Computer Science
Technical Report

Systematic Implementation of fast-i-loop in UNAfold using AlphaZ

Tomofumi Yuki, Tanveer Patahan, Gautam Gupta, and Sanjay Rajopadhye

May 31, 2012

Colorado State University Technical Report CS12-102

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

1 Introduction

In this paper, we show a detailed description of how a known optimization that reduce the complexity of RNA folding algorithm from $O(N^4)$ to $O(N^3)$ can be semi-automatically applied via an implementation of a technique called Simplifying Reductions [1].

RNA secondary structure prediction, or RNA folding, is a widely used algorithm in bio-informatics. The original algorithm has $O(N^4)$ complexity, but $O(N^3)$ algorithm has been previously proposed by Lyngso et al. [2]. However, there is no implementation of the $O(N^3)$ algorithm has been made publicly available to the best of our knowledge.

The complexity reduction takes advantage of "hidden scans" in collections of reductions, where results (possibly partial) of a reduction can be reused in computing other reductions. For example, consider the following where X_i, $0 \leq i < N$ is computed as sums of subsets of values in $A_i; 0 \leq i < N$.

$$X_i = \sum_{k=0}^{i} A_k$$

This is actually a prefix (scan) computation, and can be written as the following:

$$X_i = \begin{cases} i = 0 : A_i \\ i > 0 : A_i + X_{i-1} \end{cases}$$

Note that the former equation takes $O(N^2)$ time while the latter takes $O(N)$ time. This is the core of the algorithm, and simplifying reductions consists of collection of analyses and transformations to detect and transform such reductions to corresponding scan computations.

A much more complicated version of the above example was found by Lyngso et al. [2] in the RNA secondary structure prediction algorithm. However, implementing such optimization require significant restructuring of the program. Moreover, although the Simplifying Reductions algorithm include the necessary analyses to identify hidden scans, detecting scans from a real application is non-trivial.

This paper presents a systematic way of deriving reduced complexity implementation of a function in UNAfold software package [3], using the AlphaZ system. The paper is organized as follows. Section 2 introduces polyhedral equational model and AlphaZ [4] system that we use for manipulating RNA folding algorithm and derive reduced complexity algorithm. In Section 3, we illustrate the intuition behind the core technique, Simplifying Reductions [1] with an example. The essential elements of Simplifying Reductions are reviewed in Section 4, followed by a review of the algorithm to reach optimal complexity in Section 5. Finally, in Section 6, we step through the algorithm in Section 5 to deduce the reduced complexity algorithm.

We also include sources of equational language used in AlphaZ, before and after the transformation, as well as a script for AlphaZ that applies the sequence of transformations as appendices.

2 AlphaZ and Polyhedral Equational Model

In this section we provide the necessary background of the AlphaZ system and the polyhedral model to understand the intuition behind Simplifying Reductions.

2.1 The Polyhedral Model

The polyhedral model is a framework for program analyses and transformations. The strength of this model comes from its mathematical foundations. For example, closure properties provide composition of transformations applicable to a restricted class of programs or program sections. Feautrier [5] showed that a class of loop nests called Affine Control Loops (also called 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 computation, or dynamic programming.

In the polyhedral model, we represent each instance of each statement in a loop program as an *iteration point*, in a space called *iteration domain* of the statement. Hence, each instance of each statement is viewed

as an *operation* and *what* a program computed is completely specified by the set of operations and the interdependences between them. As noted by Feautrier, program memory and data-structures need not figure in this representation. Our IR essentially adopts this view of programs.

The iteration domain is described with 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.

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

For readability, we do not use matrices to represent the constraints and enumerate all inequalities. The dependences in the program are expressed as affine functions[1], expressed as $(z \rightarrow z')$, where z' consists of affine expressions of z.

2.1.1 Properties of Polyhedral Objects

One of the advantages of modeling the program using polyhedral objects is the rich 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. In this paper, we use one of such properties, the kernel of matrices as part of our analysis. The kernel of matrix A, $\ker(A)$, is the set of vectors x such that $Ax = 0$. The space characterized by the kernel describes the set of vectors that does not affect the result of the product. 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.

We also define the kernel of domains and affine functions to be the kernel 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 $\ker(A)$.

2.2 Polyhedral Equational Model

The polyhedral model has its origin in analyses of System of Recurrence Equations (SREs) [6], where a program is described as a system of equations, with no notion of schedule or memory. Hence, any affine control loops can be viewed as SREs using results of array dataflow analysis. Thus, polyhedral representation of programs can be given a concrete syntax and expressed as systems of equations. We use such a language, similar to `Alpha` [7] language used in MMAlpha [8].

`Alpha` can be directly written as an alternative input to our system. Since polyhedral representations extracted from loops often contain a large number of boundary conditions, directly specifying as equations can lead to better performance.

In addition, our belief is that application programmers (i.e., scientists from non-CS domains), can benefit from being able to program with equations, where performance considerations like schedule or memory remain unspecified. Therefore, what needs to be computed and implementation details for performance can be isolated.

In this paper, we focus on the equational side of the polyhedral model using `Alpha` described in Figure 1. The language resembles mathematical equation in some ways, but it associates polyhedral domains to each expression. These domains are what is necessary to perform our analysis and transformations.

2.2.1 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.

[1] In the literature of the polyhedral model, the word dependence is sometimes used to express flow of data, but here the arrow is from the consumer to the producer.

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

- $\mathcal{D}_{\mathtt{V}} \cap \mathcal{D}_{\mathtt{E}}$ 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$(\oplus, f_p, E)$.
- $\mathcal{X}_{\mathtt{E}'} \cap \mathcal{D}_{\mathtt{E}'}$ if the parent $\mathtt{E}'$ 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.

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

Figure 1: Structure of Alpha programs. Inputs, outputs, and local variables are declared after corresponding keywords. Each expression in the program also has an associated domain denoting where the expression is defined, computed using domain of its children. 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.

Each system of equations are given a name and a parameter domain that define symbolic constants (program parameters) and constraints on them. In a system, input/output/local variables are declared with an associated domain. Variables should not be confused with arrays, as it has nothing to do with the memory.

The equation for a variable defines the values to be computed for each point in the domain of a variable using expressions in Figure 1. 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.

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.

2.2.2 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 $X(i) = \texttt{reduce}(+, (i, j \rightarrow i), A(i, j))$. This is more general than mathematical notations, since reductions with non-canonic projections, such as $(i, j \rightarrow i + j)$, require an additional variable to express with mathematical notations.

2.2.3 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.

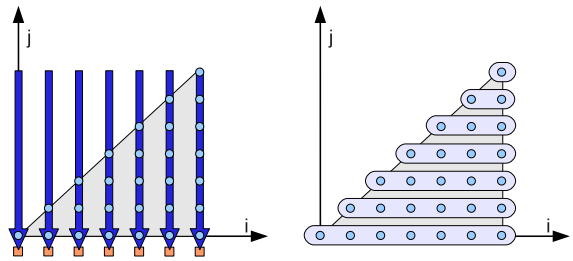
The context domain $\mathcal{X}_{\texttt{E}}$ of the expression `E` is:

- $\mathcal{D}_{\texttt{V}} \cap \mathcal{D}_{\texttt{E}}$ 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 $\text{reduce}(\oplus, f_p, E)$.
- $\mathcal{X}_{\texttt{E}'} \cap \mathcal{D}_{\texttt{E}'}$ 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.

2.2.4 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.



(a) Iteration Space and Reductions

(b) Reuse of A

Figure 2: Geometric illustration of the iteration space and reductions involved in prefix sum computation for N=8. The iteration space has a triangular domain where all integer points represent a computation. The reduction is along the vertical axis so that all points with the same i contributes to the same answer. Because A is indexed only with j, all points with the same j shares the same value.

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 $(i, j, k \to k)$, reading the value from A is $A.(i, j, k \to k)$.

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.

2.2.5 Complexity

We are interested in the asymptotic complexity of the program as a measure of complexity. Asymptotic complexity analysis needs the notion of one or more size parameter(s) and parameterized polyhedra naturally provide this: e.g., $\{i, j \mid 0 \leq (j, i) < N\}$ is implicitly a square, naturally parameterized with a size parameter, N. The cost of a reduction in the above form would directly correspond to the number of points in the domain $\mathcal{D}_E$.

To a first approximation, this is the number of index variables in the variable, e.g., the square domain has quadratic complexity, since there are two index variables, i, and j. This breaks down when the domain has equalities, e.g., a variable defined over the domain: $\{i, j \mid 0 \leq j = i < N\}$ has only linear complexity. Worse still, there me be domains with "bounded thickness" such as $\{i, j \mid 0 \leq (j, i) < N \wedge 0 \leq i - j \leq 10\}$ where there are no equalities, or others where equalities are not obvious to detect. More precise formalization of complexity for such cases is in the original article describing Simplifying Reductions [1].

3 Intuition of Simplifying Reductions

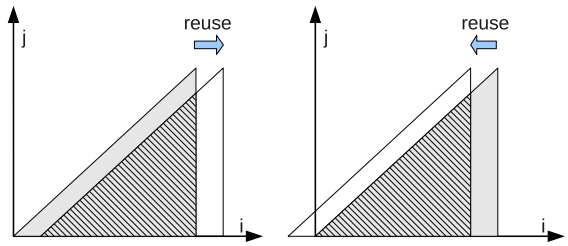
We first illustrate the intuition using a simple example. The prefix sum computation can be expressed as the following:

$$X[i] = \sum_{j=0}^{j=i} A[j] \tag{1}$$

with $\mathcal{D}_E = \{i, j | 0 \leq j \leq i < N\}$.

Figure 2 visualizes the iteration space of this program for N=8. The body of the reduction have a triangular domain $\{i, j | 0 \leq j \leq i < N\}$, and there are 7 independent reductions along the vertical axis. Because A$[j]$ is accessed within a 2D domain, it can be observed that all points along the horizontal access that has the same j but different i all share the same value. Note that the reuse space, i.e., the set of points that share the same value, is spanned by the vector [1,0].

Assume that some constant vector in the reuse space, reuse vector r_E, is given as the input and the simplification is performed so that an instance of reduction at z reuses the result of another instance at



(a) When reuse vector $(1, 0)$ is used

(b) When reuse vector $(-1, 0)$ is used

Figure 3: Visualization of the reuse and simplification. $\mathcal{D}_{\mathtt{E}'}$ is the domain translated by the reuse vector. The intersection of the two domains (striped and filled) is the value being reused. In Figure (a), the diagonal strip of filled domain that does not have the stripe, $\mathcal{D}_{add} = \mathcal{D}_E - \mathcal{D}_{\mathtt{E}'}$ is the domain that needs to be computed in addition to the reuse. In Figure (b), the diagonal strip of unfilled domain, $\mathcal{D}_{sub} = \mathcal{D}_{\mathtt{E}'} - \mathcal{D}_E$ is the domain of values that needs to be undone from the reused value.

$z - r_E$. Unless the values used at different instances of reductions are identical, reusing the result of another instance by itself is not enough. Because the iteration spaces are represented as polyhedra, the additional computation required can be computed. Figure 3 illustrates the reuse space and how the required computation in addition to the reuse is computed. Domain of additional computations are derived from the original domain $\mathcal{D}_E$ (filled domain) and its translation by the reuse vector $\mathcal{D}_{\mathtt{E}'}$ (unfilled domain). Domain with diagonal stripes is the intersection $\mathcal{D}_{int} = \mathcal{D}_E \bigcap \mathcal{D}_{\mathtt{E}'}$. $\mathcal{D}_{int}$ is where the result of two reductions r_E apart overlaps and can be reused. Thus, the diagonal strip of filled domain that does not have the stripe, $\mathcal{D}_{add} = \mathcal{D}_E - \mathcal{D}_{\mathtt{E}'}$ is the domain that needs to be computed in addition to the reuse.

Depending on the shape of the domain and the direction of reuse being exploited, some computation must be "undone" in addition to the reuse. In such cases, the reduction operator must have a corresponding inverse operator in order to undo parts of the computation. For example, if the vector[-1,0] was used instead in the above example, $P(x)$ is computed from $P(x + 1)$ by *subtracting* $A[x + 1]$. Such a domain, called *subtract domain*, can be computed as well, and it must be empty if the operator does not have an inverse.

The core of Simplifying Reductions is in precisely computing these addition and subtraction domains through geometrical analysis, by shifting polyhedra along the reuse space.

4 Simplifying Reductions

In this section, we describe the necessary elements of the Simplifying Reductions used in this paper. We first introduce the notion of *share space* that characterize sharing of values, used to determine if a given r_E is legal or not. Then we introduce the simplification transformation, followed by other transformations that enhance the applicability of Simplifying Reductions.

4.1 Sharing of Values

Consider a dependence expression E of the form:

$$X.f \tag{2}$$

The expression E has the same value at any two index points $z, z' \in \mathcal{D}_E$ if $f(z) = f(z')$ since they map to the same index point of X. We will say that the value of E at these index points is *shared*. Note, two index points in $\mathcal{D}_E$ share a value if they differ by a vector in $\ker(f)$. Thus, values of E are shared along the linear space $\ker(f)$.

However, $\ker(f)$ may not be the maximal linear space along which values are shared in E. Observe, if in turn, index points in X also share values, along $\ker(f')$ say, then a larger linear space along which values

of E are shared is $\ker(f' \circ f)$. We denote the maximal linear space along which values are shared in E as $\mathcal{S}_E$, and call it the *share space* of E. Below, we list the relationship between the share space of expressions. We assume that the specification has been transformed to have just input and computed variables, and all reductions are over expressions defined on a single integer polyhedron. The share space, S_E is equal to

- ϕ if E is a constant.
- ϕ is E is an input variable. We assume that program inputs have no sharing.
- $\mathcal{S}_X$ if E is a computed variable defined by $E = X$
- $\bigcap_{i=1}^{M} \mathcal{S}_{E_i}$ if E is $\text{op}(E_1, \ldots, E_M)$
- $\bigcap_{i=1}^{M} \mathcal{S}_{E_i}$ if E is $\text{case}\, E_1, \ldots, E_M\, \text{esac}$
- $\mathcal{S}_X$ if E is $\mathcal{D}' : X$
- $\ker(T \circ f)$ if E is $X.f$ and $\mathcal{S}_X = \ker(T)$.
- $f_p(\ker(Q) \cap \mathcal{S}_X)$ if E is $\text{reduce}(\oplus, f_p, X)$ and $\mathcal{D}_X$ is a single integer polyhedron $\mathcal{P} \equiv \{z | Qz + q \geq 0\}$.

For brevity, we denote spaces spanned by a set of basis vectors as [< list of expressions >] when indices are named in concrete examples. For example, a space spanned by basis vectors [1,0,0] and [0,1,1] where indices are named as i, j, k would be denoted as $[i, j + k]$.

4.2 Simplifying Reductions

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

$$X = \text{reduce}(\oplus, f_p, E) \tag{3}$$

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 3, 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.

We require that the reuse vector r_E to satisfy $r_E \in S_E \setminus \ker(f_p)$ for the semantic to be preserved. Since r_E is the direction of reuse it must be in the share space. However, it must not be in the kernel of the projection function f_p. This is because the transformation involves the use of the value of X at an index point to simplify the computation at another and so in order to avoid a self-dependence, we must ensure that these index points are distinct (i.e., $r_X = f_p(r_E) \neq 0$).

Note that the above transformation requires the inverse operator $\ominus$, which may not exist for some $\oplus$. Then all branch of the case in transformed that use $\ominus$ must have empty context domains.

4.3 Simplification Enhancing Transformations

We have shown a transformation that resulted in the simplification of reductions. Here, we will present transformations that, *per se*, do not simplify but enhance simplification. The goal of enhancing transformations is to increase the applicability of simplification by enlarging S_E. We only present a subset of such transformations used in simplification of RNA folding we show in Section 6.

4.4 Distributivity

Consider a reduction of the 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

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

The resultant expression can potentially have larger share space, since share space of E_1 no longer affects that of the reduction body.

4.5 Reduction Decomposition

We will now introduce a transformation that has wide applicability in enhancing simplification.

An 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 enhances simplification primarily by exposing additional opportunities to apply distributivity. When a reduction from Z^n to Z^m where m is at least 2 dimensions less than n, then some expression that cannot be distributed may be distributed once the reduction is decomposed.

For example, consider the following reduction:

$$\text{reduce}(\oplus, (i, j, k \to i), E_1 \otimes E_2)$$

where $S_{E_1} = \phi$, $S_{E_2} = [k]$, and $\ker(f_p) = [j, k]$. Since $\ker(f_p) \not\subseteq S_{E_2}$, E_2 cannot be distributed out. However, applying reduction decomposition with $f'_p = (i, j, k \to i, j)$ and $f''_p = (i, j \to i)$ to obtain:

$$\begin{aligned} X = \text{reduce}(\oplus, (i, j, k \to i, j), E_1 \otimes E_2) \\ \text{reduce}(\oplus, (i, j \to i), X) \end{aligned}$$

allows E_2 to be distributed out from the inner reduction.

Depending on its use, the reduction decomposition may or may not have side effects. However, the case without side effects only occur when domains of reduction body contain equalities (or some constant "thickness" variations of equalities) along certain dimensions. When there are equalities in the domain of reductions, the space spanned by the equalities are separated by reduction decomposition as a pre-processing. These cases, including constant "thickness" variants, are formalized as *Effective Linear Subspace* in the original article [1]. It states that a polyhedron $\mathcal{P}$ have constant thickness along any vector *not* in its effective linear subspace $\mathcal{L}_{\mathcal{P}}$.

For the domains in UNAfold, $\mathcal{L}_{\mathcal{D}_E}$ is the universe, and we focus on reduction decomposition with side effects. Reduction decomposition with side effects reduce the space of possible reuse directions, and affects if simplification is applicable later in the sequence of transformations.

Recall that distributing an expression E out from the reduction with a projection f_p requires $\ker(f_p) \subseteq \mathcal{S}_E$. Therefore, we may decompose f_p into $f''_p \circ f'_p$ to distribute an expression with available reuse space S_E outside the inner reduction by choosing f'_p such that

$$\ker(f'_p) = \ker(f_p) \cap \mathcal{S}_E$$

4.6 Normalizations

There are a number of transformations for taking equations with reductions into the form required by the simplification transformation (Equation 3). We introduce two of such transformations that are used later in Section 6.

4.6.1 Normalize Reductions

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:

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

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.

4.6.2 Permutation Case Reduce

Permutation Case Reduce, presented as a theorem by Le Verge [9], 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:

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

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.

5 Optimality and Algorithm

In the original article, we present an algorithm to apply the set of transformations, and optimality result with respect to the presented transformations. In this paper, we outline a simplified version of the algorithm used when applying the algorithm to UNAfold shown in Algorithm 1.

Algorithm 1 The Simplification Algorithm: Subset for UNAfold

Input:
An equational specification in the polyhedral model.

1. Preprocess to obtain a reduction over an expression whose domain is a single polyhedron and equal to its context domain.
2. Other pre-processing not used for UNAfold, if applicable.
3. Perform any of the following transformations, if applicable.
   (a) Distributivity.
   (b) Other side-effect free enhancing transformations not used for UNAfold.
4. Repeat steps 1-3 till convergence.
5. Dynamic Programming Algorithm to optimally choose:
   (a) The simplification transformation along some r_E.
   (b) A reduction decomposition with side effects.
   (c) Other enhancing transformations with side-effects, not used for UNAfold.
6. Repeat from step 1 on residual reductions until convergence.

Output:
An equivalent specification of optimal complexity.

For the dynamic programming in Step 5 to work, we must show that from the infinite search space of parameters for the various transformations, we need to consider only a finite set of choices and the global optima can be reached through such choices. The intuition behind our formal argument in the original article [1] is as follows:

Simplification Transformation: From the infinitely many choices of reuse vectors r_E in a share space, we show that there are only finitely many equivalence classes. The intuition is that the result of applying the simplification transformation with different reuse vectors from the same equivalence class are identical except for the "thickness" of the residual reductions. The residual reductions corresponds to the addition and subtraction domains $(\mathcal{D}_{add}, \mathcal{D}_{sub})$. The constant "thickness" of reductions do not affect the asymptotic complexity.

Reduction Decomposition with Side Effects for Distributivity: Of the infinite possible decompositions of the projection f_p, we only need to consider a finite subset, since the transformation is needed to distribute a set of subexpressions outside the inner reduction. The number of candidates are finite, with the equivalence classes formed by the basis vectors of its kernels.

6 RNA folding in UNAfold

Finally, we show that the application of the simplification algorithm described in Section 5 leads to the reduced complexity algorithm.

The RNA folding algorithm is a dynamic programming algorithm. There are multiple variations of the algorithm based on the cost model used. UNAfold [3] uses a prediction model based on thermodynamics that finds a structure with minimal free energy. For an RNA sequence of length N, the algorithm computes multiple tables of free energy for each subsequence from i to j such that $1 \leq i \leq j \leq N$. The three tables $Q(i,j)$, $Q'(i,j)$, and $QM(i,j)$ corresponds to the free energy for three different substructures that may be formed.

The following equations taken from the original algorithm:

$$Q(i,j) = \min \begin{cases} b + Q(i+1,j) \\ b + Q(i,j-1) \\ c + E_{ND}(i,j) + Q'(i,j) \\ b + c + E_{5'D}(i+1,j) + Q'(i+1,j) \\ b + c + E_{3'D}(i,j-1) + Q'(i,j-1) \\ 2b + c + E_{DD}(i+1,j-1) + Q'(i+1,j-1) \\ QM(i,j) \end{cases} \tag{4}$$

$$Q'(i,j) = \min \begin{cases} E_H(i,j) \\ E_S(i,j) + Q'(i+1,j-1) \\ \min_{i<i'<j'<j} \begin{cases} E_{BI}(i,j,i',j') \\ Q'(i',j') \end{cases} \\ a + c + E_{ND}(j,i) + QM(i+1,j-1) \\ a + b + c + E_{3'D}(j,i) + QM(i+2,j-1) \\ a + b + c + E_{5'D}(j,i) + QM(i+1,j-2) \\ a + 2b + c + E_{DD}(j,i) + QM(i+2,j-2) \end{cases} \tag{5}$$

$$QM(i,j) = \min_{i+1 \leq k \leq j-2} \Big\{ Q(i,k-1) + Q(k,j) \tag{6}$$

where a,b, and c, are constants and functions of the form E_{XX} are all energy functions for different substructures.

The third term in Equation 5 is the dominating term that makes the algorithm $O(N^4)$. Notice that the term uses four free variables i,j,i' and j' and has a 4D domain $\{i,j,i',j' | 1 \leq i < i' < j' < j \leq N\}$ and hence 4D complexity. The term corresponds to a substructure called internal loops, and the algorithm with $O(N^4)$ to evaluate this term is referred to as *fast i-loop*.

6.1 Simplification

We focus on the dominating term in calculating the energy associated with internal loops to illustrate the simplification. The term rewritten as a separate equation using our notation of reductions, and named QBI (since it is the term that involves E_{BI}) is the following:

$$QBI[i,j] = \texttt{reduce}(\min, (i,j,i',j' \rightarrow i,j), E_{BI}(i,j,i',j') + Q'(i',j')) \tag{7}$$

The sequence of transformations to obtain the above corresponds to Step 1 in Algorithm 1.

Before simplification, the energy function E_{BI} must be inlined to expose the reuse. This inlining is *not* part of the algorithm, and requires human analysis to deduce that the inlining is necessary at the moment.

E_{BI} has two different definitions, one for generic case and another to handle special cases. These special cases are when the size of the internal loop is very small (less than 4) and thus resembles other kind of substructures. Since the special case can be described as polyhedral domains, we focus on the generic case for simplicity.

The function E_{BI} for generic case is defined as follows:

$$E_{BI}(i,j,i',j') = Asym(i'-i-j+j') + S_P(i'-i+j-j'-2) + E_S(i,j) + E_S(i',j') \tag{8}$$

Inlining Equation 8 into Equation 9 gives the following:

$$QBI[i,j] = \texttt{reduce}\left(\min, (i,j,i',j' \to i,j), \begin{cases} Asym(i'-i-j+j') & + \\ S_P(i'-i+j-j'-2) & + \\ E_S(i,j) & + \\ E_S(i',j') & + \\ Q'(i',j') & \end{cases}\right) \tag{9}$$

Computing the share space for each sub-expressions in the reduction body (recall Section 4.1) gives:

$$\begin{array}{ll} Asym(i'-i-j+j') & [j-i, i+i', i+j'] \\ S_P(i'-i+j-j'-2) & [i+j, i+i', j'-i] \\ E_S(i,j) & [i', j'] \\ E_S(i',j') & [i,j] \\ Q'(i',j') & [i,j] \end{array}$$

In addition, the kernel of the projection function $\ker(f_p) = [i', j']$. Since the share space of $E_S(i,j)$ contains $\ker(f_p)$, it can be distributed out from the reduction in Step 3a of Algorithm 1 to produce: [2]

$$Q_{BI}[i,j] = E_S(i,j) + \texttt{reduce}\left(\min, (i,j,i',j' \to i,j), \begin{cases} Asym(i'-i-j+j') & + \\ S_P(i'-i+j-j'-2) & + \\ E_S(i',j') & + \\ Q'(i',j') & \end{cases}\right) \tag{10}$$

Taking the intersection of share spaces of the remaining terms gives the zero vector, and therefore no reuse can be exploited. This takes us to Step 5b of Algorithm 1. We analyze the share space of expressions in the reduction body and the projection function to find candidate decompositions. The candidate f'_p are:[3]

$$\begin{array}{rr} \ker(f'_p) = \ker(f_p) \cap S_{Asym}(i'-i-j+j') & S_{Asym}(i'-i-j+j') = [j-i, i+i', i+j'] \\ \ker(f'_p) = \ker(f_p) \cap S_{S_P}(i'-i+j-j'-2) & S_{S_P}(i'-i+j-j'-2) = [i+j, i+i', j'-i] \\ \ker(f'_p) = \ker(f_p) \cap S_{E_S}(i',j') & S_{E_S}(i',j') = [i,j] \\ \ker(f'_p) = \ker(f_p) \cap S_{Q'}(i',j') & S_{Q'}(i',j') = [i,j] \end{array}$$

where $\ker(f_p) = [i', j']$.

The latter two candidates have no feasible f'_p because $[i',j'] \cap [i,j] = \phi$. Similarly, $S_{Asym}(i'-i-j+j')$ do not have any feasible f'_p. The only feasible candidate is $\ker(f'_p) = [i'+j']$ with $S_{S_P}(i'-i+j-j'-2)$.

One instance of f'_p with $\ker(f'_p) = [i'+j']$ is $f'_p = (i,j,i',j' \to i,j,j'-i')$, and its corresponding f''_p is $(i,j,d \to i,j)$. The choice of the instance does not affect the resulting complexity. Decomposing the reduction with f'_p and naming the inner reduction QBI' gives the following two equations.

[2] $\mathcal{H}_{\mathcal{D}_E}$ is universe for this program.

[3] Note that once we have f_p and f'_p, f''_p can be deduced with standard linear algebra.

$$QBI[i,j] = E_S(i,j) + \texttt{reduce}(\min, (i,j,d \to i,j), QBI'[i,j,d]);$$

$$QBI'[i,j,d] = \texttt{reduce}\left(\min, (i,j,i',j' \to i,j,j'-i'), \begin{cases} Asym(i'-i-j+j') & + \\ S_P(i'-i+j-j'-2) & + \\ E_S(i',j') & + \\ Q'(i',j') & \end{cases}\right)$$

After the decomposition, the expression $S_P(i'-i+j-j')$ can be distributed out from the inner reduction, because its share space $[i+j, i+i', j'-i]$ contains $\ker(f'_p)$ (combining $i+i'$ and $j'-i$ gives $i'+j'$).

$$QBI'[i,j,d] = \texttt{reduce}\left(\min, (i,j,i',j' \to i,j,j'-i'), \begin{cases} Asym(i'-i-j+j') & + \\ E_S(i',j') & + \\ Q'(i',j') & \end{cases}\right) + S_P(-i+j-d-2)$$

Then the remaining expressions have a common share space $[j-i]$, the space spanned by the vector [-1,1,0,0].a Applying the simplifying reduction transformation using [-1,1,0,0] as the reuse vector yields an equivalent equation of the following form:

$$QBI'[i,j,d] = \begin{cases} \mathcal{D}_{init} & : X_{add} \\ \mathcal{D}_{add} & : \min(X_{add}, QBI'[i+1,j-1,d]) \end{cases} + S_P(-i+j-d-2) \tag{11}$$

$$X_{add} = \begin{cases} Asym(i'-i-j+j') + \\ E_S(i',j') + \\ Q'(i',j') \end{cases}$$

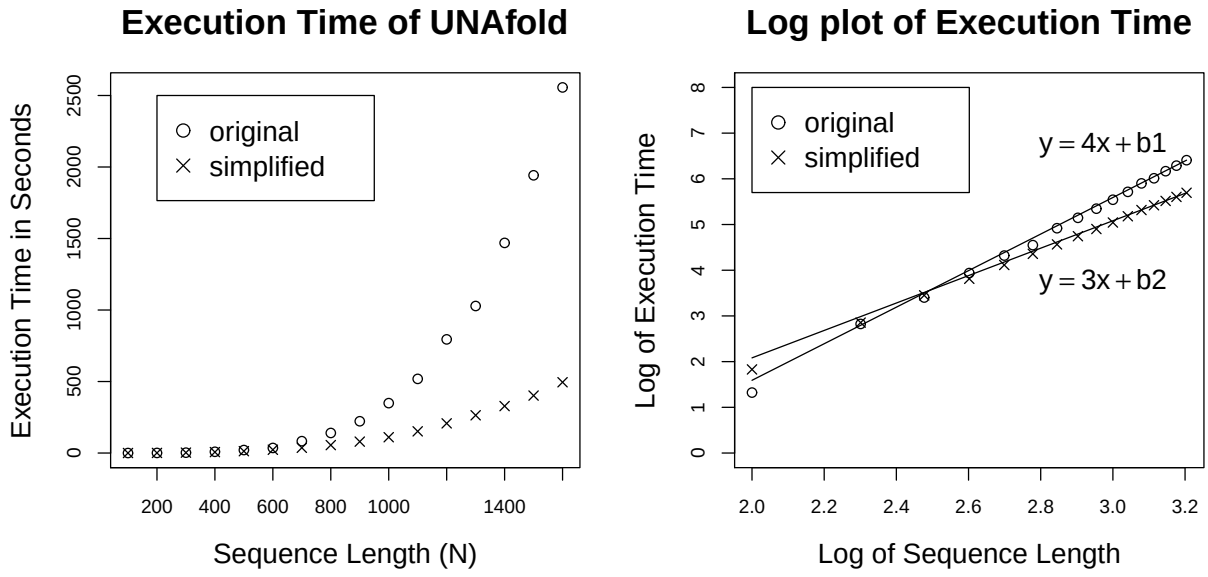
Domains D_{init}, and D_{add} are computed following the definitions in Section 4. The full Alphabets program after transformation in the appendix show the domains as the domain of corresponding Alpha variables.

In the following, we show a fragment of the Alphabets after sequence of transformations described above has been applied. We can observe that equations `QBI_SR1_init` and `QBI_SR1_add` both have equalities in the restrict expression. These equations respectively correspond to branches of QBI' in Equation 11. Because of The equalities the context domains of these reductions are 3D domains embedded in 4D space. Hence, we confirm that the complexity is reduced to $O(N^3)$. We also note that the term S_P was factored out since the simplification algorithm require reduce expression to be the top-level expression.

```
//Simplifying Reduction result
QBI_SR1[i,j,ip] = case
   {|i-j+ip+7>= 0} : QBI_SR1_init;  //D_init case
   {|j-1-ip-8>= 0} : (QBI_SR1_add min QBI_SR1[i+1,j-1,ip]); //D_add case
esac;

//X_add for D_init
QBI_SR1_init[i,j,ip] = reduce(min, (i,j,ip,jp->i,j,jp-ip),
   {|ip-i==2} || {|j-jp==4 && ip-i==3} :
      ((Ebi_stacking([jp],[ip]) + Ebi_asymmetry([ip-i-1],[j-jp-1])) + Qprime[ip,jp])
);

//X_add for D_add
QBI_SR1_add[i,j,ip] = reduce(min, (i,j,ip,jp->i,j,jp-ip),
   {|ip-i==2} || {|j-jp==4} :
      ((Ebi_stacking([jp],[ip]) + Ebi_asymmetry([ip-i-1],[j-jp-1])) + Qprime[ip,jp])
);
```



(a) Raw Execution Time

(b) Log-Log Plot

Figure 4: Execution Time of UNAfold after simplifying reduction compared with the original implementation. The two lines show are with slopes 4 and 3 with constant offsets (b1, b2) to make the lines meet the points at log(N)=3.2.

6.2 Validation

We have applied the above transformation using AlphaZ to the UNAfold 3.8 [3]. The function `fillMatrices_1` in `hybrid-ss-min.c` was written in our equational language, and the simplifying transformation was applied. One of the code generators in AlphaZ that produce sequential C code was used to generate the simplified version of `fillMatrices_1` and replaced with the original function.

Both original and the simplified versions were compiled with `GCC/4.5.1`, with `-O3` option and the execution times were measured a machine with Core2Duo 1.86GHz and 6GB of memory running Linux. Because the default option of UNAfold limits the internal loop size to 30, we also set the limit to infinity when running `hybrid-ss-min`.

Figure 4 shows the measured performance, and its log-log scaled version. The log-scale plot clearly shows the reduction in complexity, and, as expected, the speedups with transformed code becomes greater and greater as the sequence length grows.

7 Conclusion

We have presented a detailed walk-through of application of Simplifying Reductions to a RNA folding application in UNAfold package. The ability to (semi-)automatically reduce the asymptotic complexity of programs is a very powerful feature of AlphaZ. However, the algorithm and the necessary formalism to understand the simplification is complicated. We hope that this paper can help understanding the algorithm in depth.

References

[1] G. G. and R. S., "Simplifying reductions," in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '06. New York, NY, USA: ACM, 2006,

pp. 30–41. [Online]. Available: http://doi.acm.org/10.1145/1111037.1111041

[2] R. Lyngs, M. Zuker, C. Pedersen *et al.*, "Fast evaluation of internal loops in rna secondary structure prediction." *Bioinformatics*, vol. 15, no. 6, pp. 440–445, 1999.

[3] N. Markham and M. Zuker, "Software for nucleic acid folding and hybridization," *Methods Mol. Biol*, vol. 453, pp. 3–31, 2008.

[4] T. Yuki, V. Basupalli, G. Gupta, G. Iooss, D. Kim, T. Pathan, P. Srinivasa, Y. Zou, and S. Rajopadhye, "Alphaz: A system for analysis, transformation, and code generation i n the polyhedral equational model," Technical Report CS-12-101, Colorado State University, Tech. Rep., 2012.

[5] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[6] R. Karp, R. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *Journal of the ACM (JACM)*, vol. 14, no. 3, pp. 563–590, 1967.

[7] H. Verge, C. Mauras, and P. Quinton, "The ALPHA language and its use for the design of systolic arrays," *The Journal of VLSI Signal Processing*, vol. 3, no. 3, pp. 173–182, 1991.

[8] C. IRISA, "The MMAlpha environment."

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

A Alphabets version of `fill_matrices_1` in UNAfold

The following is the original Alphabets program corresponding to `fill_matrices_1` in UNAfold. The list of external functions corresponds to function calls or table look up in the C implemention. All of these functions are assumed to be side effect free functions by AlphaZ analyses.

`Ebi_asymmetry`, `Ebi_sizePenalty`, and `Ebi_stacking` respectively correspond to $Asym$, S_P, and E_S in the equations in Section 6. Aside from syntactic sugar, the Alpha program below roughly corresponds to the original equations describing the RNA secondary structure prediction algorithm. One important difference is that the Alpha program is quite verbose and precise about the domains and boundary conditions.

```
// External functions
int Es(int, int);
int Eh(int, int);
int End(int, int);
int Ed3(int, int);
int Ed5(int, int);
int Etstackm(int, int);
int a(int);
int b(int);
int c(int);
int INFINITY_VAL(int);
int Eval_isFinite(int);
int nodangle(int);
int noisolate(int);
int Eval_ssOK(int, int);
//Ebi spilt functions
int Ebi_sizePenalty(int);
int Ebi_stacking(int, int);
int Ebi_asymmetry(int, int);
int Ebi_Bulge1(int, int, int, int);
int Ebi_Bulge(int, int, int, int, int);
int Ebi_iloop1x1(int, int, int, int);
int Ebi_iloop1x2(int, int, int, int);
```

```
int Ebi_iloop2x1(int, int, int, int);
int Ebi_iloop2x2(int, int, int, int);

affine fillMatrices1_unafold {N, MAXLOOP|N>7 && MAXLOOP>7}
input
   int Qprime_ip {i,j|1<=i<=N && 2<=j<=N};
output
  int Q {i,j|1<=i<=N && 2<=j<=N};
  int Qprime {i,j|1<=i<=N && 2<=j<=N};
  int QM {i,j|1<=i<=N && 2<=j<=N};
local
  int QBI {i,j|1<=i<=N && 2<=j<=N};
  int EBI {i,j,ip,jp|1<=i<ip<jp<j<=N && ip-i-1+j-jp-1<=MAXLOOP};
  int EBI_special
          {i,j,ip,jp|ip-i==1 && j-jp==2 &&
                         1<=i<ip<jp<j<=N && ip-i-1+j-jp-1<=MAXLOOP}||
          {i,j,ip,jp|ip-i==2 && j-jp==1 &&
                         1<=i<ip<jp<j<=N && ip-i-1+j-jp-1<=MAXLOOP}||
          {i,j,ip,jp|ip-i==1 && j-jp>2  &&
                         1<=i<ip<jp<j<=N && ip-i-1+j-jp-1<=MAXLOOP}||
          {i,j,ip,jp|ip-i>2  && j-jp==1 &&
                         1<=i<ip<jp<j<=N && ip-i-1+j-jp-1<=MAXLOOP}||
          {i,j,ip,jp|ip-i==2 && j-jp==2 &&
                         1<=i<ip<jp<j<=N && ip-i-1+j-jp-1<=MAXLOOP}||
          {i,j,ip,jp|ip-i==2 && j-jp==3 &&
                         1<=i<ip<jp<j<=N && ip-i-1+j-jp-1<=MAXLOOP}||
          {i,j,ip,jp|ip-i==3 && j-jp==2 &&
                         1<=i<ip<jp<j<=N && ip-i-1+j-jp-1<=MAXLOOP}||
          {i,j,ip,jp|ip-i==3 && j-jp==3 &&
                         1<=i<ip<jp<j<=N && ip-i-1+j-jp-1<=MAXLOOP};
  int EBI_generic
          {i,j,ip,jp|ip-i>=2 && jp-ip>=1 && j-jp>=4 &&
                          1<=i<ip<jp<j<=N && ip-i-1+j-jp-1<=MAXLOOP}||
          {i,j,ip,jp|j-jp==3 && ip-i>=4  && j-ip>=4 &&
                          1<=i<ip<jp<j<=N && ip-i-1+j-jp-1<=MAXLOOP}||
          {i,j,ip,jp|j-jp==1 && ip-i==1  && j-i>=3  &&
                          1<=i<ip<jp<j<=N && ip-i-1+j-jp-1<=MAXLOOP}||
          {i,j,ip,jp|j-jp==2 && ip-i>=4  && j-ip>=3 &&
                          1<=i<ip<jp<j<=N && ip-i-1+j-jp-1<=MAXLOOP};
let
  Q[i,j] = case
              {|i>=j-3}: INFINITY_VAL(0);
              {|i<j-3}: min((b(0) + Q[i+1,j]), (b(0) + Q[i, j-1]),
                            (c(0) + End([i],[j]) + Qprime[i,j]),
                            QM[i,j]);
           esac;

  Qprime[i,j] = case
                   {|i>=j-3}: INFINITY_VAL(0);
                   {|i<j-3}: if (Eval_isFinite(Qprime_ip[i,j]) > 0) then
                                min(Eh([i],[j]),
                                    (Es([i],[j]) + Qprime[i+1,j-1]),
                                    (QBI[i,j]),
                                    (a(0) + c(0) +
                                     End([i],[j]) + QM[i+1, j-1])
                                    )
                             else
                                INFINITY_VAL(0);
                esac;
```

```
  QM[i,j] = case
              {|i>=j-8}: INFINITY_VAL(0);
              {|i<j-8}: reduce(min, [k], {|i+3+1<=k<=j-3-2}: (Q[i,k-1] + Q[k,j]));
            esac;



  EBI_special[i,j,ip,jp] = case
                      {|ip-i==1 && j-jp==2} ||
                      {|ip-i==2 && j-jp==1} : Ebi_Bulge1([i],[j],[ip],[jp]);
                      {|ip-i==1 && j-jp>2}: Ebi_Bulge([i],[j],[ip],[jp],[j-jp-1]);
                      {|ip-i>2 && j-jp==1}: Ebi_Bulge([i],[j],[ip],[jp],[ip-i-1]);
                      {|ip-i==2 && j-jp==2}: Ebi_iloop1x1([i],[j],[ip],[jp]);
                      {|ip-i==2 && j-jp==3}: Ebi_iloop1x2([i],[j],[ip],[jp]);
                      {|ip-i==3 && j-jp==2}: Ebi_iloop2x1([i],[j],[ip],[jp]);
                      {|ip-i==3 && j-jp==3}: Ebi_iloop2x2([i],[j],[ip],[jp]);
                    esac;

  EBI_generic[i,j,ip,jp] = Ebi_sizePenalty([ip-i-1+j-jp-1]) + Ebi_stacking([i],[j]) +
                           Ebi_stacking([jp],[ip]) + Ebi_asymmetry([ip-i-1],[j-jp-1]);
let


  EBI[i,j,ip,jp] = case
                       EBI_special[i,j,ip,jp];
                       EBI_generic[i,j,ip,jp];
                     esac;

  QBI[i,j] = case
              {|i>=j-6}: INFINITY_VAL(0);
              // jp = ip + d
              {|i<j-6} : reduce(min, [ip,jp],
                              {|jp-ip<=j-i-3 && jp-ip>=4 && jp-ip>=j-i-2-MAXLOOP &&
                                i<ip<j-jp+ip && ip<=N}:
                                 (EBI[i,j,ip,jp] + Qprime[ip,jp])
                          );
            esac;

.
```

B AlphaZ Script for Applying Simplifying Reductions

The following is the AlphaZ script used to appy the sequence of transformations described in Section 6. The commands used in the script is briefly described in the following.

- `ReadAlphabets`; Parse an alphabets program and return a program object.
- `Inline`; Replace references to a variable with its definition.
- `Normalize`; Apply a set of normalization rules. This transformation is often used as a pre-processing before applying transformations, so that transformations only need to support normalized (simpler) instances of Alpha programs. The resulting program do not have nested case/dependence/restrict expressions.
- `FactorOutFromReduction`; Implementation of the simplification enhancing transformation to take advantage of distributivity.

- `ReductionDecomposition`, `NormalizeReduction`, `PermutationCaseReduce`; Implementations of the corresponding simplification enhancing transformations.
- `RenameVariable`; Renames a variable. Used to rename variables with automatically generated names during other transformations to more meaningful names.
- `SplitUnion`; Splits an expression that has unions of polyhedra as its expression domain to multiple variables such that resulting variables each have a single polyhedron as its expression domain (one of the unions in the original domain).
- `SimplifyingReduction`; The simplifying transformation illustratred in this paper.
- `RemoveUnusedVariables`; Removes all variables not when computing output variables.

Fore more detail, see `http://www.cs.colostate.edu/AlphaZ/AlphaZCommandRef.pdf`.

Also, most commands take the program object and system name as the first two inputs, and the rest of the argument usually specified variable names or an expression in the AST of Alphabets. Currently we use expression ID as the most general way to specify the target expression, represent as a vector of integers. This vector uniquely identifies the target expression by specifying which "branch" to take in the AST at each level in the tree. For example "0,1,0" denotes the first system in the program (0-th branch), second equation (1-th branch), and first expression (0-th branch).

```
prog = ReadAlphabets("./unafold.ab");
system = "fillMatrices1_unafold";

#inline Ebi_A = Ebi
Inline(prog, system, "QBI", "EBI");
RemoveUnusedVariables(prog);
Normalize(prog);

#isolate QBI for EBI_generic
PermutationCaseReduce(prog, system, "QBI");
NormalizeReduction(prog, "0,5,0,1,0,1,0,1");
RenameVariable(prog, system, "NR_QBI", "QBI_generic");

#Inline EBI_generic
Inline(prog, system, "QBI_generic", "EBI_generic");
Normalize(prog);

#distribute out EBI_stacking(i,j)
FactorOutFromReduction(prog, "0,6,0,0,0,0,0,0,1");
Normalize(prog);

#decompose reductions
ReductionDecomposition(prog, "0,6,0,0", "(i,j,d->i,j)", "(i,j,ip,jp->i,j,jp-ip)");
NormalizeReduction(prog, "0,6,0,0,0");
RenameVariable(prog, system, "NR_QBI_generic", "QBI_inner");

#factor out SizePenalty
FactorOutFromReduction(prog, "0,7,0,0,0,0,0,0");
Normalize(prog);

#normalize reduction to take the QBI_generic in SR form
NormalizeReduction(prog, system, "QBI_inner");
RenameVariable(prog, system, "NR_QBI_inner", "SR_QBI");

#simplifying reductions
#split the union of polyhedra of the context domain of the reduction body
SplitUnion(prog, "0,8,0,0");
PermutationCaseReduce(prog, system, "SR_QBI");
```

```
NormalizeReduction(prog, system, "SR_QBI");

#apply SR to one of the domains split from the union
SimplifyingReduction(prog, system, "NR_SR_QBI", "1,-1,0,0");

#cannot apply SR in the other domain that was split due to subtract domain
#SimplifyingReduction(prog, system, "NR_SR_QBI_1", "1,-1,0,0");
Normalize(prog);
Simplify(prog, system);

#rename variables introduced by SR to be shorter
RenameVariable(prog, system, "NR_SR_QBI", "QBI_SR1");
RenameVariable(prog, system, "NR_SR_QBI_1", "QBI_SR2");

#get ride of extra variables
Inline(prog, system, "QBI_inner", "SR_QBI");
Inline(prog, system, "QBI_generic", "QBI_inner");
Inline(prog, system, "QBI", "QBI_generic");
Inline(prog, system, "QBI", "EBI_generic");
Inline(prog, system, "QBI", "EBI_special");
Inline(prog, system, "Qprime", "QBI");

Normalize(prog);
RemoveUnusedVariables(prog);
```

C Alphabets After Transformation

Since the Alphabets that matches UNAfold implementation has a number of complex constraints, it also requires some minor transformations not explained in the above. One is caused by the domain of QBI' being a union of polyhedra, rather than a single polyhedron. Simplifying Reductions require that each reduction have a polyhedron as its context domain, and therefore QBI' was split into two using another transformation.

Out of the two resulting reductions, one has a non-empty subtraction domain (D_{sub}), and thus cannot be simplified because inverse operator for min does not exist. However, the domain of this reduction is actually "thin", and does not affect the overall complexity of the algorithm after simplification. The equation `QBI_SR2` in the following Alphabets fragment corresponds to this reduction. Note that it does not have equalitiese, but you can observe that $j - jp$ is either 2 or 3, and hence it is an slice of size 2, and thefore, it also has $O(N^3)$ complexity.

```
int Es(int,int);
int Eh(int,int);
int End(int,int);
int Ed3(int,int);
int Ed5(int,int);
int Etstackm(int,int);
int a(int);
int b(int);
int c(int);
int INFINITY_VAL(int);
int Eval_isFinite(int);
int nodangle(int);
int noisolate(int);
int Eval_ssOK(int,int);
int Ebi_sizePenalty(int);
int Ebi_stacking(int,int);
int Ebi_asymmetry(int,int);
int Ebi_Bulge1(int,int,int,int);
```

```
int Ebi_Bulge(int,int,int,int,int);
int Ebi_iloop1x1(int,int,int,int);
int Ebi_iloop1x2(int,int,int,int);
int Ebi_iloop2x1(int,int,int,int);
int Ebi_iloop2x2(int,int,int,int);
affine fillMatrices1_unafold {N,MAXLOOP|N>7 && MAXLOOP>7}
   input
      int Qprime_ip {i,j|j<=N && 0<i && 1<j && i<=N};
   output
      int Q {i,j|j<=N && 0<i && 1<j && i<=N};
      int Qprime {i,j|j<=N && 0<i && 1<j && i<=N};
      int QM {i,j|j<=N && 0<i && 1<j && i<=N};
   local
      int QBI_SR1 {i,j,ip|0<i && 6<=j-i-ip && 4<=ip && j-i-ip-2<= MAXLOOP && j<=N};
      int QBI_SR2 {i,j,ip|0<i && 6<=j-i-ip && 4<=ip && j-i-ip-2<= MAXLOOP && j<=N};
      int Qprime_body1 {i,j|0<i && 7<=j-i && j-i<=9 && j<=N};
      int Qprime_body2 {i,j|j<=N && j-i<=10 && 0<i};
      int Qprime_body3 {i,j|j<=N && j-i<=10 && 0<i};
      int QM_body {i,j|j<=N && -i+j-9>= 0 && 0<i};
      int QBI_SR1_init {i,j,ip|0<i && 6<=j-i-ip && 4<=ip && 7<=j-i-ip && j<=N};
      int QBI_SR1_add  {i,j,ip|0<i && 8<=j-i-ip && 4<=ip &&
                                                   j-i-ip+2<=MAXLOOP && j<=N};
   let
      Q[i,j] = case
                  {|i-j+3>= 0} : INFINITY_VAL(0);
                  {|-i+j-4>= 0} : min((b(0) + Q[i+1,j]),
                                      (b(0) + Q[i,j-1]),
                                      ((c(0) + End([i],[j])) + Qprime), QM);
               esac;

      Qprime[i,j] = case
                  {|i-j+3>= 0} : INFINITY_VAL(0);
                  {|-i+j-4>= 0} :
                     if ((Eval_isFinite(Qprime_ip) > 0)) then
                        (min(Eh([i],[j]),(Es([i],[j]) + Qprime[i+1,j-1]),
                           case
                              {|i-j+6>= 0} : INFINITY_VAL(0);
                              {|-i+j-7>= 0 && i-j+9>= 0} : Qprime_body1;
                              {|-i+j-10>= 0} : min(Qprime_body2,
                                                   Qprime_body3
                                                    + Ebi_stacking([i],[j])
                                                  );
                           esac,
                        (((a(0) + c(0)) + End([i],[j])) + QM[i+1,j-1])))
                     else (INFINITY_VAL(0));
                    esac;

      QM[i,j] = case
                   {|i-j+8>= 0} : INFINITY_VAL(0);
                   {|-i+j-9>= 0} : QM_body;
                esac;

      QBI_SR1[i,j,ip] = case
                   {|i-j+ip+7>= 0} : QBI_SR1_init;
                   {|-i+j-ip-8>= 0} : (QBI_SR1_add min QBI_SR1[i+1,j-1,ip]);
                esac;

      QBI_SR2[i,j,ip] =
         reduce(min, (i,j,ip,jp->i,j,jp-ip), {|-i+4<=ip && -j+jp+3>= 0 && j-jp-2>= 0} :
```

```
      ((Ebi_stacking([jp],[ip]) +
        Ebi_asymmetry([ip-i-1],[j-jp-1])) +
        Qprime[ip,jp]));

Qprime_body1[i,j] = reduce(min, (i,j,ip,jp->i,j),
                         case
{|j-jp==2 && ip-i==1} ||
{|j-jp==1 && ip-i==2} : (Ebi_Bulge1([i],[j],[ip],[jp]) + Qprime[ip,jp]);
{|ip-i==1 && 3<=j-jp} : (Ebi_Bulge([i],[j],[ip],[jp],[j-jp-1]) + Qprime[ip,jp]);
{|j-jp==1 && 4<=ip-i} : (Ebi_Bulge([i],[j],[ip],[jp],[ip-i-1]) + Qprime[ip,jp]);
{|j-jp==2 && ip-i==2} : (Ebi_iloop1x1([i],[j],[ip],[jp]) + Qprime[ip,jp]);
{|-i+jp-6== 0 && ip-i==2 && j-i==9} :
                        (Ebi_iloop1x2([i],[j],[ip],[jp]) + Qprime[ip,jp]);
{|-i+jp-7== 0 && ip-i==3 && j-i==9} :
                        (Ebi_iloop2x1([i],[j],[ip],[jp]) + Qprime[ip,jp]);
                         esac);

Qprime_body2[i,j] = reduce(min, (i,j,ip,jp->i,j),
                         case
{|j-jp==2 && ip-i==1} ||
{|j-jp==1 && ip-i==2} : (Ebi_Bulge1([i],[j],[ip],[jp]) + Qprime[ip,jp]);
{|ip-i==1 && 3<=j-jp} : (Ebi_Bulge([i],[j],[ip],[jp],[j-jp-1]) + Qprime[ip,jp]);
{|j-jp==1 && 3<=ip-i} : (Ebi_Bulge([i],[j],[ip],[jp],[ip-i-1]) + Qprime[ip,jp]);
{|j-jp==2 && ip-i==2} : (Ebi_iloop1x1([i],[j],[ip],[jp]) + Qprime[ip,jp]);
{|j-jp==3 && ip-i==2} : (Ebi_iloop1x2([i],[j],[ip],[jp]) + Qprime[ip,jp]);
{|j-jp==2 && ip-i==3} : (Ebi_iloop2x1([i],[j],[ip],[jp]) + Qprime[ip,jp]);
{|j-jp==3 && ip-i==3} : (Ebi_iloop2x2([i],[j],[ip],[jp]) + Qprime[ip,jp]);
                        esac);

Qprime_body3[i,j] = reduce(min, (i,j,d->i,j),
                       ((QBI_SR1 min QBI_SR2) + Ebi_sizePenalty([-i+j-d-2])));

QM_body[i,j] = reduce(min, (i,j,k->i,j),
                  {|4<=k-i && 5<=j-k} : (Q[i,k-1] + Q[k,j]));

QBI_SR1_init[i,j,ip] =
  reduce(min, (i,j,ip,jp->i,j,jp-ip), {|ip-i==2} || {|j-jp==4 && ip-i==3} :
      ((Ebi_stacking([jp],[ip]) +
        Ebi_asymmetry([ip-i-1],[j-jp-1])) +
        Qprime[ip,jp]));

QBI_SR1_add[i,j,ip]  =
  reduce(min, (i,j,ip,jp->i,j,jp-ip), {|ip-i==2} || {|j-jp==4} :
      ((Ebi_stacking([jp],[ip]) +
        Ebi_asymmetry([ip-i-1],[j-jp-1])) +
        Qprime[ip,jp]));
```

.