of a program generator to implement generic transformations for high-performance computing. Sci. Comput. Program. 62, 25–46 (2006)
22. Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A., Bastoul, C.: The Polyhedral Model Is More Widely Applicable Than You Think. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 283–303. Springer, Heidelberg (2010)
23. Xue, J.: Transformations of nested loops with non-convex iteration spaces. Parallel Computing 22, 339–368 (1996)

24. Pugh, B., Wonnacott, D.: Nonlinear array dependence analysis. Technical Report CS-TR-3372, Dept. of Computer Science, Univ. of Maryland (1994)
25. Lin, Y., Padua, D.: Compiler analysis of irregular memory accesses. SIGPLAN Notices 35, 157–168 (2000)
26. Barthou, D., Collard, J.F., Feautrier, P.: Fuzzy array dataflow analysis. Journal of Parallel and Distributed Computing 40, 210–226 (1997)
27. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The omega calculator and library, version 1.1.0 (1996)
28. Chen, C., Hall, M., Venkat, A.: Omega+ (2012), `http://ctop.cs.utah.edu/ctop/?page_id=21`
29. Chen, C.: Polyhedra scanning revisited. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, pp. 499–508. ACM, New York (2012)
30. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral program optimization system. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, New York (2008)
31. Hartono, A., Norris, B., Ponnuswamy, S.: Annotation-based empirical performance tuning using Orio. In: 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS), Rome, Italy (2009)
32. Hall, M., Chame, J., Chen, C., Shin, J., Rudy, G., Khan, M.M.: Loop Transformation Recipes for Code Generation and Auto-Tuning. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 50–64. Springer, Heidelberg (2010)
33. Yuki, T., Basupalli, V., Gupta, G., Iooss, G., Kim, D., Pathan, T., Srinivasa, P., Zou, Y., Rajopadhye, S.: Alphaz: A system for analysis, transformation, and code generation in the polyhedral equational model. Technical report, Colorado State University CS-12-101 (2012)
34. Yi, Q., Seymour, K., You, H., Vuduc, R., Quinlan, D.: Poet: Parameterized optimizations for empirical tuning. In: Proceedings of the Parallel and Distributed Processing Symposium (2007)
35. Wood, S., Strout, M.M., Wonnacott, D.G., Eaton, E.: Smores: Sparse matrix omens of reordering success. Winning Poster at the PLDI Student Research Competition (2011)
36. Strout, M.M., Carter, L., Ferrante, J., Freeman, J., Kreaseck, B.: Combining Performance Aspects of Irregular Gauss-Seidel Via Sparse Tiling. In: Pugh, B., Tseng, C.-W. (eds.) LCPC 2002. LNCS, vol. 2481, pp. 90–110. Springer, Heidelberg (2005)
37. Verdoolaege, S.: An integer set library for program analysis. In: Advances in the Theory of Integer Linear Optimization and its Extensions, AMS 2009 Spring Western Section Meeting, San Francisco, California (2009)
38. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: Proceedings of the 13th Interntional Conference on Parallel Architecture and Compilation Techniques, PACT (2004)