THESIS

IEGEN: SEMI-AUTOMATIC GENERATION OF INSPECTORS AND EXECUTORS

Submitted by

Alan LaMielle

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2010

COLORADO STATE UNIVERSITY

March 8, 2010

WE HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER OUR SUPERVISION BY ALAN LAMIELLE ENTITLED IEGEN: SEMI-AUTOMATIC GENERATION OF INSPECTORS AND EXECUTORS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE.

Committee on Graduate Work

Sanjay Rajopadhye

Jiangguo Liu

Adviser: Michelle Strout

Department Chair: Darrell Whitley

ABSTRACT OF THESIS

IEGEN: SEMI-AUTOMATIC GENERATION OF INSPECTORS AND EXECUTORS

Software that simulates real-world phenomena such as heat transfer over surfaces and molecular interaction is often based on irregular computational kernels. Indirect array accesses such as `A[B[i]]` that are found in irregular computations often exhibit a memory access pattern that does not make efficient use of the memory hierarchy, reducing performance. Additionally, indirect array accesses hinder our ability to apply loop optimizations to improve data locality or introduce parallelism at compile time. One approach to solving this problem, an inspector/executor strategy, inspects the index arrays (`B`) at runtime to determine the order of accesses to the data arrays (`A`), reorders the data and index arrays, and executes a transformed computation that accesses the data arrays in a more efficient manner.

For the most part, the application of inspector/executor strategies to irregular computations has been done manually or with limited generality. This thesis presents the Inspector/Executor Generator (IEGen). This tool accepts an irregular computation specification and sequence of run-time reordering transformations to apply to that computation as input. IEGen then generates serial inspector and executor code that implements the transformed computation. We contribute an inspector intermediate representation (IR) called an Inspector Dependence Graph (IDG), a method for code generation of inspectors based on an IDG, a method for code generation of executors, and techniques for manipulating affine constraints with uninterpreted function symbol (UFS) expressions to enable code generation. We evaluate our techniques against an existing library that supports limited UFS expressions and additionally show that generalized generation of inspectors and executors that implement composed run-time reordering transformations is possible.

Alan LaMielle
Department of Computer Science
Colorado State University
Fort Collins, CO 80523
Spring 2010

ACKNOWLEDGEMENTS

First and foremost I would like to thank my advisor Michelle for her research guidance, implementation suggestions, and writing feedback and for whom I could not have produced this thesis. Many thanks to my committee members Sanjay and James who were willing to read and review this thesis. I thank my almost-wife Rachael for her love, understanding, and encouragement over the many months that it took me to implement and write. Finally, thanks and gratitude go out to my fellow grad students, family, and friends for supporting me along the way, even when you didn't know you were.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Applications such as molecular dynamics simulations and finite element analysis often contain indirect memory references such as `A[B[i]]` to support the usage of sparse data structures. Indirect array references like these make it difficult to apply loop optimizations to improve data locality at compile time. Applying loop optimizations and performing data reorderings is often a task that is desirable due to the poor use of the memory hierarchy that indirect array references exhibit.

One approach to improving data locality and introducing parallelism to irregular applications is to apply various inspector/executor strategies [15]. An inspector examines the indirect array array accesses at runtime, and may reorder data, or determine a valid tiling for the computation based on this examination. An executor utilizes the results of the inspector to improve the performance of the original computation. Consecutive packing and locality grouping [11] are examples of data and iteration permutation run-time reordering transformations. Full sparse tiling [22] is an example of a sparse tiling transformation that can be used to introduce parallelism into irregular applications. Transformations of these types are applied to a computation at compile time, but the specific reordering or tiling is not determined by the inspector until runtime. Previously, the application of inspector/executor strategies was performed manually or with limited generality.

1.1 IEGen: Inspector/Executor Code Generator

This thesis presents a serial inspector/executor code generator and transformation tool called IEGen that helps to automate the application of inspector/executor strategies to irregular applications. IEGen accepts an input file that defines an irregular computation and a sequence of run-time reordering transformations (RTRTs) to apply to that computation. The output of IEGen is an inspector function and an executor function that together implement the transformed computation. We base the code generation of inspector and executor code on the specification of irregular computations and run-time reordering transformations in a previously presented framework [21], which we now call the Sparse Polyhedral Framework (SPF[1]). Notably, this framework uses collections of integer tuples and mappings of integer tuples called sets and relations to represent irregular computations and transformations being applied to such computations. Users of IEGen can replace the code for the original computation with the generated code. A transformation writer must provide a compile-time component to transform the representation of the computation and a run-time component to calculate any reorderings or sparse tilings. In summary, IEGen accepts the specification of an irregular computation and a sequence of transformations as input, creates and transforms an internal representation of the specified computation, and generates inspector and executor code that implements the transformed computation.

The contributions of this thesis include a set of techniques used to generate inspector and executor code for irregular computations. Specifically, we contribute three main ideas. First, we have designed a data structure called the inspector

[1] This term was originally coined by Larry Carter.

dependence graph (IDG) for representing the data and tasks within an inspector that are used to perform data reorderings, iteration reorderings, and sparse tilings. Based on the IDG, we have designed an algorithm to generate inspector code. Secondly, we contribute a technique for generating executor code that is based on an existing data structure called the mapping intermediate representation (MapIR). Finally, we have designed and implemented a data structure and a set of simplification rules that allow us to represent and perform operations on sets and relations including support for uninterpreted function symbols. The simplifications enable the generation of inspector and executor code.

The remainder of this chapter will present two examples of mathematical frameworks for applying loop optimizations, called transformation frameworks. The first example is representative of polyhedral transformation frameworks and is shown as a baseline of comparison. The second is an example of representing and transforming an irregular computation using the sparse polyhedral framework and is shown in comparison to the polyhedral example. We show that representing, transforming, and generating transformed code for the irregular computation requires additional techniques. IEGen is able to represent and transform both of these examples. We utilize the sparse polyhedral framework example throughout this thesis to motivate various techniques and data structures that we present.

1.2 Polyhedral Transformation Example

Current loop transformation frameworks such as Pluto [7] represent and manipulate iteration spaces as polyhedra or unions of polyhedra. Code is generated to scan the transformed iteration spaces by (1) determining loop bounds that are affine functions of outer iterators and symbolic constants and (2) computing new memory/array accesses in terms of the new iterators. Fourier-Motzkin elimination

```
for(i=0; i<=5; i++) {
  for(j=0; j<=4; j++) {

    A[i,j]=f(...,A[i-1,j+1],...);

  }
}
```

Figure 1.1: Original affine computation

```
for(i'=0; i'<=9; i'++) {
  for(j'=max(i',0); j'<=min(i',5); j'++) {

    A[j',i'-j']=f(...,A[j'-1,i'-j'+1]+1,...);

  }
}
```

Figure 1.2: Transformed affine computation

is used to successively project out inner iterators to determine the loop bounds for outer iterators.

As an example of what polyhedral transformation frameworks can do, consider the original and transformed affine computation in Figures 1.1 and 1.2. The iteration space specification for the original 2D loop is

$$\{[\mathrm{i},\mathrm{j}] : (0 \leq \mathrm{i} \leq 5) \wedge (0 \leq \mathrm{j} \leq 4)\}.$$

After applying a skewing transformation and a loop permutation transformation, the set specification is

$$\{[\mathrm{i}',\mathrm{j}'] : (0 \leq \mathrm{i} \leq 5) \wedge (0 \leq \mathrm{j} \leq 4) \wedge (\mathrm{i}' = \mathrm{i} + \mathrm{j}) \wedge (\mathrm{j}' = \mathrm{i})\}.$$

Transforming the computation requires determining loop bounds for the new loop iterators (i.e., i' and j') and determining the new array access functions within the context of the transformed iteration space (i.e., `A[i,j]` to `A[j',i'-j']`). Fourier-Motzkin elimination is a common technique used to project out the existentials

```
   for(time=0; time<T; time++) {
     ...
      for(int tri=0; tri<R; tri++) {
        ...
S1:  ...data[n1[tri]]...
S2:  ...data[n2[tri]]...
S3:  ...data[n3[tri]]...
        ...
      }
     ...
   }
```

Figure 1.3: Original irregular computation

i and j and to project out j' to derive the loop bounds for i' with the goal of generating the transformed code.

1.3 Irregular Transformation Example

Now we show a similar example for an irregular computation using the SPF. Figure 1.3 presents a computation over multiple time-steps that iterates over the triangles in an irregular mesh accessing each of the three nodes in each triangle (based on the index arrays `n1`, `n2`, `n3`). Representing and transforming the computation in Figure 1.3 requires utilizing mathematical concepts beyond those offered by the polyhedral model. This is due to the use of the index arrays `n1`, `n2`, and `n3` to access the data array `data` since this array access is not affine. An insight introduced by Strout et al. in [21] is the use of Presburger relations with uninterpreted function symbols (UFSs) to represent run-time reordering transformations (RTRTs). Conceptually, an uninterpreted function symbol $f(p_1, p_2, ..., p_3)$ is a function whose output value is not known. Therefore, we use uninterpreted function symbols to represent index arrays (such as `n1`), permutation reordering functions, and partitioning/tiling functions at compile time even though their val-

```
/* Create input for consecutive packing */
for (tri=0;tri<=R-1;tri++) {
  cpack_input[tri]=n1[tri];
}

/* Call consecutive packing to create ’sigma’ */
cpack(cpack_input,sigma);

/* Create the inverse of ’sigma’, ’sigma_inv’ */
sigma_inv=genInverse(sigma);

/* Reorder the data array */
reorderArray(data,sigma);

/* Create input for locality grouping */
for (i=0;i<=R-1;i++) {
  locgroup_input[i]=sigma[n1[i]];
}

/* Call locality grouping to create delta */
locgroup(locgroup_input,delta);

/* Create ’delta_inv’ from ’delta’ */
delta_inv=genInverse(delta);

/* Partition the iterations of the tri loop */
proc=partition(R);
```

Figure 1.4: Inspector for transformed irregular computation

ues will not be known until runtime. The SPF’s use of Presburger sets and relations with UFSs mathematically powerful enough to represent and transform irregular computations.

IEGen reads in a specification of the original computation (Figure 1.3) and a sequence of transformations. While applying these transformations, it maintains an internal representation of the current state of the computation. After applying each transformation, it is then able to generate code for the transformed computation

```
#define S0 ...data[sigma[n1[delta_inv[tri]]]]...
#define S1 ...data[sigma[n2[delta_inv[tri]]]]...
#define S2 ...data[sigma[n3[delta_inv[tri]]]]...

 for(time=0; time<T; time++) {
   ...
   for(p=0; p<NUM_PROCS; p++) {
     for(tri=0; tri<R; tri++) {
       if(p==proc(tri)) {
         ...
         S0;
         S1;
         S2;
         ...
       }
     }
   }
   ...
 }
```

Figure 1.5: Executor for transformed irregular computation

based on its internal representation. Simplified examples of generated inspector and executor code are presented in Figures 1.4 and 1.5. The generated inspector code in Figure 1.4 performs three main tasks: reordering the data array `data` based on the consecutive packing reordering heuristic, creating the iteration permutation reordering `delta`, and partitioning the loop that iterates over triangles. The results of these tasks are used by the generated executor code in Figure 1.5.

Similarly to the polyhedral example above, we demonstrate how some components of the original and transformed irregular computation are represented. The original iteration space specification for the statement `S1` is

$$\mathrm{I} = \{[\mathrm{time}, \mathrm{tri}] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R})\}.$$

The iteration space set specification for `S1` after applying consecutive packing, locality grouping, and a computation partitioning is

$$\mathrm{I}' = \{[\mathrm{time}, \mathrm{p}, \mathrm{tri}] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge \mathrm{p} = \mathrm{proc}(\mathrm{tri}) \wedge (0 \leq \mathrm{tri} < \mathrm{R})\}.$$

Note that there are no affine bounds for the iterator `p`. The original access function for `S1`'s access to the array `data[]` is

$$\mathrm{A}_{\mathrm{orig}} = \{[\mathrm{time}, \mathrm{tri}] \rightarrow [\mathrm{out}] : \mathrm{out} = \mathrm{n1}(\mathrm{tri})\}.$$

The access function for the access to the array `data[]` after transformation is

$$\mathrm{A}'_{\mathrm{orig}} = \{[\mathrm{time}, \mathrm{p}, \mathrm{k}] \rightarrow [\mathrm{out}] : \mathrm{k} = \mathrm{delta}(\mathrm{tri}) \wedge \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{tri}))\}.$$

Note that the specification after transformation includes an existential (`tri`) involved in UFS constraints that are not in the final code. Finally, note that the UFSs `sigma`, `delta`, and `proc` are created in the inspector and utilized in the executor. Each of these are issues that must be addressed before we are able to generate the code in Figures 1.4 and 1.5.

1.4 Thesis Contributions

IEGen implements our approaches to solving the problems we highlighted with the irregular transformation framework example. The contributions of IEGen and this thesis are thus threefold:

- we show how to represent and generate code for the inspector after applying a sequence of transformations,
- we show how to generate code for the executor based on the MapIR after applying a sequence of transformations,
- and we show how to enable code generation based on our representation of sets and relations with UFSs.

Each of these contributions is discussed in this thesis. We first present a high-level introduction to IEGen in Chapter 2. Chapter 3 discusses our techniques for generating inspector and executor code after transforming the computation. We show the details of how we represent irregular computations and enable code generation in Chapter 4. The techniques for enabling code generation from sets and relations form the basis of our code generator and thus are the core contribution of this thesis. The data structure we have designed for representing sets and relations with UFSs is discussed in Chapter 5. Finally, we discuss previous research related to ours, discuss possible future work building on this project, and conclude in Chapters 6 and 7.

Chapter 2

IEGen: An Irregular Code Transformation Tool

This chapter discusses at a high level the IEGen inspector/executor generator tool for specifying irregular computations, specifying transformations on irregular computations, and generating inspector and executor functions that implement the resulting transformed computation. We show how to specify the irregular computation in Figure 1.3 and two transformations (a data reordering and an iteration reordering) as input to IEGen. We then discuss the main steps our tool takes to represent this computation, transform it using these transformations, and generate inspector and executor functions for the final transformed computation. We highlight the specific contributions of this thesis as we describe these steps.

2.1 IEGen Overview

The Inspector/Executor Generator (IEGen) consists of two main components: a middle end and a back end. The middle end applies the specified sequence of transformations to an internal program representation and constructs other intermediate representations while the back end generates the resulting code based on the final transformed program representation. These two components additionally use an internal library for representing collections of integer tuples and mappings

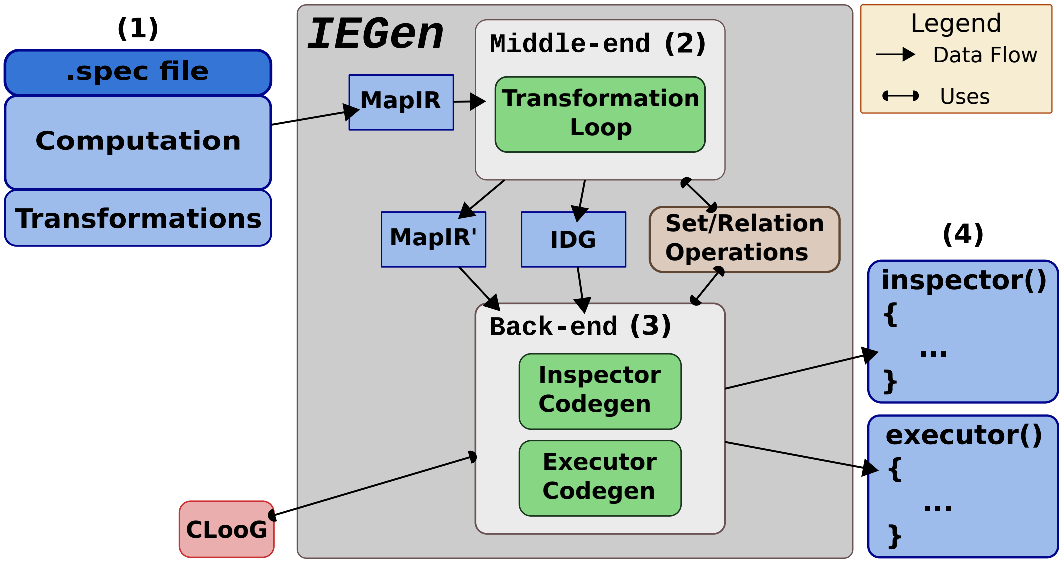

Figure 2.1: Block diagram of IEGen

of integer tuples to other integer tuples (we call these sets and relations respectively). We use this library to represent entities such as iteration spaces and array accesses. For detailed descriptions of sets and relations, we defer the reader to Chapter 4.

We present a block diagram of the structure of IEGen in Figure 2.1 and briefly describe the various components within this diagram. IEGen accepts input in the form of specification files (labeled as (1) in Figure 2.1) that define a computation and a sequence of transformations to apply to that computation. Running a command such as

```
iegen examples/example_trans.spec  −o example.c
```

invokes IEGen, tells it to read in the spec file `examples/example_trans.spec`, and write the generated code to the file `example.c`. Once IEGen is invoked, it reads in the specified spec file and creates a data structure that represents the specified computation called the mapping intermediate representation (MapIR). The middle end (2) then successively applies the specified transformations to produce a transformed MapIR (denoted as MapIR' in Figure 2.1) and an inspector dependence graph (IDG).

An IDG is a data structure that represents the tasks and data for the inspector. One may consider the MapIR' as the data structure representing the transformed executor, and the IDG as the data structure representing the inspector for this executor. Next, the back end code generator (3) uses the MapIR' and IDG to generate the associated code for each of the inspector and executor functions (4). Notice that we utilize an external tool called CLooG [3, 4, 5] to generate loops for scanning the integer points within a polyhedral space. Note also that both the middle end and the back end utilize the set/relation library for manipulation of the computation representations.

IEGen contains two common components of a typical loop transformation and code generation tool: a method for transforming computations (the middle end) and a method of generating code after transforming the computation (the back end). In addition to these two components, other more complete computation optimization tools may also include a component to determine a sequence of program transformations to apply to optimize for some particular objective function (minimize running time, maximize battery life, etc.) and a legality checker to ensure that the specified transformations are legal. At this point IEGen contains no guidance component for choosing what transformations to apply—IEGen assumes an expert user. In addition, IEGen also assumes the transformations that are specified are legal and does nothing during the application of transformations to ensure that they maintain program correctness.

2.2 Specification Files

The input to IEGen is a specification file (.spec) that defines the computation and a sequence of transformations to apply to that computation. In this section, we describe how a user of IEGen can describe a computation and a sequence of transformations. As an initial example, Figure 2.2 presents a whole computation specification file for the code in Figure 1.3. Figure 2.3 presents a second file that includes `example_comp.spec` and defines two transformations to apply to that computation. The rest of this section discusses the details of the various pieces of these files including: symbolic constants, data arrays, index arrays, statements, access relations, data dependences, and transformations.

2.2.1 Symbolics

Symbolic constants define values that are constant for a given computation but may not be known until execution-time. Therefore, we denote them with a symbol until the actual value is known. Three symbolic constants are needed for the computation in Figure 1.3: `T`, `R`, and `N`. These represent the number of time-steps, triangles, and data items respectively. We only specify lower bounds for each: `T` > 1, `N` > 1, and `R` > 1. We specify the symbolic constant `T` in a spec file with a name, type string, and lower bound as:

```
spec.add_symbolic(
    name='T',
    type='int %s',
    lower_bound=1) #Number of time steps
```

The type string is used for variable and function argument declarations and the lower (and possibly upper) bound is used by CLooG.

```
#Define the symbolic constants for the computation
spec.add_symbolic(name='T',
    type='int %s',
    lower_bound=1) #Number of time steps
spec.add_symbolic(name='R',
    type='int %s',
    lower_bound=1) #Number of triangles
spec.add_symbolic(name='N',
    type='int %s',
    lower_bound=1) #Number of data values

#Define the data arrays for the computation
spec.add_data_array(name='data',
    type='double *%s',
    elem_size='sizeof(double)',
    bounds='{[k]: 0<=k and k<N}')

#Define the index arrays for the computation
spec.add_index_array(name='n1',
    type='int *%s',
    input_bounds='{[k]: 0<=k and k<R}',
    output_bounds='{[k]: 0<=k and k<N}')
#Similar definitions for n2 and n3

#Define the statements for the computation
spec.add_statement(name='S1',
    text='...data[%(a1)s]...;',
    iter_space='''{[time,tri]:
               0<=time and time<T and 0<=tri and tri<R}''',
    scatter='{[time,tri]->[c0,time,c1,tri,c0]:
               c0=0 and c1=1}')
#Similar definitions for S2 and S3

#Define the access relations for the statements
spec.add_access_relation(statement_name='S1',
    name='a1',
    data_array='data',
    iter_to_data='{[time,tri]->[k]: k=n1(tri)}')
#Similar definitions for a2 and a3
```

Figure 2.2: Example computation specification file (example_comp.spec)

```
#Include the example computation specification
iegen.include(’example_comp.spec’)

#CPACK data reordering
spec.add_transformation(
    type=iegen.trans.DataPermuteTrans,
    name=’cpack’,
    reordering_name=’sigma’,
    data_arrays=[’data’],
    iter_sub_space_relation=
           ’{[c0,time,c1,tri,x]->[tri] : c0=0 && c1=1}’,
    target_data_arrays=[’data’],
    erg_func_name=’cpack’)

#Locality Grouping iteration permutation
spec.add_transformation(
    type=iegen.trans.IterPermuteTrans,
    name=’locgroup’,
    reordering_name=’delta’,
    iter_sub_space_relation=’{[x,s,c1,i,y]->[i]: c1=1}’,
    target_data_arrays=[’data’],
    erg_func_name=’locgroup’,
    iter_space_trans=Relation(
           ’’’{[c0,time,c1,tri,x]->[c0,time,c1,out,x]:
                      c0=0 && c1=1 && out = delta(tri)}’’’))
```

Figure 2.3: Example transformation specification file (`example_trans.spec`)

2.2.2 Data Arrays

Data arrays are collections of data relevant to the computation and generally contain inputs to the computation or results that are computed during the computation. Only one data array is referenced in Figure 1.3, `data`, and it is defined as:

```
spec.add_data_array(
    name=’data’,
    type=’double *%s’,
    elem_size=’sizeof(double)’,
    bounds=’{[k]: 0<=k and k<N}’)
```

We specify the data array's name, type string, string for calculating the size of each element, and a set that represents the bounds of the array, typically from 0 to the length of the array. Data arrays are typically one dimensional for sparse codes and therefore so too are the bounds that are specificed for each data array in a spec file.

2.2.3 Index Arrays

Index arrays introduce an additional level of indirection to the accesses of data arrays and are what make irregular computations difficult to analyze and transform. The array `n1` is an index array in the example computation:

```
spec.add_index_array(
    name='n1',
    type='int *%s',
    input_bounds='{[k]: 0<=k and k<R}',
    output_bounds='{[k]: 0<=k and k<N}')
```

We define it in the spec file using its name, type string, input bounds as a set, and output bounds as a set. Though no relation needs to be specified explicitly for this index array, it is implied for `n1` to be:

$$\{[\mathrm{k}] \rightarrow [\mathrm{j}] : \mathrm{j} = \mathrm{n1(k)}\}$$

This shows the use of an uninterpreted function symbol, `n1`, in a relation. Just as with data arrays, index arrays are typically one dimensional in sparse codes.

2.2.4 Statements

We next define each statement in the computation. Specification of a statement requires three pieces of information: the statement's text, its iteration space, and its scheduling function. The statement `S1` in the example computation is specified as:

```
spec.add_statement(
    name=’S1’,
    text=’...data[%(a1)s]...;’,
    iter_space=’’’{[time,tri]:
               0<=time and time<T and 0<=tri and tri<R}’’’,
    scatter=’{[time,tri]->[c0,time,c1,tri,c0]:
               c0=0 and c1=1}’)
```

The name (S1) is used to refer to this particular statement when defining access relations (in the next section). The statement text defines the specific computation for the statement. Notice the presence of the place holder labeled as a1 in the statement text. This place holder signifies that we will be accessing the data array data but the exact expression for the access is not yet known. The iteration space of a statement is defined as a set where each dimension represents a loop iterator (in this case time and tri). Intuitively, the scheduling function for a statement defines the statement's position in the context of the whole computation. More details about the scheduling function are discussed in Chapter 4. Also note that this information has also been referred to as a scattering function and thus why the spec file uses this terminology.

2.2.5 Access Relations

Access relations define how statements access data arrays. For each statement, we may define zero or more place holders in the statement texts for access relations. For statement S1 in the spec file, we define the single access relation a1 as:

```
spec.add_access_relation(
    statement_name=’S1’,
    name=’a1’,
    data_array=’data’,
    iter_to_data=’{[time,tri]->[k]: k=n1(tri)}’)
```

An access relation is defined by specifying the statement it is associated with, its name in that statement, the data array it is accessing, and a relation that defines how it accesses that data array. The relation is a mapping from the whole iteration space of the computation to the domain of the data array that is being accessed. While transforming the computation, the access relations will be updated according to how the transformation modifies array accesses. The separation of array accesses from statements allows these updates to be made.

2.2.6 Data Dependences

We define data dependences as relations from points in the full iteration space to other points in the full iteration space. Data dependences are updated for each transformation that is applied based on how that transformation modifies the computation. As noted earlier, we assume an expert user and thus do not automatically extract data dependences. The data dependences must be specified explicitly by the user of IEGen.

2.2.7 Using Existing Transformations

The final component to define in a spec file is a sequence of transformations to apply to the computation. Figure 2.3 defines two transformations, a data permutation (cpack), and an iteration permutation (locgroup). IEGen currently supports data permutation transformations, iteration permutation transformations, and iteration alignment.

As an example, consider the specification of the data permutation transformation cpack in Figure 2.3:

```
spec.add_transformation(
    type=iegen.trans.DataPermuteTrans,
    name='cpack',
    reordering_name='sigma',
    data_arrays=['data'],
    iter_sub_space_relation=
          '{[c0,time,c1,tri,x]->[tri] : c0=0 && c1=1}',
    target_data_arrays=['data'],
    erg_func_name='cpack')
```

The type of transformation is defined as a `DataPermuteTrans`, which specifies a data permutation transformation. Other transformation types, such as `IterPermuteTrans`, are available as well. The name of the transformation, `cpack`, is used internally. The only requirement is that all transformation names be unique. The reordering name specifies the name of the uninterpreted function symbol that will be used at compile time to represent the reordering that will be calculated at run-time by the specified function name (`cpack` in this case). The data arrays collection defines the names of the previously specified data arrays that will be reordered based on this transformation. The target data arrays collection defines the names of the data arrays that will be used by the reordering heuristic to determine the new reordered array. Finally, the iteration subspace relation defines what subspace of the full computation iteration space will be examined for data array accesses.

2.2.8 Writing New Transformations

We are working towards supporting a broad class of transformations, but not all possibilities are currently supported. However, if the particular transformation you desire is not natively supported in IEGen, all is not lost. It is possible to implement custom transformations for use in IEGen. This process involves two components, a compile-time component and a run-time component.

Adding support for a new transformation at compile time consists of implementing a new transformation component based on the base `Transformation` class. `Transformation` is the base class of all transformations in IEGen. This class defines an interface of four methods that must be implemented when writing a new transformation:

- calc_inputs: This method calculates the compile-time inputs that are needed for the transformation. Additionally, any IDG nodes that need to be created for the input to the transformation will be created and added to the IDG here.

- calc_outputs: This method calculates the compile-time description of the inputs that are needed for the transformation. IDG nodes pertinent to the output from the transformation will be created and added to the IDG here.

- update_mapir: This method updates various components of the MapIR (such as statement access relations and scattering functions).

- update_idg: This method updates the IDG with new nodes that are related to this transformation.

Additionally, there are four fields that may be needed: `_set_fields`, `_relation_fields`, `_data_array_fields`, and `_data_arrays_fields`. These fields enable the IEGen spec files to be much more user friendly. You may notice that IEGen spec files are fairly syntactically simple. Sets and relations are specified as strings. Data arrays, symbolics, and statements are referenced by name. These features are supported in part by these four fields. Each should define a collection of field names (strings) that are names of parameters that are specified to the constructor of the transformation. For example, when one of these parameters is a Set, the `_set_fields` field should contain the name of this parameter. By including this name, users are able to specify only a string for the set, but IEGen knows that it should be created as a Set object before the transformation is instantiated. The relation and data array fields work similarly.

In addition to compile-time components, a new transformation needs to (potentially) provide a run-time component. For example, if the user is implementing a new sparse tiling method, the code that determines the tiles for a given input will need to be implemented.

One major component of the run-time portion of IEGen is a data structure called an explicit relation. Explicit relations are the runtime manifestation of compile-time relations. This data structure represents relations from tuples of one arity to tuples of a different arity, explicitly storing the tuple mappings. Any run-time functions will need to be written using explicit relations. Explicit relations provide the IEGen code generator and run-time transformation writers a common interface with which to communicate.

Note that it may be necessary for a user to provide either an additional compile-time transformation class, run-time code, or both. For example, if a user desires to use a new data permutation heuristic with IEGen, no compile-time components will need to be rewritten. However, a run-time function that implements this heuristic using explicit relations will need to be provided.

2.3 IEGen Middle End: Transforming the Computation

After the user specifies a computation and a sequence of transformations to apply to that computation, the middle end of IEGen can take over. The crux of IEGen's middle end is the transformation loop, which successively applies the specified transformations to the computation. The main transformation loop is expressed in Algorithm 1. For each transformation, we perform four main tasks. These tasks are implemented within the transformation using the methods `calc_input`, `calc_output`, `update_mapir`, and `update_idg`. The calculate input and output methods construct any necessary data structures used to represent runtime entities that are either input to or output from the transformation.

The update IDG and MapIR methods modify the data structures that represent the transformed computation. One of the main contributions of our work in this thesis is the inspector dependence graph (IDG). This is a data structure that represents the data and tasks that will be used and performed by the inspector. During the application of transformations to the computation, we build the IDG. More information about the IDG can be found in Chapter 3. In addition to creating and updating the IDG, we also manipulate the MapIR.

The result of the middle end of IEGen are two final data structures: the IDG and the MapIR. These two data structures represent the inspector and executor and will be used by the back end of IEGen to generate code.

```
for each transformation t do
    Calculate inputs to t;
    Calculate outputs from t;
    Update the MapIR based on t;
    Updated the IDG based on t;
end
```

Algorithm 1: Transformation algorithm

2.4 IEGen Back End: Generating Code

After transforming the computation, the final step is to generate code based on the data structures that represent the inspector and executor: the IDG and the MapIR. Code generation based on these two data structures is another main contribution of our work in this thesis and is covered in detail in Chapter 3. The result of the back end of IEGen is a source file containing functions for the inspector and the executor. This pair of functions can be called from driver code or be called as a replacement for the original computation.

2.5 IEGen Usage Summary

In summary, IEGen is a tool for specifying irregular computations, transforming these irregular computations, and generating the transformed inspector and executor code that implements the transformed computations. This chapter has described the major components of IEGen, including the input (spec files), the middle end that transforms the computation, the back end that generates the code, and the output (generated inspector and executor code). A user of IEGen thus needs to provide a spec file that defines an irregular computation. This spec file contains symbolic constants, data arrays, index arrays, statements, access relations for those statements, and data dependences between statement instances. The user also specifies a sequence of transformations to apply to this computation using existing transformations that are a part of IEGen or new transformations the user has created. When developing a new transformation, the user will typically provide a compile-time transformation component and a run-time reordering component. Once the spec file has been created, the user invokes IEGen with a command such as

```
iegen examples/example_trans.spec −o example.c
```

This command specifies that IEGen should read the spec file `examples/example_trans.spec`, transform the computation specified in that spec file, and write the generated code to the file `example.c`. The user may then replace the generated code with the untransformed computation in the application that the user is attempting to optimize.

Chapter 3

Inspector/Executor Code Generation

As noted in Chapter 1, one of the core contributions of this thesis is a code generator for inspectors and executors that implement a transformed irregular computation. This chapter discusses the details of our code generation techniques. We present three discussions related to the generation of inspectors and executors: a data structure called the inspector dependence graph (IDG) that we build while applying transformations to the computation (Sections 3.1 and 3.2), generation of inspector code based on the IDG (Section 3.3), and generation of executor code based on the MapIR (Section 3.4).

3.1 An Example Inspector Dependence Graph

We base the generation of inspector code on a new intermediate representation (IR) called an inspector dependence graph (IDG). The IDG represents data used by the inspector, tasks performed by the inspector, and dependences between the data and tasks. As an example, consider Figure 3.1, an IDG that corresponds to inspector code in Figure 3.2 that we previously presented in Chapter 1. The rectangular nodes represent data, the oval nodes represent tasks, and edges between nodes represent dependences between the nodes. For example, the call to `cpack`

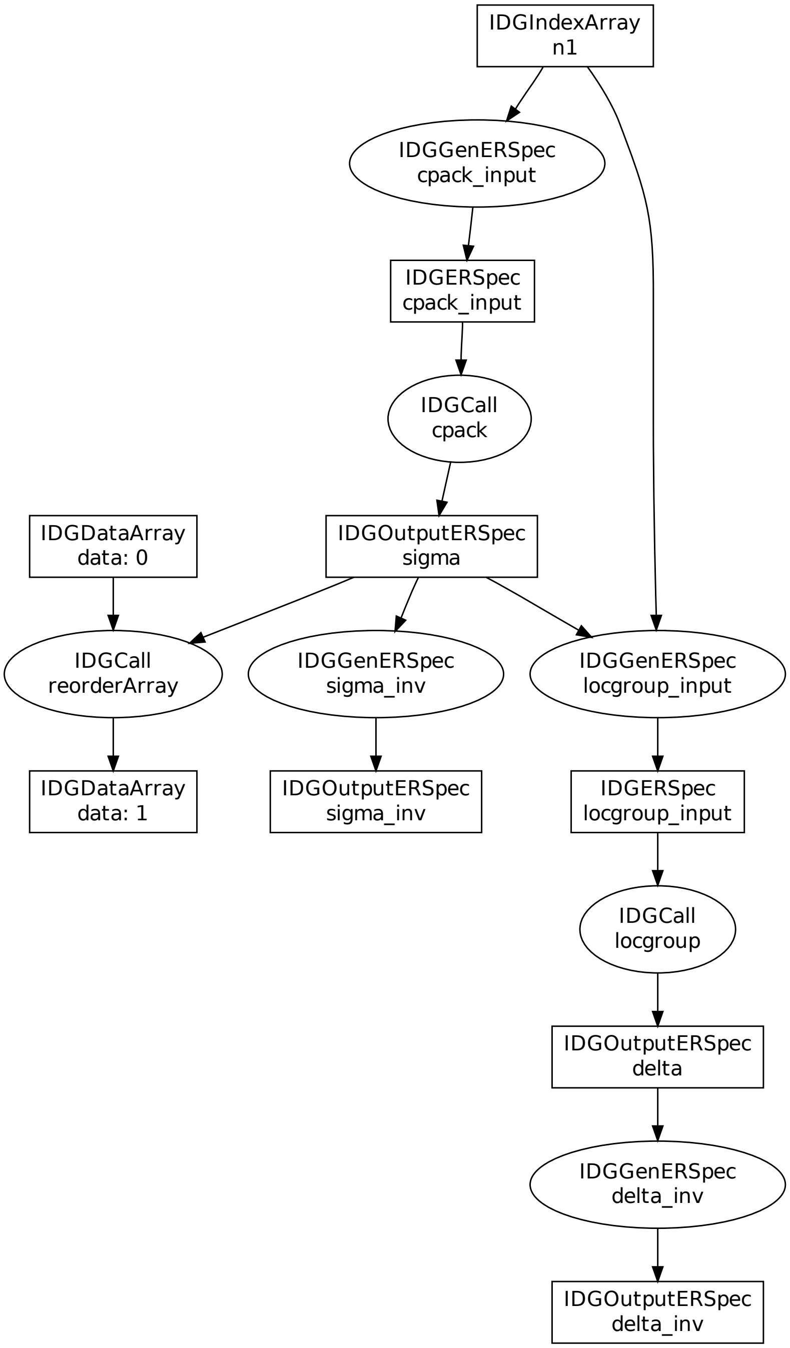

Figure 3.1: Example IDG for the example computation in Figure 3.2 after applying the cpack and locgroup transformations

```
/* Create input for consecutive packing */
for (tri=0;tri<=R-1;tri++) {
  cpack_input[tri]=n1[tri];
}

/* Call consecutive packing to create ’sigma’ */
cpack(cpack_input,sigma);

/* Create the inverse of ’sigma’, ’sigma_inv’ */
sigma_inv=genInverse(sigma);

/* Reorder the data array */
reorderArray(data,sigma);

/* Create input for locality grouping */
for (i=0;i<=R-1;i++) {
  locgroup_input[i]=sigma[n1[i]];
}

/* Call locality grouping to create delta */
locgroup(locgroup_input,delta);

/* Create ’delta_inv’ from ’delta’ */
delta_inv=genInverse(delta);

/* Partition the iterations of the tri loop */
proc=partition(R);
```

Figure 3.2: Inspector for the transformed irregular computation

in Figure 3.2 corresponds to the IDGCall task node for cpack in Figure 3.1. This call produces sigma, a data node.

The IDG in Figure 3.1 represents an inspector for two transformations: cpack and locgroup. cpack is a data reordering heuristic that reorders a data array to improve temporal and/or spatial locality based on the accesses to the data array. The input to cpack is an explicit relation that maps each iteration of the triangle loop in Figure 1.3 to the indices of the data array that are accessed in that iteration.

Based on this input, the function cpack determines a new order of elements in the data array. The index array n1 is used to calculate the input to cpack. The call to the function cpack produces the sigma explicit relation, which is then used to reorder the data array data and to create sigma_inverse. Notably, the task node labeled reorderArray takes data nodes sigma and data:0 as input and produces a new data node data:1 that is a reordered version of the array data. We annotate data arrays in the IDG with version numbers to signify that a node such as the call to reorderArray modifies the data array.

The second transformation expressed in the IDG in Figure 3.1 is locgroup, an iteration permutation transformation. Similarly to cpack, locgroup is a heuristic that attempts to improve data locality based on how statements within a loop access data. It differs in that it reorders the iterations of the loop rather than the data itself (this is actually achieved by reordering the entries in the index arrays). Just as with cpack, the input to locgroup is an explicit relation that maps iterations of a loop to the indices of the data array that are accessed. The result of the call to locgroup is the explicit relation delta. Finally, note that nodes for the symbolic constants T, R, and N are present in the IDG created by IEGen, but that we have omitted these nodes for clarity. The symbolic constant nodes that were omitted are data (rectangle) nodes that were input to every task (oval) node present in Figure 3.1 and thus introduced many edges that obscured other details of the IDG.

3.2 General Description of Inspector Dependence Graphs

We now discuss the inspector dependence graph (IDG) in a more general setting including the semantics of the IDG, its nodes, and its edges. At a high level, the

IDG is an abstract representation of an inspector. Specifically, the IDG represents tasks, data, and their interaction within the inspector. This means an IDG consists of two main types of nodes: task nodes and data nodes. Task nodes represent tasks the inspector will perform, such as calling a function. Data nodes represent data, such as index arrays and explicit relations, that will be used by tasks. The edges of the IDG thus represent the relationships (dependences) between data and tasks. A few simple statements define the rules for the existence of an edge between a data and a task node:

- An edge from a data node to a task node represents a data dependence and data consumption relationship within the inspector: The task cannot be performed until all incoming data is available—the task will make use of the data when it is performed.

- An edge from a task node to a data node signifies a data creation relationship: the task will calculate or produce the data.

- Edges from a data node to another data node or from a task node to another task should not exist and have no meaning.

It is useful to amend this more general discussion of the IDG semantics by a few more details concerning data nodes. We utilize Figure 3.1 to identify some of these details. First, nodes that represent data have one of the following lifetimes:

- *Input*: Some nodes represent data that is input to the inspector, such as symbolic constants and index arrays. The IDGIndexArray node `n1` in Figure 3.1 is an example of this situation. One may recognize these nodes easily as they are not the output of any task node in the IDG (i.e., have no incoming edges).

- *Output*: Some nodes represent data that is computed within the inspector and is needed by the executor once the inspector completes. The IDGOutputERSpec node `sigma` in Figure 3.1 is an example of this. This node represents the reordering calculated by a call to `cpack`. This information is needed by the executor once the inspector has completed. All IDGOutputERSpec nodes, and only nodes of this type, will be outputs from the inspector.

- *Internal*: Some nodes represent data that is computed within the inspector and are not needed after the inspector tasks are completed. The IDGERSpec node `cpack_input` in Figure 3.1 is an example of this. This node is needed as input to the call to `cpack` but is not needed again after this call. All data nodes that do not fall into the previous two lifetimes are members of this group.

These lifetimes are useful to know when generating code so that we may deallocate certain data structures when they are no longer needed. Internal nodes may be deallocated after the inspector is complete (or possibly earlier in the inspector when they are no longer useful). Output nodes are utilized by the executor and thus cannot be deallocated until after the executor completes.

Second, a special case exists for the way data arrays are represented in the IDG. Note in Figure 3.1 that two separate nodes refer to the data array `data`: one with the number 0 and one with the number 1. Both of these nodes refer to the same data array object, `data`, however they represent this data array in two different states. The IDGCall node for `reorderArray` permutes a data array given a permutation reordering explicit relation. They reorder the incoming data array to produce a data array with a new version number.

Creating an intermediate representation (IR) for the inspector provides us with various benefits, such as the ability to answer questions required for generating inspector code. Specifically, the IDG approach provides us with the following benefits:

- We can examine the IDG to determine the parameters to the inspector and necessary variable declarations using simple topological traversals of the IDG and maintaining information about the visited nodes.

- Language agnostic code generation can be done by traversing the IDG since it is not specific to any particular programming language. Note that this is conceptually possible but that our IDG implementation has certain language-specific information embedded in various nodes and thus we can currently only target C-like languages.

- Coarse-grain parallelism within the inspector may be discovered in the IDG. Since we explicitly express the tasks and data as individual units in the IDG, we may easily discover parallel tasks and thus execute them in parallel.

- We can perform effective memory management of data structures within the inspector and utilize global information about the composed inspector; it is obvious when a given memory allocation is no longer needed.

- We can express inspector optimizations (such as collapsing nested index array accesses) as transformations of the IDG, which makes them easier to understand and implement.

It is important to note that this work does not examine language-agnostic inspector code generation, code generation of parallel inspectors, or efficient memory management techniques within inspectors. However, as we are using the IDG-based approach, extending IEGen to support these ideas will be far easier than if we did not have this intermediate representation.

Table 3.1: IDG Node Types

Node Type	Description
IDGNode	Root node type (base class) for all IDG nodes (not directly present in IDG)
IDGSymbolic	Represents a symbolic scalar value
IDGIndexArray	Represents an index array
IDGDataArray	Represents a data array
IDGERSpec	Represents an explicit relation
IDGOutputERSpec	Represents an explicit relation that will be used outside of the inspector (usually the output of a reordering function, e.g. `cpack`)
IDGCall	Represents a call to a function

Finally, Table 3.1 summarizes the possible node types found in an IDG and their descriptions.

3.3 Inspector Code Generation

Inspector code generation is based on traversal of the inspector dependence graph (IDG). The IDG represents the data and tasks of an inspector. For code generation, we are able to examine the IDG and extract information about function parameters to the inspector, variable declarations, and tasks to be performed. Based on this information, we can generate the final inspector code.

Answering various questions about an IDG, such as ‘what parameters are needed by the inspector’, is made possible by a topological traversal of the nodes in the IDG. Algorithm 2 presents our algorithm for visiting the nodes of the IDG in a topologically-valid order. We have implemented the visitor design pattern allowing us to implement various visitors to answer the questions required for code generation orthogonally to the details of this algorithm. These visitors include:

- Topological Visitor: The base visitor class that implements Algorithm 2.

- Parameter Visitor: Determines and generates the parameter list that will be needed by the inspector function.
- Declaration Visitor: Determines and generates the variables declarations for the inspector.
- Code Generation Visitor: Generates the body of the inspector.
- Dot Printing Visitor: Prints a `.dot` file useful for visualizing an IDG.

For a given IDG, we can thus utilize the parameter, declaration, and code generation visitors to generate the inspector. Figure 3.2 is an example of code that will be generated based on the IDG in Figure 3.1.

```
depcounts=mapping from node to number of dependences;
while more nodes left in depcounts do
    nodeps=all nodes in depcounts with 0 dependences;
    if nodeps is empty then
        fail due to cycle in IDG;
    end
    for each node n in nodeps do
        visit n;
        reduce the dependence count for nodes that depend on n;
        remove n from depcounts;
    end
end
```

Algorithm 2: IDG topological visit algorithm

3.4 Executor Code Generation

We now discuss the details of executor code generation, which is more straightforward than code generation for the inspector and is based on the MapIR data structure. Overall, we must perform two distinct tasks: generate loops for the

```
#define S0 ...data[sigma[n1[delta_inv[tri]]]]...
#define S1 ...data[sigma[n2[delta_inv[tri]]]]...
#define S2 ...data[sigma[n3[delta_inv[tri]]]]...

 for(time=0; time<T; time++) {
   ...
   for(p=0; p<NUM_PROCS; p++) {
     for(tri=0; tri<R; tri++) {
       if(p==proc(tri)) {
         ...
         S0;
         S1;
         S2;
         ...
       }
     }
   }
   ...
 }
```

Figure 3.3: Executor for transformed irregular computation

transformed computation (Section 3.4.1) and generate statements for the transformed computation (Section 3.4.2) Transforming the computation produces a modified MapIR (labeled as MapIR' in Figure 2.1) that represents the code for the transformed executor. The relevant pieces of information contained in the MapIR for each statement include iteration spaces, scattering functions, and access relations. Executor code generation thus seeks to generate loops that scan the full iteration space for all statements and statement text filled in with transformed access relation expressions. The previously presented executor code in Figure 3.3 is an example of code that will be generated.

3.4.1 Executor Loop Generation

We utilize CLooG [3, 4, 5] for generating a loop nest that scans all points in the full iteration space for the computation. For example, the loops for the iterators time, p, and tri in Figure 3.3 were generated by CLooG. After transformation, each statement will have a scattering function that is a relation that places the statement in the full context of the computation based on the statement's iteration space. We translate all statement iterations spaces and scattering functions into a format that CLooG accepts.

Chapter 4 presents our techniques for translating the sets and relations that represent the iteration spaces and scattering functions for input to CLooG. Specifically, we present techniques for manipulating set and relation constraints with uninterpreted function symbols into a form that CLooG can handle. One of the manipulations that we must perform is to introduce affine bounds on the iterator p for the transformed iteration space of the first statement in the triangle loop in Figure 3.3. The constraint p=proc(tri) is not supported by CLooG and thus must be handled by our techniques before being passed to CLooG for loop generation.

The output from CLooG is a complete loop nest that scans the full iteration space for all statements. Within the loop nest CLooG inserts statement place-holders (such as S0 in Figure 3.3) in the appropriate positions based on how it generated the nesting. We use this loop nest as the main loop for the executor.

3.4.2 Executor Statement Generation

The statement place-holders that CLooG inserts into the generated loop nest are where the text for each statement will be inserted. Additionally, a statement's text may have been specified with one or more 'holes' for array accesses (access relations). For example, the statement text for S1 is defined in Figure 2.2 as '...data[%(a1)s]...; which contains a hole denoted by %(a1)s. This hole corresponds to the access relation named a1.

After the computation has been transformed, the final access relations define how a statement accesses a data array for each of these holes. For statement S1, the access relation `a1` is

$$\{[\mathrm{c_0}, \mathrm{time}, \mathrm{c_1}, \mathrm{k}, \mathrm{c_2}] \to [\mathrm{out}] : \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{delta}^{-1}(\mathrm{k})))\}.$$

The details of how we obtain this relation are discussed in Chapter 4. Generating the statements for the executor thus involves determining the expression that the single output tuple variable is equal to for each of the access relations. In this example, we must determine that the output tuple variable `out` is equal to the expression $\mathtt{sigma}(\mathtt{n1}(\mathtt{delta}^{-1}(\mathtt{k})))$. We then replace the holes in the statement texts with the appropriate expression string.

Chapter 4

Enabling Code Generation in the Sparse Polyhedral Framework

This chapter discusses our techniques for manipulating sets and relations with uninterpreted function symbols to enable code generation. Existing libraries and techniques such as Fourier-Motzkin elimination do not support projecting variables out of constraints involving uninterpreted function symbols. This chapter discusses techniques to project out such existentials and to introduce affine bounds to enable code generation for inspectors and executors (Section 4.2). Before introducing these techniques, in Section 4.1 we discuss the mathematical framework this work is based on. Section 4.3 evaluates these techniques by comparing our implementation's performance and ability to project out existentials to Omega, a similar library.

4.1 The Sparse Polyhedral Framework (SPF)

Strout et al. originally introduced the sparse polyhedral framework in [21] 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 introduce a new problem that arises in this context. The SPF provides a mathematical framework for representing and transforming irregular computations in a way that is analogous to the polyhedral model for regular computations. SPF builds on the work discussed by Kelly and Pugh in [13], which is based on the concept of Presburger sets and relations. The Kelly and Pugh framework has the ability to express computations and transformations in the polyhedral model as well non-affine memory references and control flow using uninterpreted function symbols (UFSs) [20]. The key addition to the SPF over the Kelly and Pugh framework is the usage of UFSs to express run-time entities such as index arrays and run-time reordering transformations at compile-time and the requirement that the input and output domains of UFSs be specified to enable code generation. We now review the concept of sets and relations, a few operations on these sets and relations, and finish with a discussion of the problems that inhibit code generation.

4.1.1 Sets

We use sets to represent computation spaces. A set represents an unordered collection of integer tuples in $\mathbb{Z}^m$. For the example in Figure 1.3, the computation space is specified with the set

$$\mathrm{I} = \{[\mathrm{time}, \mathrm{tri}] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R})\}.$$

In general sets have the form

$$s = \{[x_1, \ldots, x_m] : c_1 \wedge \ldots \wedge c_n\}, \tag{4.1}$$

where each x_i is a tuple variable/iterator and each c_j is a constraint. The constraints in a set are equalities and inequalities that are affine expressions involving the tuple variables, symbolic constants, and existentials. A symbolic constant represents a constant value that does not change during the computation, but may not be known until runtime. An existential is any variable in the constraints that is not a tuple variable or a symbolic constant. Our definition of sets supports a single level of existential quantification and no support for universal quantifiers. Tuple variables and existentials are existentially quantified variables. The above set s is said to have *arity* m as it has m tuple variables.

Sets can also be unions of collections of integer tuples and have the form

$$s = \{\vec{x} : C_1\} \vee \{\vec{x} : C_2\} \vee \ldots \vee \{\vec{x} : C_p\}, \tag{4.2}$$

where $\vec{x}$ represents the vector of tuple variables and each C_i represents the constraints for each component of the union. We refer to the complete union of multiple sets as a disjunction and each individual component of the union as a conjunction. This arises from the fact that the constraints of a set are in disjunctive normal form.

Sets in the SPF can also have constraints with uninterpreted function symbol (UFS) expressions. The following iteration space for the example computation in Figure 1.5 contains the UFS `proc`:

$$\mathrm{I}' = \{[\mathrm{time}, \mathrm{p}, \mathrm{tri}] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge \mathrm{p} = \mathrm{proc}(\mathrm{tri}) \wedge (0 \leq \mathrm{tri} < \mathrm{R})\}.$$

An uninterpreted function symbol has the form $f(a_1, a_2, \ldots, a_m)$, where f is the name of the function and a_1 through a_m are the m arguments to the function. Just as with constraints, the arguments are affine expressions involving the tuple variables, symbolic constants, existentials, and other uninterpreted function symbols (allowing for the possibility of nested functions). Since a UFS is a function, it holds that for a UFS f, if $i = j$ the $f(i) = f(j)$. We use UFSs to represent entities in irregular computations that will not be known until runtime, such as index arrays (`n1`) and run-time reorderings (`proc`). The domain and range of a UFS must be a union of polyhedra and thus cannot be expressed in terms of other UFSs. Also, the arguments to UFS must be UFS expressions (i.e., affine and UFS) of tuple variables and symbolic constants. An existential can only be an argument to a UFS that is a bijection with only one argument.

4.1.2 Relations

We use relations to represent memory access functions, scheduling functions, and transformation functions. A relation represents an unordered mapping of integer tuples from $\mathbb{Z}^m$ to $\mathbb{Z}^n$. For the example in Figure 1.3, the first access to the data array `data[]` is defined by the relation

$$\mathrm{A_{orig}} = \{[\mathrm{time}, \mathrm{tri}] \rightarrow [\mathrm{k}] : \mathrm{k} = \mathrm{n1}(\mathrm{tri})\}.$$

This access function defines the indices of the data array `data[]` that are accessed given the loop iterators `time` and `tri`. Notice here the use of the UFS `n1`, an example of using a UFS to represent an *index array.* Index arrays introduce a layer of indirection when accessing a data array and thus only at runtime can we know exactly what elements of the data array are accessed.

In general relations have the form

$$r = \{[x_1, \ldots, x_m] \rightarrow [y_1, \ldots, y_n] : c_1 \wedge \ldots \wedge c_p\}, \tag{4.3}$$

where each x_i is an input tuple variable, each y_j is an output tuple variable, and each c_k is a constraint. The constraints of a relation follow the same restrictions as set constraints. The above relation r is said to have an *input arity* of m and an *output arity* of n as it has m input tuple variables and n output tuple variables.

Unions of relations are also possible and have the following form (similar to unions of sets)

$$\{\vec{x} \rightarrow \vec{y} : C_1\} \vee \{\vec{x} \rightarrow \vec{y} : C_2\} \vee \ldots \vee \{\vec{x} \rightarrow \vec{y} : C_q\}, \tag{4.4}$$

where $\vec{x}$ is the vector of input tuple variables, $\vec{y}$ is the vector of output tuple variables, and each C_i represents the constraints for each component in the union.

As a second example, the scheduling/scattering [5, 3] function for the statement `S1` in Figure 1.3 is defined by the relation

$$\mathrm{S} = \{[\mathrm{time}, \mathrm{tri}] \rightarrow [0, \mathrm{time}, 1, \mathrm{tri}, 0]\},$$

where we have denoted that the first and fifth output tuple variables are equal to zero and the third is equal to one. A scheduling function maps iteration points of a specific loop nest to points in an iteration space for the whole computation [13, 1, 5, 3]. The lexicographic order of the points in the full iteration space defines the order of execution for the whole computation.

Figure 1.5 shows a transformed version of the original computation after applying three transformations: consecutive packing, locality grouping, and a partitioning. Consecutive packing (`cpack`) and locality grouping (`locgroup`) are data and iteration reordering heuristics used to improve data locality by optimizing the order of data accesses and were introduced by Ding and Kennedy [11]. In this example, we permute the data array `data[]` and permute the iterations of the tri loop based on the run-time computed permutations determined by `cpack` and `locgroup`. The partitioning introduces potential parallelism into the computation by adding a processor loop and a guard that ensures that the computation for each triangle is only executed on the processor that the triangle is mapped to (this is commonly referred to as an owner-computes strategy of parallelism). We note that the introduction of the processor 'guard' is sub-optimal. We are developing techniques for removing such guards, but the discussion of these techniques is beyond the scope of this thesis. This loop could be annotated using an OpenMP `#pragma` to introduce parallelism across multiple processors. We represent these three transformations in the SPF using the following relations:

$$\begin{aligned}
\mathrm{R}_{\texttt{cpack}} &= \{[\mathrm{in}] \rightarrow [\mathrm{out}] : \mathrm{out} = \mathrm{sigma}(\mathrm{in})\}, \\
\mathrm{T}_{\texttt{locgroup}} &= \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{tri}, \mathrm{c}_2] \rightarrow [\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{k}, \mathrm{c}_2] : \mathrm{k} = \mathrm{delta}(\mathrm{tri})\}, \text{ and} \\
\mathrm{T}_{\texttt{part}} &= \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{tri}, \mathrm{c}_2] \rightarrow [\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{p}, \mathrm{c}_2, \mathrm{tri}, \mathrm{c}_3] : \mathrm{p} = \mathrm{proc}(\mathrm{tri})\}.
\end{aligned}$$

Note the use of the UFSs `sigma`, `delta`, and `proc` to represent mappings that will not be known until runtime.

4.1.3 The Apply Operation

Building on our definitions of sets and relations from the previous sections, next we describe three operations on sets and relations that are required by our transformation framework: first we discuss the apply operation in this section followed by the inverse and compose operations in Section 4.1.4. To generate code a transformation framework must be able to apply a schedule to a statement's iteration space to determine the statement's position in the context of the full computation. A statement's iteration space is represented with a set (e.g., $\mathrm{I} = \{[\mathrm{time}, \mathrm{tri}] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R})\}$). The scheduling function is represented as a relation (e.g., $\mathrm{S} = \{[\mathrm{time}, \mathrm{tri}] \rightarrow [0, \mathrm{time}, 1, \mathrm{tri}, 0]\}$). By applying the original scheduling function to the statement's iteration space, we can derive the full iteration space for that statement, including the nesting and ordering of that statement relative to other statements. For example:

$$\begin{aligned}
&\mathrm{F} = \mathrm{S}(\mathrm{I}) \\
&\quad = \{[\mathrm{time}, \mathrm{tri}] \rightarrow [0, \mathrm{time}, 1, \mathrm{tri}, 0]\}(\ \{[\mathrm{time}, \mathrm{tri}] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R})\}\ ) \\
&\quad = \{[0, \mathrm{time}, 1, \mathrm{tri}, 0] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R})\}.
\end{aligned}$$

In general, the resulting set s of the apply operation $s = r_1(s_1)$ has the same tuple variables as the output tuple from the relation r_1, has constraints from both the relation r_1 and the input set s_1, and additionally has constraints that the input tuple variables from r_1 are pairwise equal to the tuple variables of s_1. Mathematically, the apply operation is defined as $(\vec{x} \in s) \iff (\exists \vec{y} \text{ s.t. } \vec{y} \in s_1 \wedge \vec{y} \rightarrow \vec{x} \in r_1)$. The general form of the apply operation is

$$\begin{aligned} s =& r_1(s_1) \\ =& \{[y_1, \dots, y_m] \rightarrow [z_1, \dots, z_n] : c_1 \wedge \dots \wedge c_k\}(\\ & \quad \{[x_1, \dots, x_m] : c_{k+1} \wedge \dots \wedge c_{j+k+1}\}\ ) \\ =& \{[z_1, \dots, z_n] : c_1 \wedge \dots \wedge c_{k+j+1} \wedge x_1 = y_1 \wedge \dots \wedge x_m = y_m\}. \end{aligned} \tag{4.5}$$

The apply operation is only legal when the arity of s_1 matches the input arity of r_1. The x_i and y_j variables become existentials in the resulting set, which later simplification will project out. If s_1 and/or r_1 involve more than one conjunction, then the above operation is performed on the Cartesian product of the two disjunctions of conjunctions. Note that in the following formulas, C_i and D_i denote conjunctions of constraints rather than single constraints:

$$r_1 = \{\vec{y} \rightarrow \vec{z} : C_1\} \vee \{\vec{y} \rightarrow \vec{z} : C_2\} \vee \dots \vee \{\vec{y} \rightarrow \vec{z} : C_k\}, \tag{4.6}$$

$$s_1 = \{\vec{x} : D_1\} \vee \{\vec{x} : D_2\} \vee \dots \vee \{\vec{x} : D_j\},$$

$$\begin{aligned} s =& r_1(s_1) \\ =& \{\vec{z} : C_1 \wedge D_1 \wedge \vec{x} = \vec{y}\} \vee \{\vec{z} : C_1 \wedge D_2 \wedge \vec{x} = \vec{y}\} \vee \dots \vee \{\vec{z} : C_k \wedge D_j \wedge \vec{x} = \vec{y}\}. \end{aligned}$$

4.1.4 The Inverse and Compose Operations

We now describe both the inverse and compose operations. Just as with iteration spaces, a transformation framework must be able to place access functions within the whole context of the computation being represented and manipulated. The access function $\mathrm{A_{orig}} = \{[\mathrm{time}, \mathrm{tri}] \rightarrow [\mathrm{k}] : \mathrm{k} = \mathrm{n1}(\mathrm{tri})\}$ is specified in terms of only the iterators of a single statement (in this case `time` and `tri`). Using the scheduling function for the statement (e.g., $\mathrm{S} = \{[\mathrm{time}, \mathrm{tri}] \rightarrow [0, \mathrm{time}, 1, \mathrm{tri}, 0]\}$) with the inverse and compose operations allows us to construct an access function that is in terms of the full iteration space `F`. The scheduling function is a relation from the iterations of a single statement to the full iteration space. We use the inverse operation to produce a new relation from the full iteration space to the iterators for the single statement. Finally, we compose the original access function (a relation from the single statement iterators to an array position) with this new relation to produce the full access function from the full iteration space to an array position:

$$\begin{aligned}
\mathrm{A_{full}} =& \mathrm{A_{orig}}(\mathrm{S}^{-1}) \\
=& \{[\mathrm{time}, \mathrm{tri}] \rightarrow [\mathrm{k}] : \mathrm{k} = \mathrm{n1}(\mathrm{tri})\}(\ (\{[\mathrm{time}, \mathrm{tri}] \rightarrow [0, \mathrm{time}, 1, \mathrm{tri}, 0]\})^{-1}\ ) \\
=& \{[\mathrm{time}, \mathrm{tri}] \rightarrow [\mathrm{k}] : \mathrm{k} = \mathrm{n1}(\mathrm{tri})\}(\ \{[0, \mathrm{time}, 1, \mathrm{tri}, 0] \rightarrow [\mathrm{time}, \mathrm{tri}]\}\ ) \\
=& \{[0, \mathrm{time}, 1, \mathrm{tri}, 0] \rightarrow [\mathrm{k}] : \mathrm{k} = \mathrm{n1}(\mathrm{tri})\}.
\end{aligned}$$

In general, the resulting relation r of the inverse operation $r = r_1^{-1}$ is a relation with the same constraints as r_1 but with input and output tuple variables swapped. Mathematically, the inverse operation is defined as $(\vec{x} \rightarrow \vec{y} \in r) \iff (\vec{y} \rightarrow \vec{x} \in r_1)$. In general, inverse has the form

$$\begin{aligned}
r =& r_1^{-1} \qquad (4.7) \\
=& (\{\vec{x} \rightarrow \vec{y} : C_1\} \vee \{\vec{x} \rightarrow \vec{y} : C_2\} \vee \ldots \vee \{\vec{x} \rightarrow \vec{y} : C_k\})^{-1} \\
=& \{\vec{y} \rightarrow \vec{x} : C_1\} \vee \{\vec{y} \rightarrow \vec{x} : C_2\} \vee \ldots \vee \{\vec{y} \rightarrow \vec{x} : C_k\}.
\end{aligned}$$

The calculation of the full access function also utilizes the compose operation. The resulting relation r of the compose operation $r = r_1(r_2)$ has the input tuple variables from r_2 and the output tuple variables from r_1, has constraints from both relations r_1 and r_2, and additionally has constraints that the output tuple variables from r_2 are pairwise equal to the input tuple variables from r_1: Mathematically, the compose operation is defined as $(\vec{x} \to \vec{y} \in r) \iff (\exists \vec{z}\,\text{s.t. } \vec{x} \to \vec{z} \in r_2 \wedge \vec{z} \to \vec{y} \in r_1)$. In general, compose has the form

$$\begin{aligned} r =& r_1(r_2) \\ =& \{[d_1, \ldots, d_i] \to [e_1, \ldots, e_n] : c_{k+1} \wedge \ldots \wedge c_{j+k+1}\}(\\ & \quad \{[a_1, \ldots, a_m] \to [b_1, \ldots, b_i] : c_1 \wedge \ldots \wedge c_k\}) \\ =& \{[a_1, \ldots, a_m] \to [e_1, \ldots, e_n] : c_1 \wedge \ldots \wedge c_{k+j+1} \wedge b_1 = d_1 \wedge \ldots \wedge b_i = d_i\}. \end{aligned} \tag{4.8}$$

The compose operation is only legal when the output arity of r_2 is equal to the input arity of r_1. The b_i and c_i variables become existentials in the resulting set, which later simplification will attempt to project out. If r_1 and/or r_2 involve more than one conjunction, then the above operation is performed on the Cartesian product of the two collections of conjunctions as follows:

$$r_1 = \{\vec{d} \to \vec{e} : F_1\} \vee \{\vec{d} \to \vec{e} : F_2\} \vee \ldots \vee \{\vec{d} \to \vec{e} : F_j\}, \tag{4.9}$$

$$r_2 = \{\vec{a} \to \vec{b} : C_1\} \vee \{\vec{a} \to \vec{b} : C_2\} \vee \ldots \vee \{\vec{a} \to \vec{b} : C_k\},$$

$$\begin{aligned} r =& r_1(r_2) \\ =& \{\vec{a} \to \vec{e} : C_1 \wedge F_1 \wedge \vec{b} = \vec{d}\} \vee \{\vec{a} \to \vec{e} : C_1 \wedge F_2 \wedge \vec{b} = \vec{e}\} \vee \\ & \ldots \vee \{\vec{a} \to \vec{e} : C_k \wedge F_j \wedge \vec{b} = \vec{d}\}. \end{aligned}$$

4.1.5 The Problems Arise: Transforming the Computation

This section presents two examples of situations where an operation produces a set or relation for which we are unable to generate code using existing code generation techniques. We have shown in Sections 4.1.1, 4.1.2, 4.1.3, and 4.1.4 how to represent the computation in Figure 1.3 and how to manipulate this representation to prepare it for transformation. Now we show that applying the transformations $\mathtt{R_{cpack}}$, $\mathtt{T_{locgroup}}$, and $\mathtt{T_{part}}$ to the data and the computation results in sets and relations that inhibit code generation.

We first apply the consecutive packing transformation to the computation. This involves composing the access function transformation (e.g., $\mathrm{R}_{\mathtt{cpack}} = \{[\mathrm{in}] \to [\mathrm{out}] : \mathrm{out} = \mathrm{sigma}(\mathrm{in})\}$) with the full access function for the statement (e.g., $\mathrm{A}_{\mathrm{full}} = \{[0, \mathrm{time}, 1, \mathrm{tri}, 0] \to [\mathrm{k}] : \mathrm{k} = \mathrm{n1}(\mathrm{tri})\}$) to derive the transformed access function:

$$\begin{aligned}\mathrm{A}'_{\mathrm{full}} =& \mathrm{R}_{\mathtt{cpack}}(\mathrm{A}_{\mathrm{full}})\\ =& \{[0, \mathrm{time}, 1, \mathrm{tri}, 0] \to [\mathrm{out}] : \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{tri}))\}.\end{aligned}$$

When applying an iteration permutation RTRT such as locality grouping, we must update the access functions of the statements that are within the loop whose iterations are being permuted using a transformation relation (e.g., $\mathrm{T}_{\mathtt{locgroup}} = \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{tri}, \mathrm{c}_2] \to [\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{p}, \mathrm{c}_2, \mathrm{tri}, \mathrm{c}_3] : \mathrm{p} = \mathrm{proc}(\mathrm{tri})\}$). As an example, consider the following operations that update $\mathrm{A}'_{\mathrm{full}}$:

$$\begin{aligned}\mathrm{A}''_{\mathrm{full}} =& \mathrm{A}'_{\mathrm{full}}(\mathrm{T}^{-1}_{\mathtt{locgroup}})\\ =& \{[0, \mathrm{time}, 1, \mathrm{tri}, 0] \to [\mathrm{out}] : \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{tri}))\}(\\ & \quad \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{k}, \mathrm{c}_2] \to [\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{tri}, \mathrm{c}_2] : \mathrm{k} = \mathrm{delta}(\mathrm{tri})\})\\ =& \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{k}, \mathrm{c}_2] \to [\mathrm{out}] : \mathrm{k} = \mathrm{delta}(\mathrm{tri}) \wedge \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{tri}))\}.\end{aligned}$$

Notice this operation results in the presence of the existential `tri` as the input to the UFS `delta`. This is a problem as existing techniques such as Fourier-Motzkin do not support projecting out existentials that are inputs to UFSs. To generate efficient code for this access function, we need the output tuple variable `out` expressed as a function of the input tuple variables.

Finally, we apply a partitioning transformation (e.g., $\mathrm{T}_{\texttt{part}} = \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{tri}, \mathrm{c}_2] \rightarrow [\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{p}, \mathrm{c}_2, \mathrm{tri}, \mathrm{c}_3] : \mathrm{p} = \mathrm{proc}(\mathrm{tri})\}$) to the computation by updating the iteration spaces of the affected statements (e.g., $\mathrm{F} = \{[0, \mathrm{time}, 1, \mathrm{tri}, 0] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R})\}$):

$$\begin{aligned}
\mathrm{F}' =& \mathrm{T}_{\texttt{part}}(\mathrm{F}) \\
=& \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{tri}, \mathrm{c}_2] \rightarrow [\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{p}, \mathrm{c}_2, \mathrm{tri}, \mathrm{c}_3] : \mathrm{p} = \mathrm{proc}(\mathrm{tri})\}(\\
& \quad \{[0, \mathrm{time}, 1, \mathrm{tri}, 0] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R})\}) \\
=& \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{p}, \mathrm{c}_2, \mathrm{tri}, \mathrm{c}_3] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R}) \wedge \mathrm{p} = \mathrm{proc}(\mathrm{tri})\}.
\end{aligned}$$

Note the fact that the iterator `p` does not have explicit affine bounds but rather is equal to the output of the UFS `proc`. In order to generate code that iterates over all points in this set, we need affine bounds for each iterator/tuple variable.

The application of both the locality grouping and partitioning transformations resulted in situations where code generation is inhibited. In the first case, the resulting access function contains a constraint where an existential is an input to a UFS. In the second case, we do not have explicit bounds on a loop iterator. These types of situations inhibit the generation of polyhedral scanning loop nests by tools such as CLooG [3, 4, 5] or the generation of expressions for array accesses based on access functions. Section 4.2 introduces techniques to rectify these situations to enable code generation.

4.2 Constraint Simplification in the SPF

Section 4.1 showed examples of how a computation can be represented and transformed using the Sparse Polyhedral Framework. After applying RTRTs to the computation it is possible to produce constraints that contain existentials as inputs to UFSs (e.g., $\mathtt{A}''_{\mathtt{full}}$) or tuple variables without explicit affine bounds (e.g., $\mathtt{F}'$)—situations that inhibit code generation. This section introduces remedies for these situations.

We identify three general situations where code generation is inhibited. In the following, let v be an existential, let t be a tuple variable, let f be a UFS:

- $t = f(v)$: An existential is an input to a UFS,
- $v = \text{expr}$: An existential is equal to an affine expression, possibly containing a UFS instance, and
- $t = f(\ldots)$: A tuple variable is equal to a UFS instance and no affine bounds are present for that variable in other constraints.

These situations were generalized from specific instances of each that arose during our work with the SPF on an example benchmark. This list is thus necessary but may not be sufficient for enabling code generation—there may be other situations involving UFSs that we have not encountered that make it difficult to generate code. We examine each of these general cases in the following subsections.

4.2.1 Inverse Function Simplification

When performing operations on sets and relations to represent and transform computations in the SPF, it is possible to introduce an existential that is an input to a UFS. We have shown that this is possible by applying the locality grouping iteration permutation RTRT to our example computation. Specifically, we produced the following access function:

$$\mathrm{A}''_{\mathrm{full}} = \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{k}, \mathrm{c}_2] \to [\mathrm{out}] : \mathrm{k} = \mathrm{delta}(\mathrm{tri}) \wedge \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{tri}))\}.$$

Notice in the constraint k = delta(tri) that the argument to the function `delta`, `tri`, is an existential. Therefore we need to manipulate these constraints in some way so that we can remove the existential.

If we are given the fact that the UFS `delta` is invertible (with its inverse named $\texttt{delta}^{-1}$ for example), we can utilize this information to remove the existential from the constraints. More specifically, we can utilize information that `delta` is a bijection to perform an obvious simplification on the equality constraint. We often deal with permutations such as those that are the result of `cpack` or `locgroup`. Since permutations are bijections, they can be inverted automatically at runtime. For this simplification to apply, we only require to know that a given UFS is a bijection and what the inverse of the UFS is named.

When we apply this simplification rule to the constraint in the relation $\mathrm{A}''_{\mathrm{full}}$, we obtain the following relation:

$$\mathrm{A}''_{\mathrm{full}} = \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{k}, \mathrm{c}_2] \to [\mathrm{out}] : \mathrm{delta}^{-1}(\mathrm{k}) = \mathrm{tri} \wedge \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{tri}))\}.$$

This relation is now one step closer to having all existentials removed from the constraints. At this point, assuming we knew `sigma` and `n1` were also bijections, we could apply the inverse simplification rule twice to the constraint $\texttt{out} = \texttt{sigma}(\texttt{n1}(\texttt{tri}))$ to obtain the constraint $\texttt{n1}^{-1}(\texttt{sigma}^{-1}(\texttt{out})) = \texttt{tri}$. In order to generate code, however, we require constraints be in terms of the output tuple variable `out`. Additionally, the permutability of `n1` is not known and therefore we could not apply this rule for this UFS to expose the existential `tri`. The next section will discuss how we completely remove the existential `tri` from the remaining constraints using a different approach.

In general, the inverse function simplification applies to equality constraints of the form:

$$t = f(v),$$

where t is a tuple variable, v is an existential, and we know that f is a function that admits an inverse and that inverse is f^{-1}. In this situation, we can transform the constraint into:

$$f^{-1}(t) = v,$$

to produce a constraint with tuple variable t as input to the inverse function f^{-1}.

We show that this is a legal constraint manipulation using the fact that f admits an inverse:

$$\begin{aligned} t =& f(v) \\ \implies \quad f^{-1}(t) =& f^{-1}(f(v)) \\ \implies \quad f^{-1}(t) =& v. \end{aligned} \tag{4.10}$$

4.2.2 Existential Equality Simplification

We utilize a second simplification rule to remove existentials involved in equalities with UFSs. The use of the inverse function simplification in the previous section produced the following relation:

$$\mathrm{A''_{full}} = \{[\mathrm{c_0, time, c_1, k, c_2}] \to [\mathrm{out}] : \mathrm{delta^{-1}(k) = tri \wedge out = sigma(n1(tri))}\}.$$

This relation still contains the existential `tri` We remove this variable using the existential equality simplification. If an existential is present in an equality constraint, we can remove that constraint and replace all instances of that variable's equivalent value. In this example, we replace tri with $\mathrm{delta}^{-1}(\mathrm{k})$ to produce the following relation:

$$\mathrm{A}''_{\mathrm{full}} = \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{k}, \mathrm{c}_2] \rightarrow [\mathrm{out}] : \mathrm{out} = \mathrm{sigma}(\mathrm{n1}(\mathrm{delta}^{-1}(\mathrm{k})))\}.$$

After applying this simplification, we have removed all existentials from the constraints of this relation and thus may generate code for this access function.

In general, the existential equality simplification can be used with equality constraints of the form

$$v = \mathrm{expr},$$

where v is an existential and expr is any expression, possibly involving UFSs (though this is not required). If an equality constraint of this form is found, we can:

1. remove the equality constraint from the collection of constraints of the set or relation and
2. replace all instances of the existential v with the expression that v is equal to (expr).

The legality of this simplification can be ensured by considering the fact that we apply this simplification to equality constraints. Since the existential in question is equal to the expression we are replacing it with, we know that simply removing that constraint and modifying all other constraints that reference this existential is legal. Note that we do not apply this simplification rule in situations where the existential in question is not used in other constraints.

4.2.3 Affine Approximation

In this section, we introduce a method for introducing affine bounds for tuple variables that are constrained by an equality with a UFS. Consider the following full iteration space that we derived at the end of Section 4.1.5:

$$\mathrm{F}' = \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{p}, \mathrm{c}_2, \mathrm{tri}, \mathrm{c}_3] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R}) \wedge \mathrm{p} = \mathrm{proc}(\mathrm{tri})\}.$$

One method of code generation requires that all iterators of an iteration space have affine bounds. However, note that the tuple variable `p` is only involved in an equality constraint that contains a UFS.

To fix the situation when a tuple variable has no explicit affine bounds, we utilize additional information about the UFS to which it is equal. Specifically, we note that if we know the range of the UFS `proc` in the above example, we can introduce inequalities that bound the tuple variable `p`:

$$\begin{aligned}\mathrm{F}' =& \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{p}, \mathrm{c}_2, \mathrm{tri}, \mathrm{c}_3] : \\ & (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R}) \wedge \mathrm{p} = \mathrm{proc}(\mathrm{tri}) \wedge (0 \leq \mathrm{p} < \mathrm{n\_procs})\}.\end{aligned}$$

Here we have introduced two inequalities that bound `p` from above and below. We utilize the information that the UFS `proc` ranges from `0` to `n_procs` to allow us to approximate the value of `p`. After this approximation, we are able to generate a loop that scans the range of possible values for `p`.

In general, suppose we have a constraint of the form

$$t = f(...),$$

where t is a tuple variable, f is some UFS for which we have compile-time range information and no other affine bounds are available for t. In this case, since we know the range of f, we can add approximate constraints on t. Specifically, we can conservatively say that t will fall somewhere within the upper and lower bounds of the range of f, or

$$\mathrm{lower\_bound}(\mathrm{range}(f)) \leq t \leq \mathrm{upper\_bound}(\mathrm{range}(f))$$

in constraint form. We may safely introduce these inequality constraints without changing the meaning of the set or relation since we are only bounding the values of the tuple variable t and not restricting its value beyond the original equality constraint.

4.2.4 Simplification Algorithm

In the previous sections we described three techniques for manipulating set and relation constraints that are required to enable code generation of the transformed code in Figure 1.5. Our approach for applying these three techniques as a complete simplification algorithm is shown as Algorithm 3. Our implementation uses this algorithm in two situations: at the time a set or relation is constructed from a user's specification and on a set or relation that is the result of an operation such as apply, inverse, or compose.

This algorithm attempts to successively project out each existential present in a set or relation's constraints. For each existential we attempt to apply the inverse function simplification and existential equality simplification as many times as they are applicable. Once these have been applied, we check that no existential is present in constraints containing a UFS. We then use Fourier-Motzkin elimination to project the existential out of any remaining affine constraints. Finally, we apply the affine approximation technique to introduce affine bounds for tuple variables where possible.

4.3 Evaluation

In this section, we present a discussion concerning the complexity and performance of the set and relation operations and simplifications. We first discuss the complexity of each operation and the simplification algorithm presented in Section 4.2.4.

```
for each existential v do
    Inverse function simplifications;
    Existential equality simplifications;
    if v present in a UFS constraint then
        fail
    end
    Project out v from the constraints using Fourier-Motzkin;
end
Affine approximation;
```

Algorithm 3: Set/Relation Simplification Algorithm

The second half of this section will present the performance of our implementation of the set and relation concepts from this thesis.

4.3.1 Complexity Analysis

Understanding the algorithmic complexity of the set and relation operations and simplifications allows us to have some idea for the upper bound on the expected running time of a code generation tool based on our techniques. We first present the complexity of the constraint simplification techniques followed by the overall complexity of the set/relation operations. For the following discussion, take N to be the number of constraints and M to be the number of conjunctions.

In general when performing an operation such as union or compose, we first perform the steps required by the operation to produce a new set of constraints. Following this we apply our simplification techniques to attempt to project out existentials from the constraints. Therefore, the complexity of each operation is a function both of the operation itself and of simplification. Since simplification is common to all operation complexity, we discuss this first.

We have discussed our simplification algorithm in Section 4.2.4 and have presented it as Algorithm 3. For each existential present in a set or relation's constraints, this algorithm attempts to project out that variable. It first applies the inverse function simplification and the existential equality simplification to remove existentials from constraints containing UFSs. It then uses Fourier-Motzkin elimination to project the variable out of the constraints that do not contain UFSs. The complexity of Fourier-Motzkin elimination is double exponential in the number of constraints and variables. Therefore, this is the dominating term for our simplification algorithm. However, for the sake of completeness, we note that both the inverse function simplification and the existential equality simplification are $O(N)$ for each existential since we must perform a linear scan through all constraints. Also, as an optimization, we note that we can utilize the existential equality simplification when projecting out existentials involved in equality constraints. We have found that this optimization avoids running full FM in the majority of cases, avoiding the worst-case exponential complexity of FM.

The complexity of both the apply and compose operations is $O(M^2)$ in the total number of conjunctions since we must create a new conjunction for every pair of conjunctions between operands. Union is $O(M)$ since we must combine all conjunctions into a single set or relation. Inverse is $O(1)$ since we only change the order of the tuple variables and no manipulation of the conjunctions or constraints is necessary.

4.3.2 Implementation Performance

We evaluate the performance and power of our set and relation implementation by comparing to Omega [13, 12] where possible. We base our evaluation on a suite of set/relation operations taken from a real-world run of our tool, IEGen. This run consisted of the application of three transformations (`cpack`, `locgroup`, and an iteration alignment) and one optimization (pointer update, or collapsing nested UFSs) to a molecular dynamics benchmark. The benchmark itself, `moldyn` [16], consisted of approximately 100 lines of C code with three main loops nested within an outer timestep loop. We extracted only the sets and relation operations from this run to form our suite of operations for benchmarking. We measured various aspects of performing these operations after constructing the arguments to each operation. We first present information about these operations only related to our implementation followed by a discussion of our implementation compared to Omega.

The whole run of IEGen of the transformation sequence on the `moldyn` computation took a total of 1.85 seconds. Running just the set and relation operations that this consisted of (the operations suite) took 1.46 seconds. Figure 4.1 presents the number of each operation type for 77 total operations that the operations suite contains. Figure 4.2 shows a histogram of the numbers of tuple variables in the results of the operations. Figure 4.3 shows a histogram of the numbers of existentials that need to be projected out for the operations. The time to run each of the collections of individual operation types is shown in Figure 4.4. Across all operations, a total of 246 existentials were projected out. Most of these were projected out using one or more applications of the existential equality simplification. Only 16 existentials required using the full Fourier-Motzkin algorithm to be projected out of the constraints. The inverse simplification rule was applied a total of 30 times.

Omega's support for UFSs is limited [20, 26]. Syntatically, it requires that inputs to UFSs be prefixes of the tuple variables. Additionally, any attempt to project out an existential that is the input to a UFS results in an UNKNOWN constraint being added to the result signifying that the result is only an approximation and may not be correct. Due to these restrictions, Omega is not able to perform all of the 77 operations that the operations suite contains. We filtered out the operations that Omega could not support to create a subset of operations that Omega can support. Of the 136 sets and relations that were arguments to the operations 81 did not contain UFSs. The total number of operations where all arguments contain no UFSs is 33, composed of:

- Inverse: 4
- Apply: 16
- Compose: 4
- Set Union: 9

We use these 33 operations to compare our implemementation performance to Omega's. Figure 4.5 presents the times taken for our implementation and Omega to perform all operations from each of the four types above. In most cases we are two orders of magnitude slower than Omega. We note that our initial implementation goal was not raw performance but rather supporting the operations and simplifications required for code generation. To improve the performance, we could utilize techniques implemented in Omega [19]. The performance goal for our implementation is to be fast enough to support interactive use of IEGen. For moldyn, we've come very close to meeting this goal, but there is clearly still room for improvement.

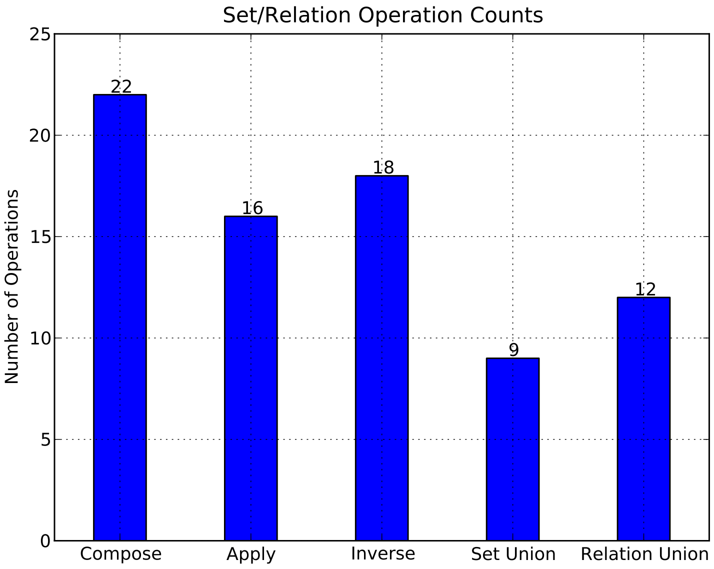

Figure 4.1: Counts of number of operations of 5 types in the operations suite

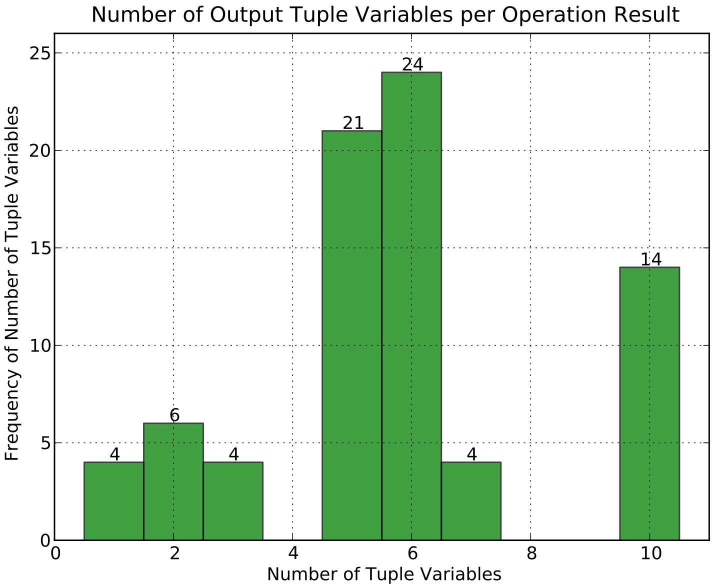

Figure 4.2: Histogram of numbers of tuple variables in the results of all operations in the operations suite

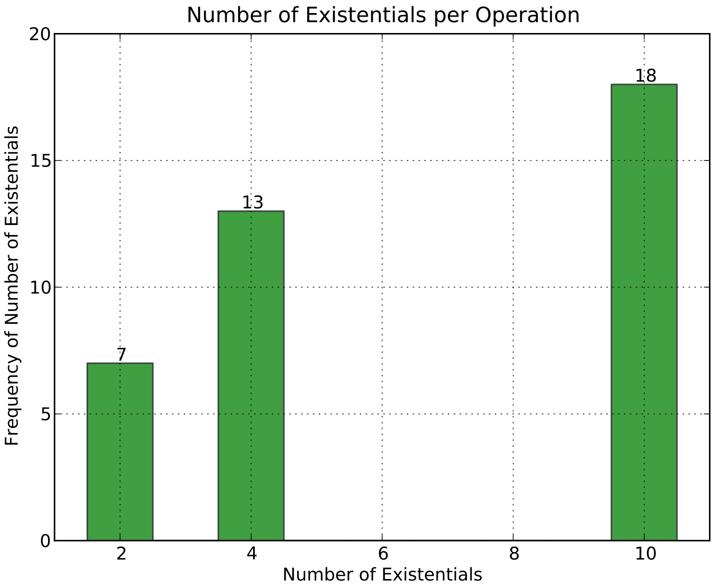

Figure 4.3: Histogram of total number of existentials for all operations in the operations suite

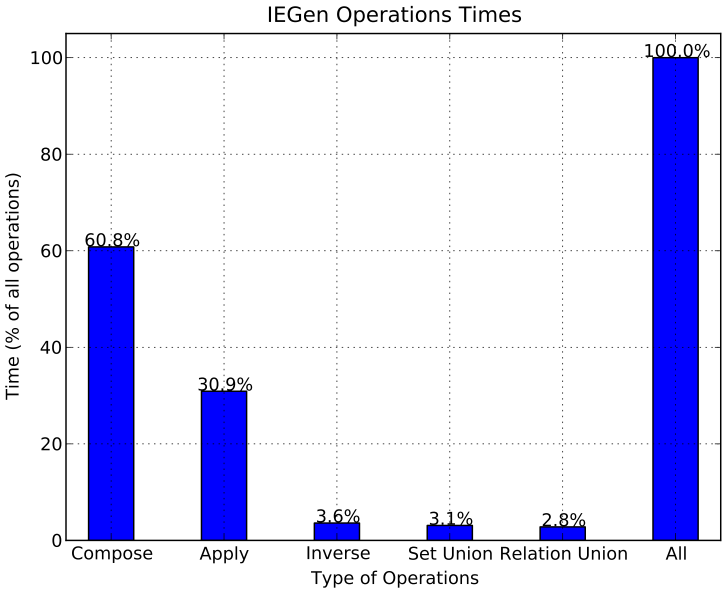

Figure 4.4: Times for performing various collections of operations in the operations suite grouped by type

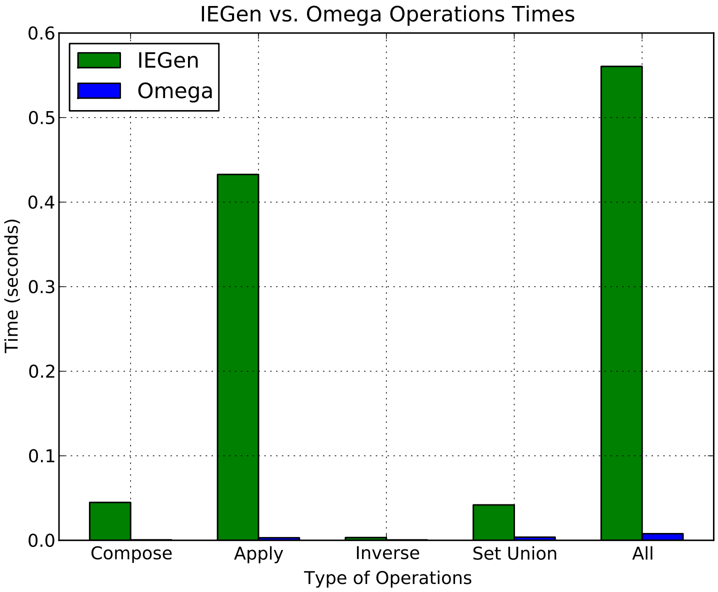

Figure 4.5: Times for performing various collections of operations that Omega could run

Chapter 5

Sparse Formula Data Structure

A number of libraries have been developed that support the specification and manipulation of polyhedra and unions of polyhedra including Polylib [25, 14], ISL [24], PPL [2], and Omega [13, 12, 19]. Traditionally, these libraries have not included support for uninterpreted function symbols (UFSs), with the exception being Omega. However, even Omega's support for UFSs is limited in the ability to support arbitrary arguments and utilize additional information about individual UFSs.

To support IEGen and implement the techniques discussed in Chapter 4, we developed a data structure that supports specifying sets and relations with constraints containing tuple variables, existentials, symbolic constants, and arbitrarily nested UFSs. We use the term formula to refer to both sets and relations. Since we support UFSs in the formula constraints, we refer to this data structure as the sparse formula data structure or simply sparse formulas. We've implemented operations such as apply, inverse, and compose (discussed in Section 4) on top of the sparse formula data structure discussed in this section. This chapter describes the details of our sparse formula implementation starting with an example of the overall implementation approach (Section 5.1) followed by a more general description of the details of the our sparse formula implementation (Section 5.2).

5.1 A Brief Example

To begin we present an example of how our we represent the set

$$\mathrm{F}' = \{[\mathrm{c}_0, \mathrm{time}, \mathrm{c}_1, \mathrm{p}, \mathrm{c}_2, \mathrm{tri}, \mathrm{c}_3] : (0 \leq \mathrm{time} < \mathrm{T}) \wedge (0 \leq \mathrm{tri} < \mathrm{R}) \wedge \mathrm{p} = \mathrm{proc}(\mathrm{tri})\}.$$

from a previous chapter as a sparse formula. This set was constructed in Chapter 4 and represents the full transformed iteration space for the code in Figure 1.5.

First notice that this set specifies a collection of integer 7-tuples restricted by the following five constraints: $\texttt{0} \leq \texttt{time}$, $\texttt{time} < \texttt{T}$, $\texttt{0} \leq \texttt{tri}$, $\texttt{tri} < \texttt{R}$, and $\texttt{p} = \texttt{proc}(\texttt{tri})$. Sparse formulas represent inequality and equality constraints as single expressions that are either $\geq$ or $=$ to zero. Therefore, we actually represent these five constraints as: $\texttt{time} \geq \texttt{0}$, $-\texttt{1}*\texttt{time}+\texttt{T}-\texttt{1} \geq \texttt{0}$, $\texttt{tri} \geq \texttt{0}$, $-\texttt{1}*\texttt{tri}+\texttt{R}-\texttt{1} \geq \texttt{0}$, and $\texttt{p} - \texttt{proc}(\texttt{tri}) = \texttt{0}$. Each of the five expressions for the constraints are stored as mappings from a term type to a coefficient. For example, the expression $-\texttt{1} * \texttt{time} + \texttt{T} - \texttt{1}$ has three types of terms: a tuple variable, a symbolic constant, and a constant value. Each of these terms has a coefficient, namely -1, 1, and -1 respectively. We signify tuple variables by their position in the tuple—in the case of this set we have positions zero through six. We signify symbolic constants by their name—in this case `T`. There can only be a single constant per expression (multiple constants may simply be added to obtain the single constant) and thus signifying the constant term is done with the name `Constant`. The coefficient mapping for this expression then is $\texttt{TupleVar}(\texttt{1}) \to -\texttt{1},\ \ \texttt{Symbolic}(\texttt{T}) \to \texttt{1},\ \ \texttt{Constant} \to -\texttt{1}$. Similar coefficient mappings exist for the expressions of the other four constraints and are presented in Table 5.1.

Representing uninterpreted function symbols is slightly more complicated than the other types of terms. We use the name UFCall to signify UFS terms in expressions. Each UFCall has a name and an ordered list of arguments associated

Table 5.1: Example sparse expressions for equality and inequality constraints

Constraint	**Expression Term→Coefficient Mappings**
$\texttt{time} \geq 0$	TupleVar(1)→1
$-1 * \texttt{time} + \texttt{T} - 1 \geq 0$	TupleVar(1)→-1 Symbolic(T)→1 Constant→-1
$\texttt{tri} \geq 0$	TupleVar(5)→1
$-1 * \texttt{tri} + \texttt{R} - 1 \geq 0$	TupleVar(5)→-1 Symbolic(R)→1 Constaint→-1
$\texttt{p} - \texttt{proc}(\texttt{tri}) = 0$	TupleVar(3)→1 UFCall(proc,TupleVar(5)→1)→-1

with it. Each argument is another expression represented using the same coefficient mapping concept we describe above. Since our expressions can contain UFSs, this method supports nested functions. The UFS proc in the constraint $\texttt{p} - \texttt{proc}(\texttt{tri}) = 0$ has only a single argument, tri. The coefficient mapping for the single argument is $\texttt{TupleVar}(5) \rightarrow 1$. The constraint expression is represented as $\texttt{TupleVar}(3) \rightarrow 1, \texttt{UFCall}(\texttt{proc}, \texttt{TupleVar}(5) \rightarrow 1) \rightarrow -1$ and is also shown in Table 5.1.

In summary, we build equality and inequality constraints using the coefficient mapping expressions. We build conjunctions and disjunctions on top of these constraints. Conjunctions are unordered collections of equality and inequality constraints. Disjunctions are unordered collections of conjunctions. Finally, we store a bijection of tuple variable name to tuple position—in this case $\texttt{c}_0 \leftrightarrow 0, \texttt{time} \leftrightarrow 1, \texttt{c}_1 \leftrightarrow 2, \texttt{p} \leftrightarrow 3, \texttt{c}_2 \leftrightarrow 4, \texttt{tri} \leftrightarrow 5, \texttt{c}_3 \leftrightarrow 6$. This bijection is used for tasks like set/relation printing.

5.2 Data Structure Details

After presenting a brief example of the sparse formula data structure in Section 5.1, we now describe it more generally and completely. As we previously showed, sparse formulas are based on representing expressions as mappings from term types to coefficients for those terms. Constraints (both equalities and inequalities) are represented with a single expression that is assumed to be ≥ 0 or $= 0$. Conjunctions are unordered collections of constraints. Disjunctions are unordered collections of conjunctions.

We show the classes of the sparse formula implementation with a diagram in Figure 5.1. This figure shows the `Set` and `Relation` classes with fields for the tuple/position bijection (`tuple_vars`), and fields for the collection of symbolic constants. Since relations have an input and output arity, the `Relation` class stores the input arity in the field `arity_in`. The output arity may be calculated based on the value of this field the the total number of tuple variables (obtained based on the `tuple_vars` field). The `disjunction` field of the `Set` and `Relation` classes stores an instance of the `Disjunction` class. `Disjunction` contains an unordered set of `Conjunction` instances in the `conjunctions` field. The `Conjunction` class has a `constraints` field that stores an unordered set of `Constraint` instances. The `Equality` and `Inequality` classes derive from the base `Constraint` class. The `Constraint` class is defined, as noted earlier, using the `Expression` class that contains a mapping of terms to coefficients.

The types of terms supported by the `Expression` class are shown in Figure 5.2. These terms include `TupleVar`, `ExistVar`, `Symbolic`, `Constant`, and `UFCall`. `TupleVar` terms correspond to a particular tuple variable position, stored

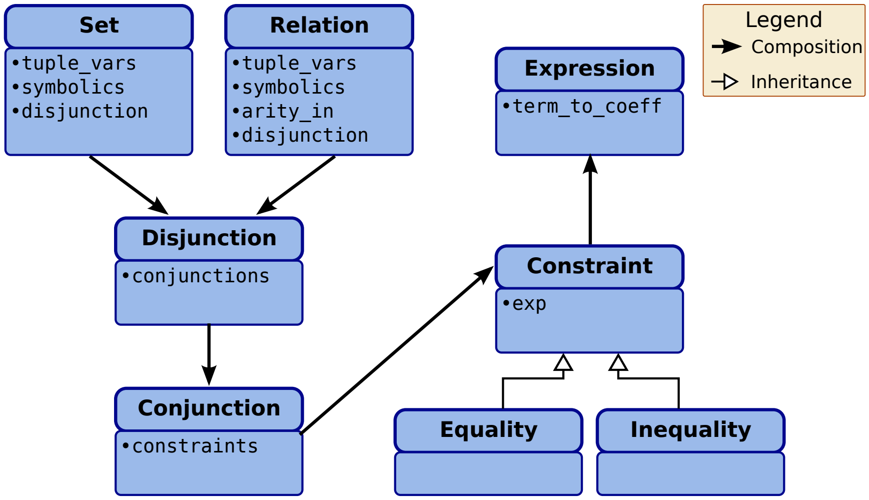

Figure 5.1: Sparse formula classes

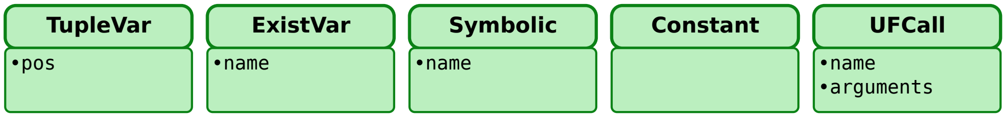

Figure 5.2: Sparse expression term types

in the `pos` field. `Symbolic` and `ExistVar` terms have a name associated with them, stored in the `name` field. Our support for uninterpreted function symbols is introduced with the `UFCall` expression term type. A `UFCall` has a `name` and an ordered list of arguments. Each argument to a `UFCall` is an `Expression`, providing support for nested uninterpreted function symbols.

We make two assumptions in our sparse formula implementation. First, we assume that we have all symbolic and existential names available at the time the formula is created (this excludes the compose and apply operations where certain tuple variables become existentials). Second, we do not overload function names. In other words, if a function 'f' has two arguments, all instances of it must have two arguments.

In summary, sparse formulas provide a method of representing polyhdera and unions of polyhedra whose constraints additionally can contain uninterpreted function symbols. We base our implementation on the concept of an expression containing terms that map to their coefficients. This coefficient mapping approach allows us to implement operations such as apply, inverse, and compose without the need to reallocate columns that a fixed column representation might require since we do not impose an arbitrary order on the terms in an expression. We utilize sparse formulas throughout IEGen to represent entities such as iteration spaces and access functions. The support for UFSs that is present in our sparse formula implementation allows us to implement the simplifications discussed in Chapter 4 and thus enable inspector executor code generation.

Chapter 6

Related Work

The techniques we have developed during the development of IEGen build upon many ideas and projects within the field of loop transformation and optimization. This section describes how the work we present in this thesis relates to other projects, including libraries for computation representation, an intermediate representation similar to our inspector dependence graph (IDG), and other tools for program optimization and code generation. The techniques presented in Chapter 4 and data structure presented in Chapter 5 form the basis of a library for representing irregular computations and transformations on these computations. This library is similar in many ways to other polyhedral libraries. Our inspector code generation is based on the IDG graph data structure and is similar to another graph-based inspector/executor strategy called a slice graph. Finally, our irregular computation transformation and code generator, IEGen, has functionality similar to many other program optimization and code generation tools. We discuss similar projects in this chapter.

We note four polyhedral libraries that provide representations and operations on polyhedra: Omega, Polylib, PPL, and ISL. The Omega project [13, 12, 19] is most closely related to the work we present in this thesis. Used primarily for data dependence analysis, Omega's implementation of sets and relations (based on

Presburger arithmetic) supports limited usage of UFSs. Unlike the techniques we have presented in this thesis, Omega does not utilize any additional information about UFSs such as the domain and range of UFSs or knowledge that UFSs are bijections. Therefore, the library is unable to perform operations like the inverse function simplification or affine approximation. Other limitations are that inputs to uninterpreted functions in Omega can only be prefixes of input and/or output tuples and composition of two relations with uninterpreted functions results in an `UNKNOWN` constraint. To a limited extent, uninterpreted function symbols have been used in Omega to represent indirect array accesses for determining more accurate data dependence information. In comparison, we use UFSs to represent index arrays and run-time reordering functions whose precise data is not known until runtime.

Polylib [25, 14] is a library that provides a way to represent and manipulate unions of parameterized polyhedra. Polylib maintains both constraint and generator representations of the polyhedra as some operations are more efficient on one representation than the other. Our implementation uses an approach similar to the constraint representation. Polylib does not have support for UFSs and thus is not able to support the class of programs we are targeting. Another library that represents polyhedra is PPL [2]. Similarly to Polylib, PPL maintains dual representations of polyhedra. Just as with Polylib, PPL represents rational polyhedra as well. PPL has no support for representing UFSs. Finally, ISL [24] is a fourth library that represents polyhedra. This library was recently adopted as the default polyhedral backend for CLooG.

Our inspector code generation technique is based on a graph data structure called in inspector dependence graph (IDG). Das et al. [9] described a program transformation for reducing multiple indirect array accesses to a single indirect array access. This program transformation used an inspector executor strategy to, at runtime, inspect the multiple levels of indirection and produce a single new indirection array, thus avoiding multiple layers of indirection. The authors describe a data structure called a slice graph that in many ways resembles our IDG. The slice graph is used to generate inspector code that creates the single indirection array and executor code that uses this new array. The code generation process is also very similar to our topological traversal of the nodes in the IDG.

CLooG, Pluto, Graphite, FADA, and PoCC are tools used for code generation, program analysis, and optimization that utilize one or more of the polyhedral libraries we have referenced above. CLooG [3, 4, 5] is a tool for generating code that scans a specified union of polyhedra. We use CLooG for generating loop nests that scan the iteration spaces of our computations. Pluto [7] is an automatic polyhedral loop optimizer and parallelizer that utilizes a large set of tools from the polyhedral community including CLooG for generating scanning loop nests. Graphite [17] is used as a polyhedral optimizing back end for the popular GCC compiler suite. FADA [8, 6] (Fuzzy Array Dataflow Analysis) is a tool for performing dataflow analysis on irregular programs. It performs an 'instance wise dependence analysis for non-static control programs' which provides data dependence information for irregular computations. The FADA Toolkit is a partial implementation of this analysis. FADA offers full support for static control programs as it extends Feautrier's Array Dataflow Analysis (ADA). Additionally, FADA supports while loops and conditionals, scalar and array references, and indirect array accesses (such as we require in SPF) and non-affine array accesses. FADA does not support pointer indirection and array cell aliasing. The output from FADA is a set of dataflow dependences in the form of quasts (quasi-affine selection tree). Our work may be able to utilize a tool like the FADA Toolkit to support dataflow analysis to determine if a specified transformation is legal. Finally, PoCC [18] (the Polyhedral Compiler Collection) is a source-to-source optimizing compiler.

Chapter 7

Future Work and Conclusions

This chapter looks to the future of IEGen and discusses a few potential additions that could be made to extend the functionality and scope of IEGen. Following our discussion of future work, we conclude by summarizing the contributions of this thesis and of the IEGen tool.

7.1 Future Work

It would seem that most large projects are never complete, but only have milestones marking significant progress—IEGen is no exception to this observation. This section presents three potential avenues of exploration beyond the current state of IEGen. First, we suggest implementing additional inspector/executor strategies using IEGen. Second, we could improve the automation of IEGen to support extraction of computations from existing code would make using IEGen easier. Third, we may apply IEGen to a real-world application or simplified application (‘mini-app’).

Many inspector/executor strategies have been developed since the original idea was developed [15]. IEGen currently supports a limited subset of these strategies, including data and iteration reorderings. One obvious type of inspector/executor strategy, sparse tiling, attempts to improve data locality or introduce parallelism.

Supporting this type of transformation would be very useful as we could compare the relative performance benefits of techniques like full sparse tiling [23] and communication avoiding tiling techniques [10]. To extend IEGen to support additional transformations like tiling may require modifications to the IDG, such as adding new node types, modifications to the inspector code generation algorithm based on the IDG, and modifications to the executor code generator.

The second extension piece that would improve the usability of IEGen is to improve its automation aspects. Currently IEGen accepts input as a spec file that defines the irregular computation in question and a sequence of transformations to apply to this computation. However, creating a spec file from an existing piece of code is often a non-trivial task. We have developed a graphical user interface called IEGenCC that provides a graphical method for specifying spec files and invoking IEGen. Tools like Pluto and PoCC allow the user to mark a piece of code to be extracted and transformed. A similar interface to IEGen would greatly improve its usability and lower its barrier to entry. Some of the tasks that would be required are extraction of loop nests, statements, access relations for each statement, and data dependences from the code. The user would then only need to define the transformations to apply to this computation. The question of how to specify these transformations orthogonally to the base computation is an ongoing research area.

Finally, we suggest using IEGen to optimize a real-world application or 'mini-app'. Currently we have only looked at the transformation of small benchmark computational kernels (around 100 lines of code). A mini-app is a larger piece of code (around 1000 lines) that exhibits the behavior of a full scientific application yet has all of the non-performance-critical pieces removed. By applying IEGen to a mini-app, we would show that our techniques are more widely applicable in a real-world setting and are not just solving academic problems. Additionally, a performance improvement of a real-world application or mini-app would have a much more direct impact on the scientific research related to the application.

7.2 Conclusions

Many scientific computations utilize indirect array accesses such as `A[B[i]]`, which make poor use of the memory hierarchy which leads to poor performance. Inspector/executor strategies for performing run-time reordering transformations are one approach to optimizing irregular computations. This thesis presented a tool, IEGen, for applying inspector/executor strategies to irregular computations in a semi-automated way. IEGen accepts the specification of an irregular computation and a sequence of transformations to apply to that computation and generates inspector/executor code that implements the transformed computation.

Implemented in IEGen are three main techniques (the contributions of this thesis) that enable the generation of this code. First, we have created a new intermediate representation called an inspector dependence graph (IDG) that represents the inspector code for the transformed computation. The IDG enables us to answer various questions about the inspector and we use the answers to these questions to generated code. Secondly, we devised a method for generating executor code for the transformed computation based on the MapIR data structure. Executor code generation consists of two pieces: generating loop nests for scanning the full transformed iteration space and generating statement strings with transformed data array accesses. Finally, we have developed techniques to manipulate the constraints of sets and relations containing uninterpreted function symbols to enable the generation of code based on these sets and relations. These techniques are implemented with our data structure for representing sparse sets and relations.

By implementing our contributions within IEGen, we are now able to generate transformed irregular computations that implement composed inspector/executor strategies. IEGen currently supports the specification and generation of data and iteration permutation reorderings. By supporting these transformations with IEGen, we are able to generate code rather than manually implementing it for each sequence of transformations. This would save programmer time and will allow us to perform experiments to determine the best of a collection of transformation sequences that optimizes a computation.

REFERENCES

[1] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *Int. J. Parallel Program.*, 29(5):493–544, 2001.

[2] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

[3] C. Bastoul. Generating loops for scanning polyhedra. Technical Report 2002/23, PRiSM, Versailles University, 2002.

[4] C. Bastoul. Efficient code generation for automatic parallelization and optimization. pages 23–30, october 2003.

[5] C. Bastoul. 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.

[6] Marouane Beloucha, Denis Barthou, and Sid-Ahmed-Ali Touati. *FADA Toolkit Users Guide*. University of Versailles Saint-Quentin en Yvelines, France, October 2009.

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, 2008.

[8] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy array dataflow analysis. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 92–101, New York, NY, USA, 1995. ACM.

[9] Raja Das, Joel H. Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect accesses to distributed arrays. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, number CRPC-TR93319-S, pages 152–168, London, UK, 1994. Springer-Verlag.

[10] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine A. Yelick. Avoiding communication in sparse matrix computations. In *IPDPS*. IEEE, April 2008.

[11] Chen Ding and Ken Kennedy. 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*, pages 229–241, Atlanta, Georgia, May 1999.

[12] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. *The Omega Calculator and Library*, November 1996.

[13] Wayne Kelly and William Pugh. A unifying framework for iteration reordering transformations. Technical report, University of Maryland, College Park, MD, USA, October 1995.

[14] Vincent Loechner. Polylib: A library for manipulating parameterized polyhedra, 1999.

[15] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. Principles of runtime support for parallel processors. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 140–152, New York, NY, USA, 1988. ACM.

[16] Ravi Ponnusamy, Yuan-Shin Hwang, Raja Das, Joel Saltz, Alok Choudhary, and Geoffrey Fox. Supporting irregular distributions in fortran 90d/hpf compilers. Technical report, College Park, MD, USA, 1994.

[17] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G. A. Silber, and N. Vasilache. Graphite: Loop optimizations based on the polyhedral model for gcc. In *Proc. of the 4th GCC Developper's Summit*, pages 179–198, June 2006.

[18] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Hybrid iterative and model-driven optimization in the polyhedral model. Technical Report 6962, INRIA Research Report, June 2009.

[19] William Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992.

[20] William Pugh and David Wonnacott. Nonlinear array dependence analysis. Technical Report CS-TR-3372, College Park, MD, USA, November 1994.

[21] Michelle Mills Strout. Compile-time composition of run-time data and iteration reorderings. In *In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[22] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Rescheduling for locality in sparse matrix computations. In *Proceedings of the 2001 International Conference on Computational Science, Lecture Notes in Computer Science*, pages 28–30. Springer-Verlag, 2001.

[23] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. Combining performance aspects of irregular gauss-seidel via sparse tiling. In *in 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2002.

[24] Sven Verdoolaege. 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, April 2009.

[25] Doran K. Wilde. A library for doing polyhedral operations. Technical report, 1993.

[26] David G. Wonnacott. *Constraint-Based Array Dependence Analysis.* PhD thesis, University of Maryland, College Park, 1995.