A New Control Structure for Transformation-Based Generators

Ted J. Biggerstaff

Email: tbiggerstaff@austin.rr.com

Abstract. A serious problem of most transformation-based generators is that they are trying to achieve three mutually antagonistic goals simultaneously: 1) deeply factored operators and operands to gain the combinatorial programming leverage provided by composition, 2) high performance code in the generated program, and 3) small (i.e., practical) generation search spaces. The hypothesis of this paper is that current generator control structures are inadequate and a new control structure is required. To explore architectural variations needed to address this quandary, I have implemented a generator in Common LISP. It is called the Anticipatory Optimization Generator (AOG[1]) because it allows programmers to **anticipate** optimization opportunities and to prepare an abstract, distributed plan that attempts to achieve them. The AOG system introduces a new control structure that allows differing kinds of knowledge (e.g., optimization knowledge) to be anticipated, placed where it will be needed, and triggered when the time is right for its use.

Problems

A serious problem of most transformation-based generators is that they are trying to achieve three mutually antagonistic goals simultaneously: 1) deeply factored operators and operands to gain the combinatorial programming leverage provided by composition, 2) high performance code in the generated program, and 3) small (i.e., practical) generation search spaces. This paper will make the argument that this quandary is due in large measure to the control structure characteristics of large global soups of pattern-directed transformations. While pattern-directed transformations make the specification of the transformations easy, they also explode the search space when one is trying to produce highly optimized code from deeply factored operators and operands. Since giving up the deep factoring also gives up the combinatorial programming leverage provided by the composition, that is not a good trade off.

There are other problems in addition to the search space explosion. Pattern-directed transforms provide relatively few tools for grouping sets of transformations, coordinating their operation, and associating them with a large grain purpose that transcends the fine grain structural aspects of the target program. For example, there are few and crude tools to express the idea that some subset of tightly related,

[1] Much of this work was done at Microsoft Research and the author gratefully acknowledges the support of Microsoft.

cooperating transformations is designed for the narrow purpose of creating, placing and merging loops in the target program.

Along a similar vein, it is difficult to generate target code optimized for differing machine architectures from a single canonical specification. For example, one would like to be able to generate code optimized for a non-SIMD architecture and with the flip of a switch generate code optimized for a SIMD architecture. With conventional systems this is difficult.

There is no way currently to record knowledge about the reusable components that may lead to optimization opportunities during the generation process. For example, the writer of a reusable component may know that a property of his component ensures that part of its code can be hoisted above a loop that will be generated to host the component. Conventional transformation systems allow no easy way to express and then later exploit such information.

Similary, most conventional transformation systems provide no mechanism to record knowledge of future optimization opportunities that become known in the course of executing early transformations. Further, there is no way to condition such optimization opportunities upon optimization events (e.g., a substitution of a particular structure) or expected properties of the generated code (e.g., the requirement that an expression must have been simplified to a constant).

Finally, there is no way to intermix (abstractly) planned optimization operations with opportunistic optimization operations that cannot be planned because the opportunities for these optimizations arise unpredictably as a consequence earlier transformations manipulating the target program code.

We will introduce a new kind of transformation and a new control structure (called a *tag-directed control structure*) that can overcome much of the search space explosion problem and also can address these other problems. Let us review convention transformation architectures, contrast the new architecture with respect to these conventional systems and then examine how this new architecture addresses these problems.

The New Control Structure

Overview

Conventional Transformation Systems. Conventionally, generic transformation systems store knowledge as a single global soup of transformations represented as rules of the form

`syntactic pattern` $\Rightarrow$ ***`reformulated structure`***

The left hand side of the rule recognizes the syntactic form and binds matching elements of the program being transformed to transformation variables (e.g., **`?operator`**) in the pattern. If that is successful, then the right hand side of the rule (instantiated with the variable bindings) replaces the matched portion of the program. Operationally, rules are chosen (i.e., triggered) based largely on the syntactic pattern of the left hand side, which may include type constraints as well as purely syntactic patterns. Rules may include some set of additional constraints (often called *enabling* conditions) that must be true before the rule can be triggered. However, all of this is not entirely sufficient since pure declarative forms are often inadequate to express complex procedural transformations. Therefore, the rules of some systems also allow for arbitrary computations to occur during the execution of the rules.

The operational form of the program being transformed may be text or, more typically in modern systems, an Abstract Syntax Tree (AST). In summary, the major control structure of generic transformation systems is based largely on the pattern of the program portion being transformed. Hence, we call such systems *pattern-directed.*

Draco. Some transformation systems add control variations to enhance efficiency. For example, Draco [10] separates its rules into two kinds: 1) refinement rules, and 2) optimization rules. Refinement rules add detail by inlining definitions. Optimization rules reorganize those details for better performance. Further, the refinement rules are grouped into subsets that induce a set of translation "stages" by virtue of the fact that each group translates from some higher level domain specific language (e.g., the language of the relational algebra) into one or more lower level domain specific languages (i.e., ones that are closer to conventional programming languages, such as a tuple language). After each such refinement stage, optimization rules are applied to reorganize the current expression of the program into a more optimum form within the current domain specific language. The final output is a form of the program expressed in a conventional programming language such as C. While such generators usually produce programs with adequate performance and do so within an acceptable period of time, in some domains, the search space of the optimization phases tends to explode.

Tag-Directed. A key insight of AOG (i.e., the use of tags to identify and trigger optimizing transformations) arose from an analysis of several transformation-based derivations of graphics functions. The derivations were quite long with a series of carefully chosen and carefully ordered transformations that prepared the program for the application of key optimizing transforms. The need to choose exactly the right transform at exactly the right point over and over again seemed like a long series of miracles that implied a huge underlying search space. Further, I noticed that the choice of the various transformations depended only weakly on the patterns in the program. Rather they depended on other kinds of knowledge and relationships. Some of this was knowledge specific to the reusable components and could be attached to the components at the time of their creation. Other knowledge arose in the course of transformation operation (e.g., the creation of a data flow introduced by a preparatory transformation). Tags serve to capture such knowledge and thereby they became a key element of the AOG control structure. Such use of tags motivates the moniker of *tag-directed* transformations for those optimizing transformations that are triggered largely because of the tags attached to the AST. In triggering these optimizing transformations, patterns play a lesser role.

Event Triggering. The transformation name in a tag indicates **what** transformation to fire but not **when**. Thus, the tag control structure includes the idea of *local and global events* associated with the tag structure that indicate when to fire the transformations. Global events, which apply to the whole subtree being operated on, provide a way to induce a set of optimization stages. Each stage has a general optimization purpose, which assures certain global properties are true upon completion of the stage. For example, all loop merging is complete before commencing the stage that triggers certain kinds of code hoisting outside of loops. Local events, on the other hand, are events specific to an AST subtree such as its movement or substitution. Local events occur in the course of some other transformation's operation. Local events allow opportunistic transformations to be interleaved among the transformations triggered during a stage.

Components. Another control structure innovation is to represent passive and active components by different mechanisms. Passive components (just called *components*) are those for which one can write a concrete, static definition. These comprise the library reusable piece parts from which the target program will be built and they are represented in an Object Oriented hierarchy. For example, one of the data types specific to the graphics domain language that I use is a *Neighborhood*, which is a subset of pixels in a larger image centered on some specific pixel within that larger image. Specific instances of neighborhoods are defined via functional methods that compute: 1) the indexes of neighborhood pixels in terms of the image's indexes (**`Row`** and **`Col`**), 2) a set of convolution weights associated with individual pixel positions within the neighborhood (**`W`**), and 3) the range of the relative offsets from the centering pixel (**`PRange, QRange`**). These methods, like conventional refinement rules, will be inlined and thereby will form portions of the target program.

The operator components (e.g., the convolution operator ⊕) are defined in the operator subtree of this OO hierarchy and act like multi-methods whose definition is determined by the type signature of the operator expression. For example, "(⊕ array [iterator, iterator], neighborhood)" is a signature that designates a method of ⊕ with a static, inline-able definition which will become the inner loop of a convolution.

Transformations. The transformations are the active components and are represented as executable functions rather than isolated declarative rules. This allows them to handle high degrees of AST variation; compute complex enabling conditions; and recognize why enabling conditions are failing and take actions to fix them (e.g., by directly calling another transformation). Representing such transformations as conventional rules would require splitting them up into a number of individual transformations which would thereby explode the generator search space. Larger grain transformations that are implemented as programmable functions prevent this explosion.

Kinds of Transformations. AOG transformations come in two flavors – pattern-directed and tag-directed. Pattern-directed transformations are used for program refinement stages and tag-directed are used for optimization stages. Pattern-directed transformations are organized into the OO hierarchy to help reduce the number of candidate transformations at each point in the AST. That is to say, the OO hierarchy captures a key element of the pattern – the type of the subtree being processed, which saves looking at a large number of transformations that might syntactically match but semantically fail.

Partial Evaluation. Like mathematical equations, reformulations of program parts require frequent simplication. If simplifications are represented as isolated transformations in a global soup of transformations, they explode the search space because they can be applied at many points, most of which are inappropriate. Therefore, AOG contains a partial evaluator that is called for each new AST subtree to perform that simplication. A partial evaluator is a specialized agent that simplifies expressions without exploding the search space.

Now let us examine these ideas in the context of an example.

Related Interdependent Knowledge

Sometimes *a priori* knowledge about the interdependence of several aspects of a problem implies a future optimization opportunity and using this knowledge in course of generation can reduce the generator's search space. For example, a component writer may know that in a particular architectural context (e.g., a CPU without parallel processing instructions) a specific optimization will be enabled within a component and will be desirable at a particular stage of generation. How should such knowledge be captured? Where should it be kept? And how should the optimization be triggered? Let us look at a concrete example of this situation.

Suppose that the component writer is creating a description of a pixel neighborhood (call it **s**) within a graphics image and that description will be used in the context of graphics operators such as the convolution[2] operator. This neighborhood description comprises a set of methods of **s** that describe the size and shape of the neighborhood, the weights associated with each relative position in the neighborhood, how to compute the relative positions in terms of the **[i,j]** pixel of the image upon which the neighborhood is centered, and any special case processing such as the special case test for the neighborhood hanging off the edge of the image. Let us consider the method that defines the weights for a particular neighborhood **s**. If any part of the neighborhood is hanging off the edge of the image, the weight will be defined as 0. Otherwise, the weights will be defined by the matrix:

$$\begin{array}{cc} & \overbrace{\begin{array}{ccc}-1 & 0 & 1\end{array}}^{Q} \\ P\left\{\begin{array}{c}-1\\ 0\\ 1\end{array}\right. & \begin{bmatrix}-1 & 0 & 1\\ -2 & \langle 0\rangle & 2\\ -1 & 0 & 1\end{bmatrix}\end{array}$$

where **p** and **q** define the pixel offset from image pixel upon which the neighborhood is centered. The diamond bracketed entry indicates the center of the neighborhood. Then the definition of the weight method **w** for pixel **[p,q]** in the neighborhood **s** is defined by the following pseudo-code, where **s** is centered on the **[i,j]** pixel of an **m x n** image:

```
w.s(i,j,m,n,p,q) ⇒
{if ((i==0) || (j==0) || (i==(m - 1)) || (j==(n - 1)))
    then 0;
```

[2] A convolution is a graphics operator that computes an output image **b** from an input image **a** by operating on each neighborhood around each pixel **a[i,j]** to produce the corresponding **b[i,j]** pixel. Specifically, the operation is a sum of products of each of the pixels in the neighborhood around **a[i,j]** times the weight value defined for that pixel in the neighborhood.

```
      else {if ((p!=0) && (q!=0))
             then q;
             else {if ((p==0) && (q!=0))
                       then (2 * q); else 0 }}}
```

What does the component writer know about this component that might be helpful in the course of code generation? He knows that the eventual use of the weight calculation will be in the context of some image operator (e.g., a convolution). Further, that context will comprise a 2D loop iterating over the neighborhood that will be nested within another 2D loop iterating over the whole image. Further, the component writer knows that the special case test is not dependent on the neighborhood loop and, therefore, the test can be hoisted outside of the neighborhood loop. Or equivalently, the loop can be distributed over the **`if`** statement producing an instance of the loop in both the **`then`** and **`else`** clauses.

Since the specific instance of the image operation has not been chosen and these loops have not yet been generated, the component writer cannot execute the potential transformation yet. The component writer can only use the abstract knowledge that defines, in general terms, the eventual solution envelope. What he would like to do is to associate a piece of information with the **`if`** statement that would indicate the transformation to be invoked and some indication of when it should be invoked. AOG provides a mechanism for accomplishing this by allowing the component writer to attach a tag (to be used by the generator) to the **`if`** statement. In this case, the tag has the form:

```
(_On SubstitutionOfMe
     (_PromoteConditionAboveLoop ?p ?q))
```

This tag is like an interrupt that is triggered when the **`if`** statement gets substituted in some context, i.e., when a *local substitution event* happens to the **`if`** statement. When triggered, the tag will cause the **`_PromoteConditionAboveLoop`** transform to be called with the name of the target loop control variables (i.e., the values **`p`** and **`q`**, which will be bound to the generator variables **`?p`** and **`?q`**) as parameters. The transform will find the loops controlled by the values of **`p`** and **`q`**, check the enabling conditions, and if enabled, perform the distribution of the loop over the if statement. Thus, **`_PromoteConditionAboveLoop`** will transform a form like

```
{Σp,q  (p ∈ [-1:1] ) (q ∈ [-1:1] ) :
         { if(i==0 || j==0 || i==m-1 || j==n-1)
              /* Off edge */
              then <special case processing>;
              else <general case processing> }}
```

into a form like

```
{ if(i==0 || j==0 || i==m-1 || j==n-1) /* Off edge */
  then {Σp,q (p ∈ [-1:1] ) (q ∈ [-1:1] ):
```

```
                 <special case processing>}
    else {Σp,q (p ∈ [-1:1] ) (q ∈ [-1:1] ):
                 <general case processing>} }
```

Thus, with these ideas, we have introduced a new kind of transformation, which we will call a *tag-directed transformation*. Such transformations are triggered by events, in contrast to conventional *pattern-directed transformations* that are triggered largely by patterns in the AST (Abstract Syntax Tree). Another difference from pattern-directed transformations is that tag-directed transformations and their host tags can capture knowledge that is not easily derivable from the AST patterns or operator/operand semantics. Such knowledge would require some deep inference, some sense of the optimization opportunities particular to the evolving program or some knowledge that is fleetingly available in the course of transformation execution. For example, the tag-directed example we examined takes advantage of several interrelated knowledge nuggets that have little to do with the structure of the AST:

1) the knowledge of the case structure of the reusable component **`s`** where the general purpose of the branching is known (i.e., one branch is a special case),
2) the knowledge that the reusable component will be used in the context of neighborhood loops whose general purpose is known *a priori*,
3) the knowledge that the **`if`** condition within the component is independent of the anticipated neighborhood loops,
4) the optimization knowledge that executing an **`if`** test outside of the loop instead of within the loop is more computationally efficient, and
5) the generation knowledge that attaching an interrupt-like tag (incorporating all of this special knowledge) to the reusable component will produce search space reduction advantages by eliminating the search for which transform to fire, where to apply it in the AST, and when to fire it.

The key objective of tag-directed transformations is to reduce the generator search space explosion that arises when each transformation is just one of many in a global soup of transformations. By using all of this knowledge together, we can eliminate much of the branching in the search space (i.e., eliminate all of the alternative transformations that might apply at a given point in the generation) because tags supply all of the information needed to make the choice of transformation. They determine:

1) **which** transformation is called (i.e., it is explicitly named in the tag),
2) **when** it is called (i.e., when the named event occurs), and
3) **where** in the AST to apply the transform (i.e., it is applied to the structure to which the tag is attached).

The Likelihood of Optimization Opportunities

Not all knowledge that one might want to use is deterministic. Often, there is just the likelihood that some optimization friendly condition may occur as a result of code manipulation. This knowledge too is valuable in keeping the search space from exploding while simultaneously achieving important generation goals such as eliminating redundant code. An example of this situation is illustrated by the expression for Sobel edge detection in bitmapped images.

```
DSDeclare image a, b :form ( array m n) :of bwpixel;
b = [ (a ⊕ s)² + (a ⊕ sp)2]^(1/2) ;
```

where **`a`** and **`b`** are **`(m X n)`** grayscale images and ⊕ is a convolution operator that applies the template matrices **`s`** and **`sp`** to each pixel **`a[i,j]`** and its surrounding neighborhood in the image **`a`** to compute the corresponding pixel **`b[i,j]`** of **`b.`** **`s`** (whose **`w`** method was defined earlier) and **`sp`** are OO abstractions that define the specifics of the pixel neighborhoods. It is possible and indeed, even likely that the special case processing seen in the **`w`** method of **`s`** will be repeated in the **`w`** method of **`sp`**. If so, it would be desirable to share the condition test code if possible by applying the **`_MergeCommonCondition`** transformation:

```
{ if ?a then ?b else ?c; if ?a then ?d else ?e }
=>  if ?a then {?b; ?d} else {?c ; ?e}
```

where the **`?x`** syntax represents generator variables that will be bound to subtrees of the AST. In the above example, **`?a`** will be bound to the common special case condition code that tests for the neighborhood hanging partially off the edge of the image. Because the component writer anticipates this possibility, he would like to hang a tag on the **`if`** statement in the **`w`** definitions of **`s`** and **`sp`** that will cause the **`_MergeCommonCondition`** transformation to be called. If the condition code is common and all other enabling conditions are met, it will perform the transformation. However, this raises a question. What event should this transformation be triggered on?

Local events like substitution of the subtree would be a bad choice because there is no easy way to assure ordering of the strategic computational goals of the overall optimization process. For example, we want all code sharing across domain specific subexpressions (i.e., global code manipulations) to be completed before any in-place optimizations (i.e., mostly local manipulations) begin. The approach used by AOG is to separate the strategic processing into phases that each have a general purpose such as

1. inlining definitions (e.g., substituting the method definition of **`w.s`**),
2. sharing code across expressions (e.g., sharing common test conditions),
3. performing in-place optimizations (e.g., unrolling loops), and
4. performing clean up optimizations (e.g., eliminating common subexpressions).

These phases behave like an abstract algorithm where the details of the algorithmic steps come from the transformations mentioned in the tags attached to the components. The start of each stage is signaled by the generator posting a global event that will cause all tags waiting on that event to be scheduled for execution. This is how **`_MergeCommonCondition`** gets called to share the condition test code common to **`w.s`** and **`w.sp`**. It is scheduled when the global event signalling the start of the cross expression code sharing is posted by AOG.

So, here we have a new control structure construct that allows a useful separation of concerns. The generator writer provides the broad general optimization strategy by defining stages, each with a narrow optimization goal. The component writer at library creation time (or a transformation during generator execution) adds tags that supply the details of the steps. This means that the generator writer does not have to account for all the possible combinations of purposes, sequences, enabling conditions, etc. He only has to create one (or more) abstract algorithms (i.e., define a set of stages) that are suitable for the classes of optimization that might occur. Similarly, the component writer (or transformation writer in the case where the tags are dynamically added to the AST) can add tags that take advantage of every bit of knowledge about the components, even possibilities that may or may not arise in any specific case. This kind of separation of concerns avoids much of the search space explosion that occurs when all of the strategic goals, component-specific details, and their interdependencies reside in one central entity such as the generator's algorithm or in a global soup of transformations.

The view that tag-directed transforms are like interrupts is an apt simile because they mimic both the kind of design separation seen in interrupt-driven systems as well as the kind of operational behavior exhibited by interrupt-driven systems.

Simplification Opportunities

Not all optimizations fit the tag-directed model. There are many opportunities for simplification by partial evaluation. In fact, any newly formed AST subtree is a candidate for such simplification and each transformation that formulates new subtrees immediately calls the partial evaluator to see if the subtrees can be simplified. For example, during the in-place optimization phase, one of the neighborhood loops is unrolled by a tag-directed transformation. The pseudo-code of the internal form of the loop looks like:

```
{_sum (p q) (_suchthat (_member p (_range -1 1))
                       (_member q (_range -1 1)))
            {if ((p!=0) && (q!=0))
               then (a[(i + p),(j + q)]*p);
               else {if ((p!=0) && (q==0))
                       then (a[(i + p),(j + q)]*(2*p));
                       else 0;}}}
```

where the **_sum** operator indicates a summation loop and the **_suchthat** clause indicates that **p** and **q** range from –1 to +1. This produces two levels of loop unwrapping. In the course of the first level of unwrapping (i.e., the loop over **p**), one of the terms (i.e., the one for **(p==1)**) has the intermediate form:

```
{_sum (q) (_suchthat (_member q (_range -1 1)))
          {if  (q!=0)
            then a[(i+1),(j+q)] ;
            else {if (q==0) then (a[(i+1),(j+q)]*2);
                             else 0;}}}
```

When the remaining loop over **q** is subsequently unrolled, we get the expression

```
({if (-1!=0)
     then a[(i+1),(j-1)] ;
     else {if (-1==0) then (a[(i+1),(j-1)]*2);
                      else 0;}}
+ {if (0!=0)
      then a[(i+1),(j+0)];
      else {if (0 == 0) then (a[(i+1),(j+0)]*2);
                        else 0;}}
+ {if (1!=0)
      then a[(i+1),(j+1)] ;
      else {if (1 == 0) then (a[(i+1),(j+1)]*2);
                        else 0;}})
```

The unroll transform calls the partial evaluator on this expression which produces the final result for this subloop:

```
(a[(i+1),(j-1)] + (a[(i+1),j]*2) + a[(i+1),(j+1)])
```

In the same way, the other derived, nested loops produce zero or more analogous expressions for a total of six terms for the original loop over **p** and **q**.

Partial evaluation is critical because it allows future transformations to execute. Without it, many future transformations would fail to execute simply because of the complexity of detecting their enabling conditions or the complexity of manipulating un-simplified code.

Partial evaluation is the most executed transformation. For the Sobel edge detection expression, 44 out of a total of 92 transformations required to generate code are partial evaluation.

Architectural Knowledge

No aspect can have a larger effect on the final form of the generated code than the architecture of the CPU. Consider the two different sets of code produced by AOG for a CPU without parallel instructions and one with parallel instructions (i.e., the MMX instructions of the Pentium^{TM} processor).

For a single CPU Pentium machine without MMX instructions (which are SIMD instructions that perform some arithmetic in parallel), the AO generator will produce code that looks like

```
for (i=0; i < m; i++)  /* Version 1 */
    {im1=i-1; ip1= i+1;
     for (j=0; j < n; j++)
          { if(i==0 || j==0 || i==m-1 || j==n-1)
              then b[i, j] = 0; /* Off edge */
              else {jm1= j-1; jp1 = j+1;
                    t1 = a[im1,jm1]*(-1)+a[im1,j]*(-2) +
                         a[im1,jp1]*(-1)+a[ip1,jm1]*1 +
                         a[ip1,j]*2+a[ip1,jp1]*1;
                    t2 = a[im1,jm1]*(-1)+a[i,jm1]*(-2) +
                         a[ip1,jm1]*(-1)+a[im1,jp1]*1 +
                         a[i,jp1]*2+a[ip1,jp1]*1;
                     b[i,j] = sqrt(t1*t1 + t2*t2 )}}}
```

This result requires 92 large grain transformations and is produced in a few tens of seconds on a 400 MHz Pentium. In contrast, if the machine architecture is specified to be MMX, the resultant code is quite different:

```
{int s[(-1:1), (-1:1)]={{-1, 0, 1}, {-2, 0 , 2},
                        {-1, 0, 1}};/* Version 2 */
int sp [(-1:1), (-1:1)]={{-1, -2, -1}, {0, 0, 0},
                         {1, 2, 1}};
for (j=0; j<n; j++) b[0,j] = 0; /*Zero image edge */
for (i=0; i<m; i++) b[i,0] = 0; /*Zero image edge */
for (j=0; j<n; j++) b[(m-1),j] = 0;/*Zero image edge */
for (i=0; i<m; i++) b[i,(n-1)] = 0;/*Zero image edge */
{ for (i=1; i < (m-1); i++)    /*Process inner image */
  { for (j=1; j < (n-1); j++)
      {t1 = unpackadd(padd2(padd2(pmadd3(&(a[i-1,j-1]),
                                         &(s[-1, -1])),
                                  pmadd3(&(a[i, j-1]),
                                         &(s[0, -1]))),
                            pmadd3(&(a[i+1,j-1]),
                                   &(s[ 1, -1])));
       t2 = unpackadd(padd2 (pmadd3 (&(a[i-1, j-1]),
                                     &(sp [-1, -1])),
                             pmadd3 (&(a[i+1, j-1]),
                                     &(sp [0,-1])))));
       b[i,j] = sqrt(t1*t1 + t2*t2);}}}
```

where the routines **unpackadd**, **padd2**, and **pmadd3** correspond to MMX instructions and are defined as **pmadd3 ((a0, a1, a2) , (c0, c1, c2)) = (a0*c0+a1*c1, a2*c2+0*0**[3]**), padd2 ((x0, x1) , (x2, x3)) = (x0+x2, x1+x3),** and **unpackadd((x0, x1)) = (x0+x1).** All lend themselves to direct translation into MMX instruction sequences. In this example, **s** and **sp** have become pure data arrays to optimize the use of the MMX instructions. Notice, that the special case that tests to see if the template is hanging over the edge of the image, has completely disappeared. Transformations have split the main loop on that test, turning the single loop of the previous version into five loops by

[3] MMX instructions use registers containing an even number of operands to be operated on in parallel. Thus, **pmadd3** is just a special case of **pmadd4((a0, a1, a2, a3), (c0, c1, c2, c3))** with a zero padded fourth operand in each of the registers **a** and **c**. **pmadd4** is defined as **(a0*c0+a1*c1, a2*c2+a3*c3)**. Thus, **pmadd3 ((a0, a1, a2) , (c0, c1, c2))** is actually the special case **pmadd4 ((a0, a1, a2, 0) , (c0, c1, c2, 0))**.

incorporating the special case test logic into the loop control logic. Four of the loops plug zeros into the four edges of the image (i.e., the new form of the special case processing) and one loop processes the inside of the image (i.e., the non-special case processing). The fundamental difference in the derivation of the two versions is in the tag driven optimization phase. Up to that stage, the transformations that fire are the same, resulting in two programs that are the same except for the tags.

How do we get such dramatic differences from the same specification expression? The short answer is that there are differently tagged reusable components for each architecture. The reusable component definitions look like

```
if (?MMX == 'MMX) then <MMXDefinition>
                  else <NonMMXDefinition> ;
```

Given a global generator variable (say **?MMX**) bound to a value indicating the target architecture, partial evaluation of the reusable components will result in differently tagged definitions for each architectural variation. Thus, after this partial evaluation, the weight method of **s** for the MMX architecture will have some additional tags that are MMX specific.

In order to exploit MMX instructions, the generated code needs to have a couple of important architectural properties. The weights need to be formed into a vector to exploit the vector processing of the MMX instructions. Ideally, this should happen at generation time but failing that, it could happen at target program execution time by computing all weights first and then using the resulting vector to compute the convolution. The second important architectural property is a branch free expression of the convolution computation so that the vector processing instructions will not be interrupted by branch instructions.

Let's us sketch how these properties are achieved. We will start with getting **s** and **sp** into vector form. This is accomplished by attaching the tag

```
(_On SubstitutionOfMe  (_MapToArray))
```

to the non-special case branch of **w.s** and **w.sp**. For **w.s**, this branch would be the expression

```
{if ((p!=0) && (q!=0)) then q;
     else {if ((p==0) && (q!=0)) then (2*q); else 0 }}
```

When triggered, **_MapToArray** replaces this expression with a reference to a vector **s[p,q]** which it then sets about creating. First, it formulates the data declaration

```
int s[(pRange.s), (qRange.s)]=
{∀p,q (p ∈ pRange.s ) (q ∈ qRange.s ) :
         {if ((p!=0) && (q!=0))
              then q;
             else {if ((p== 0)&&(q!=0)) then (2*q);
                                          else 0 }}}
```

which after substitution of the definitions of **pRange.s** and **qRange.s** followed by partial evaluation, becomes

```
int s[(-1:1),(-1:1)]={{-1, 0, 1},{-2, 0 , 2},
                      {-1, 0, 1}}
```

The next job is to get rid of the branching in the loop body. This is handled by the attached tag[4]

```
(_On CFWrapUpEnd(0) ( _SplitLoopOnCases))
```

to the top level **if** statement in **w.s**. This will effect the incorporation of each of the cases of the condition test

```
((i==0) || (j==0) || (i==(m - 1)) || (j==(n - 1)))
```

into a different loop, thereby producing the five loops in the MMX code shown earlier. **_MapToArray** checks the enabling conditions, deconstructs both the loop control information and the branching test, and reformulates the single loop structure into

```
{∀i,j ((i∈[0:(m-1)]) && (j∈[0:(n-1)]) && (i==0)):
         b[i,j] = 0 }
{∀i,j ((i∈[0:(m-1)]) && (j∈[0:(n-1)]) && (j==0)):
         b[i,j] = 0 }
{∀i,j ((i∈[0:(m-1)]) && (j∈[0:(n-1)]) && (i==(m-1)):
         b[i,j] = 0 }
{∀i,j ((i∈[0:(m-1)]) && (j∈[0:(n-1)]) && (j==(n-1)):
         b[i,j] = 0 }
{∀i,j ((i∈[0:(m-1)]) && (j∈[0:(n-1)]) && (i!=0) &&
       (j!= 0) && (i!=(m - 1)) && (j!=(n - 1))):
         … non-special process…}
```

To simplify the control expressions, **_MapToArray** invokes some lightweight inference and uses AOG's built-in schema language to perform that inference. (See 6 for more details of the schema language and this inference operation.) It uses a set of rules that recognize an iteration pattern and thereby, determine how to simplify the loop control and the loop body. For example, suppose that the control variable **i** is really a fixed constant (e.g., **(i∈[0:(m-1)]) && i==0)**). This would engender elimination of the control variable **i** from the loop control and substitution of **0** everywhere **i** appears in the loop body. Other rules recognize patterns such as those that require clipping the upper and/or lower limits of the control range, which is how the non-special processing loop control is generated.

---

[4] **CFWrapUpEnd** is the event signalling the clean up optimization phase and the zero parameter on the event indicates that it should be scheduled before any other transforms in that phase that have larger parameters.

There is still more massaging to do. At this point, the non-special processing loop body looks like

```
{t1= {Σp,q (p∈[-1:1])(q∈[-1:1]):a[(i+p),(j+q)]*s[p,q]}
 t2= {Σp,q (p∈[-1:1])(q∈[-1:1]):a[(i+p),(j+q)]*sp[p,q]}
 b[i,j] = sqrt(t1*t1 +  t2*t2);}
```

The loops over **p** and **q** (i.e., **{Σp,q: … }**) arose from the definition of the convolution operator, which was defined by the "(⊕ array [iterator, iterator], neighborhood)" method of the ⊕ operator

```
(a[i,j] ⊕  s) ⇒
   {Σp,q (p∈ pRange.s) (q∈qRange.s ):
         {a[row.s(i,j, m, n, p,q),col.s(i,j, m,n,p,q)]
          * w.s(i,j,m,n,p,q) }}
```

and which has a tag of the form

```
(_ON CFWrapUpEnd(1)  (_MMXLoop))
```

attached to its loop. Because the tag was preserved during definition inlining, this tag is now attached to both of the loops over **p** and **q** since they are derived from the definition of the convolution operator. The **_MMXLoop** transformation will massage these loop controls and loop bodies into the form shown in the final MMX pseudo-code shown earlier.

Creation, Placement, and Merging of Loops

The interrupt style control structure of tag-directed transformations is not ideal for all generator operations. There is still a role for pattern-directed transformations for some refinement-like operations. Even so, a large global soup of pattern-directed transformations is not optimal from a search-space point of view so we will adapt that model to the job required by AOG.

The pattern-directed processing (which precedes the tag-directed processing) performs loop introduction, placement and merging. In the course of that, it may also perform some opportunistic optimizations (e.g., reduction in strength of the square operator). At the end of the pattern-directed processing, the example has the following form expressed in a C-ish pseudo-code.

```
for (i=0; i < m; i++)
 {for (j=0; j < n; j++)
   {t1 = (a[i,j] ⊕ s);
    t2 = (a[i,j] ⊕ sp);
    b[i,j] = sqrt(t1*t1 +  t2*t2)  }}
```

Like APL, AOG operators use implicit looping. This simplifies the specification code in that most domain operations can be written as loop-free expressions of operators and operands. It also makes manipulation of the code much simpler than if the programmer where allowed to program arbitrary loops. However, it makes AOG's job harder in that it must generate the loops and figure out how to eliminate redundant looping operations. It does this by executing an abstract algorithm comprising three phases.

1) Walk down the expression tree reducing operators and adding tag information that describes the implicit looping,

2) Walk back up the tree moving and merging the looping information to eliminate redundant loops, and

3) Walk over the tree again generating loop code, adding tags needed by the tag-directed phase, and executing opportunistic optimizations (e.g., the reduction in strength optimization of the expressions $\mathbf{(a[i,j]\oplus s)^2}$ and $\mathbf{(a[i,j]\oplus sp)^2}$).

Like the tag-directed phases, the details of the abstract algorithm operations are contained in the code of the transformations. However, the control structure inherent to loop creation and placement does not lend itself to the interrupt-like design of the tag-directed transformations. As the looping data is moved up the expression tree, the rules by which it is transformed and merged are largely determined by the patterns and semantics of the operators and operands in the tree. This suggests that pattern-directed transformations would provide a more suitable control structure. However, a large global soup of pattern-directed transformations tends to explode the search space because too much computation is required to discover which transformations to fire. Since the transformation of the looping information is naturally organized as case-based logic dependent almost exclusively on the semantics of the operators and operands, a small set of candidate transformations is immediately obvious from the operator and operand types in the AST. Thus, the most natural organization for the transformations is to attach them to the domain specific operator and operand classes in the inheritance hierarchy (and group them by phase). This significantly reduces the search space. Operationally, the loop placement control simply walks over the expression tree and executes any phase-specific transformations attached to the operator and operand definitions. The pattern-directed phases of AOG are described in detail elsewhere [4].

Summary

In summary, the new control structure is designed to reduce the generator's search space as much as possible while still allowing deep factorization of reusable componentry and high degrees of optimization of the target code. There are two major phases (i.e., pattern-directed refinement and tag-directed optimization) and the control constructs of each are optimized for the goals of the underlying phases. Each major phase is subdivided into minor phases, each of which has a narrowly specific generation goal.

The pattern-directed control structures are designed to optimize the generation, placement, and merging of looping constructs. Because the operator and operand types naturally separate the relevant transformations into small groups, the inheritance hierarchy is used as an index to these small groups. The transformations are further grouped by minor phase, which further diminishes the number of choices at any point. By these techniques, the generator's search space is significantly reduced because there are usually only a handful of transformations to check at any point in the AST. This organization thwarts most of the search space explosion induced by a generator architecture that relies on global sets of transforms using pattern-directed triggering.

The minor phases behave like an abstract algorithm (i.e., a strategy) with the pattern-directed transformations supplying the tactical details. This separation of the strategic from the tactical makes the addition of new operator and operand types quite easy.

Optimization is a process of reorganizing the code to accommodate the architectural properties of the CPU. Such code movement requires a different control structure – a tag-directed control structure -- to minimize its search space. Like the pattern-directed control structure used to generate and optimize loops, the tag-directed control structure also separates the strategic from the tactical. The strategic elements are defined by a set of minor phases with very specific computational goals. However, the tag-directed mechanism differs from the pattern-directed in its interrupt-like triggering mechanism and in its organization. The search space reduction with tag-directed transformations is wrought by

- Locating the tag at the point of application in the reusable components,
- Naming the specific transformation so that no search is required to determine which transformation should be applied at a given point in time,
- Using as much *a priori* knowledge (e.g., domain specific knowledge) as possible to further narrow the computational purpose and therefore, the search space, and
- Using an arbitrarily rich model of local and global events to trigger the transformations named in the tags.

Fundamentally, the tag-directed process is a scheduler that keeps a queue of events that have triggered tags. It simply runs until the event queue is empty. Each minor phase is initiated by the posting of a global event naming the phase. When there are no more phases and the queue is empty, tag-directed processing terminates. This control structure relies on the transformations

1. To post local events (e.g., substitution, migration, simplification),
2. To handle their own enabling condition checking and to directly call other transformations to fix up enabling conditions that do not quite meet their specifications, and
3. To call the partial evaluator for each new subtree that is constructed in the AST.

Contributions

AOG makes several contributions.

- It uses **tag-directed transformations** to exploit knowledge and operations that are ill suited to pattern-directed transformations.

- It **stages transformation phases** so that each is organized to achieve a narrowly defined translation or optimization purpose.
- It provides **event-driven tags** that allow for opportunistic optimizations as well as for interdependent, anticipated optimizations that can be organized into stages to assure consistency.
- It **reasons over the domain specific operators and operands** early in the translation process to produce tags that can thereby take advantage of that domain specific knowledge later in the optimization process when, conventionally, all of the domain specific knowledge would have been translated away.
- It **localizes pattern-directed transforms and component definitions** to specific abstractions within an inheritance hierarchy thereby reducing the opportunity for them to explode the search space by being applied in inappropriate situations.
- It uses a **schema language** to insulate the transformations from the physical details and variations in the AST.

Related Research

This work bears the strongest relation to Neighbors work. [10] The main differences are 1) the fact that the AOG pattern-directed transformations are organized into an inheritance hierarchy which guides the choice of which transformations to try, and 2) the use of the tag-directed approach for program optimization. Neighbors uses pattern-directed transformations during his optimization.

The work bears a strong relationship to Kiczales' Aspect Oriented programming at least in terms of its objectives but the optimization machinery appears to be quite different. [9] Kiczales' optimization mechanism seems not to be distributed over the AST and the optimization algorithms do not appear to be manipulated by the transformations. In contrast, the AOG's tags are distributed over the program and they undergo transformations as the generator reasons about the domain, the program, and the optimization tags.

This work is largely orthogonal but complementary to the work of Batory. [1] Batory optimizes his type equations to choose the optimum components from which to assemble classes and methods. AOG inlines and interweaves the bodies of methods invoked by compositions of method calls (i.e., expressions). Thus, Batory's generation focus is at the class level and AOG's is at the instance level. For details of the relationship see [6].

AOG and Doug Smith's work are similar in that they make heavy use of domain specific information in the course of generation. [11] They differ in the machinery used. Smith's work relies more heavily on inference machinery than does AOG. The reasoning that AOG does is narrowly purposeful and is a somewhat rare event (e.g., the transformation that splits the loop in the MMX example above does highly specialized reasoning about loop limits). However, partial evaluation (a form of inference) is heavily used in AOG, which is how three level if-then-else expressions (which are interweavings of the definitions of **`W, Row, Col`** and **`ConvolutionOp`**) get reduced to expressions like "**`a[im1,j]*(-2)`**".

The organization of the transformations into goal driven stages is similar to Boyle's TAMPR [7]. However, Boyle does not use tags.

The schema language is most similar to the work of Wile [14, 15] and Crew [8]. Popart leans more toward an architecture driven by compiling and parsing notions. As such, it is influenced less by logic programming. On the other hand, ASTLOG is more similar to the AOG schema language in that it is heavily influenced by logic programming. However, ASTLOG's architecture is driven by program analysis objectives and is not really designed for dynamic change and manipulation of the AST. It assumes that its target is a set of link files produced by a compile and link operation, thereby producing a batch-oriented model of AST manipulation. Such a model is not well suited to dynamic manipulation and change of the AST under the control of a transformation-based generator.

There are a variety of other connections that are beyond the space limitations of the paper. For example, there are relations to metaprogramming [14], logic programming based generation, formal synthesis systems (e.g., Specware) [12], deforestation [13], transformation replay [2] and other procedural transformation systems (e.g., Refine). The differences are greater or lesser across this group and broad generalization is hard. However, the most obvious general differences between AOG and most of these systems is AOG's use of transformations that operate in the optimization problem domain and are triggered based on optimization-specific events. Additionally, the AOG control structure is unusual. The overall optimization process behaves like an abstract algorithm where the algorithmic steps are stages and where the details of the steps (i.e., what operations are performed, what part of the AST they affect, and when they get called) are determined by tags on the AST itself. In addition, the event driven transforms behave like interrupts that allow for operations whose invocation details cannot be planned in advance and whose effect is largely simplification.

Conclusions

AOG is being developed to study of the effects of architectural variations of generators on programming leverage, variability, performance, and search space size. While still early, it has demonstrated that some operators and types can be deeply factored to allow highly varied re-compositions while simultaneously allowing the generation of high performance code without huge search spaces.

References

1. Batory, Don, Singhal, Vivek, Sirkin, Marty, and Thomas, Jeff: Scalable Software Libraries. Symposium on the Foundations of Software Engineering. Los Angeles, California (1993)
2. Baxter, I. D.: Design Maintenance Systems. Communications of the ACM, Vol. 55, No. 4, (1992) 73-89
3. Biggerstaff, Ted J.: Fixing Some Transformation Problems. Automated Software Engineering Conference, Cocoa Beach, Florida (1999)

4. Biggerstaff, Ted J.: Anticipatory Optimization in Domain Specific Translation. International Conference on Software Reuse, Victoria, B. C., Canada (1998) 124-133
5. Biggerstaff, Ted J.: Composite Folding in Anticipatory Optimization. Microsoft Research Technical Report MSR-TR-98-22 (1998)
6. Biggerstaff, Ted J.: Pattern Matching for Program Generation: A User Manual. Microsoft Research Technical Report MSR-TR-98-55 (1998)
7. Boyle, James M.: Abstract Programming and Program Transformation—An Approach to Reusing Programs. In: Biggerstaff, Ted and Perlis, Alan (eds.): Software Reusability, Addison-Wesley/ACM Press (1989) 361-413
8. Crew, R. F.: ASTLOG: A Language for Examining Abstract Syntax Trees. Proceedings of the USENIX Conference on Domain-Specific Languages, Santa Barbara, California (1997)
9. Kiczales, Gregor, Lamping, John, Mendhekar, Anurag, Maede, Chris, Lopes, Cristina, Loingtier, Jean-Marc and Irwin, John: Aspect Oriented Programming. Tech. Report SPL97-08 P9710042, Xerox PARC (1997)
10. Neighbors, James M.: Draco: A Method for Engineering Reusable Software Systems. In: Biggerstaff, Ted and Perlis, Alan (eds.): Software Reusability, Addison-Wesley/ACM Press (1989) 295-319
11. Smith, Douglas R.: KIDS-A Knowledge-Based Software Development System. In: Lowry, M. & McCartney, R., (eds.): Automating Software Design, AAAI/MIT Press (1991) 483-514
12. Srinivas, Y. V.: Refinement of Parameterized Algebraic Specifications. In: Bird, R. and Meertens, L. (eds.): Proceedings of a Workshop on Algorithmic Languages and Calculii. Alsac FR. Chapman and Hill. (1997) 164-186.
13. Wadler, Philip: Deforestation: Transforming Programs to Eliminate Trees. Journal of Theoretical Computer Science, Vol. 73 (1990) 231-248
14. Wile, David S.: Popart: Producer of Parsers and Related Tools. USC/Information Sciences Institute Technical Report, Marina del Rey, California (1994) (http://www.isi.edu/software-sciences/wile/Popart/ popart.html)
15. Wile, David S.: Toward a Calculus for Abstract Syntax Trees. In: Bird, R. and Meertens, L. (eds.): Proceedings of a Workshop on Algorithmic Languages and Calculii. Alsac FR. Chapman and Hill (1997) 324-352