Fixing Some Transformation Problems[1]

Ted J. Biggerstaff
Email: Ted_Biggerstaff@msn.com

ABSTRACT

Defining domain specific abstractions for generator systems leads to a quandary between choosing abstractions that exhibit powerful programming amplification through the combinatorial opportunities provided by composition, and choosing abstractions that can be easily transformed into high performance code. Most generators opt for abstraction to improve programming productivity, which usually compromises target program performance. Transformation-based generators widen the quandary through deep factorization of operators and operands to amplify expressive power, but this explodes the search space. My hypothesis is that existing architectures are inadequate to achieve simultaneously high levels of abstraction, high performance target programs and small solution search spaces. To explore architectural variations to address this quandary, I have implemented a generator in Common LISP designed specifically to address these problems. It is called the Anticipatory Optimization Generator (AOG) because it allows programmers to ***anticipate*** *optimization opportunities and to prepare an abstract, distributed plan that attempts to achieve them.*

Keywords

Domain specific, program generation, transformations, logic programming, pattern-directed, tag-directed

1 PROBLEMS

Program generators compile high level, compact, (and usually domain specific) languages into conventional programming languages like C or Java thereby improving programming productivity, code safety, ease of understanding, and so forth. The same properties that make such domain specific language (DSL) representations appealing make compiling them into high performance code difficult. The performance problems arise because DSLs tend to delocalize performance-related elements of the target code and introduce high levels of redundancy. The various approaches to re-localize (i.e., reorganize) the code and remove redundancy to achieve high degrees of optimization (e.g., approaches like conventional optimization, optimizing transformations, specialization, etc.) all have practical problems that call into question their feasibility. The fundamental problem is that dependencies between distant uses of operators and operands make re-localization and redundancy removal hard. When attempting to reorganize components for near optimum performance, the architectures of conventional transformation systems induce very large search spaces. Therefore, commercial transformation systems that must deal with large programs rely on forward refinement as their fundamental mode of operation and forego significant program reorganization and optimization. For example, CAPE, a generator built on Draco [8] and specialized to communications problems, exploits few or no optimization transformations and relies on the problem domain to allow solutions that are acceptable with limited or no optimization. In fact, Neighbors asserts that often he either ignores optimizations completely and depends on default forward refinement transformations or he uses the programmer as an oracle to guide the generator through the myriad choices of optimizing transforms.

The problems of conventional transformation systems are inherent to their architectures. In this paper, we will explore the architectural characteristics that lead to these large search spaces and explore how their architecture can be changed to overcome these problems.

2 PROBLEMS INDUCED BY ARCHITECTURE

Scale-Variation-Performance-Search Space Dilemma: The fundamental problem is that transformation-based generators are trying to achieve four mutually antagonistic goals. They want to use big building blocks or equivalently high degrees of composition (i.e., scale) to achieve high programming leverage. They also want to provide high degrees of variation in the target program so that the resulting programs will be widely applicable. Finally, they want to produce programs with sufficiently good performance to be practical in the real world without engendering huge solution search spaces. These goals are mutually antagonistic.

Using big components impedes the variation goal because the combinatorial explosion of pre-coordinated design decisions for big components explodes the number of variants needed. Using small components and composition both nullifies the programming leverage achieved and explodes the search space induced by composing and organizing them for acceptable performance. Performance is a global property of a program that relies on coordinated design of the componentry from which the program is assembled. Such

[1] Much of this work was done at Microsoft Research and the author would like to acknowledge the support of Microsoft.

coordination limits the degree of variation that is possible.

As a practical matter, compromises of one or more of these goals are the usual approach with the consequences being that the results are often not satisfactory to the user.

Fundamentally, this dilemma arises from the goal to build universal canonical components that can be used everywhere and the fact that universal canonical components are often ill-tuned to work with other universal components. Tuning such components so that they work well together (e.g., have acceptable performance) is a process of revising the structure of each component to impose inter-component coordination such that certain desired global properties are achieved. Such tuning is hard and often induces impracticably large search spaces. In the example we will examine, this tuning process would involve a chain of 92 carefully ordered transformations where at each point in the chain, many other legal transformations are possible. But if different transformations are applied, the process is likely to dead-end in a local minimum and never achieve the full performance potential. It is a bit like the requirement to have 92 carefully ordered miracles. Such difficulties are inherent to the architecture of conventional transformation systems. What architectural aspects cause these problems?

Syntactic, pattern-directed bias: Conventional transformation systems suffer from a syndrome that is suggested by the aphorism: "To a hammer, all problems look like a nail." Much of the technology underlying conventional transformation systems evolved out of compiler and programming language research. Compiler theory has developed mature and powerful tools for dealing with syntax and that legacy has influenced the approaches to transformation systems. That is to say, most transformation specifications are very good at dealing with program syntax and structure (including the structure of abstract syntax of ASTs or Abstract Syntax Trees) and are far less good at dealing with non-structural information, e.g., such as the non-structural, large grain purposes of the transformations. In fact, the commonly used term "pattern-directed transformation" suggests this predisposition. 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, cooperating transformations is designed for the narrow purpose of creating and placing loops in the target program.

A second aspect of the syntactic orientation is the desire to force all transformation specification into a declarative rule oriented form. High variation in the ASTs induces a combinatorial explosion in the number of rules needed to account for the variation. Even adding a small amount of programming capability to the rule forms (e.g., Prolog like backtracking and arbitrary embedded computations) reduces this explosion to a degree. Other facilities (e.g., tag-driven transformations) can be added to further reduce the explosion by dealing with the cross rule dependencies.

The syntactic bias induces the huge search spaces because it is a little like trying to find one's way through a maze with only a small knot hole view of the local area. Unless the space has some global property that one can exploit, finding the global maximum of such a space requires exhaustive search.

Organization for architectural purpose: A second aspect of this problem is that sets of transformations are seldom organized into groups of related, cooperating transformations aimed at achieving a specific global architectural purpose or result[2]. Ideally, an architectural purpose, either expressed at the outset of overall generation or associated with a single generation phase, should have global effects. It should enable sets of related, cooperating transformations that are designed as a group to achieve that purpose and disable groups that are designed to achieve different architectural purposes. For example, the broad organizational strategies employed to exploit a machine with SIMD (Single Instruction, Multiple Data stream) instructions (e.g., the Intel Pentium™ with MMX instructions) will differ markedly from those employed for a machine without such instructions (e.g., a vanilla Pentium™ without MMX instructions).

In the first case, the global strategy employed by the transformation set will be to derive constant data values (e.g., weights of for a convolution operator) and organize them into arrays. Arrays (used by loops) enable optimal use of the SIMD parallelism. Likewise, a second strategy will be to organize the formulas that operate on those data values into loops that have bodies without internal branching operations. The loops enable the parallelism but intra-loop branching may disrupt or slow the streaming of the constant data and thereby thwart the parallelism. In the example that we will examine later in this paper, such a strategy will require splitting a single loop with branching logic in its body into several distinct loops where the branching conditions are incorporated into the loop control logic. Finally, the bodies of the loops will need to be restructured into expressions built out of operators that are structurally similar to the parallel instructions (e.g., an

[2] Interpreting this most broadly, CAPE is an exception to this in that it allows the loading of different sets of transformations at the outset that are very purpose directed. One set is designed to produce code, another to produce a simulation, another to produce a model for use with model checking software, another to produce various kinds of graphic documentation, and so forth.

operator that adds four pairs of operands in parallel).

In the second case (i.e., the PentiumTM without MMX instructions), the global strategy will be aimed eliminating inner loops to allow simplification of complex branching. For example, the branching tests within small inner loops of convolution operators that are known *a priori* not to depend on the indexes of the small inner loops may be moved outside the inner loops. This reduces the number of times the branching tests are computed. It also opens up the opportunity to eliminate residual branching tests (still within the small inner loops) that only depend on the loop indexes. Unrolling the inner loops allows these residual branch tests to be calculated at generation time thereby eliminating the branching.

Anticipation of future optimization opportunities: This latter example raises the issue of recording properties of the componentry that may lead to reorganization opportunities once the componentry is assembled into a program. Conventional transformation systems allow no easy way to express and then later exploit such information. For example, the writer of a method describing the formula for computing the weights associated with a convolution neighborhood knows *a priori* that if the neighborhood is hanging even one pixel off the edge of the image, all weights will be zero. He also knows that the test to see if the neighborhood is hanging off the edge of the image does not depend on the indexes of any (anticipated) loop that will traverse the neighborhood since these indexes will just be offsets from some central pixel of the neighborhood. The hanging test depends only the position of the neighborhood centering pixel with respect to the coordinates of the overall image. Therefore, whenever the formula is substituted into some loop iterating over the neighborhood, the neighborhood loop can be distributed over the then and else branches of the edge hanging test thereby effectively moving the branching test outside of the neighborhood loop. Most transformation systems provide no easy way to tell future optimization transformations about such **anticipated** opportunities.

Similary, most conventional optimization systems provide no mechanism to record knowledge of future optimization opportunities that become known in the course of executing early transformations. For example, in the examples to be examined later in the paper, an early transformation discovers the opportunity to perform a *reduction in strength* optimization by turning a square operation into a multiply operation. Unfortunately, the operand of the square is an expression, so to prevent multiple computations of the expression, it must be moved out of line and stored in some temporary variable such as "**t1**". Thus, the square operation can now be represented by the more efficient form "**t1 * t1**". However, there is the distinct possibility that future reorganizations of the program might reduce the expression assigned to **t1** to a constant value thereby opening up the opportunity for reducing the multiplication to a data constant by two coordinated transformations: 1) a code movement optimization that enables 2) a subsequent *constant folding* optimization. We would like any simplification of **t1**'s value expression to trigger an attempt at such an optimization. The way AOG handles this is by adding *tags* to the code that:

1) explicitly describe the newly introduced data flow dependencies to reduce the computational effort of the anticipated code movement and constant folding transformations,

2) describe the *optimization event* that should trigger the anticipated transformations (e.g., the simplification of **t1**'s assigned value to a constant), and

3) provide the name of the transformation (i.e., **_CheckFoldFlows**) that should be attempted if and when the optimization event occurs.

The reason that a generic constant folding pass over the program will not find this optimization opportunity and thereby trigger a cascade of succeeding simplifications is that the optimization depends on a series of loosely cooperating transformations[3]. The early transformations create the enabling conditions for the later transformations. This kind of coordinated application is not easy to accomplish in a pure pattern directed transformation system without significant computational costs.

Opportunistic application of anticipated optimizations: Even if conventional transformation systems were to represent anticipated optimizations, one can never really be certain about when to apply them. The target program will undergo significant reorganization between the anticipation event and the opportune time to apply the optimization transformations. In fact, the opportunity is likely to depend less on the local structure of the program and more on the events of the reorganization process itself. For example, the code movement and constant folding transformation discussed earlier is unlikely to succeed unless one of the expressions has been simplified to a data constant. Thus, in AOG, triggering this tag-driven transformation is conditioned upon this

[3] In the example we will examine, transformations occurring after the reduction in strength optimization but before the opportunity to apply the tag-driven simplification will have moved and duplicated the source of the data flows associated with **t1** so that they emanate from both the then and else clauses of a preceding if statement. To enable constant folding, the tag-driven transformation will have to copy the statement containing the "**t1 * t1**" expression into both branches of the if. Only on the then branch will constant folding be successfully enabled.

optimization event. We call such transformations *event-driven* transformations.

Other event-driven transformations (e.g., those that perform code movement in the presence of loops) may be organized into stages, each of which achieves a well-defined organizational purpose and prepares the target program for the next stage. Thus, each stage expects a certain set of abstract, global program properties to hold before it performs its operations. For example, the transformations that manipulate loops (e.g., loop unrolling) expect that all transformations that perform code movement in the presence of loops have been completed. Thus, staging is a way to impose an abstract optimization algorithm on the program where the stages are the algorithm's steps 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. The steps are anticipated and recorded by the programmer or by earlier transformations. In addition, the event driven transforms behave like interrupts that allow for operations whose invocation details cannot be planned in advance and whose affect is largely simplification. Stages are just a series of optimization events organized on a strict timeline.

In contrast to AOG, conventional transformation systems are not organized to anticipate optimization opportunities, or to defer their actual application, or to condition that application on events specific to the optimization process itself. This leads to their exploring optimization opportunities at the wrong time because there is no global sense of purpose to filter out ill-timed or fruitless explorations.

In summary, the search space induced by conventional program generation strategies is often too large to allow search for optimal versions (or even near-optimal) of generated programs. Conventional transformation systems lack the tricks that mimic a programmer's strategy to find near optimal solutions, tricks such as:

1) purpose-based organization that relates groups of cooperating transformations,
2) the use of global architectural guidance in choosing which sets of transformations to enable,
3) the anticipation and recording of future optimization opportunities,
4) the deferred application of optimizations, and
5) the ability to inter-mix transformations aimed at global architectural purposes with opportunistic transformations that dynamically exploit optimization opportunities.

This paper will describe a system designed specifically to exploit such tricks to reduce the search space and thereby produce near optimal programs.

3 THE ANTICIPATORY OPTIMIZATION SYSTEM

The most distinct architectural feature of the AOG generator is the ability to deal with tags attached to any element of the Abstract Syntax Tree (AST) of the target program and use those tags to trigger late stage optimizing transformations. The tags can be thought of as an abstract, distributed optimization plan. They capture knowledge about the program and optimization process that is not easily derived or represented using pattern-directed transformations.

The generator both manipulates the tags (i.e., reasons about the tags, the domain information, and the program constructs) and eventually executes the optimizing transformations denoted by tags (e.g., the **_PROMOTECONDITIONABOVELOOP** tag triggers a transformation that tries to move a condition test outside of a loop). Furthermore, some tags may depend on optimization events (e.g., substitution of a tagged expression) for their triggering condition, which may, in turn, trigger a whole cascade of other opportunistic transformations. Through this tag-driven optimization process, delocalized code gets relocalized and redundant code gets shared.

In short, the approach is to plan optimization strategies abstractly in the problem domain, record the plan via tags added to program components, and execute the plan in the low level programming domain using tag-driven transformations.

4 APPROACH

4.1 Example

An example of a domain specific programming expression is illustrated by the expression for Sobel edge detection in bitmapped images.

Dsdeclare image a, b :form (array m n) :of bwpixel;

$b = [(a \oplus s)^2 + (a \oplus sp)^2]^{1/2}$;

where **a** and **b** are **(m X n)** grayscale images and $\oplus$ 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** and **sp** are OO abstractions that define the specifics of the pixel neighborhoods (i.e., their sizes and shapes, the weights associated with each position in the neighborhood, and any special processing such as the special case test for **s** or **sp** hanging off the edge of the image). The convolution operator iterates over the image **a** performing a sum of the products of each of the neighborhood's weights times each of the pixels in the neighborhood of **a[i,j]**. For this example, the neighborhood weights of **s** and **sp** are defined as

s [(-1:1), (-1:1)] = {{-1, 0, 1}, {-2, 0 , 2}, {-1, 0, 1}} and
sp [(-1:1), (-1:1)] = {{-1, -2, -1}, {0, 0, 0}, {1, 2, 1}}

In the case of any neighborhoods that are partly off the edge of the image, the resultant pixel is defined to be 0.

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) /* Off edge */
              then b[i, j] = 0;
              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. This is not particularly fast at the moment but redesigning the implementation data structures should reduce generation time to somewhere near the range of optimizing compiler times. In any case, the AO generator can write the code a good deal faster than I can.

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, $padd_2$, and $pmadd_3$ correspond to MMX instructions and are defined as $pmadd_3$ ((a0, a1, a2) , (c0, c1, c2)) = (a0*c0+a1*c1, a2*c2 +0*0), $padd_2$ ((x0, x1) , (x2, x3)) = (x0+x2, x1+x3), $pmadd_3$ ((a0, a1, a2) , (c0, c1, c2)) = (a0*c0+a1*c1, a2*c2 +0*0), 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. Four 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.

4.2 Operation

How does the AO generator accomplish this? The generator is a multi-phase, transformation system. The early phases use *pattern-directed transformations* (i.e., transformations that trigger based on AST patterns) to translate the high level operands into lower level conventional programming constructs. For example, these transformations will refine images into pixels, pixels into channels, and channels into integers. In the course of this, they reason over the domain specific information to create, place, move, and fuse the looping constructs implied by the various operators. These loop creation transformations encode loop information as tags that are attached to AST expressions, moved over the AST and merged based on the semantics of the AST expressions involved. [2] These early stage transformations may also perform opportunistic optimizations. For example, in the convolution example above, a pattern-directed transformation recognized that the expressions $(a \oplus s)^2$ and $(a \oplus sp)^2$ allow a reduction in strength optimization to **(t1 * t1)** and **(t2 * t2)** , if the temporary assignments **t1 =(a ⊕ s)** and **t2 =(a ⊕ sp)** are created and moved out of line.

The later phases use *tag-directed transformations* to reorganize the program for high performance. The tags trigger transformations that incorporate operator definitions (e.g., the convolution operator and calls to the methods of **s** and **sp**), reorganize the resulting forms (e.g., move a loop into an if-then-else), and simplify the resulting code. These reorganizations may cause optimization events (e.g., a substitution event) that further trigger event-based transformations, i.e., via the mechanism of tags with explicit triggering conditions. These may cascade to completely reorganize the program. For example, substitution of the convolution operator definitions in the earlier example starts a cascade of transformations. It moves the neighborhood loop into the then and else legs of the if-then-else expression that computes the weights, moves the multiplication of the **a[i,j]** pixel into the then and else legs (recursively), and triggers constant folding that reduces the loop in the then leg to zero. [3]

Tags may be added at any time. Some are pre-positioned on reusable library components (e.g., on the definition of the convolution operator) in anticipation of potential optimizations. For example, the programmer of the reusable library might add the tag

```
(_On SubstitutionOfMe (_PromoteConditionAboveLoop ?piter ?qiter))
```

to the special case test within the definition of **W** of **s** and **sp** because he knows that the conditional expression of the if statement will not depend on the indexes of the inner most convolution loops. This tag is conditioned upon the optimization event "**SubstitutionOfMe**" which means that the transformation **_PromoteConditionAboveLoop** will only get triggered when the if statement to which the tag is attached gets substituted in the AST.

Tags are also added and deleted by other transformations in the course of generation. Transformations may discover knowledge fleetingly in the midst of their computation that suggests a candidate optimization. For example, the reduction in strength transformation describe above anticipates the likelihood that the data flows it introduces can be re-manipulated to enable constant folding when certain optimization events occur. Such knowledge is captured by adding a tag such as:

(_On MigrationOfMe (_CheckFoldFlows))

which will cause the **_CheckFoldFlows** transformation to be triggered if the subtree to which the tag is attached gets moved[4].

Tags capture knowledge that is not easily derivable from the AST patterns or from operator or operand semantics. These are usually properties that 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.

4.3 A Tag-Directed Transformation Scenario

This section will overview the tag-driven reorganization that produces the non-MMX version of the example. The pattern-directed phases of AOG perform loop introduction, placement and fusion. In the course of that, they may also perform some opportunistic optimizations (e.g., reduction in strength of the square operator). This phase of AOG is described elsewhere [2] and we will not consider it further here. At the end of the pattern-directed phases, 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)  }
```

This expression gets transformed as follows:

The start of the tag-directed phases triggers inlining of the definition of ⊕ in the expression **(a[i,j] ⊕ s)**. The definition contains calls to the methods of **s** and **sp** defining the neighborhood size (**prange**, and **qrange**), the relative pixel locations (**row , col**), and the pixel specific weights (**w**).These method definitions of **s** and **sp** are recursively inlined.

The substitution of the **w** method triggers the tag **(_PromoteConditionAboveLoop p q)** where **p** and **q** are the indexes of the neighborhood loop. **_PromoteConditionAboveLoop** fails to meet its enabling condition because it is inside of an expression **(a[i,j] * ...)** and so calls **_IncorporateContext** to try to fix the problem, which distributes the expression over the then and else clauses. After that, the enabling conditions are met and the neighborhood loop too is distributed over the then and else branches. When the loop formed on the then branch is partially evaluated, it collapses to a constant **0**. This completes the work of **_PromoteConditionAboveLoop.**

The else branch had the tag **(_on substitutionofme (_IncoporateContext))** which is triggered causing the expression **(a[i,j] * <else leg>)** to be rewritten with the **(a[i,j] * ...)** moved inside the else branch. **_IncorporateContext** is recursively applied for each embedded if-then-else statement. Partial evaluation of resulting expressions reduces results to their simplest forms.

The expression **(a[i,j] ⊕ sp)** undergoes an analogous set of transformations. At this point, each of the two temporary variable assignment statements look like:

```
(t2 = (if ((i == 0) || (j == 0) || (i == (m - 1)) || (j == (n - 1)))
       then 0
       else (_sum (p q)
                  (_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 highest level branching test is there to detect the case of the neighborhood hanging partially off the edge of the image. At this point, there are no more transformations to be triggered, so the system moves to the next stage, which is designed to try to share code across the subexpressions of the original expression. It does this by posting an artificial event naming the stage.

A tag attached to the outer if-then that is conditioned on this event has been waiting. It will call **_MergeCommonCondition** whose objective is to eliminate redundant tests by performing the rewrite:

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

The transformation is unable to meet its enabling conditions and recognizing why, calls **_IncoporateContext** to distribute the assignments over the relevant then and else branches. At this point, the conditions of the ifs can be merged.

Recall from our early discussion, a reduction in strength optimization created **t1** and **t2**, and moved the

[4] A new event is planned (i.e., **SimplificationToConstant**) to reduce the number of times **_CheckFoldFlows** is triggered and fails before it finally succeeds.

expressions out of line. In the course of that rewrite, it added tags to the **t1** and **t2** assignments that trigger an optimization (i.e., **_CheckFoldFlows**) when the right hand side of the assignment becomes a constant. If all of the data flow and dependency enabling conditions are met, the transformation removes the expression **b[i,j] = sqrt(t1*t1 + t2*t2)** from its current position and copies it into both legs of the if-then, from whence the data flows now originate. With constant folding, the then branch is transformed from **{t1=0; t2=0; b[i,j] = sqrt(t1*t1 + t2*t2)}** to **b[i,j] = 0.** The else branch does not qualify for constant folding, so **_CheckFoldFlows** performs no further manipulation of it.

No other opportunistic transformations are triggered, so the system posts the event signaling the final stage. Its job is to achieve in-place simplifications under the assumption that the major code movement stages have completed their work. For the non-MMX architecture, the **_UnWrapIfSmallConstant** transformation will be triggered twice which with partial evaluation will reduce the temporary variable assignments to forms like:

```
t2 = ( (a[(i + 1) , (j + 1)]  + (a[(i + 1) , j]  * 2) +
       a[(i + 1) , (j -1)]  - a[(i -1) , (j + 1)] -
       (a[(i -1) , j] * 2)   - a[(i  -1) , (j -1)] ) )
```

The final two remaining tags will trigger the transformations **_PromoteAboveLoop** and **_PromoteToTopOfLoop** which will hoist the code for the common indexing expressions to temporary assignments and replace them with the temporary variables **im1, ip1, jm1,** and **jp1.**

5 THE SCHEMA LANGUAGE

5.1 Purpose and Essence of the language

AOG is built in terms of highly purposeful, large-grain programmatic transformations, e.g., transformations that create, place, and fuse loops. This design induces a large degree of variation in the AST patterns that must be dealt with. For example, the **_MergeCommonCondition** rule discussed earlier must account for variations including: two orderings for the two if statements, optional else clauses, optional tag structures on the overall if as well as its subexpressions, and optional forms for leaf structures (e.g., "**0**" or **"(leaf 0 (tags ....))**"). To insulate the transformations from this variability and thereby reduce their complexity, AOG uses a unification-based, Prolog-like schema language in which to specify the AST patterns. It thereby allows the parameterization of the transformations by *logical or conceptual schemas*. These schemas insulate the transformations from the details and variations in the AST in much the same way that logical and conceptual schemas insulate database management systems from the physical details and format variations of databases. As a result, the transformations are performing their operations on *conceptual patterns* within the AST thereby reducing the case logic within the transformations.

The schema language [4] is a fully backtracking Prolog-like language that performs pattern-matching, parameter binding, enabling condition checking, and logical inference for the transformations. The schemas are first class objects that are created dynamically, stored within the AOG's data structures, and executed by the transformations. The schema language provides the major Prolog-like operators (e.g., and, or, not, mark, cut, bind, is, succeed, fail, prove, and rule definition). It also provides operators tailored to the job of generation including operators that span sections of an AST, bind the remainder of a list, search top-down or bottom-up, rename local variables, use dynamically computed variable values as patterns, invoke and use rule sets for inference, navigate object structures, perform recursive matching, and execute arbitrary Lisp expressions. It is fully backtracking thereby allowing a schema execution to fail, backup to the last choice point and restart with the next choice. This allows schema executions to explore all possible bindings in their search for a desired match. Schemas are always executed in the context of a current position in the AST. Any structures, literal data, variables, or matching operations in the schema must match the structures, literal data, CLOS objects, and atoms at the current point of the AST subtree.

A schema is fundamentally a pattern of literal data with pattern matching and logical operations embedded. The style of schema expression is *inverse quoting* which means that literal data is expressed directly whereas variables and operations are expressed with some syntactic sugar to distinguish them from the literal data. Specifically, schema variables[5] are symbols preceded by a question mark, e.g., **?x**. Operator expressions are preceded by a dollar sign. For example, the **or** operator expression **$(por ?x ?y)** requires that the value of the variable **?x** or the value of **?y** must match the item at the current position in the tree. As a further example, the matching operation **$(ptest numberp)** requires that the item matched is a Lisp number. Matching is accomplished by unification of schema variables with literal data or variables in the AST. This means that if a variable is already bound to a value at some earlier point in the matching process, then the current item in the AST must be that same value or a variable that is bound to that value. If a variable is unbound at the time it is matched, unification will cause it to become bound to the current item in the AST.

5.2 Application of schemas

The schema engine is used in a variety of ways within the generator. For example, many of generator computations exhibit a stereotypical pattern. They fetch the schema value from some slot in a CLOS object

[5] We must distinguish these *pattern or schema variables* (e.g., **?x**) from the variables in the target program (e.g., **x**). Pattern variables exist only at generation time and their values are often target program variables or expressions that will become part of the target program.

representing an operator or operand (e.g., the **backwardconvolution** operator of Figure 1), execute that schema, and then use the resultant bindings to direct the subsequent computation. For example, when selecting a particular pattern-directed transformation to run, they execute the schema in the slot representing the current translation stage (e.g., the **fusion2** or **fusion3** slots in Figure 1)[6]. If successful, the resultant bindings name the transformation to be run (e.g., the **backwardconvolutiononleaves** or **reduceconvolutionoperator** slots in Figure 1). Similarly, when selecting refinements for operator or method expressions (e.g., calls to the **W**, **Row**, or **Col** methods of **s** or **sp**), the slot of **s** or **sp** containing the replacement for the operator or method expression is determined by executing a schema in the **formals** slot of **s** or **sp**. In addition to providing the slot name containing the refinement, the schema execution also determines what bindings should be applied to that refinement before it is substituted

Schemas are also used in a variety of other ways. For type inference, a schema in the operator's definition (in the **infertype** slot) is run and the result is the inferred type bound to the pattern variable **?itype**. Additionally, schemas are used 1) to search over the expression AST to find tags that will trigger transformations, 2) to do some lightweight reasoning about the programs, and 3) to recognize expressions to be simplified by the partial evaluator.

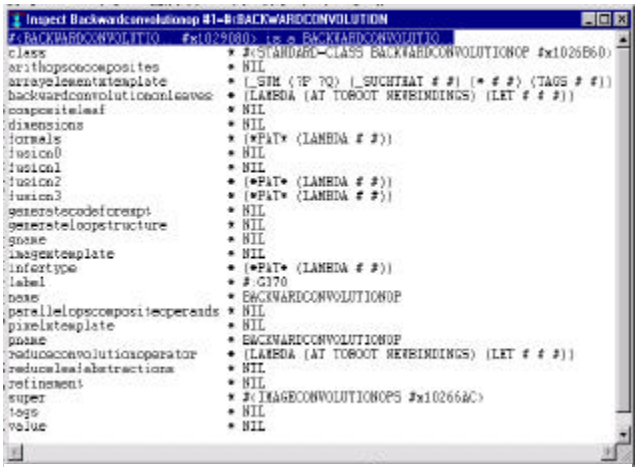

Figure 1 - Definition of the ⊕ Operator

5.3 The Reusable Components

The code for some operators must be generated dynamically (e.g., most loops) but some can be statically defined *a priori* when their arguments have been refined to implementation levels (e.g., the ⊕ operator applied to implementation level pixels). For example, the ⊕ operator applied to an integer array that is the implementation form of an image and a template type that is the implementation form of a neighborhood description is defined by the **arrayelementXtemplate** method of the ⊕ operator:

```
(DefComponent arrayelementXtemplate
  (#.BackwardConvolutionOp #.ArrayReference
                           #.TemplateReference)
  (_Sum (?p ?q)
    (_SuchThat
      (_Member ?p
               (PRange ?template (aref ?aname ?iter1 ?iter2)
                       ?plow ?phigh ?p))
      (_Member ?q
               (QRange ?template (aref ?aname ?iter1 ?iter2)
                       ?qlow ?qhigh ?q)))
    (* (aref ?aname
           (row ?template (aref ?aname ?iter1 ?iter2) ?p ?q)
           (col ?template (aref ?aname ?iter1 ?iter2) ?p ?q))
       (w ?template (aref ?aname ?iter1 ?iter2) ?p ?q))
    (tags (_On CFWrapUpEnd (_UnWrapIfSmallConstant (?p)
                                                   (abs (- ?plow ?phigh))))
          (_On CFWrapUpEnd (_UnWrapIfSmallConstant (?q)
                                                   (abs (- ?qlow ?qhigh)))))))
```

where the parameters **BackwardConvolutionOp**, **ArrayReference**, and **TemplateReference** are schema patterns defined elsewhere (see [4]) that respectively match the CLOS definition of the ⊕ operator (i.e., the CLOS object of Figure 1), an integer array reference (e.g., **a [i, j]** which is expressed in the AST as **(aref a i j)**), and a template abstraction (e.g., **s**). This **arrayelementXtemplate** definition is the canonical form for the convolution's inner loop that computes the convolution value for the neighborhood around some pixel **?aname[?iter1, ?iter2]**. Notice that the tags in this definition anticipate the unwrapping of the loops controlled by **?p** and **?q** if the expression "**(abs (- ?plow ?phigh))**" is a small constant for some definition of small. The variables **?plow** and **?phigh** are the upper and lower range values of **?p** and will be bound by the **TemplateReference** pattern. The transformation **_UnWrapIfSmallConstant** will be triggered when the event **CFWrapUpEnd** is posted by the generator. Such events are used to sequence stages of the tag-driven process.

Defcomponent or-s a doctored version of the parameter list schema onto the schema value already in the **formals** slot of the **BackwardConvolutionOp** object[7] of Figure 1 and stores the body of the definition in the **arrayelementXtemplate** slot. Thus, the **formals** slot contains a pattern that when matched to an AST structure will determine which method (if any) is applicable and what AST values should be bound to the pattern variables such as **?aname, ?iter1, ?iter2**, etc.

The neighborhood **s** is defined in terms of the methods **PRange**, **QRange**, **Row**, **Col**, and **W** that respectively compute the ranges for the row and columns, the row and column

[6] Only the top few levels of the list structures are shown in Figure 1. The elided portions are indicated by the "**#**" symbol. Schema values are shown as **"(*pat* (lambda (...) ...)**" which is the compiled form of "**$(oper ...)**" expressions.

[7] **$(por** *ParmlistPattern* **(formals BackwardConvolutionOp))**

indexes and the weights associated with each pixel of the neighborhood. Example definitions are:

```
(Defcomponent Row (s #.ArrayReference ?piter ?qiter)
          (+ ?iter1 ?piter))

(Defcomponent W (s #.ArrayReference ?piter ?qiter)
  (if (or (== ?iter1  ?i1low) (== ?iter2  ?i2low)
          (== ?iter1 ?i1high) (== ?iter2 ?i2high))
      (then 0)
      (else (if (and (/= ?piter 0) (/= ?qiter 0))
                (then ?qiter)
                (else (if (and (== ?piter 0)   (/= ?qiter 0))
                          (then (* 2 ?qiter))
                          (else 0)))
                (tags (_On SubstitutionOfMe
                                  (_IncorporateContext)))))
      (tags (_On SubstitutionOfMe
                 (_PromoteConditionAboveLoop ?piter ?qiter))
            (_On CFWrapUp (_MergeCommonCondition)))))
```

Not all parameters are explicitly acquired from the argument list. The **ArrayReference** pattern matches **?aname[?iter1, ?iter2]** and then navigates via object relationship links to get the range values **[?i1low : ?i1high]** and **[?i2low : ?i2high]** defined for **?iter1** and **?iter2**. Analogous to the ⊕ operator, **s** is represented as a CLOS object. As with the definition of the ⊕ operator, Defcomponent or-s a doctored form of each parameter list onto the existing schema in the **formals** slot of **s** and stores the bodies in slots **Row, Col** and **W** of **s.**

Notice that **W**'s definition anticipates several kinds of optimizations to be tried whenever the tagged expressions get substituted into a different expression. For example, the substitution of **W**'s body will trigger the **_PromoteConditionAboveLoop** transformation.

These defcomponents are conceptually analogous to what the transformation community would call *forward refinements* and would represent in a rule form like

```
(Row  s #.ArrayReference ?piter ?qiter) => ( + ?iter1 ?piter)
```

The key difference is that such forward refinement rules are often global and more general in form (e.g., the pattern might contain a variable in place of **s**). This increases the opportunities for them to be applied at inappropriate points thereby exploding the search space. Defcomponent localizes such rules to specific abstractions (e.g., **s**) thereby allowing them to be considered only when their associated abstractions are being processed. This reduces the search space explosion.

6 CONTRIBUTIONS

AOG make several contributions. It uses **tag-driven transformations** to exploit knowledge and operations that are ill suited to pattern-directed transformations thereby allowing planning in the problem domain and optimization execution in the programming and optimization domains. 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 on a time line to assure their consistent application. 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.

7 RELATED RESEARCH

This work bears the strongest relation to Neighbors work. [8] 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. [7] 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 [4].

AOG and Doug Smith's work are similar in that they make heavy use of domain specific information in the course of generation. [9] 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 **BackwardConvolutionOp**) get reduced to expressions like "**a[im1, j] * (-2)**".

The organization of the transformations into goal driven stages is similar to Boyle's TAMPR [5]. However, Boyle does not use tags.

The schema language is most similar to the work of Wile [11, 12] and Crew [6]. 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 [12], logic programming based generation, formal synthesis systems (e.g., Specware) [10], deforestation 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.

8 CONCLUSIONS

AOG is being developed to study of the effects of architectural variations 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. It suggests that one day it may be possible to simultaneously achieve the mutually antagonistic goals of high programming leverage, high variability, adequate performance and small search spaces.

9 REFERENCES

1. Batory, Don, Singhal, Vivek, Sirkin, Marty, and Thomas, Jeff, "Scalable Software Libraries," Symposium on the Foundations of Software Engineering. Los Angeles, CA, December, 1993.
2. Biggerstaff, Ted J., "Anticipatory Optimization in Domain Specific Translation," International Conference on Software Reuse, Victoria, B. C., Canada, June 1998, pp. 124-133.
3. Biggerstaff, Ted J., "Composite Folding in Anticipatory Optimization," Microsoft Research Technical Report MSR-TR-98-22, June 1998, pp. 10.
4. Biggerstaff, Ted J., "Pattern Matching for Program Generation: A User Manual," Microsoft Research Technical Report MSR-TR-98-55, December 1998, pp. 46.
5. Boyle, James M., "Abstract Programming and Program Transformation—An Approach to Reusing Programs," In Ted J. Biggerstaff and Alan Perlis (Eds.), Software Reusability, Addison-Wesley/ACM Press, 1989, pp. 361-413.
6. Crew, R. F., ASTLOG: A Language for Examining Abstract Syntax Trees, In Proceedings of the USENIX Conference on Domain-Specific Languages, "ASTLOG: A Language for Examining Abstract Syntax Trees," Santa Barbara, California, October 1997.
7. 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.
8. Neighbors, James M., "Draco: A Method for Engineering Reusable Software Systems." In Ted J. Biggerstaff and Alan Perlis (Eds.), Software Reusability, Addison-Wesley/ACM Press, 1989, pp. 295-319.
9. Smith, Douglas R., "KIDS-A Knowledge-Based Software Development System," in Automating Software Design, M. Lowry & R. McCartney, Eds., AAAI/MIT Press, 1991, pp.483-514.
10. 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. February, 1997, pp. 164-186.
11. Wile, David S., "Popart: Producer of Parsers and Related Tools," USC/Information Sciences Institute Technical Report, Marina del Rey CA 1994. (http://www.isi.edu/software-sciences/wile/Popart/popart.html)
12. 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. February, 1997, 324-352.