Reduction Transformations

This tutorial covers transformations on reductions. Reductions are first-class expressions in Alphabets, enabling compact specification of computations as well as powerful optimizations. In particular, we will see an extremely powerful transformation that allows us to change the asymptotic complexity of the program. In addition, we will also see two other transformations that may be necessary pre-processing in order to enable simplifying reductions.

Simplifying Reductions

Simplifying Reductions takes a reduction and reuse vector as inputs. The reductions must be in a “Normalized” form before applying this transformation. Normalized form for reduction is when the reduce expression is the entire RHS of an equation. Another transformation called NormalizeReduction (described later) is available in case the reduction is not in normalized form.

The command for simplifying reductions has the following signature:
SimplifyingReduction(program, systemName, equationName);

The following in an example, using an input alphabets where simplifying reductions is already applicable.

```
affine SRExample {N|N>0}
given
    int X {i|0<=i<N};
returns
    int Y {i|0<=i<N};
through
    Y[i] = reduce(+, (i,j->i), {|j<=i} : X[j]);
.
```

The original program has a reduction that computes, for each `i` in the range, 0<=i<N, the sum of X from 0 to `i`. This is our old friend, the prefix sum, and it is easy see that the value of `Y[i]` is `Y[i-1] + X[i]`.

The reuse vector to be exploited is specified by comma delimited string. The reuse vector exploited in the script below is [-1,0]. The vector is 2D because the domain of the body of reduction is 2D.

```
prog = ReadAlphabets("SRExample.ab");
system = "SRExample";
AShow(prog);
SimplifyingReduction(prog, system, "Y", "-1,0");
Normalize(prog);
AShow(prog);
```

The resulting alphabets program is shown below. Although may seem more complicated (at leas as far as readability is concerned), but if you focus on the number of dimensions, you can observe that the reduction has one less dimension.

```
Y[i] = case
   {|i== 0} : reduce(+, (i,j->i), {|-j>= 0} : X[i]);
   {|i-1>= 0} : (reduce(+, (i,j->i), {|-i+j== 0} : X[i]) + Y[i-1]);
esac;
```

The branch `{|i==0}` is the initial computation that computes sum of X for 0<=i<=0, or `X[0]`. The other branch `{|i>=1}` uses `Y[i-1]`, and adds it to the result of another reduction. This reduction is still on 2D space, but notice the RestrictExpression `{|i==j}` that effectively decreases the dimensionality of this reduction to 1D.

Normalize Reduction

In some cases, the reduction you wish to simplify may not be in Normalized form. For example, the following is an Alphabets program with nested reductions. The inner reduction computes the prefix sum, and it can be simplified like the above example, but there is also an outer reduction that takes the max of the prefix sums.

```
affine NRExample {N|N>0}
given
    int X {i|0<=i<N};
returns
    int Y;
through
    Y[] = reduce(max, [i], {|0<=i<N} : reduce(+, [j], {|j<=i} : X[j]));
.
```

NormalizeReduction is a transformation that transforms reductions that are not in normalized form into normalized reductions. The command NormalizeReduction is overloaded and allows us to choose different scopes where the transformation is applied: all reductions in an equation, or a specific ReduceExpression. For this example, specifying the equation is sufficient (because there is only one reduce expression that is not normalized).

```
prog = ReadAlphabets("NRExample.ab");
system = "NRExample";
AShow(prog);
NormalizeReduction(prog, system, "Y");
AShow(prog);
```

In the resulting code, the body of the outer reduction is replaced with a new variable. The RHS of the new variable is the inner reduction in the original program. In other words, this transformation introduced a new variable, determined its declaration and simply renamed the reduce by this name.

This process is necessary for SimplifingReduction, because the answer of the reduction to simplify must be named in order to apply the transformation. Note that introducing new variables is not necessarily connected to extra storage in Alphabets, it simply eases mathematical reasoning.

```
Y[] = reduce(max, (i->), {|i>= 0 && N-i-1>= 0} : NR_Y);
NR_Y[i] = reduce(+, (i,j->i), {|i-j>= 0} : X[j]);
```

Reduction Decomposition

In some cases, reduction must be decomposed into reductions of lower dimensions before SimplifyingReduction is possible. The following is one such example.

The body of the reduction here has a 3D domain. Because the only variable accessed is `X[j,k]`, and it's value is independent of `i`, we can see that there is reuse along the `i` dimension. However, because the operator of the reduction is max, reuse with “[-1,0,0]” is not possible. If you try, you should see an exception saying the subtract domain is not empty but the reduction operator does not have inverse.

```
affine RDExample {N|N>2}
given
    int X {j,k|0<=(j,k)<N};
returns
    int Y {i|0<=i<N};
through
    Y[i] = reduce(max, [j,k], {|0<=j<=i && 0<=k<N-i}: X[j,k]);
.
```

Even with an operator that does not have an inverse, one dimension of reuse can still be exploited in the above example. The domain that is shrinking (= the source of the need for subtract) is the `k` dimension. Thus, the reduction along `j` can still be exploited.

The command ReductionDecomposition takes two functions that serve as new projection function for each reductions resulting from the decomposition. The first function composed with the second one should be equal to the projection function of the original reduction. For this example, we would like to make the reduction along the `j` dimension the inner reduction, so that subsequent simplification is possible.

```
prog = ReadAlphabets("RDExample.ab");
system = "RDExample";
AShow(prog);
ReductionDecomposition(prog, "0,0,0", "(i,k->i)", "(i,j,k->i,k)");
AShow(prog);
NormalizeReduction(prog, system, "Y");
SimplifyingReduction(prog, system, "NR_Y", "-1,0,0");
Normalize(prog);
AShow(prog);
```

After the decomposition, the program now contains two reductions, one along `k` and another along `j`. Now the inner reduction can be normalized and simplified.

The second argument given to the ReductionDecomposition command is the nodeID corresponding to the reduction. For now, interpret it as “1st system, 1st equation, 1st expression”, indexed from 0. How to use nodeID for specifying transformations is explained next.

```
Y[i] = reduce(max, (i,k->i), reduce(max, (i,j,k->i,k), {|i-j>= 0 && j>= 0} :
X[j,k]));
```

FactorOutFromReduction

FactorOutFromReduction is a transformation that factorizes expressions inside reduction. This step may expose more opportunities for applying SimplifyingReduction. Because the expression inside a reduction can form arbitrary sub-tree, the target expression must be specified with the nodeID like in the previous example.

In the following example program, adding values of A is added to B inside the reduction. However, the value of A used in each instance of the reduction is the same value. This reuse can be exploited by taking advantage of the distributive property (addition is distributive over max).

```
affine FactorizeExample {N|N>0}
given
    int A,B {i|0<=i<N};
returns
    int X {i|0<=i<N};
through
    X[i] = reduce(max,[j], {|j<=i} : A[i]+B[j]);
.
```

The following script uses the FactorOutFromReduction command and specify VariableExpression A as the target expression to factor out using nodeID.

```
prog = ReadAlphabets("FactorizeExample.ab");
AShow(prog);
FactorOutFromReduction(prog, "0,0,0,0,0,0");
Normalize(prog);
AShow(prog);
```

After the transformation, the new equation for X should look like the following. Addition of A has been factored out from the reduction, and the reduction now only involves B. Variable B also has a reuse along the `i` dimension, across multiple instances of the reduction. This reuse can now be exploited because the access to A using the `i` dimension has been factored out.

```
X[i] = {|N-i-1>= 0 && i>= 0} : (reduce(max, (i,j->i), {|i-j>= 0} : B[j]) +
A);
```

Using other transformations covered in this tutorial, the sequence of transformations from factoring out A to applying SimplifyingReducion can be done through a script.

```
prog = ReadAlphabets("FactorizeExample.ab");
system = "FactorizeExample";
AShow(prog);
```

```
FactorOutFromReduction(prog, "0,0,0,0,0,0");
Normalize(prog);
AShow(prog);
NormalizeReduction(prog, system, "X");
SimplifyingReduction(prog, system, "NR_X", "-1,0");
Normalize(prog);
AShow(prog);
```

Now the program is simplified (although it has many more lines of code).

```
X[i] = {|N-i-1>= 0 && i>= 0} : (NR_X + A);
NR_X[i] = case
{|i== 0} : reduce(max, (i,j->i), {|j== 0} : B[j]);
{|i-1>= 0} : (reduce(max, (i,j->i), {|-i+j== 0} : B[j]) max NR_X[i-1]);
esac;
```

How to Specify an Expression in AST

In the previous examples, nodeID was used to specify the target expression. For transformations that can be applied to expressions that may be in arbitrary locations, simply specifying the target system or variable may not be enough. NodeID is a mechanism to specify exactly which expression to apply the transformation from the script interface.

The nodeID is a vector of integers that uniquely identifies expressions in Alphabets program. It mostly corresponds to the xth branch to traverse in the AST. PrintAST can be used to find the nodeID for every Expression in the AST. You could also traverse the AST yourself, which may be a good way for simple programs like the above example.

The following is the output of PrintAST for reduction decomposition example (initial state).

```
_Program
   |_AffineSystem
   |   |nodeId = (0)
   |   |_RDExample
   |   |_ParameterDomain
   |   |   |+-- {N|N-3>= 0}
   |   |_VariableDeclaration
   |   |   |+-- X
   |   |   |+-- {j,k|N-3>= 0 && j>= 0 && N-j-1>= 0 && k>= 0 && N-k-1>= 0}
   |   |   |+-- int
   |   |_VariableDeclaration
   |   |   |+-- Y
   |   |   |+-- {i|N-3>= 0 && i>= 0 && N-i-1>= 0}
   |   |   |+-- int
   |   |_StandardEquation
   |   |   |nodeId = (0,0)
   |   |   |+-- Y
   |   |   |_ReduceExpression
```

```
   |   |   |   |nodeId = (0,0,0)
   |   |   |   |+-- (i,j,k->i)
   |   |   |   |_RestrictExpression
   |   |   |   |   |nodeId = (0,0,0,0)
   |   |   |   |   |+-- {i,j,k|N-3>= 0 && j>= 0 && i-j>= 0 && k>= 0 && N-i-
k-1>= 0}
   |   |   |   |   |_DependenceExpression
   |   |   |   |   |   |nodeId = (0,0,0,0,0)
   |   |   |   |   |   |+-- (i,j,k->j,k)
   |   |   |   |   |   |_VariableExpression
   |   |   |   |   |   |   |nodeId = (0,0,0,0,0,0)
   |   |   |   |   |   |   |+-- X
```

The following is the output of PrintAST for factor out example (initial state).

```
_AffineSystem
   |nodeId = (0)
   |_FactorizeExample
   |_ParameterDomain
   |   |+-- {N|N-1>= 0}
   |_VariableDeclaration
   |   |+-- A
   |   |+-- {i|N-1>= 0 && i>= 0 && N-i-1>= 0}
   |   |+-- int
   |_VariableDeclaration
   |   |+-- B
   |   |+-- {i|N-1>= 0 && i>= 0 && N-i-1>= 0}
   |   |+-- int
   |_VariableDeclaration
   |   |+-- X
   |   |+-- {i|N-1>= 0 && i>= 0 && N-i-1>= 0}
   |   |+-- int
   |_StandardEquation
   |   |nodeId = (0,0)
   |   |+-- X
   |   |_ReduceExpression
   |   |   |nodeId = (0,0,0)
   |   |   |+-- (i,j->i)
   |   |   |_RestrictExpression
   |   |   |   |nodeId = (0,0,0,0)
   |   |   |   |+-- {i,j|N-1>= 0 && i-j>= 0}
   |   |   |   |_BinaryArithmeticExpression
   |   |   |   |   |nodeId = (0,0,0,0,0)
   |   |   |   |   |+-- ADD
   |   |   |   |   |_DependenceExpression
   |   |   |   |   |   |nodeId = (0,0,0,0,0,0)
   |   |   |   |   |   |+-- (i,j->i)
   |   |   |   |   |   |_VariableExpression
   |   |   |   |   |   |   |nodeId = (0,0,0,0,0,0,0)
   |   |   |   |   |   |   |+-- A
   |   |   |   |   |_DependenceExpression
```

```
|   |   |   |   |   |nodeId = (0,0,0,0,0,1)
|   |   |   |   |   |+-- (i,j->j)
|   |   |   |   |   |_VariableExpression
|   |   |   |   |   |   |nodeId = (0,0,0,0,0,1,0)
|   |   |   |   |   |   |+-- B
```