AlphaZ: A System for Analysis, Transformation, and Code Generation in the Polyhedral Equational Model

Tomofumi Yuki Vamshi Basupalli Gautam Gupta
Guillaume Iooss DaeGon Kim Tanveer Pathan
Pradeep Srinivasa Yun Zou Sanjay Rajopadhye

May 31, 2012

Colorado State University Technical Report CS12-101

Contents

1 Introduction **3**

2 The AlphaZ System **3**

2.1 Compiler Scripts . . . 4

2.2 Implementation . . . 4

3 The Polyhedral Model **5**

3.1 Properties of Polyhedral Objects . . . 5

3.2 Lexicographical Order . . . 5

3.3 Memory-Based Dependences . . . 5

3.4 Polyhedral Equational Model . . . 6

3.5 Polyhedral Reduced Dependence Graph . . . 6

4 The Alpha Language **7**

4.1 Domains and Functions . . . 7

4.1.1 Parameter Domains . . . 8

4.1.2 Index Names . . . 8

4.2 Affine Systems . . . 8

4.3 Alpha Expressions . . . 9

4.3.1 Reductions in Alpha . . . 10

4.3.2 Context Domain . . . 10

4.4 Normalized `Alpha` . . . 10

4.4.1 Normalization Rules . . . 11

4.5 Array Notation . . . 12

4.6 Example . . . 12

4.7 Subsystems and Use Equations . . . 13

5 AlphaZ Transformations **14**

5.1 Expression Identifier . . . 15

5.2 Basic Commands . . . 15

5.2.1 `ReadAlphabets` . . . 15

5.2.2 `RenameSystem` and `RenameVariable` . . . 16

5.2.3 `RemoveUnusedVariables` . . . 16

5.2.4 `Normalize` . . . 16

5.2.5 `Show`, `AShow`, `Save`, and `ASave` . . . 16

5.2.6 `PrintAST` . . . 16

5.3 `CoB`: Change of Basis . . . 16

5.4 `Inline` . . . 17

5.5 `Split` . . . 17

5.6 `SplitUnion` . . . 17

5.7 `Simplify` . . . 18

5.8 `BuildPRDG` and `ExportPRDG` . . . 18

6 Reduction Transformations **18**

6.1 `SimplifyingReduction` . . . 18

6.2 `PermutationCaseReduce` . . . 19

6.3 `ReductionDecomposition` . . . 19

6.4 `NormalizeReduction` . . . 20

6.5 `FactorOutFromReduction` . . . 20

6.6 `SerializeReduction` . . . 20

6.7 `SplitReductionBody` . . . 21

6.8 `TransformReductionBody` . . . 21

7 Target Mapping 21
7.1 Space-Time Mapping 21
7.1.1 Schedule 22
7.1.2 Processor Allocation 22
7.1.3 Ordering Dimensions 23
7.2 Memory Mapping 23
7.2.1 Reductions 23
7.3 Additional Specifications 23
7.3.1 Tiling 24

8 The Verifier 24
8.1 Overview 24
8.2 Verification of Space-Time Mappings 24
8.2.1 Verification of Schedule 24
8.2.2 Verification of Processor Allocation 25
8.2.3 Reductions 25
8.2.4 Tiling 25
8.3 Verification of Memory Mappings 26

9 Code Generators 26
9.1 WriteC 26
9.2 ScheduledC 27
9.3 ScanC 27
9.4 WrapperC 27
9.5 Makefile 28

10 Related Work 28

11 Future Additions 29

1 Introduction

The *polyhedral model* is a formalism for automatic parallelization of an important class of programs. This class includes *affine control loops* (ACLs), a common target for aggressive program optimizations and transformations. It also includes programs in `Alpha` a polyhedral equational language [22] that subsumes ACLs, and also includes reductions as first class expressions [19]. Many optimizations, including loop fusion, fission, tiling, and skewing, can be expressed as transforming one polyhedral specification into another. Vasillache et al. [27, 34] make a strong case that a polyhedral representation of programs is especially needed to avoid the blowup of the intermediate program representation (IR) when many transformations are repeatedly applied, as is becoming prevalent in auto-tuners and iterative compilation.

A number of "polyhedral" tools and components for generating efficient code are now available [3, 4, 13, 16, 17, 18, 23, 28]. In the typical flow of such tools, a source program in an imperative language (usually a subset of `C`) is first analyzed to extract a section of code that is amenable to polyhedral analysis. This is followed by a sequence of analyses and transformations, and finally code is generated.

In this paper, we present `AlphaZ`, an open source research system to enable users and researchers experiment with analysis, transformation, and code generation in the polyhedral model. Our aim is to expose, as much as possible, the choice of transformations to apply to polyhedral programs, and to provide the infrastructure for rapid development of back-end code generators. `AlphaZ` is not (yet) intended to be a fully automatic compiler, although certain elements in it are automatic. There are two main motivations for our particular view.

First, research in polyhedral program optimization—what specific transformation(s) to apply to an initial program so as to maximize "performance"—has remained an open and active research question for well over two decades. The notion of "performance" is not cast in stone, and may vary: from *speed*, which itself could be either throughput, or latency, or some combination, through *cost*—number of processors or other critical resources, silicon area, memory footprint, etc., to the increasingly important *power/energy.* We may either leave these as independent cost metrics among which Pareto optimal solutions are sought, or seek to optimize a combination, such as the energy-delay product. In a fully automatic system these choices are aften made early, and hardwired deep inside the system.

Second, the choice of the optimal transformation also depends crucially on target architectures, which themselves continue to evolve: multi-core, many-core, distributed memory, accelerators, FPGAs, GPUs, etc. The "code" for these different targets often varies widely, and is written in multiple different target languages: from `C/FORTRAN` possibly with OpenMP pragmas, or with MPI library calls, possibly even hybrid MPI-OpenMP, through other modern languages like X10 or Chapel, to `Verilog/VHDL` for accelerators and FPGA targets. Developing "back-end" code generators for this diverse array of targets is a daunting prospect. Most tools make the choice of the target early on and hardwire it into the system.

`AlphaZ` is intended to serve as a framework for all such objectives, where researchers may choose their optimization criteria and target architectures. All transformations are exposed to the user, and the system is "merely" a framework for possible exploration of optimization strategies. It builds on a "clean" intermediate representation (IR) that describes declaratively, at the highest level of abstraction possible, *what* is to be computed.[1] It also uses the notions of *target mappings*, very high level specification of the desired final transformation, *verifiers* that check the legality of proposed mappings, and finally a *target-agnostic* code generation framework.

2 The AlphaZ System

In this section, we present an overview of the AlphaZ system. The input program is specified in a polyhedral equational language[2] called `Alpha` proposed by Mauras in 1989, extended by Leverge in 1992 to include reductions as first-class expressions [19], and further by Dupont de Dinechin to include modular structure in the form of *subsystems* [7]. Given the input `Alpha` program, the basic flow is the following:

1. transform the program using available transformations

[1]Because of this we, in fact, adopt the view that the IR should actually be the input specification.

[2]We are working on an extension of `Alpha`, called `Alphabets`, to handle non-polyhedral domains [30]. In this report however, any reference `Alphabets` should simply be taken to mean `Alpha`.

2. specify execution strategies, such as schedules and memory mappings, and
3. generate code

AlphaZ also provides a number of semantics-preserving transformations for manipulating Alpha programs, that go beyond simply specifying an execution strategy. These transformations can be used to expose more opportunities for scheduling/memory allocation, or incorporate user insight such as common subexpression elimination, inlining, etc. Transformations in AlphaZ are described in Section 5. Various analyses of the program may also be performed to guide the transformations. Furthermore, AlphaZ works with a program representation that emphasizes reductions, and provides a separate set of transformations for manipulating reductions, described in Section 6.

Before generating executable code from Alpha, various execution strategies, such as schedule, memory mapping, and processor allocation, can be specified. These execution strategies are collectively called the *target mapping* (TMap), as described in detail in Section 7.

One critical analysis given Alpha program and its TMap is its verification. Since AlphaZ allows users to specify the TMap, the given TMap may be incorrect. The verifier, depicted in Section 8, is a module in AlphaZ to verify the given TMap, and give detailed feedback on what may have caused violations in case violations were detected.

Once a program is given a TMap, code generators may be invoked to generate programs to obtain codes that executes the Alpha program using specified strategies. AlphaZ provides different code generators for different purposes as described in Section 9.

2.1 Compiler Scripts

The sequence of operations that are to be performed by AlphaZ are written as scripts called *compiler scripts*. The scripts consists of function calls and dynamically typed variables. For example, the following script will read and print out an Alpha program [3] example.ab.

```
prog = ReadAlphabets("example.ab");
Show(prog);
```

The complete list of commands is at http://www.cs.colostate.edu/AlphaZ/AlphaZCommandRef.pdf. Also, when using Eclipse to edit compiler scripts (.cs files), you can use the Eclipse content-assist feature (control-space by default key bindings) to show a dialog with available commands with the matching prefix.

2.2 Implementation

The system is written in Java (version 1.6) and is integrated to an open source IDE, Eclipse, which provides very convenient features like syntax highlighting, completion, context sensitive debug/error information, and code navigation. In addition to the script interface, the user may alternatively write Java code and use AlphaZ through the same commands as in the script, since they are also provided as static methods.

We chose Java for a number of reasons. When speed is not the primary concern, modern high-level languages provide much benefits in terms of programmer productivity. Since our primary goal is to provide an environment for rapid prototyping, the efficiency of the system itself is not critical. Java is a verbose language, where the code explains itself more than high-level scripting languages such as Perl, Python, or Ruby. The verbosity may slow the development slightly, but when a number of developers are involved, less ambiguous code helps code sharing and reuse. In addition, using Java makes our system a good fit to the eclipse environment.

We chose not to re-implement basic polyhedral operations for which stable libraries are already available [1, 8, 17, 36, 35]. We use the Integer Set Library [35] in AlphaZ through Java Native Interface. Currently, we only provide bindings for Linux and Mac environments, and thus the use of AlphaZ is limited to these platforms as well.

[3] ab is the file extension for Alphabets, but are also used for the current Alpha-like language accepted by AlphaZ.

3 The Polyhedral Model

The strength of the polyhedral model as a framework for program analysis and transformation are its mathematical foundations for two aspects that should (but are often not) viewed separately: program *representation/transformation* and *analysis*. Feautrier [9] showed that a class of loop nests called Affine Control Loops (or Static Control Parts) can be represented in the polyhedral model. This allows compilers to extract regions of the program that are amenable to analyses and transformations in the polyhedral model, and to optimize these regions. Such code sections are often found in kernels of scientific programs, such as dense linear algebra, stencil computations, or dynamic programming.

In the model, each instance of each statement in a program is represented as an *iteration point*, in a space called *iteration domain* of the statement. Each such point is hence, an *operation*. The iteration domain is described by a set of linear inequalities forming a convex polyhedron using the following notation, where z is iteration point, A is a constant matrix, and b is a constant vector.

$$D = \{z \mid Az + b \geq 0,\ z \in \mathbb{Z}^n\}$$

Dependences are affine functions, expressed as[4] $(z \to z')$, where z' consists of affine expressions of z. *What* a program computes is completely specified by the set of operations and the (flow) dependences between them. As noted by Feautrier, program memory and data-structures need not figure in this representation.

3.1 Properties of Polyhedral Objects

One of the advantages of modeling the program using polyhedral objects is the rich set of closure properties that polyhedra and affine functions enjoy as mathematical objects. Preimage by function f, or image by its relational inverse f^{-1}, of a domain $\mathcal{D}$ is the set of points x such that $f(x) \in \mathcal{D}$. Polyhedral domains (unions of polyhedra) are closed under set operations. It is also closed under image by the relational inverse of an affine function, also called preimage. Because of this closure property, transformations described as affine functions can be guaranteed to produce another polyhedra after its application.

In addition, a number of properties from linear algebra can be used to reason about the program. For some of the analyses in this paper, we use one class of such properties, namely the kernels of matrices, and by implication, of afine functions and domains. The kernel of matrix A, $\ker(A)$, is the set of vectors x such that $Ax = 0$. Note that if $\rho \in \ker(A)$ then $Az = A(z + \rho)$, so the space characterized by the kernel describes the set of vectors that do not affect the value of an affine function. This can be used to find the set of points that share the same value, characterizing reuse as we will show later in the paper.

With an abuse of notation, we define the *kernel*s of domains and affine functions to be the respective kernels of the matrix that describes the linear part of the domain and affine functions. The kernel of domain $\mathcal{D}$ represented as $Ax + b \geq 0$ in matrix representation, is $\ker(A)$.

3.2 Lexicographical Order

Lexicographical ordering is used to describe the relation between two vectors. In this paper we use $\ll$ and $\gg$ to denote lexicographical ordering. Given two vectors z and z', $z \ll z'$ if

$$\exists k; \forall i < k, z_i = z'_i, z_k < z'_k$$

In words, z lexicographically precedes z' if some i-th element of z is less than z', and for all elements i that are before k, z_i and z'_i are equal.

Lexicographical ordering is the base notion of "time" in multi-dimensional polyhedra used in the polyhedral literature.

3.3 Memory-Based Dependences

The results of array dataflow analysis are based on the values computed by instances of statements, and therefore do not need any notion of memory. Therefore, program transformation using dataflow analysis

[4]In the literature of the polyhedral model, the word dependence is sometimes used to express flow of data, but our convention in this paper the arrow is from the consumer to the producer.

results usually requires re-considering memory allocation of the original program. Most existing tools have made the decision to preserve the original memory allocation, and include memory-based dependences as additional dependences to be satisfied. Note that there are sometimes good reasons for this. For example, in a production compiler like gcc for an existing imperative language like C, the compiler must be absolutely sure that there is no aliasing before considering such optimizations and/or transformations. However, research tools like Pluto could make such chices easily, since they already assume that external functions are side-effect free, and that sections of code that are amenable to polyhedral optimization are explicitly marked.

3.4 Polyhedral Equational Model

The AlphaZ system adopts an *equational* view, where programs are described as mathematical equations using the Alpha language [22]. After array dataflow analysis of an imperative program, the polyhedral representation of the flow dependences can be directly translated to an Alpha program. Furthermore, Alpha has reductions as first-class expressions [19] providing a richer representation.

We believe that application programmers (i.e., non computer scientists), can benefit from being able to program with equations, where performance considerations like schedule or memory remain unspecified. This enables a separation of what is to be computed, from the mechanical, implementation details of *how* (i.e., in which order, by which processor, thread and/or vector unit, and where the result is to be stored).

To illustrate this, consider a Jacobi-style stencil computation, that iteratively updates a 1-D data grid over time, using values from the previous time step. A typical C implementation would use two arrays to store the data grid, and update them alternately at each time step. This can be implemented using modulo operations, pointer swaps, or by explicitly copying values. Since the first two are difficult to describe as affine control loops, the Jacobi kernel in PolyBench/C 3.2 [26] uses explicit copying, and the code (jacobi_1d_imper) looks as follows:

```
for (t = 0; t < T; t++)
   for (i = 1; i < N-1; i++)
      A[i] = foo(B[i-1] + B[i] + B[i+1]);
   for (i = 1; i < N-1; i++)
      B[i] = A[i];
```

However, if writen equationally, the same *computation* would be specified as:

$$A(t,i) = \begin{cases} t = 0: & B_{\texttt{init}}(i); \\ t > 0 \leq i < N-1: & \texttt{foo}(A(t-1,i-1), A(t-1,i), A(t-1,i+1)); \\ t > 0 = i: & A(t-1,i); \\ t > 0 \wedge i = N-1: & A(t-1,i); \end{cases}$$

where A is defined over $\{t,i|0 \leq t < T \wedge 0 \leq i < N\}$, and B_{init} provides the initial values of the data grid. Note how the loop program is already influenced by the decision to use two arrays, an implementation decision, not germane to the computation.

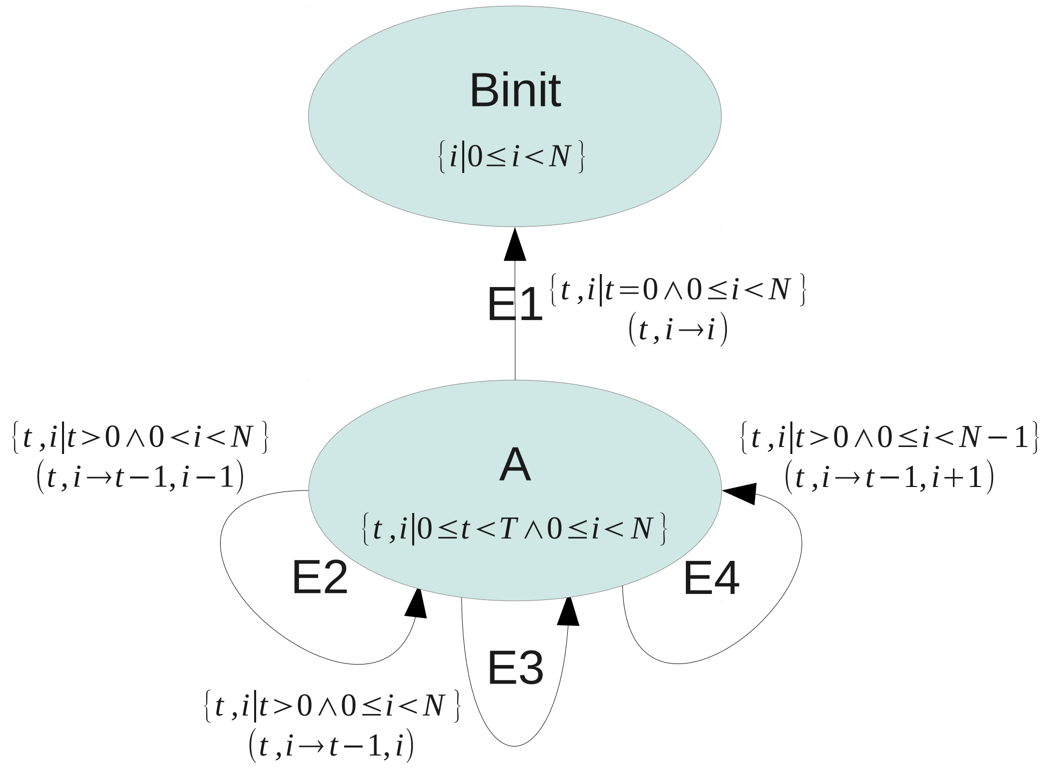
3.5 Polyhedral Reduced Dependence Graph

Polyhedral Reduced Dependence Graph (PRDG), sometimes called Generalized Dependence Graph, is a concise representation of dependences in a program. Each node of the PRDG represents an equation in Alpha (or a statement of a C program, when PRDG is constructed from C). Nodes are connected with edges that represent dependences between equations (statements).

Nodes in PRDG have an attribute, its domain, which is the domain of the corresponding equation. Edges have two attributes, its domain and the dependence function.[5] The domain of edge represents the set of points where the dependence exists. The dependence function is an affine function that map a point in the consumer to a point in the producer. The direction of the edge is the same as the dependence function, consumer points to the producer.

For example, the following is the PRDG for Jacobi stencil equation above.

[5]These two attributes are sometimes jointly represented as a dependence polyhedra. In AlphaZ, they are represented as two separate attributes.


Binit
$\{i|0\le i<N\}$
E1
$\{t,i|t=0\wedge 0\le i<N\}$
$(t,i\rightarrow i)$
A
$\{t,i|0\le t<T\wedge 0\le i<N\}$
$\{t,i|t>0\wedge 0<i<N\}$
$(t,i\rightarrow t-1,i-1)$
E2
$\{t,i|t>0\wedge 0\le i<N-1\}$
$(t,i\rightarrow t-1,i+1)$
E4
E3
$\{t,i|t>0\wedge 0\le i<N\}$
$(t,i\rightarrow t-1,i)$

Edge `E1` in the figure corresponds to the dependence to $B_{\texttt{init}}$ when $t = 0$ in the equation. The edges `E2`, `E3`, and `E4` corresponds to dependence to A itself, where the domain and functions are different based on the access functions and the branches of the case. The node `Binit` is an input, and may be omitted from the PRDG since input dependences are usually not considered for many analyses, such as scheduling.

4 The Alpha Language

In this section we describe the `Alpha` language used in AlphaZ. The language we use is a slight variant of the original `Alpha` [19]. In addition, an extension to the language to represent *while loops* and *indirect accesses*, called `Alphabets` has been proposed but is not fully implemented [30]. For the purposes of this paper, references to `Alphabets` should be considered the synonymous to `Alpha`.

4.1 Domains and Functions

Before introducing the language, let us first define notations for polyhedral objects, domains and functions. The textual representation of domains and functions resembles the notations used in Section 3 with the following changes to use standard characters:

- `&&` denotes intersection and `||` denotes union
- $\rightarrow, \le, \ge$ are written `->, <=, >=` respectively

We use the above textual representation when referring to a code fragment or when describing `Alpha` syntax.

When writing constraints for polyhedral domains, some short-hand notations are available. Constraints such as `a <= b` and `b <= c` can be merged as `a <= b <= c` if the constraints aligned ($<$ and $\le$ or $>$ and $\ge$). If two indices share the same constraints, it can be expressed concisely by using a list of indices surrounded by parenthesis (e.g., `a <= (b,c)`).

4.1.1 Parameter Domains

Polyhedral objects may involve program parameters that represent problem size (e.g., size of matrices) as symbolic parameters. Except for where the parameters are defined, `Alpha` parser treats parameters as implicit indices. For example, a 1D domain of size N is expressed as `{i|0<=i<N}`, and not `{N,i|0<=i<N}`. Similarly, functions that involve parameters are expressed like `(i->N-i)`, and not `(N,i->N-i)`.

4.1.2 Index Names

Although textual representation of domains and functions use names to distinguish indices from each other, the system internally do not use the index names when performing polyhedral operations. The indices are distinguished from each other by *dimensions*. For example, domains:

- `{i,j| 0<=i<N && 0<=j<M}`
- `{x,y| 0<=x<N && 0<=y<M}`
- `{j,i| 0<=j<N && 0<=i<M}`
- `{i,x| 0<=i<N && 0<=x<M}`

are all equivalent, since constraints on the first dimension is always 0 to N, and constraints on the second dimension is always 0 to M. Similarly, the index names can be different for each polyhedron in a union of polyhedra. For example, `{i,j| 0<=i<N && 0<=j<M} || {x,y| 0<=x<P && 0<=y<Q}` is valid. The system does make an effort to preserve index names during transformations, but it cannot be preserved in general.

4.2 Affine Systems

An `Alpha` program consists of one or more affine systems. The high-level structure of a system in `Alpha` programs is as follows:

```
affine <name> <parameter domain>
   input
      (<type> <name> <domain>;)*
   output
      (<type> <name> <domain>;)*
   local
      (<type> <name> <domain>;)*
   let
      (<name> = <expr>;)*
.
```

Each system corresponds to a System of Affine Recurrence Equations (SARE). The system consists of a name, a parameter domain, variable declarations, and equations that define values of the variables.

Parameter Domain is a domain with indices and constraints that are true for all domains in the system. The indices in this domain are treated as program parameters mentioned above, and are implicitly added to all domains in the rest of the system.

Variable Declarations specify the type and domain of each variable. We currently support the following types `int`, `long`, `float`, `double`, `char`, `bool`. The specified domain should have a distinct point for each value computed throughout the program, including intermediate results. It is important not to confuse variables domains with memory, but rather as simply the set of points where the variable is defined. Some authors my find is useful to view this as single assignment memory allocation, where every memory location can only be written once.[6]

[6]We contend that so called, "single assignment" languages are actually zero-assignment languages. Functional language compilers almost always reuse storage, so nowhere does it make sense to use the term "single" assignment.

External Functions may additionally be declared in the beginning of an `Alpha` program. External function declarations take the form of C function prototypes/signatures with scalar inputs and outputs. Declared external functions can be used as point-wise operators, and are assumed to be side effect free.

4.3 Alpha Expressions

The following table summarizes expressions in `Alpha`.

Expression	**Syntax**	**Expession Domain**
Constants	Constant name or symbol	$\mathcal{D}_P$
Variables	`V` (variable name)	$\mathcal{D}_{\texttt{V}}$
Operators	$\texttt{op}(\texttt{Expr}_1, \ldots, \texttt{Expr}_M)$	$\bigcap_{i=1}^{M} \mathcal{D}_{\texttt{Expr}_i}$
Case	$\texttt{case}\,\texttt{Expr}_1; \ldots; \texttt{Expr}_M\ \texttt{esac}$	$\biguplus_{i=1}^{M} \mathcal{D}_{\texttt{Expr}_i}$
If	$\texttt{if Expr}_1\ \texttt{then Expr}_2\ \texttt{else Expr}_3$	$\mathcal{D}_{\texttt{Expr}_1} \cap \mathcal{D}_{\texttt{Expr}_2} \cap \mathcal{D}_{\texttt{Expr}_3}$
Restriction	$\mathcal{D}' : \texttt{Expr}$	$\mathcal{D}' \cap \mathcal{D}_{\texttt{Expr}}$
Dependence	$f\texttt{@Expr}$	$f^{-1}(\mathcal{D}_{\texttt{Expr}})$
Index Expression	$\texttt{val}(f)$ (range of f must be $\mathbb{Z}^1$)	$\mathcal{D}_P$
Reductions	$\texttt{reduce}(\oplus, f, \texttt{Expr})$	$f(\mathcal{D}_{\texttt{Expr}})$

Expressions in `Alpha` also have an associated domain computed from the leaf (either constants or variables, where the domain is defined on its own) using domains of its children. These domains denote where the expression is defined and could be computed. Domain $\mathcal{D}_P$ in the table above, shown as the domain of constants and index expressions, is the parameter domain. These expressions can be evaluated for the full universe, and thus its expression domain is the intersection of universe with the parameter domain.

The semantics of each expression when evaluated at a point z in its domain is defined as follows:

- a constant expression is the associated constant.
- a variable is either provided as input or given by an equation; in either case, it is the value, at z, of the expression on its RHS.
- an operator expression is the result of applying `op` on the values of its arguments at z. `op` is an arbitrary, strict point-wise, single valued function.
- a case expression is the value at z of that branch whose domain contains z. Branches of a case expression are defined over disjoint domains to ensure that the case expression is not uniquely defined.
- an if expression `if` E_C `then` E_1 `else` E_2 is the value of E_1 at z if the value of E_C at z is true, and the value of E_2 at z otherwise. E_C must evaluate to a boolean value. Note that the else clause is *required*.
- a restriction of E is the value of E at z.
- the dependence expression $f@E$ is the value of E at $f(z)$. The dependence expression in our variant of `Alpha` use function joins instead of compositions. For example, $f@g@E$ is the value of E at $g(f(z))$, where the original `Alpha` wrote $E.g.f$.
- the index expression $\texttt{val}(f)$ is the value of f evaluated at point z.
- $\texttt{reduce}(\oplus, f, E)$ is the application of $\oplus$ on the values of E at all points in its domain $\mathcal{D}_E$ that map to z by f. Since $\oplus$ is an associative and commutative binary operator, we may choose any order of application of $\oplus$.

It is important to note that the restrict expression only affects the domain, and not what is computed for a point. This expression is used in various ways to specify the range of values being computed for an equation. In addition, identity dependence is assumed for variable expressions with out a surrounding dependence expression. Similarly, function to zero-dimensional space from the surrounding domain is assumed for constant expressions.

4.3.1 Reductions in Alpha

Reductions, associative and commutative operators applied to collections of values, are explicitly represented in the intermediate representation of AlphaZ. Reductions often occur in scientific computations, and have important performance implications. For example, efficient implementations of reductions are available in OpenMP or MPI. Moreover, reductions represent more precise information about the dependences, when compared to chains of dependences.

The reductions are expressed in the following form as $\texttt{reduce}(\oplus, f_p, \texttt{Expr})$, where *op* is the reduction operator, f_p is the projection function, and E is the expressions/values being reduced. The projection function f_p is a affine function that maps points in $\mathbb{Z}^n$ to $\mathbb{Z}^m$, where m is usually smaller than n. When multiple points in $\mathbb{Z}^n$ is mapped to a same point in $\mathbb{Z}^m$, those values are combined using the reduction operator. For example, commonly used mathematical notations such as $X_i = \sum_{j=0}^{n} A_{i,j}$ is expressed as $X(i) = \texttt{reduce}(+, (i, j \to i), A(i, j))$. This is more general than mathematical notations, since reductions with non-canonic projections, such as $(i, j \to i + j)$, require an additional variable to express with mathematical notations.

4.3.2 Context Domain

Each expression is associated with a domain where the expression is defined, but the expression may not need to be evaluated at all points in its domain. Context domain is another expression attribute, denoting the set of points where the expression must be evaluated. Context domain of an expression E is computed from its domain and the context domain of its parent.

The context domain $\mathcal{X}_\texttt{E}$ of the expression E is:

- $\mathcal{D}_\texttt{V} \cap \mathcal{D}_\texttt{E}$ if the parent is an equation for variable V.
- $f(\mathcal{X}_{E'})$ if E' is $E.f$.
- $f_p^{-1}(\mathcal{X}_{E'}) \cap \mathcal{D}_E$ if E' is reduce$(\oplus, f_p, E)$.
- $\mathcal{X}_{\texttt{E}'} \cap \mathcal{D}_{\texttt{E}'}$ if the parent $\texttt{E}'$ is any other expression.

This distinction of what *must* be computed and what *can* be computed is important when the domain and context domain are used to analyze the computational complexity of a program.

4.4 Normalized Alpha

Alpha programs can become difficult to read, especially as program transformations are composed, and may have complicated expressions such as case or if expressions. For example, consider the equation below (drawn from [19]).

```
U = case
       {i,j|j==0} : X;
       {i,j|j>=1} : (i,j->i+j)@(Y+Z)
                  * case
                       {i,j|i==0 && j>0} : W1;
                       {i,j|i>=1 && j>0} : W2;
                    esac;
    esac;
```

it would be much more readable if it were rewritten as:

```
U = case
       {i,j|j==0} : X;
       {i,j|i==0 && j>=1} : ((i,j->i+j)@Y + (i,j->i+j)@Z) * W1;
       {i,j|i>=1 && j>=1} : ((i,j->i+j)@Y + (i,j->i+j)@Z) * W2;
    esac;
```

Note how the case expressions are now “flattened.” This flattening is the result of a transformation called *normalization*, as proposed originally by Mauras [22]. Normalized programs are usually easier to read since the branching of the cases are only at the top-level expression, and the reader do not have to think about restrict domains at multiple level of case. The important properties of normalized `Alpha` programs are:

- Case expressions are always the top-level expression of equations or reductions, and there is no nesting of case expressions.
- Restrictions, if any, are always just inside the case, and are also never nested. The expression `inside` a restriction has neither case nor restriction, but is a simple expression consising of pointwise operators and dependence expressions.
- The child of dependence expressions are either a variable, a constant, or a reduce expression.

4.4.1 Normalization Rules

The following rules are used to normalize `Alpha` programs.

1. $f@E \Rightarrow E$, if $f(z) = z$
2. $f@(E_1 \oplus E_2) \Rightarrow f@(E_1) \oplus (f@E_2)$
3. $(\mathcal{D} : E_1) \oplus E_2 \Rightarrow \mathcal{D} : (E_1 \oplus E_2)$
4. $E_1 \oplus (\mathcal{D} : E_2) \Rightarrow \mathcal{D} : (E_1 \oplus E_2)$
5. $f_2@(f_1@E_1) \Rightarrow f@E$, where $f = f_1 \circ f_2$
6. $\mathcal{D}_1 : (\mathcal{D}_2 : E) \Rightarrow \mathcal{D} : E$, where $\mathcal{D} = \mathcal{D}_1 \cap \mathcal{D}_2$
7. $f@(\mathcal{D} : E) \Rightarrow \mathcal{D}' : E$, where $\mathcal{D}' = f^{-1}(\mathcal{D})$
8. $f@u(E) \Rightarrow u(f@E)$
9. $u(\mathcal{D} : E) \Rightarrow \mathcal{D} : u(E)$
10. $case\, E_1^1; \ldots case\, E_1^2; \ldots E_n^2;\, esac \ldots E_m^1;\, esac \Rightarrow case\, E_1^1; \ldots E_1^2; \ldots E_n^2; \ldots E_m^1;\, esac$
11. $E \oplus (case\, E_1; \ldots E_n;\, esac) \Rightarrow case\, (E \oplus E_1); \ldots (E \oplus E_n);\, esac$
12. $(case\, E_1; \ldots E_n;\, esac) \oplus E \Rightarrow case\, (E_1 \oplus E); \ldots (E_n \oplus E);\, esac$
13. $u(case\, E_1 \ldots E_n;\, esac) \Rightarrow case\, u(E_1); \ldots u(E_n);\, esac$
14. $f@(case\, E_1; \ldots E_n;\, esac) \Rightarrow case\, (f@E_1); \ldots (f@E_n);\, esac$
15. $\mathcal{D} : (case\, E_1; \ldots E_n;\, esac) \Rightarrow case\, \mathcal{D} : E_1; \ldots \mathcal{D} : E_n;\, esac$
16. $f@(if\, E_1\, then\, E_2\, else\, E_3) \Rightarrow if\, (f@E_1)\, then\, (f@E_2)\, else\, (f@aE_3)$
17. $if\, (\mathcal{D} : E_1)\, then\, E_2\, else\, E_3 \Rightarrow \mathcal{D} : (if\, E_1\, then\, E_2\, else\, E_3)$
18. $if\, E_1\, then\, (\mathcal{D} : E_2)\, else\, E_3 \Rightarrow \mathcal{D} : (if\, E_1\, then\, E_2\, else\, E_3)$
19. $if\, E_1\, then\, E_2\, else\, (\mathcal{D} : E_3) \Rightarrow \mathcal{D} : (if\, E_1\, then\, E_2\, else\, E_3)$
20. $if\, (case\, E_1^1; \ldots E_n^1; esac)\, then\, E_2\, else\, E_3$
    $\Rightarrow case\, (if\, E_1^1\, then\, E_2\, else\, E_3); \ldots (if\, E_n^1\, then\, E_2\, else\, E_3);\, esac$
21. $if\, E_1\, then\, (case\, E_1^2; \ldots E_n^2; esac)\, else\, E_3$
    $\Rightarrow case\, (if\, E_1\, then\, E_1^2\, else\, E_3); \ldots (if\, E_1\, then\, E_n^2\, else\, E_3);\, esac$
22. $if\, E_1\, then\, E_2\, else\, (case\, E_1^3; \ldots E_n^3; esac)$
    $\Rightarrow case\, (if\, E_1\, then\, E_2\, else\, E_1^3); \ldots (if\, E_1\, then\, E_2\, else\, E_n^3);\, esac$

4.5 Array Notation

For readability, an abbreviated notation is used for dependence expressions in parts of the paper. In the examples we encounter, the parent of a variable expression is almost always a dependence node. For example, let A be a variable with one-dimensional domain, and it is used by another expression with 3D domain. Then the variable must be accessed as $A.f$, where f is an affine function from $\mathbb{Z}^3$ to $\mathbb{Z}^1$. For example, if the dependence function is $(i, j, k \rightarrow k)$, reading the value from A is $A.(i, j, k \rightarrow k)$.

However, when the index names are unambiguous from the context, we use array notations and only write the RHS of the function. For the above example, the corresponding expression in array notation is $A[k]$ when it is clear from the "context" that the indices for 1st to 3rd dimensions are named i, j, k.

Array notation was created to allow dependence to variables resemble array accesses. In addition to dependences, there are other syntactic convenience that rely on context information. These context-sensitive syntax are collectively called array notations. Other array notations in `Alpha` are:

- Index Expressions may use array notations similar to dependences. (e.g., `val`$(i, j, k \rightarrow i - j + 10)$ may be written as $[i - j + 10]$)
- Restrict Expressions may omit the index names in its restrict domain. Index names of restrict domains must either be fully given or fully omitted, and names from the context are used when omitted. (e.g., `{|0<=i<N}`).
- Reduce Expressions may specify its projection function using array notation. Array notation for projection functions is only applicable when the function is canonic. With canonic projections, index names for new dimensions may be specified with array notation. For example, `reduce(+, (i,j->i), ...)` may be written as `reduce(+, [j], ...)` .

Context of array notations are either defined by the LHS of the surrounding equation, or by a surrounding reduce expression. In the LHS of the equation, the variable name may be accompanied with a list of index names. When index names are given in the LHS, then those names are treated as the context in its RHS expressions. However, this context may be over written by a reduce expression. When a reduce expression is encountered, all its children will now be in a different context, defined by the LHS of the projection function. New index names will be simply appended if the projection function was specified with array notation.

For example, consider the following equation:

```
A[i,j] = X[i,j] + reduce(+, (x,y,z->x,z), Y[x-y+z]);
```

Access to `X` uses the context defined by the LHS of the equation, but access to `Y` uses the context defined by the reduce expression.

4.6 Example

We take a simple computation, matrix multiplication to illustrate basic syntax of the language. Matrix multiplication using reduction is written as follows in alphabets:

```
affine matrix_product {N|N>0}
   input
      double A,B {i,j|0<=(i,j)<N};
   output
      double C {i,j|0<=(i,j)<N};
   let
      C = reduce(+, (i,j,k->i,j), (i,j,k->i,k)@A * (i,j,k->k,j)@B);
.
```

Note that we only use one program parameter in the above example to avoid clutter, making it a square matrix multiplication.

Matrix multiplication can be written as follows without using reductions:

```
affine matrix_product {N|0<N}
   input
      double A,B {i,j|0<=(i,j)<N};
   output
      double C {i,j|0<=(i,j)<N};
   local
      double temp_C {i,j,k|0<=(i,j,k)<N};
   let
      C = (i,j,k->i,j,N-1)@temp_C;
      temp_C = case
             {i,j,k|k==0} : (i,j,k->i,k)@A * (i,j,k->k,j)@B;
             {i,j,k|k> 0} : (i,j,k->i,k)@A * (i,j,k->k,j)@B
                              + (i,j,k->i,j,k-1)@temp_C;
          esac;
.
```

Without reductions, the program must explicitly specify dependences for accumulation of the result matrix C. In the above program, accumulation is performed in a local variable `temp_C`.

The matrix multiplication example in array notation is written as follows:

```
affine matrix_product {N|0<N}
   input
      double A,B {i,j|0<=(i,j)<N};
   output
      double C {i,j|0<=(i,j)<N};
   let
      C = reduce(+, [k], A[i,k] * B[k,j]);
.
```

```
affine matrix_product {N|0<N}
   input
      double A,B {i,j|0<=(i,j)<N};
   output
      double C {i,j|0<=(i,j)<N};
   local
      double temp_C {i,j,k|0<=(i,j,k)<N};
   let
      C[i,j] = temp_C[i,j,N-1];
      temp_C[i,j] = case
             {|k==0} : A[i,k] * B[k,j];
             {|k> 0} : A[i,k] * B[k,j] + temp_C[i,j,k-1];
          esac;
.
```

4.7 Subsystems and Use Equations

A *Subsystem* [6] is an affine system called by another affine system. Calls to subsystems are expressed as a different type of equations, named *Use Equation.* Subsystems are introduced to allow structured/modularlized specification of equations. The syntax of use equation is:

```
use <Extension Domain> <name>[<parameters>] (<inputs>) returns (<outputs>);
```

where

- *Extension Domain* specifies the instances of the subsystem. The subsystem is called for each point in the extension domain.

- `<name>` is the name of the subsystem being called. The signature of the subsystem (parameter domain, input and output variables) must be known. The current parser requires the full subsystem to be defined in the same file.
- `<parameters>` are the values of the subsystem parameters. The parameters are given as affine expressions of the (caller) system parameters, and the indices of the extension domain.
- `<inputs>` are a list of `Alpha` expressions that correspond to the inputs of *all instances* of the subsystem. The first dimensions correspond to the extension domain, and the rest to the domain of the inputs of the subsystem.
- `<outputs>` are a list of variables in the caller system, where the result of use equation becomes its value. Similar to the inputs, all instances of subsystems are mapped to the same variable.

For example, matrix multiplication can be implemented using subsystem that computes dot product of two vectors. The following `Alpha` program is an example of such implementation.

```
affine mat_product {N,M,K | N>0 && M>0 && K>0}
input
    long A {i,k | 0<=i<N && 0<=k<K};
    long B {k,j | 0<=k<K && 0<=j<M};
output
    long C {i,j | 0<=i<N && 0<=j<M};
let
    use {row,col|0<=row<N && 0<=col<M} dot_product[K]
            ((r,c,k->r,k)@A,(r,c,k->k,c)@B) returns (C);
.

affine dot_product {N | N>0}
    input
        double vector1, vector2 {i | 0<=i<N };
    output
        double result;
    let
        result = reduce(+, (i->), vector1[i] * vector2[i]);
.
```

In the example above, there are $N \times M$ instances of subsystems being called by the use equation. Each instance (row, col) computes the value $C[row, col]$ using the subsystem.

Valid use equations must satisfy the following:

- The number of parameters/inputs/outputs must match the subsystem.
- The parameters specified must be in the parameter domain of the subsystem.
- The expression domain of the input expressions must include the product of the extension domain and of the domain of the inputs of the subsystem.
- The variable domain of the outputs of the subsystem must include the product of the extension domain and of the domain of the output variables in the main system.
- There must be no cycles in the call chain of subsystems.

5 AlphaZ Transformations

In this section, we provide an overview of available transformations in AlphaZ. These transformations are made available in the script environment as commands. Most commands take an instance of Program object, and a String specifying target system name as its first two input arguments. Some commands that affect the entire program, rather than a system, may only take Program object as its only argument.

The following is a common first two lines of a compiler script, where the first line reads an `Alpha(bets)` program and assigns to `prog`, and the second line assigns the name of target system to apply transformations to `system`.

```
prog = ReadAlphabets("example.ab");
system = "exampleSystem";
```

Note that the complete list of overloadings available for convenience is not presented in this paper. Please refer to the command reference (`http://www.cs.colostate.edu/AlphaZ/AlphaZCommandRef.pdf`) for details.

5.1 Expression Identifier

Inputs to some of the transformations may require specifying an expression in the program. We use an unique identifier of expressions for specifying expressions in such cases. Expression IDs are based on the *path* from the root node to the target expression in the Abstract Syntax Tree of `Alpha`. The ID is represented as a vector of integers, each integer representing which branch to take (first branch being 0) in the traversal of the AST. For example, consider the following `Alpha` program:

```
affine example {N|N>10}
    input A {i|10<=i<N};
    output X {i|0<=i<N};
    let
    X[i] = case
                {|i==0} : 0;
                {|0<i<10} : [i-1];
                A[i];
           esac;
.
```

Expression ID $[0, 0, 0]$ corresponds to the case expression; the first system, first equation, and first expression. Expression ID $[0, 0, 1, 0]$ corresponds to the index expression `[i-1]`; the first system, first equation, second branch, first expression.

These identifiers, also called nodeIDs, are also shown in the AST printed out by `PrintAST` command.

5.2 Basic Commands

There are a number of basic commands, including cosmetic transformations, and commands for visualizing the current state of `Alpha` programs.

5.2.1 `ReadAlphabets`

`ReadAlphabets(String filePath)`

Reads an `Alpha(bets)` program from a file. The expression domains and context domains are computed at the time of parsing, and incorrectly specified dependences or domains may cause exceptions [7] to be thrown. One common cause of such exceptions is mis-matched number of dimensions, when performing polyhedral operations.

[7]Error messages some times may be cryptic, returning raw messages from ISL or other back-ends.

5.2.2 `RenameSystem` and `RenameVariable`

```
RenameSystem(Program program, String system, String newSystemName)
RenameVariable(Program program, String system, String varName, String newVarName)
```

The above commands will replace system/variable name and update all references as well. Some of the transformations in AlphaZ introduce new systems/variables, and may use default naming scheme when doing so. These commands may be used to give names that make more sense to the programmer.

5.2.3 `RemoveUnusedVariables`

```
RemoveUnusedVariables(Program program)
```

This commands removes unused variables from a program or a system. The unused variables are determined after inspection of Reduced Dependence Graph, and are defined as variables that are not reachable from any outputs, following the dependence edges in the RDG.

5.2.4 `Normalize`

```
Normalize(Program program)
```

Applies the normalization rules described in Section 4.4.

5.2.5 `Show`, `AShow`, `Save`, and `ASave`

```
Show(Program program)
AShow(Program program)
Save(Program program, String outDir)
ASsave(Program program, String outDir)
```

`Show` and `AShow` pretty prints the `Alpha` program. `AShow` uses array notations when applicable. `Save` and `ASave` saves the result of `Show` and `AShow` respectively to the specified file.

5.2.6 `PrintAST`

```
PrintAST(Program program)
```

This command prints out the Abstract Syntax Tree of `Alpha` programs. AST also includes expression and context domains.

5.3 `CoB`: Change of Basis

```
CoB(Program program, String systemName, String varName, String function)
```

Change of Basis (CoB) is a transformation that has a very wide applicability. This transformation takes as inputs, the target variable X and a transformation function $\mathcal{T}$. The transform $\mathcal{T}$ must admit a *left inverse* for all points in the domain of X, $\mathcal{D}_X$. Given these inputs the following transformations are applied:

- Replace the variable domain $\mathcal{D}_X$ by $\mathcal{T}(\mathcal{D}_X)$, the image of $\mathcal{D}_X$ by $\mathcal{T}$.
- On the RHS of the equation for X, replace each dependence f by $f \circ \mathcal{T}^{-1}$; the composition of f and $\mathcal{T}^{-1}$.
- On the RHS of *any* equation, replace $f\texttt{@}X$ (dependence on the target variable) by $(\mathcal{T} \circ f)\texttt{@}X$.

The occurrences of X on the RHS of the equation for X itself constitute a special case where the last two rules are both applicable, and we replace the dependence f by $\mathcal{T} \circ f \circ \mathcal{T}^{-1}$.

The effect of this transformation is essentially transforming the domain of X by $\mathcal{T}$, and the latter two rules are necessary modifications to the dependences to preserve semantics.

5.4 Inline

`Inline(Program program, String systemName, String targetEq, String inlineEq)`

Inline is a transformation that replaces variables in a target equation with its definition. The following example illustrates one of its uses. Consider the following two equations:

```
X[i] = foo(A[i], B[i]);
Y[i] = bar(X[i]);
```

Applying `Inline` of `X` to `Y` yields the following:

```
X[i] = foo(A[i], B[i]);
Y[i] = bar(foo(A[i], B[i]));
```

As a result of inlining, the dependence from `Y` to `X` has been replaced to dependences to `A` and `B`. If `A` and `B` are input variables, then there are less dependences to be considered at scheduling, and hence may expose more opportunities. However, since `X[i]` is no longer used, `foo(A[i], B[i])` is re-computed when computing `Y[i]`. Depending on the cost of `foo`, re-computation may be too costly to exchange for increased scheduling flexibility.

5.5 Split

`Split(Program program, String systemName, String varName, String sepDomain)`

`Split` transforms a variable `V` into two, where the split is defined by the split domain $\mathcal{D}_{\mathcal{S}}$ The equation of the form

$$X = E;$$

is replaced by two equations

$$X_1 = E;$$
$$X_2 = E;$$

where the domains of X_1 and X_2 are $\mathcal{D}_X \setminus \mathcal{D}_S$ and $\mathcal{D}_S$ respectively.

The transformation can be used to separate the main part of the equation from boundary conditions. In addition, splitting an equation to multiple pieces is a way to express *piece-wise affine* schedules that are not currently expressible as Target Mapping.

5.6 SplitUnion

`SplitUnion(Program prog, String nodeID)`

`SplitUnion` takes an expression E with domain $\mathcal{D}_E$ is a *disjoint* [8] union of polyhedra $\mathcal{D}_1 \cup \mathcal{D}_2 \cup \cdots \cup \mathcal{D}_n$, and replaces it with the following case expression:

$$\begin{aligned} &\textit{case} \\ \mathcal{D}_1 &: E; \\ \mathcal{D}_2 &: E; \\ &\ldots \\ \mathcal{D}_n &: E; \\ &\textit{esac} \end{aligned}$$

[8] We apply a pre-processing call to ISL to convert unions into disjoint unions of polyhedra

5.7 `Simplify`

`Simplify(Program prog)`

`Simplify` is a transformation that attempts to *simplify* `Alpha` programs in various ways. The current implementation applies the simplifications listed below.

- Simplify in Context: Replaces the restrict domain $\mathcal{D}$ of each restrict expression E by $\mathcal{D}'$ such that $\mathcal{D}' \cap \mathcal{X}_{E'} = \mathcal{D}$, where $\mathcal{X}_{E'}$ is the context domain of its parent. This operation on domains, also called *gist*, essentially removes redundant constraints available in the context from a domain. Thus, the resulting program after simplification may appear to have less constraints, but the expression/context domains of expressions remain the same.
- Detection of Scalar Reductions: Detects reductions that reduce over domains that consist of a single point, and replaces the reduction with its body evaluating the single point. The result of applying Simplifying Reductions [12] may contain equalities such that the domain of the reduction body is a single point. The detection of scalar reductions is imprecise, and there is no guarantee that all scalar reductions are replaced after `Simplify`.

5.8 `BuildPRDG` and `ExportPRDG`

```
BuildPRDG(Program program, String system
ExportPRDG(PRDG prdg, String filename)
```

`BuildPRDG` constructs a PRDG that represents dependences of an affine system. By defefault, input variables are excluded from the PRDG.

`ExportPRDG` takes a PRDG and outputs the PRDG as a `DOT` specification for visualizing the PRDG with tools such as `GraphViz`.

6 Reduction Transformations

One of the key features of AlphaZ is the explicit representation of reductions, and rich set of tools to manipulate them. There is a separate technical report that describes the derivation of $O(N^3)$ algorithm from $O(N^4)$ initial version using AlphaZ commands [39].

6.1 `SimplifyingReduction`

`SimplifyingReduction(Program program, String system, String varName, String reuseVector)`

Implementation of the Simplifying Reductions [12] transformation. The transformation take advantage of the reuse in the reduction and replaces reductions with scan computations. This result in a decrease in the asymptotic complexity, observed as domains of fewer dimensions (may be embedded into higher dimensions with equalities). Current implementation assumes that the given reuse vector is correct.

The input reduction is required to be in the following form:

$$X = \text{reduce}(\oplus, f_p, E) \tag{1}$$

where $\mathcal{D}_E$ is a single integer polyhedron and equal to $\mathcal{X}_E$. For simplicity of explanation, we have the reduction named by a computed variable X.

The Simplifying Reduction transformation takes as inputs; a reduction in the form of Equation 1, where $\mathcal{D}_E$ is a single integer polyhedron, and a *legal* vector specifying the direction of reuse r_E; and returns a semantically equivalent equation:

$$\begin{array}{rcl}
X & = & \\
\text{case} & & \\
(\mathcal{D}_{add} - \mathcal{D}_{int}) & : & X_{add}; \\
(\mathcal{D}_{int} - (\mathcal{D}_{add} \cup \mathcal{D}_{sub})) & : & X.(z \rightarrow z - r_X); \\
(\mathcal{D}_{add} \cap (\mathcal{D}_{int} - \mathcal{D}_{sub})) & : & (X_{add} \oplus X.(z \rightarrow z - r_X)); \\
(\mathcal{D}_{sub} \cap (\mathcal{D}_{int} - \mathcal{D}_{add})) & : & (X.(z \rightarrow z - r_X) \ominus X_{sub}); \\
(\mathcal{D}_{add} \cap \mathcal{D}_{int} \cap \mathcal{D}_{sub}) & : & (X_{add} \oplus X.(z \rightarrow z - r_X) \ominus X_{sub}); \\
\text{esac}; & & \\
X_{add} & = & \text{reduce}(\oplus, f_p, (\mathcal{X}_E - \mathcal{X}_{E'}) : E) \\
X_{sub} & = & \text{reduce}(\oplus, f_p, \\
 & & f_p^{-1}(\mathcal{D}_{int}) : (\mathcal{X}_{E'} - \mathcal{X}_E) : E')
\end{array}$$

where $E' = E.(z \rightarrow z - r_E)$, $r_X = f_p(r_E)$, $\ominus$ is the inverse of $\oplus$, $\mathcal{D}_{add}$, $\mathcal{D}_{sub}$ and $\mathcal{D}_{int}$ denote the domains $f_p(\mathcal{X}_E - \mathcal{X}_{E'})$, $f_p(\mathcal{X}_{E'} - \mathcal{X}_E)$ and $f_p(\mathcal{X}_E \cap \mathcal{X}_{E'})$ respectively, and X_{add} and X_{sub} are defined over the domains $\mathcal{D}_{add}$ and $\mathcal{D}_{int} \cap \mathcal{D}_{sub}$ respectively.

6.2 `PermutationCaseReduce`

`PermutationCaseReduce(Program program, String systemName, String targetVar)`

Permutation Case Reduce, presented as a theorem by Le Verge [20], takes reduce expression of the form

$$E = \text{reduce}(\oplus, f_p, \text{case } E_1; E_2;\ \text{esac})$$

and returns a semantically equivalent equation:

$$\begin{array}{l}
E = \text{case} \\
\quad \mathcal{D}_1 : X_1; \\
\quad \mathcal{D}_{12} : (X_1 \oplus X_2); \\
\quad \mathcal{D}_2 : X_2; \\
\quad \text{esac};
\end{array}$$

where $\mathcal{D}_{12} = f_p(\mathcal{D}_{E_1}) \cap f_p(\mathcal{D}_{E_2})$, $\mathcal{D}_1 = f_p(\mathcal{D}_{E_1}) \setminus f_p(\mathcal{D}_{E_2})$, $\mathcal{D}_2 = f_p(\mathcal{D}_{E_2}) \setminus f_p(\mathcal{D}_{E_1})$, and X_1, X_2 are defined as follows:

$$X_1 = \text{reduce}(\oplus, f_p, E_1)$$
$$X_2 = \text{reduce}(\oplus, f_p, E_2)$$

The transformation essentially moves case expressions out of the reduction. Since the simplification transformation requires that the domain of the reduction body to be a single polyhedron, and not unions of polyhedra, case expressions must be moved out.

6.3 `ReductionDecomposition`

`ReductionDecomposition(Program program, String nodeID, String f1, String f2)`

An reduce expression of the form

$$\text{reduce}(\oplus, f_p, E)$$

is semantically equivalent to

$$\text{reduce}(\oplus, f''_p, \text{reduce}(\oplus, f'_p, E))$$

where $f_p = f''_p \circ f'_p$.

This transformation uses the above to decompose a reduction into two reductions. $\mathtt{f1} \circ \mathtt{f2}$ should match the original projection function f_p ($\mathtt{f1} = f''_p$, $\mathtt{f2} = f'_p$).

6.4 `NormalizeReduction`

`NormalizeReduction(Program program)`

Normalize Reductions is a transformation that takes expression containing reductions

$$E = \cdots \text{reduce}(\oplus, f_p, E_1) \cdots$$

and isolates reductions by adding vairables:

$$E = \cdots X \cdots$$
$$X = \text{reduce}(\oplus, f_p, E_1)$$

After this transformation, all reduce expression in the Alphabets program will be top-level expressions (the first expression in the right hand side of an equation). This is purely a pre-processing to obtain reductions of the form required by the simplification algorithm. We also provide another transformation, called Inline, to replace variables with its definition, so that the variables introduced by this transformation can eventually be removed.

6.5 `FactorOutFromReduction`

`FactorOutFromReduction(Program program, String nodeID)`

This transformation takes advantage of distributivity to factor out expressions from reductions. The number of operations may be significantly reduced, and it can also expose additional opportunities for applying SimplifyingReduction. The transformation is more formally discussed by Gupta and Rajopadhye [12]. The following is a stripped down description of the transformation.

Consider a reduction of the following form

$$E = \text{reduce}(\oplus, f_p, E_1 \otimes E_2)$$

where $\otimes$ distributes over $\oplus$.

If one of the expressions is constant within the reduction (E_1, say), we would be able to distribute it outside the reduction. For the expression E_1 to be constant within a reduction by the projection f_p, we require

$$\mathcal{H}_{\mathcal{D}_E} \cap \ker(f_p) \subseteq \mathcal{H}_{\mathcal{D}_E} \cap \mathcal{S}_{E_1}$$

where $\mathcal{H}_{\mathcal{D}}$ is defined as the linear part of the smallest affine subspace containing $\mathcal{H}_{\mathcal{D}}$. $\mathcal{H}_{\mathcal{D}}$ becomes important when the domains contain equalities. After distribution, the resultant expression is

$$E_1 \otimes \text{reduce}(\oplus, f_p, E_2)$$

6.6 `SerializeReduction`

`SerializeReduction(Program program, String nodeID, String schedule)`

`SerializeReduction` takes a reduction and a full-dimensional schedule (schedule that give all points a unique time stamp) for the reduction body, and serializes the reduction using the schedule. Serialization is a transformation of reductions to accumulation via chains of dependences by selecting the order of accumulation.

For example, consider the following reduction:

$$X = \text{reduce}(\oplus, (i \rightarrow), A(i))$$

where A is defined over $\{i|0 \leq i < N\}$. The above reduction can be serialized given a schedule $(i \rightarrow i)$ to obtain the following equations:

$$X = X'(N-1)$$
$$X'(i) = \begin{cases} i = 0 : A(i); \\ i > 0 : X(i-1) \oplus A(i); \end{cases}$$

The first and last points are respectively the lexicographical minimum and maximum of the domain of the reduction body, with respect to the given schedule.

6.7 `SplitReductionBody`

`SplitReductionBody(Program program, String nodeID, String splitDomain)`
`SplitReductionBody` takes a split domain $\mathcal{D}_S$ and a reduction of the form

$$\text{reduce}(\oplus, f_p, E)$$

with the expression domain of the reduction denoted as $\mathcal{D}_{E_{orig}}$, and transforms the reduction into two reductions combined by the reduction operator

$$\begin{array}{c}\text{reduce}(\oplus, f_p, \mathcal{D}_1 : E) \\ \oplus\text{reduce}(\oplus, f_p, \mathcal{D}_2 : E)\end{array}$$

where $\mathcal{D}_1 = \mathcal{D}_S$, and $\mathcal{D}_2 = \mathcal{D}_{E_{orig}} \setminus \mathcal{D}_S$.

6.8 `TransformReductionBody`

`TransformReductionBody(Program program, String nodeID, String T)`

`TransformReductionBody` takes a affine function $\mathcal{T}$ and a reduce expression of the form

$$\text{reduce}(\oplus, f_p, E)$$

and applies the following transformations:

- Replace E with $\mathcal{D} : E$ where $\mathcal{D} = \mathcal{T}(\mathcal{X}_E)$, the image of context domain of the reduction body by $\mathcal{T}$.
- Replace all variable expressions `V` in the sub-tree of E with $\mathcal{T}^{-1}$@`V`

The domain of the reduction body is transformed by $\mathcal{T}$, while all dependences on variables are composed by the inverse of $\mathcal{T}$ to preserve semantics.

7 Target Mapping

In this section we describe the Target Mapping (TMap) for specifying execution strategies. TMap consists of three main specifications, schedule to define when to compute, processor allocation for where to compute, and memory allocation for where to store the results. We will use a simple 3-point stencil computation with 1D data, shown in Figure 1, as an example in this section.

TMap is used for exploring transformations described as the combination of schedule, processor allocation, and memory allocations. In addition to these main axes, additional specifications such as tiling and synchronization is also specified in TMap. The main specifications are specified per "variable" in alphabets programs, which corresponds to statements in ACLs.

7.1 Space-Time Mapping

We use multi-dimensional affine functions to jointly represent schedules and processor allocation. We call this multi-dimensional mapping the space-time (ST) mapping. Space-time mapping maps domain of variables to another domain, where each dimension will eventually corresponds o a loop in the generated code.

The following restrictions apply to the given mapping:

- The destination dimension must have the same number of dimensions for all variables. Since all statements must be placed relative to each other, all statements must be scheduled in a common space.

```
affine jacobi1D {N,T|N>0 && T>0}
   input
      double Ain {i|0<=i<N}
   local
      double A {t,i|0<=i<N && 0<=t<=T}
   output
      double Aout {i|0<=i<N}
   let
      A[t,i] = case
                 {t==0} : Ain[i,j]
                 {t>0 && 1<=i<N-1} :
                        (A[t-1,i] + A[t-1,i-1] + A[t-1,i+1]) * 0.333333;
                 {t>0 && i==0} || {t>0 && i==N-1} : A[t-1,i];
               esac;
     Aout[i] = A[T,i];
.
```

Figure 1: `Alpha` specification of 3-point Jacobi stencil.

- The mapping must be bijective, so that the statements are executed for the appropriate point in its domain.

Space-time mapping can be seen as multi-dimensional schedule [10], and some times referred to as full-dimensional schedule. In this paper, the non-parallel dimensions of the space-time mapping are referred to as schedules.

7.1.1 Schedule

We first describe the case when the entire space-time mapping is the schedule; sequential programs. A large number of loop transformations can be expressed with multi-dimensional affine schedules, and they are used by most program transformation tools using the polyhedral model.

For example, the schedule `(i,j->j,i)` can be viewed as a loop permutation when compared to another schedule `(i,j->i,j)`. Similarly, the schedule `(t,i->t,t+i)` corresponds to loop skewing, skewing i loop by t.

How a user may specify the above schedule for loop skewing in our script is illustrated below:

```
prog = ReadAlphabets("jacobi1D.ab");
SetSTMap(prog, "A", "(t,i->t,i+t)");
```

`SetSTMap` command is used to specify space-time mapping for each variable.

7.1.2 Processor Allocation

In AlphaZ, processor allocation is part of the space-time mapping, and are distinguished from the time component of the ST mapping by annotations given to dimensions. After specifying the space-time mapping, certain dimensions may be flagged as parallel using `SetDimensionType` command.

SetSTMap command is first used to specify lexicographic schedule, and then `SetDimensionType` command is used to flag the inner dimensions as parallel. Dimensions of ST mappings are specified as integer index starting from 0.

The script to specify ST mapping for executing the inner loop of Jacobi 1D stencil is the following:

```
prog = ReadAlphabets("jacobi1D.ab");
SetSTMap(prog, "A", "(t,i->t,i)");
SetSTMap(prog, "Aout", "(i)->(T-1,i)");
SetDimensionType(prog, 1, "parallel");
```

7.1.3 Ordering Dimensions

In addition to sequential and parallel dimensions, another possible dimension type is called the ordering dimension. When generating imperfect loop nests from polyhedral representation, the common practice is to use additional dimensions, with constant values, to denote ordering of the loops. For example, given two statements `S1` and `S2` with the same domain $\{i, j | 0 \leq (i, j) < N\}$, generating code with schedules $\theta_{S1} = (i, j \rightarrow i, 0, j)$ and $\theta_{S2} = (i, j \rightarrow i, 1, j)$ will produce:

```
for (i=0; i < N; i++)
  for (j=0; j < N; j++)
    S1
  for (j=0; j < N; j++)
    S2
```

7.2 Memory Mapping

Memory allocation is specified through multi-dimensional affine functions with dimension-wise modulo operation, called modular mapping or pseudo projections. This representation is commonly used by existing approaches for optimal memory allocation [21, 29, 31, 32]. Another representation using integer lattices may also be represented as modular mappings [5].

For the 3-point Jacobi stencil example, memory allocation corresponding to the commonly used "ping-pong" style allocation is specified by the following script:

```
prog = ReadAlphabets("jacobi1D.ab");
SetMemoryMapping(prog, "A", "(t,i->t,i)", "2,0");
```

Two arrays of size `N` are used alternately in each iteration of the time loop, achieving the ping-pong style computation of Jacobi stencil. The convention is to treat modulo factor 0 as projection without modulo operations.

7.2.1 Reductions

The reduction expressions have two possible ways to specify schedules. One is to view each reduction as "atomic" operations, and schedule the domain of the answers of reductions. The other is to assign time stamps to each point in the body of reductions.

The latter provides more control over how the computations are executed, while the former is a simpler abstraction. Existing scheduling techniques for programs with reductions only handle the former case, and scheduling reduction bodies is an open problem.

The latter option is only applicable if the reduce expression is the top-most expression of an equation. Then the space-time mapping given to the variable on the LHS of the equation can be specified to schedule the reduction body. In such cases, the LHS of the STmap must have the dimensionality equal to that of the reduction body.

Since reduction involves projection, the dimensionality is usually different between the reduction body and the answer space, and thus the same command `SetSTMap` may be used for both purposes.

7.3 Additional Specifications

TMap described above forms the basis for specifying execution strategies orthogonal to the specification of *what* is computed. However, such a TMap is not yet complete. Additional, optional, specifications will be needed for code generators to accommodate other strategies such as tiling and synchronization, and possibly code generator specific options. The purpose of TMap is to decouple optimization strategies from the input specification and code generators, so that design space exploration and orthogonal specification are both possible. Thus, it is best if options for future code generators are also decoupled and exposed through TMap.

7.3.1 Tiling

Tiling is a well known loop transformation for data locality and extracting coarse grained parallelism [15, 37]. The additional specification required for tiling is which dimensions are to be tiled, and tile sizes.

Currently, the ScheduleC code generator assumes that all dimensions are tilable when applying tiling. Tiling can be specified by the command `SetTiling`, and can be either sequential or parallelized by OpenMP.

8 The Verifier

Verifier is a key component of AlphaZ that verifies the Target Mapping specified for a program. Since AlphaZ allow user to specify the target mapping, the specified target mapping may be incorrect. The verifier is invoked by the command `VerifyTargetMapping`, which takes the program, system name, and verbosity option as inputs.

8.1 Overview

The verification is used before the code generator is called to ensure the given TM is legal for the input specification. In addition, the verifier can be used to check if a TM is legal while a user is exploring possible executions of a program. The user/auto-tuner may iterate a number of times between the verifier and TM selection to find a legal TM.

The verifier first checks for legality of space-time mapping. This is because almost all the currently known algorithms for checking the legality of a memory allocation require that the schedule be given. We use formulation of legality used in the literature for scheduling and finding memory allocations.

The verifier checks all dependences one-by-one, dimension-by-dimension, to provide detailed feedback to the user. For example, the user will receive feedbacks like: the causality of dependence $S1 \rightarrow S2$ is violated by the d-th dimension of the space-time mapping for domain $\mathcal{D}$.

In the following, we briefly describe how legality checks are performed. For technical details of how the verifier works, there is a Master's thesis that describes the verifier and its implementation citevamshi-thesis.

8.2 Verification of Space-Time Mappings

As we discussed in Section 7, Space-Time mapping is an affine function that represents both schedule and processor allocation. However, a valid Space-Time mapping is also a valid schedule, and thus we first describe the verification of schedule, when all dimensions of the space-time mapping corresponds to time or statement ordering.

8.2.1 Verification of Schedule

We use Feautrier's formulation [10, 11] to verify schedule. The causality condition formulated by Feautrier [11, 10] states that $\forall z \in \mathcal{D}_E, \theta_C(z) \gg \theta_P(E_f(z)) + \delta$ must hold for the dependence E to be satisfied, where

- E is a PRDG edge,
- P is the producer PRDG node,
- C is the consumer PRDG node,
- $\mathcal{D}_E$ is the domain of the edge,
- E_f is the dependence function,
- θ_P is the schedule of producer node,
- θ_C is the schedule of producer node, and
- δ is a constant vector denoting the delay of the scheduled operation.

The delay δ above is assumed to be 1 in the last dimension and 0 otherwise, since scheduling with delays were more applicable in hardware synthesis context.

Using the above, we can classify each dependence in the PRDG in to the following three states:

- Strictly Satisfied: The causality condition is satisfied with non-zero δ.
- Weakly Satisfied: The causality condition is satisfied with δ being the zero vector.
- Violated: The causality condition cannot be satisfied, even with $\delta = \vec{0}$.

Furthermore, the above three states may refer to a particular dimension of the schedule. We call a dependence to be strictly satisfied at dimension d, if the causality condition holds on dimension d, and there are no dimensions before d that also strictly satisfies the dependence.

For example, consider an edge in the PRDG from node `A` to `B`, with a dependence function $(i, j, k \rightarrow i, j-1, k+1)$. Let the schedules for `A` and `B` be the identity function $(i, j, k \rightarrow i, j, k)$, then the following observations can be made.

- The dependence is strictly satisfied.
- The dependence is strictly satisfied at dimension 1.
- The dependence is weakly satisfied if only dimension 0 is considered.
- The dependence is violated if only dimension 2 is considered.

The above notions become a useful building blocks for reasoning about legality of various Target Mapping specifications.

8.2.2 Verification of Processor Allocation

Verifying the space-time mapping requires a few additional checks to be performed.

We require that the dimensions annotated as parallel to satisfy the following:

- The dependence must *not* be strictly satisfied by the parallel dimension. If the dependence is satisfied by the parallel dimension then that dependence is carried by the parallelized loop.
- All dimensions that share the common ordering dimension prefix must all be annotated as parallel. These dimensions with common statement/loop ordering is likely to become a single loop in the generated loop nest. Therefore, this condition is necessary to avoid situations where a loop is both flagged as sequential and parallel.

8.2.3 Reductions

The schedules given to reductions are verified differently depending on how the reductions are scheduled. Recall that the reductions may be treated as an atomic operation, where the schedule is given for the answer of reductions, or the body may be scheduled.

In the former case, the verifier checks that all inputs to a reduction is available before the entering the reduction. In the latter case, the verifier first serializes the reduction using the given schedule (with `SerializeReduction`), and the resulting equation is treated normally.

8.2.4 Tiling

Tiling requires that all loops are fully permutable. This translates to ensuring all dimension being tiled to be weakly satisfied by the schedules.

8.3 Verification of Memory Mappings

The verifier will ensure that the given memory mapping do not overwrite a value before its last use according to the given schedule. However, we current do not handle the legality of modulo factors, and we also do not handle the legality of memory mapping when tiling is applied.

We use the Quillér-Rajopadhye formulation [29] for verifying memory allocation. However, the formulation is slightly different as we formulate the "illegal" cases, and ensure that it is empty.

The main idea is to ensure that a value is not overwritten before its last use. The values used by the consumer C at z is overwritten by a point z' of the producer P if there exists z' that satisfies:

$$\begin{aligned}
&z \in \mathcal{D}_E \\
&z' \in \mathcal{D}_P \\
&M_P(E_f(z)) = M_P(z') \\
&\theta_C(z) \gg \theta_P(z') \gg \theta_P(E_f(z))
\end{aligned}$$

where, in addition to the notations in Section 8.2.1,

- M_P is the memory mapping of producer node.

In words, if z' writes to the same memory location as the value used by the consumer $(E_f(z))$, and if it is scheduled between the consumer and the producer, then there is a write conflict.

9 Code Generators

There are several code generators available in AlphaZ, and more are being developed. The code generator takes a program with Target Mapping and produces executable code. Current code generators all target C as the target output language.

How the Target Mapping is used largely depends on the code generator. Some specialized code generators will ignore certain specifications, or partially modify to fit their purpose.

In this section, we describe the three (relatively) stable code generators, and a code generator that produces template wrappers for calling the codes generated.

- WriteC: Demand driven code generator that do not require any TMap specification.
- ScheduledC: Code generator that fully respects the TMap specification. This code generator is intended to be the core for more specialized code generations.
- ScanC: Generates OpenMP parallel programs that parallelize scan computations.
- WrapperC: Code generator to produce code to interface with codes generated by other code generators.

The basic interface to AlphaZ generated code is function calls. The generated code includes a function for each system in `Alpha`, where all the inputs and parameters are given as function arguments. In the generated C code, we also take pointers to output arrays as function arguments.

9.1 WriteC

```
generateWriteC(Program prog, String system, String outDir)
```

WriteC is designed to provide executable code for any legal input specification without specifying any additional information, in other words, not even a schedule is specified. This code generator does not need TMap to be given, indeed, if this code generator is called and a TMap is given, most of it will be ignored. However, it respects the memory allocation for input and output variables because this affects how the generated code interfaces with existing code. If the given program contains cyclic dependences, it is detected and flagged at run-time.

Demand driven code produced by this code generator traverses the dependences "backwards" from the outputs to find out the values required to compute the output. The order of evaluation is specified implicitly when a value is "demanded," hence the name demand-driven code generator.

However, a naive implementation of this execution strategy may introduce inefficiencies when a value is used multiple times. For example, the Fibonacci series `fib` defined as $\texttt{fib}(\texttt{n}) = \texttt{fib}(\texttt{n}-\texttt{1}) + \texttt{fib}(\texttt{n}-\texttt{2})$ will require the result of $\texttt{fib}(\texttt{n}-\texttt{2})$ when computing $\texttt{fib}(\texttt{n})$ and also when computing $\texttt{fib}(\texttt{n}-\texttt{1})$. To avoid such redundant computation, the demand-driven code uses memoization to keep track of previously computed values.

9.2 ScheduledC

`generateScheduledC(Program prog, String system, String outDir)`

ScheduledC produces loop programs that execute the computations in the order specified by the TMap. The STmap, memory mapping, and tiling specifications are all respected. The program produces C programs and uses OpenMP for parallelism.

CLooG [2] is used to generate loop nest that scans all equations in `Alpha`. The `Alpha` program is first normalized, and then each branch of the top-level case expression, if any, becomes a separate statement in the generated loop program. We use D-Tiling [18] to generate both sequential tiled code and tiled code with wave-front parallel execution of tiles.

9.3 ScanC

`generateScanC(Program prog, String system, String outDir)`

ScanC generates parallelized OpenMP code for a *Scan* (prefix computation). A scan is an operation that takes an associative binary operation $\oplus$ and a list of expressions

$$[e_0, e_1, \cdots, e_n]$$

and returns a sequence

$$[e_0, e_0 \oplus e_1, \cdots, e_0 \oplus e_1 \cdots \oplus e_n]$$

We adapted the approach by Merrill and Grimshaw [24], where the scan is performed in the following three phases:

I Local Reduction

II Global Scan

III Local Scan

In Phase I, each thread performs a local scan on the set of expressions they are responsible for. Then a thread performs a scan on the result of the local reductions. Finally in Phase III, each thread does a local scan with the proper seeding value from the global scan.

There are a number of additional optimizations for generating efficient OpenMP parallel implementation of scan. These optimizations as well as some performance evaluations are available in a separate paper [40].

9.4 WrapperC

`generateWrapper(Program prog, String system, String outDir)`

WrapperC is a code generator primarily used for testing purposes. The code generators presented above produce C functions that perform computations of an affine system in `Alpha` programs. WrapperC generated code takes care of allocating memory for input and output variables, initializing inputs, measuring execution time, and printing/verifying outputs.

There are a number of options for providing inputs and verifying outputs. In the generated code, C macro definitions are used to toggle the following options:

- `RANDOM`; if defined, initial values are randomly generated.
- `CHECKING`; if defined, prompts for initial values of input arrays to be manually entered, and prints out all values of the output.
- `NO_PROMPT`; if defined, suppresses texts prompting for values of initial array elements, as well as array indices the printed output. Intended to be used in conjunction with `CHECKING` when input/output is redirected from/to files.
- `VERIFY`; if defined, assumes the existence of `xxx_verify.c` and `xxx_verify` function, where `xxx` is the generated system name, and compares the outputs of `xxx` against those of `xxx_verify`. The `xxx_verify` function is a user provided implementation that should be used as the verification target.

9.5 Makefile

`generateMakefile(Program prog, String system, String outDir)`

Template Makefile for compiling generated programs can be generated by this command. The generated Makefile assumes that the wrapper was generated by `WrapperC`, in addition to the code produced by other code generators.

The generated Makefile provides rules for following combinations of `WrapperC` options:

- `make` compiles with `-DRANDOM`
- `make check` compiles with `-DCHECKING`
- `make check-noprompt` compiles with `-DCHECKING -DNO_PROMPT`
- `make verify` compiles with `-DCHECKING -DVERIFY`
- `make verify-rand` compiles with `-DRANDOM -DVERIFY`

10 Related Work

The polyhedral model has a long history, and there are many existing tools that utilize its power. Moreover, it is now used internally in the IBM X10 compiler family. We now contrast `AlphaZ` with such tools. The focus of our framework is to provide an environment to try many different ways of transforming a program. Since many automatic parallelizers are far from perfect, manual control of transformations can sometimes guide automatic parallelizers as we show later.

PLuTo is a fully automatic polyhedral source-to-source program optimizer tool that takes C loop nests and generates tiled and parallelized code [3]. It uses the polyhedral model to explicitly model tiling and to extract coarse grained parallelism and locality. Since it is automatic, it follows a specific strategy in choosing transformations.

Graphite is an optimization framework for high-level optimizations that are being developed as part of `GCC` now integrated to its trunk [25]. Its emphasis is to extract polyhedral regions from programs that `GCC` encounters, significantly more complex task than what research tools address, and to perform loop optimizations that are known to be beneficial.

`AlphaZ` is not intend to be full fledged compiler. Instead, we focus on intermediate representations that production compilers may eventually be able to extract. Although codes produced from our system can be integrated into a larger application, we do not insist that the process has to be fully automatic, thus expanding the scope of transformations.

PIPS is a framework for source-to-source polyhedral optimization using interprocedural analysis [14]. Its modular design supports prototyping of new ideas by *developers*. However, the end-goal is an automatic parallelizer, and little control over choices of transformations are exposed to the user.

Polyhedral Compiler Collections (PoCC) is another framework for source-to-source program optimizations, designed to combine multiple tools that utilize the polyhedral model [28]. Like `AlphaZ`, POCC also seeks to provide a framework for developing tools like Pluto, and other automatic parallelizers. However, their focus is oriented towards automatic optimization of C codes, and they do not explore memory (re)-allocation.

MMAlpha is another early system with similar goals to `AlphaZ` [16]. It is also based on the `Alpha` language. The significant differences between the two are that MMAlpha emphasizes hardware synthesis (therefore, considers only 1-D schedules, nearest-neighbor communication, etc.) It does not treat reductions as first class (the first thing an MMAlpha user does is to replace the reductions by "serialization"), and does no tiling. Moreover, it is based on Mathematica, and this limits its potential users by its learning curve and licensing cost. MMAlpha does provide memory reuse in pronciple, but in its context, simple projections that directly follow processor allocations are all that it needs to explore.

RStream from Reservoir Labs performs automatic optimization of `C` programs [23]. It uses the polyhedral model to translate `C` programs into efficient code targeting multi-cores and accelerators. Vasillache et al. [33] recently gave an algorithm to perform a limited form of memory (re)-allocation (the new mapping must *extend* the one in the original program). In addition, RStream is also fully automatic, while our focus is on being able to express and explore different optimization strategies. Moreover, their tool is a commercial, closed-source system (although they do mention source-level collaborations are possible).

CHiLL is a high-level transformation and parallelization framework using the polyhedral model [4]. It also allows users to specify transformation sequences through scripts. However, it does not expose memory allocation.

POET is a script-driven transformation engine for source-to-source transformations [38]. One of its goals is to expose parameterized transformations via scripts. Although this is similar to `AlphaZ`, POET does not check validity of the transformations, and relies on external analysis to verify the transformations in advance.

11 Future Additions

In this paper, we have presented the AlphaZ system for analysis, transformation, and code generation in the polyhedral model. The system is still under active development, and there are a number of extensions/additions planned.

Some of these additions are:

- Support for the full `Alphabets` language.
- Support for MPI based parallelization for distributed memory computing.
- Support for CUDA code generation to target GPGPUs.

[TODO:not complete]

References

[1] R. Bagnara, P.M. Hill, and E. Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1):3–21, 2008.

[2] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. pages 7–16, 2004.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Notices*, volume 43, pages 101–113. ACM, 2008.

[4] C. Chen, J. Chame, and M. Hall. Chill: A framework for composing high-level loop transformations. *U. of Southern California, Tech. Rep*, pages 08–897, 2008.

[5] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.

[6] F. de Dinechin, P. Quinton, and T. Risset. Structuration of the alpha language. In *Programming Models for Massively Parallel Computers, 1995*, pages 18 –24, oct 1995.

[7] F. Dupont de Dincehcin. *Systmes structurs d'quations rcurrentes : mide en œuvre dans le langage Alpha et applications*. PhD thesis, Universit de Rennes, IRISA, Rennes, janvier 1997.

[8] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[9] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[10] P. Feautrier. Some efficient solutions to the ane scheduling problem, Part II, Multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, 1992.

[11] Paul Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *Int. J. Parallel Program.*, 21(5):313–348, 1992.

[12] G. Gautam and S. Rajopadhye. Simplifying reductions. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 30–41, New York, NY, USA, 2006. ACM.

[13] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 147–157, New York, NY, USA, 2009. ACM.

[14] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *Proceedings of the 5th international conference on Supercomputing*, pages 244–251. ACM, 1991.

[15] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329. ACM, 1988.

[16] C. IRISA. The MMAlpha environment.

[17] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. page 332, 1995.

[18] DaeGon Kim and Sanjay Rajopadhye. Efficient tiled loop generation: D-tiling. In *The 22nd International Workshop on Languages and Compilers for Parallel Computing*, 2009.

[19] H. Le Verge. Reduction operators in alpha. In D. Etiemble and J-C. Syre, editors, *Parallel Algorithms and Architectures, Europe*, LNCS, pages 397–411, Paris, June 1992. Springer Verlag. See also, Le Verge Thesis (in French).

[20] H. Le Verge and P. Quinton. Un environnement de transformations de programmes pour la synthèse d'architectures régulières. 1992.

[21] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3-4):649–671, 1998.

[22] C. Mauras. *ALPHA: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. PhD thesis, L'Université de Rennes I, IRISA, Campus de Beaulieu, Rennes, France, December 1989.

[23] B. Meister, A. Leung, N. Vasilache, D. Wohlford, C. Bastoul, and R. Lethin. Productivity via automatic code generation for PGAS platforms with the R-Stream compiler. In *APGAS09 Workshop on Asynchrony in the PGAS Programming Model*, 2009.

[24] Duane Merrill and Andrew Grimshaw. Parallel scan for stream architectures. Technical report, Technical Report CS2009-14, Department of Computer Science, University of Virginia, 2009.

[25] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.A. Silber, and N. Vasilache. Graphite: Loop optimizations based on the polyhedral model for gcc. 2006.

[26] Louis-Noël Pouchet. Polybench/c 3.2. http://www.cse.ohio-state.edu/ pouchet/software/polybench/.

[27] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *IEEE/ACM Fifth International Symposium on Code Generation and Optimization (CGO'07)*, pages 144–156, San Jose, California, March 2007. IEEE Computer Society press.

[28] 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.

[29] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. Program. Lang. Syst.*, 22(5):773–815, 2000.

[30] S. Rajopadhye, G. Gupta, and DG. Kim. Alphabets: An Extended Polyhedral Equational Language. In Fujiwara Nakano, Bordim, editor, *Advances in Parallel and Distributed Computational Models*, Anchorage, AK, May 2011. IEEE Press.

[31] M.M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independent storage mapping for loops. *ACM SIGOPS Operating Systems Review*, 32(5):24–33, 1998.

[32] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. *ACM SIGPLAN Notices*, 36(5):232–242, 2001.

[33] N. Vasilache, B. Meister, A. Hartono, M. Baskaran, D. Wohlford, and R. Lethin. Trading off memory for parallelism quality. In *International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2012.

[34] Nicolas Vasilache. *Scalable Program Optimization Techniques In The Polyhedral Model.* PhD thesis, University of Paris-Sud 11, 2007.

[35] S. Verdoolaege. isl: An integer set library for the polyhedral model. *Mathematical Software–ICMS 2010*, pages 299–302, 2010.

[36] D. Wilde. A library for doing polyhedral operations. Technical report, IRISA, 1993.

[37] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. *ACM Sigplan Notices*, 26(6):30–44, 1991.

[38] Q. Yi. Poet: a scripting language for applying parameterized source-to-source program transformations. *Software: Practice and Experience*, 2011.

[39] T. Yuki, G. Gupta, T. Pathan, and S. Rajopadhye. Systematic implementation of fast-i-loop in UNAfold using AlphaZ. Technical report, Technical Report CS-12-102, Colorado State University, 2012.

[40] Y. Zou and S. Rajopadhye. Scan detection and parallelization in "inherently sequential" nested loop programs. In *International Symposium on Code Generation and Optimization (CGO)*, 2012.