A New Architecture for Transformation-Based Generators[1]

Ted J. Biggerstaff[2]
SoftwareGenerators.com

tbiggerstaff@austin.rr.com

Abstract - A challenge of many 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 Anticipatory Optimization Generator (AOG) has been built to explore architectures and strategies that address this challenge. The fundamental principle underlying all of AOG's strategies is to solve separate, narrow and specialized generation problems by strategies that are narrowly tailored to specific problems rather than a single, universal strategy aimed at all problems. A second fundamental notion is the preservation and use of domain-specific information as a way to gain extra leverage on generation problems. This paper will focus on two specific mechanisms: 1) *Localization*: The generation and merging of implicit control structures, and 2) *Tag-Directed Transformations:* A new control structure for transformation-based optimization that allows differing kinds of retained domain knowledge (e.g., optimization knowledge) to be anticipated, affixed to the component parts in the reuse library, and triggered when the time is right for its use.

Index Terms - Backtracking, domain-specific architectures, image processing, inference engines, logic programming, optimization, partial evaluation, pattern matching, program synthesis, reusable software, search, program transformations.

1 Introduction

1.1 The General Problem

A serious problem of most pattern-directed, transformation-based program generators is that they are trying to achieve three mutually antagonistic goals simultaneously: 1) deeply factored and highly abstract operators and operands to gain the combinatorial programming leverage provided by compositions of abstractions, 2) high performance code in the generated program, and 3) small (i.e., practical) generation search spaces. Various program generators focus on one or two of these goals thereby often compromising the other goal(s). This paper will make the argument that this quandary is due in large measure to deficiencies of conventional pattern-directed transformation models. While

[1] This paper is derived from material in two papers, one chosen as best paper of conference and one tied for best paper of conference in two successive International Conferences on Software Reuse. [11, 12] Some of the early work on this project was done at Microsoft Research and the author gratefully acknowledges the support of Microsoft.

[2] Address: 3801 Far View Drive, Austin, Texas 78730; email: tbiggerstaff@austin.rr.com., website: www.softwaregenerators.com.

pattern-directed (PD) transformations (also called *rules*) make the specification of the transformation steps easy and, in fact, often do a pretty good job of refining (i.e., translating) abstractions into code, they just as often explode the search space when one is later trying to produce highly optimized code from compositions of deeply factored, abstract operators and operands. Since giving up the deep factoring of abstractions to reduce the search spaces also gives up the combinatorial programming leverage provided by the composition, it is not a good trade-off.

Before addressing the detailed sub-problems and proposing solutions, the next two sections will provide a short introduction to PD transformation systems and discuss the nature of the search space explosion problem.

1.2 Pattern-Directed Control Regimes

In the simplest form, generic pattern-directed transformation systems store knowledge as a global soup of transformations[3] represented as rewrite rules of the form[4]

Pattern ⇒ RewrittenExpression

The left hand side of the rule (i.e., ***Pattern***) matches a subtree of an Abstract Syntax Tree (AST) and binds matching elements of that subtree to variables (e.g., **?operator**) in the pattern. If successful, the right hand side (rhs) (i.e., ***RewrittenExpression***), instantiated with the variable bindings, replaces the matched portion of the subtree. A simple, concrete example of such a rule might be a distributed law for the arithmetic operators * (i.e., times) and + (i.e., plus):

```
  ?X * (?Y + ?Z) => (?X * ?Y) + (?X * ?Z)
```

If this example transformation is applied to the AST expression

```
          A * (10 + B)
```

[3] Most non-toy transformation systems use various specialized designs to attempt to overcome the inefficiencies of the “global soup” model of transformations while retaining the convenience of viewing the set of rules as a set of more or less independent atoms. For example, Neighbor’s Draco system [32-34] compiled all transformations such that associated with each transformation was a list of just those transformations that could possibly follow a successful application of that transformation. Thus, potential follow-on transformations that could be determined *a priori* to be impossible for all possible data expressions, would never be tried.

[4] The meta-language for definitions is: 1) **bold for code**, 2) ***bold italic for psuedo-code***, and 3) *non-bold italic underlined for comments*.

then the pattern variable **?X** would match **A**, **?Y** would match **10** and **?Z** would match **B** and the rewritten result would be

```
(A * 10)  + (A * B)
```

Operationally, rules are chosen (i.e., triggered) based largely on the pattern of the left hand side, thereby motivating the moniker “Pattern-Directed” for such systems. Beyond syntactic forms, rules may also include 1) semantic constraints (e.g., type restrictions), and 2) domain constraints that must be true before the rule can be triggered. Such constraints are called *enabling conditions*. The checking of enabling conditions and other translation chores (e.g., generating translator variables) are often handled by a separate procedure associated with the rule.

One of the key questions with transformation systems is what is the organization and control regimes underlying the application of the rules. That is, how is rule storage organized and how are the transformations triggered? The question of rule organization will be deferred to a later section but triggering issues will be considered here. In general, control regimes are some mixture of two kinds of triggering strategies: PD triggering and metaprogram controlled triggering [17, 19, 35, 37, 44]. In its simplest implementation, PD triggering produces a control regime that looks like an exhaustive search process directed mostly by syntactic or semantic information local to AST subtrees. While PD regimes allow easy addition of new rules because rules can be treated as independent atoms, pure PD regimes have the problem that the triggering of the rules is based on pattern matching that is largely local to AST subtrees. This strategy leads to an overall process that is strategically blind and often induces very large search spaces.

On the other hand, the triggering choices may be made by a metaprogram that codifies some strategic goal and often employs heuristics to shortcut the search process. Metaprograms [17, 37] are algorithms and therefore, have state. This allows them to make design choices based on the earlier successes or failures. Their heuristic character, computational purposefulness and use of state information tends to reduce the search space over that of a pure PD search. However, the algorithmic rigidity and heuristic approach makes extensions more difficult than just dropping in a few new transformations. Also, surprise interactions among design choices are less likely with metaprograms than with PD search because typically some combinations of design choices have been pruned away in the design of the metaprogram. Finally, pruning the search space precludes exhaustive searches and therefore, some important answers may be missed.

Nevertheless, both PD rules and metaprograms may lead to searches that tend to explode. Why? The short answer is *constraint propagation*, i.e., separated parts of a generated program must be coordinated in order to produce a correct program. The next section will look at this problem more closely.

1.3 The Constraint Propagation Problem

Constraint propagation [27] is the process whereby design choices made in one part of the generated program must be coordinated with design choices made in other parts of the program. For example, suppose that a generator is refining a container abstraction in the target program specification and the generator chooses to implement that container as a linked list. Elsewhere in the evolving program, the generator will have to choose an implementation algorithm for the container's search method. Suppose that the reusable library used by the generator allows two kinds of searches, sequential and Boyer-Moore. A Boyer-Moore search is precluded by the previous choice of a linked list implementation because for a Boyer-Moore search, the container must have the property that every element of the container can be accessed in an equal amount of time. An array has this property but a linked list does not. This constraint must be communicated between the two places in the program where these related decisions will be made.

Katz and Volper [27] have shown that the problem of finding a consistent set of refinements for a program specification is NP complete. Operationally, this means that automating the development and optimization programs in the most completely general form is likely to face exploding spaces in the search for a consistent set of refinements and optimizations. The remainder of this paper will describe strategies and compromises that reduce this search space explosion.

2 Controlling Search Space Explosions

2.1 Overview

A central thesis of this paper is that the constraint propagation problem is best approached by solving specialized sub-problems that lend themselves to the use of specialized control regimes and metaprograms. Such strategies:

1) Solve narrower, more specific problems that are not NP complete (e.g., the problem of generating and integrating "implicit" target program control structures),

2) Use transformation control regimes that are customized to those narrower problems (e.g., break the overall translation into phases with narrow translation goals and trigger optimizing transformations based on event tags that are attached to reusable components[5]),

3) Employ domain knowledge to further prune the number of search space choices (e.g., use domain knowledge to pre-tag reusable components with calls to desirable optimizations),

4) Limit the area of the program over which constraints must be propagated (e.g., within a single Domain-specific Language – DSL – expression), and

5) Organize the transformations in ways that reduce the number to be tried at any given point (e.g., group transforms in a two dimensional space that exposes only a few transforms at each point in the translation process). For example, organizing DSL translations into a series of PD phases reduces the number of transformations that need to be tried in each phase and thereby reduces the number of pattern matches needed. The order of the phases also determines the order in which the phase-specific groups of transforms are enabled.

Some of these strategies have been employed by existing generators (e.g., Draco [32-34] and TAMPR [13, 21] employ two notions of phased translation) and some are introduced by the AOG generator [6-12] (e.g., tagging reusable components with desirable optimizations that are triggered by translation events). The remainder of the paper enlarges on these strategies.

2.1.1 Phased DSL to DSL Refinements

One way to reduce the search space is by employing **implicit** rule subsetting mechanisms to make irrelevant rules invisible in a particular situation. Systems like Draco employ distinct DSLs that can be translated in stages from high level DSLs to lower level DSLs and eventually to conventional programming languages such as C++ or Java. These distinct DSLs induce an implicit subsetting of rules that reduces the search space at each translation stage by hiding rules not relevant to the specific DSL constructs being translated. Thus, *refinement* – the process of translating from one DSL to lower level DSLs and eventually to code – produces a series of small search spaces rather than one large search space because at each translation stage only a small number of relevant rules are available to be attempted.

2.1.2 Inter-Refinement Optimization Phases

At each DSL to DSL translation stage, overly complex code is often generated, which explodes the number of pattern cases that need to be used in the next translation stage. Periodic AST expression simplification is needed to prevent the rules from becoming so complex that refinement progress is

[5] Components are definitions of domain-specific classes, methods and operators. Methods and operators are implemented as transformations that are special in that they get triggered only at a predefined phase in the overall translation process.

impeded. Program generation often uses a kind of form *simplification* or *specialization* that basically removes redundant expressions in a DSL (e.g., (X + 0) => X) without attempting any sophisticated expression reorganization or extended inference. In AOG, this step is attempted as each new code expression is generated in an attempt to keep the expressions as simple and canonical as possible. Without this step, subsequent refinement rules become inordinately complex and thereby, explode the search space. AOG uses a *customized partial evaluator*[6] [25] that is designed to specialize code for which some data values have become known (e.g., unrolling a loop will make loop indexes known constant values).

What is more, DSL expressions often reveal opportunities to execute certain *domain-specific optimizations* that would be impractical without the domain-specific (DS) vantage point. [32-34] For example, an optimization rule using knowledge of an *Augmented Transition Network (ATN)* parser domain may remove an ATN state (equivalent to removing a parse rule in a conventional parser) and thereby make a significant optimization. Such an optimization would be impractical to perform once the target program is translated into a conventional programming language because the optimizer would be swamped by low-level details and low-level transformations and would no longer have the abstract, domain knowledge to guide it. The domain level knowledge provides a view of the "forest" whereas the code level provides only a view of the "trees." Thus, DS knowledge also plays an important role by allowing *domain-specific optimizations* that map from a domain to itself because the rules can use the domain knowledge to significantly improve the target computation while the program specification is still at an abstract, domain-specific level. Once the abstract, DS level is translated to the conventional code, the result of the optimizations can often be seen to be quite sweeping and difficult at that level.

Thus, interlaced between each DSL to DSL refinement stage is a stage that simplifies the generated code and applies optimizations specific to the current DSL.

[6] AOG's partial evaluator (PE) is a restricted *online* partial evaluator with some customized features. Its focus is restricted to a small portion of the target program and it uses a restricted set of PE operations. Classically, the opportunity for PE arises due to some knowledge of the input data (e.g., some input values are known constants) and this knowledge allows code elimination or specialization. In AOG, such knowledge arises specifically as a result of the manipulation of the program by transforms (e.g., loop unrolling) and is highly localized within the target program (e.g., localized to the body of the unrolled loop). AOG's partial evaluator avoids many of the expression expansions that a classical partial evaluator would perform (e.g., unfolding of recursive function calls or specializing functions and replacing the calls to them). In AOG, expression expansion decisions are relegated to the generation transforms (which trigger the partial evaluator). Thus, these expansion decisions arise for reasons other than the possibility of partial evaluation specializations and any such specializations are simply serendipitous side effects of the transformations. For example, a transform may unroll a loop to enable some later transformation but that unrolling may allow expression simplification in some of the new body instances.

2.1.3 Localization

Domain-specific languages excel at programming productivity improvements[7] because they provide large-grain composite data structures (e.g., a graphics image) and large-grain composition operators (e.g., image addition). As a result, extensive computations can be written as APL-like one-line expressions that are equivalent to tens or hundreds of lines of code (LOC) when written in a conventional language like Java. Refining such expressions step-by-step into programming language operators and operands introduces implied control structures (e.g., loops or enumerations) for each large grain composite or large grain operator. These implied control structures are distributed (i.e., de-localized) across an expression of operators and operands. Relationships among these operators and operands invite full or partial control structure sharing across a multiple operator expression. Human programmers recognize the relation among these distributed control structures and merge them into customized control structures that minimize the redundancy of control. For example, while the controls implied by each large-grain item in a DSL expression may imply a series of passes over the data, customized control structures may be able to perform several operations on large-grain data structures in a single pass. This generation of custom control structures from the individual control elements implied by the operators and operand that are scattered across a DSL expression is called *control localization*. In AOG, control localization is automated via PD rules. A later section will follow through an extended localization example.

2.1.4 Architectural Shaping

Inter-component (i.e., cross-domain and cross-component) optimization tasks suffer several problems: 1) they are coordinating remote but related pieces of the target program, 2) they are less influenced by local AST patterns than by global optimization goals, and 3) they often cannot be performed until the target program has been refined to the programming language domain by which time virtually all DS leverage is lost because the DS knowledge has been translated away. Because of these problems, conventional PD strategies for such optimization tasks may explode the search space.

These problems arise largely because the generator is trying to establish global operational properties in the target program, e.g., trying to optimize the overall code for differing machine architectures while starting out with a canonical target program specification and canonical reusable piece-parts. For example, one would like to be able to generate code optimized for a non-SIMD[8] architecture and with the flip of a switch generate code optimized for a SIMD architecture. Domain knowledge provides unique and valuable information about the computational goals and interrelationships of the program parts that can be used to simplify the optimization process. For example, the control structure of an

[7] Evidence for generation-based programming productivity improvement and performance improvements is found in [2-3].

[8]SIMD is defined as Single Instruction stream, Multiple Data stream architecture.

image convolution operator[9] will have an outer two-dimensional (2D) loop iterating over the image and an inner 2D loop iterating over a pixel neighborhood within that image. Further, the neighborhood computations are likely to have a special case computation method for neighborhoods that are hanging off the edge of the image. To truly exploit a SIMD machine with a parallel sum of products operator and a parallel addition operator, a human programmer would likely apply an optimization that would split the outer 2D loop into two kinds of outer loops, one to handle special case computations (e.g., neighborhood partially off the edge of the image) and one to handle the default case (e.g., neighborhood completely within the image). Such a control design will provide better pipelining of data onto the bus (i.e., no bus stalls induced by the conditional branches that check for the off-edge condition) and therefore, more optimal exploitation of the parallel operators. This "optimization goal" can be accomplished by a metaprogram whose job is to discover the pieces (i.e., the outer 2D loop associated with the convolution and the conditional test for the special case computation) and transform them into the separate control structures. In AOG, this metaprogram is a large grain transformation named **_SplitLoopOnCases**.

With conventional systems such optimizations are difficult because they cannot occur until the generator gets to the code level and integrates the code pieces that provide the programming details required by the loop splitting optimization. Further, the optimization (e.g., **_SplitLoopOnCases**) is ideally expressed in terms of abstract domain structures (e.g., the abstract form of a typical convolution and its special cases) but these structures correspond to low level programming language structures of the program that have lost any connection to that domain knowledge. That is, the generator no longer knows that a specific two-dimensional (2D) code level loop is a convolution and a specific if-then test in the body of that loop is a special case test associated with that convolution definition. The **_SplitLoopOnCases** optimization only works if the generator has retained a connection between the domain abstractions in which the optimization is expressed and the programming structures into which the domain abstractions are translated. While conventional optimization algorithms can (in theory) re-discover such connections, it is computationally complex and when the interrelationships among many individual optimizations are considered, re-discovery may tend to produce a search space explosion. For example, tens or hundreds of preparatory transformations may have to be discovered and run in a specific order to enable **_SplitLoopOnCases** and allow it to be successful.

Ironically, the creator of these reusable components (e.g., convolution definition) has the key domain knowledge in hand at the time he puts the components into the reusable library. Further, he also knows that on a SIMD machine the **_SplitLoopOnCases** will be a good transformation to try on these

[9] A convolution is the sum of the products of the pixels in a pixel neighborhood and the weights associated with the pixel positions in that neighborhood, where a neighborhood of an image is a subset of pixels centered on some pixel in the image. Neighborhoods are used to provide the specifics of various kinds of image convolution operations. Aspects of a neighborhood are defined by methods that give its size, weight values associated with various neighborhood pixel positions, special cases for dealing with the neighborhood, and so forth.

components. But conventional transformation systems allow no easy way to express early and then later exploit such information about desired optimizations.

To retain such domain knowledge so that it can be directly applied once the composed components have been refined to a conventional programming language level, AOG introduces a new kind of transformation and a new control regime (i.e., tag-directed or TD Transformations[10]). The TD control regime preserves such early domain knowledge by adding tags to the reusable components. These tags provide direct information as to which TD transformation to invoke (i.e., the tag contains an explicit call with parameters that connect the domain concepts to the code details), when to invoke it (i.e., TD-transformations are triggered by generator events) and where to focus its activity (i.e., on the component to which it is attached). This helps to avoid using complex algorithms to (partially) infer lost domain knowledge and discover the sequence of needed preparatory transformations, which in turn helps to eliminate the search spaces that can arise from the interactions of many such individual optimizations.

The remainder of the paper will examine these strategies with emphasis on those that are particular to AOG.

3 Localization

3.1 The Problem

DSLs significantly improve program productivity because they deal with large-grain data structures and large-grain operators and thereby allow a programmer to say a lot (i.e., express a lengthy computation) with a few symbols. Large-grain data structures (e.g., images, matrices, arrays, structs, strings, sets, etc.) can be decomposed into finer and finer grain data structures until one reaches data structures that are atomic with respect to some conventional programming language (e.g., field, integer, real, character, etc.). Thus, operators on large-grain data structures imply some kind of extended control structure such as a loop, a sequence of statements, a recursive function call, and so forth. As one composes large-grain operators and operands together into longer expressions, each subexpression implies not only some computation (e.g., pixel addition) that will eventually be expressed in terms of atomic operators (e.g., modular integer addition), but it also implies some control structure to sequence through those computations. Those implied control structures are typically distributed (i.e., de-localized) across the whole expression.

[10] Microsoft Patent Number 6,314,562.

For example, if one defines an addition operator for images in some graphics domain and if **a** and **b** are defined to be graphic images, the expression **(a + b)** will perform a pixel-by-pixel addition of the images. To keep the example simple and limit the number of definitions that must be introduced, suppose that the pixels are integers (i.e., **a** and **b** are grayscale images) and **a** and **b** are the same size. Then the expression **(a + b)** implies a 2D loop over **a** and **b**. Squaring each pixel in resulting image (represented as $\mathbf{(a + b)}^2$) implies a second outer 2D loop. Human programmers easily identify this case as one that can be dealt with in a single 2D pass over the image.

That kind of transformation seems simple enough but the real world is much more complex and when all of the cases and combinations are dealt with, it may require tricks to avoid the search space becoming intractably large. More complex operators hint at some of this complexity. For example, consider a *convolution operator*[11] **⊕,** which for each pixel **a[i,j]** in some image **a**, performs a sum of products of all the pixels in a neighborhood of that pixel times weights associated with the pixel positions of that neighborhood. The weights are defined separately from ⊕. Suppose the weights are defined by a domain object **s** that is called a *neighborhood* of a pixel, where the actual pixel position defining the center of the image neighborhood will be a parameter of **s**. Then **(a ⊕ s)** would define a sum of products operation for each neighborhood around each pixel in **a** where the details of the neighborhood would come from **s**. Thus, **s** will contribute (among other data) the neighborhood size and the definition of the method for computing the weights. The ⊕ operator definition will contribute the control loop and the specification of the centering pixel that is to be the parameter of **s**. The translation rules not only have to introduce and merge the control structures, they have to weave together (in a consistent manner) the implied connections among the loop control, the definition of ⊕ and the definition of **s**.

Thus, localization can be fairly complex because it is coordinating the multi-way integration of specific information from several large-grain components. The greater the factorization of the operators and operands (i.e., the separation of parts that must be integrated), the more numerous and complex are the rules required to perform the (re-) localization. As a consequence, localization has the potential to explode the solution search space. To thwart this explosion, AOG groups localization rules in special ways and makes use of domain-specific knowledge to limit the explosion of choices during the localization process. Both of these strategies reduce the search space.

While this paper will focus on the Image Algebra domain [36], the de-localization problem is universal over all complex domains with large-grain operators and operands. Localization is required when the

[11] A more formal definition of the convolution operator is given in the next section.

domain's language factors and compartmentalizes partial definitions of large-grain operators and data structures and then allows compositional expressions over those same operators and data structures. Other domains that exhibit DSL induced de-localization are: 1) the User Interface domain, 2) the network protocol domain, 3) various middleware domains (e.g., transaction monitors), and so forth.

3.2 An Example Mini-Domain

To provide a concrete context for discussing the issues of localization, this section will define a tiny portion of the Image Algebra (IA) as a mini-DSL for writing program specifications.

Domain Entity	**Description**	**Definition**	**Comments**
Image	An composite data structure in the form of a matrix with pixels as elements	**a** = {**a [i , j]: a[i , j]** is a pixel} where **a** is a matrix of shape **[[imin : imax], [jmin : jmax]]**.	Subclasses include images with grayscale or color pixels. To simplify the discussion, assume all images have the same size.
Neighborhood[12]	A matrix template overlaying a region of an image and centered on an image pixel such that the matrix associates a numerical weight with each overlay position	A neighborhood **s** is defined by a set of methods. For example, its weights are defined by a method **w.s** that computes elements of the set **w(**$s_{a[i,j]}$**)** = { **w[p , q] : w[p , q]** is a numerical weight associated with the **[p , q]** position of **s** centered on pixel **[i,j]** of some image **a** where **s** is a neighborhood of shape **[[pmin : pmax] , [qmin : qmax]]** and **a** is an image of shape **[[imin : imax] , [jmin : jmax]]** }	Neighborhoods are objects with methods. The methods define the weights, neighborhood size, special case behaviors, and methods that compute a neighborhood position in terms of image coordinates. Notice that all methods (e.g., **w.s**) may depend upon the image size and shape, neighborhood size and shape as well as the position of the neighborhood in the image.
Convolution	The convolution **(a ⊕ s)** applies the neighborhood **s** to each pixel in **a** to produce a new image	**(a ⊕ s) = {∀i,j (b[i , j] : b[i , j] = (**$\sum_{p,q}$ **(w[p , q] * a [i+p , j+q])) }** where **w[p , q]** ∈ **w(**$s_{a[i,j]}$**)**, **p** and **q** range over the	Variants of the convolution operator are produced by replacing the $\sum_{p,q}$ operation with $\Pi_{p,q}$, **Min** $_{p,q}$, **Max** $_{p,q}$, and

[12] To avoid unneeded complexity, this definition of Neighborhood is a slightly relaxed version of the actual IA definition.

		neighborhood **s**; **i** and **j** range over the images **a** and **b**)	others; and the + operation with *****, **max**, **min** and others.
Matrix Operators	**(a+b)** , **(a-b)** , **(k*a)** , **a**^{n}, **√a** where **a** & **b** are images, **k** & **n** are numbers	These operations on matrices have the conventional definitions, e.g., **(a+b) = {∀i,j (ai,j + bi,j)}**	

Define the weights for concrete neighborhoods **s** and **sp** to be 0 if the neighborhood is hanging off the edge of the image, or to be

$$w(s) = \begin{array}{cc} & \overbrace{\begin{array}{ccc}-1 & 0 & 1\end{array}}^{Q} \\ \mathrm{P}\left\{\begin{array}{c}-1\\0\\1\end{array}\right. & \begin{bmatrix}-1&-2&-1\\0&0&0\\1&2&1\end{bmatrix}\end{array} \qquad w(sp) = \begin{array}{cc} & \overbrace{\begin{array}{ccc}-1 & 0 & 1\end{array}}^{Q} \\ \mathrm{P}\left\{\begin{array}{c}-1\\0\\1\end{array}\right. & \begin{bmatrix}-1&0&1\\-2&0&2\\-1&0&1\end{bmatrix}\end{array}$$

if it is not. Given these definitions, one can write an expression for a Sobel edge detection method [36] that has the following form:

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

This expression de-localizes loop controls and spreads them over the expression in the sense that each individual operator introduces a loop over some image(s), e.g., over the image **a** or the intermediate image **(a ⊕ s)**. Consider what control structures are implicit in this expression, how they are related and how these separate implicit control structures can be woven into a minimal combined control structure.

3.3 A Localization Example

Implicit in the definitions of the operator and operands are the following control information and relationships[13]:

1) The instances of **a** imply 2D loop controls that iterate through the pixels of **a** – e.g., $(\forall \mathbf{i,j}: \mathbf{a}_{\mathbf{i,j}})$ and $(\forall \mathbf{v,z}: \mathbf{a}_{\mathbf{v,z}})$ – where the generator creates the index variables **i,j** and **v,z**, and **a** supplies information about the ranges of the index variables;

[13] To keep the example simple, it will assume that **a** and **b** have the same dimensions.

2) The convolution expressions $(a \oplus s)$ and $(a \oplus sp)$ imply two additional 2D loop controls that must be identical to the $(\forall_{i,j}: a_{i,j})$ and $(\forall_{v,z}: a_{v,z})$ loop controls;

3) The convolution expressions also imply 2D loops over the neighborhoods (e.g., $(\Sigma_{p,q}: (w.s\ (a[i,j],m,n,p,q)\ *\ a_{i+p,j+q})))$ that are nested within the **i,j** and **v,z** loops, where **s** and **sp** supply both the ranges of **p** and **q**, and the neighborhood weight method (e.g., **w.s (a[i,j],m,n,p,q)**, which computes the weight at the **p,q** offset of neighborhood **s**, when **s** is positioned at the pixel **i,j** in the **m** by **n** image **a**);

4) The instance of **b** implies a 2D loop with generated index variables **d,e**, i.e., $(\forall_{d,e}: b_{d,e})$ but the generator will have to infer the relationship between this loop and the other loops; and

5) The generator will have to infer that the 2D loop controls implied by the image instances may be merged with 2D loop controls implied by the convolution, square, plus, square root, and assignment operators based on the semantics of those operators and their operands. Operationally, this means that some of the generated index variables will be discarded and others used in their place.

Now, consider an *idealized* example of AST rewrites that will perform localization. The example will ignore many implementation complexities. Also, the AST rewrites will be re-ordered slightly to simplify the presentation. Even though the AST is a tree, the example will use a text based publication form where the tree structure is implied by the parenthetical nesting of expressions. For reference, rules will be given names. The beginning example AST is:

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

A rule named **RefineComposite** will rewrite the AST to refine **image** instances (e.g., **b**) into **pixel** instances (e.g., $b_{d,e}$) by introducing the control structures implied by the image instances. Three applications of the rule transforms the AST into the form:

$$(\forall_{d,e}: b_{d,e}) \quad = [((\forall_{i,j}: a_{i,j}) \quad \oplus s)^2 + ((\forall_{v,z}: a_{v,z}) \quad \oplus sp)^2]^{1/2}$$

Next, the **ConvolutionOnLeaves** rule will introduce the definitions of the outer loop of ⊕ and infer the equivalence of the outer loops of ⊕ and the loops (i.e., $\forall_{i,j}$ and $\forall_{v,z}$) already introduced. Because ⊕ is overloaded, the new expression is using the definition of ⊕ that operates on pixels whereas the previous expression was using the definition that operates on images.

$$(\forall_{d,e}: b_{d,e}) \quad = [(\forall_{i,j}: a_{i,j} \quad \oplus s)^2 + (\forall_{v,z}: a_{v,z} \quad \oplus sp)^2]^{1/2}$$

Next, the **FunctionalOpsOnComposites** rule processes the square operator applied to the intermediate images (e.g., the image represented as $(\forall_{i,j}: a_{i,j} \quad \oplus s)$). Since square is a pure arithmetic function, no new loop needs to be introduced. Square can be immediately applied to each of the pixel values as they are computed by the **i,j** and **v,z** loops. Operationally, this just propagates the loops above the respective square operators.

$$(\forall_{d,e}: b_{d,e}) \quad = [(\forall_{i,j}: (a_{i,j} \oplus s)^{2}) + (\forall_{v,z}: (a_{v,z} \oplus sp)^{2})]^{1/2}$$

The next rewrite (the **FunctionalOpsOnParallelComposites** rule) determines that the **+** operator is adding two intermediate images whose loops can be merged. It chooses to retain the **i,j** index variables and discard the **v,z** variables. In effect, this propagates the **i,j** loop above the **+** operator and replaces the **v,z** indexes with **i,j**.

$$(\forall_{d,e}: b_{d,e}) \quad = [\forall_{i,j}: ((a_{i,j} \oplus s)^{2} + (a_{i,j} \oplus sp)^{2})^{1/2}]$$

Like the square operator, the semantics of the square root operator allows the **i,j** loop to be propagated above it.

$$(\forall_{d,e}: b_{d,e}) \quad = \forall_{i,j}: [((a_{i,j} \oplus s)^{2} + (a_{i,j} \oplus sp)^{2})^{1/2}]$$

Like the earlier case that combined loops over the + operator, the next rewrite (the **FunctionalOpsOnParallelComposites** rule) will merge the loops over the assignment operator, retaining the **i,j** index variables and discarding the **d,e** variables. The final form of loop localization is:

$$\forall_{i,j}: [b_{i,j} \quad = ((a_{i,j} \oplus s)^{2} + (a_{i,j} \oplus sp)^{2})^{1/2}]$$

A following section will exhibit the form of the **RefineComposite** rule used in this example, but first, the form and storage organization of the PD transformation rules must be explained.

3.3.1 Defusing Search Space Explosions

As discussed earlier, AOG avoids NP complete approaches to program generation by solving narrower, more specialized problems with methods that are polynomial in some aspect of the program (e.g., number of nodes in an expression tree). Localization is one such specialized solution. While this narrowing of the problem is by far the most important technique for defusing search space explosions, AOG uses two additional tricks to reduce search space explosion: 1) It groups the localization rules in ways that make irrelevant rules invisible, and 2) It uses domain knowledge (e.g., knowledge about the general design of the code to be generated) to further prune the search space.

The discussion will focus on item 1 and defer discussion of item 2. Grouping transformations so that at each decision point only a small number of relevant transformations need to be tried is a good way to reduce the search space. AOG implements this idea by allowing rules to be stored under any object (e.g., a “type” object) and allows additional discrimination by further grouping the rules under an arbitrary translation phase name. The phase name captures the strategic objective or job that those rules as a group are intended to accomplish (e.g., the Localize phase performs control localization). In addition, the object under which the rules are stored often provides some key domain knowledge that further prunes the search space. For example, in order for loop localization to move loops around, it needs to know the data flow design for the various operators. The general design of the operator’s data flow is knowable by knowing the resulting type of the expression plus the details of the expression. Thus, the localization rules are stored on type objects. The individual rules determine the details of the expression (e.g., operator and operand structure) via pattern matching. As a consequence, the localization process for a specific expression of type X is a matter of trying all rules in the Localize group of the type X and in the Localize group of all super types of X. Notice that this means that AOG transformations support a form of inheritance if they are attached to types. An AOG rule will be attached to the most general type to which it applies and it will apply to all subtypes as well.

Operationally, AOG rules provide this organization by the rule format:

```
(⇒ XformName PhaseName ObjName Pattern RewrittenExpression Pre Post)
```

The transform’s name is ***XformName*** (e.g., **RefineComposite**). The rule is stored as part of the ***ObjName*** object structure, which in the case of localization will be a type object, e.g., the **image** type. The rule is enabled only during the ***PhaseName*** phase, which in this context is **Localize**. ***Pattern*** is used to match an AST subtree and upon success, the subtree is replaced by ***RewrittenExpression*** instantiated with the bindings returned by the pattern match. ***Pre*** is the name of a routine that checks enabling conditions and performs bookkeeping chores (e.g., creating translator variables and computing equivalence classes for localization). ***Post*** performs various computational chores after the rewrite. ***Pre*** and ***Post*** are optional.

For example, a trivial but concrete example of a PD rule would be

```
(⇒ FoldZeroXform SomePhaseName dsnumber `(+ ?x 0) `?x)
```

This transform is named **FoldZeroXform**, is stored on the type **dsnumber**, is enabled only in phase **SomePhaseName**, and rewrites an expression like **(+ 27 0)** to **27**. The pattern variable **?x** will match anything in the first position of expressions of the form **(+ ___ 0)**. Now, let’s examine an example localization rule.

3.3.2 RefineComposite Rule

Among the rules used in the earlier example is **RefineComposite**, which refines an instance of a black and white image (e.g., **a**) into an instance of a black and white pixel (e.g., **a[i,j]**) and generates the implied control structure (e.g., **(∀i,j: ...)**) needed to iteratively compute the pixel values. The idealized forms shown in the example are designed for publication but gloss over some of the operational details needed for localization. In order to understand the example rule, these details must be defined. For example, each node in the AST tree has a LISP-like property list (called a *tags* list) that is used to keep translation data specific to that AST node. The tags lists are simply appended to an AST node list. For example, the expression **(+ a b)** might have a tags list that contains an attribute value pair **(itype image)** indicating the type of the expression is **image**. That AST node would be represented as **(+ a b (tags (itype image))**. All AST leaves consisting of atomic items are represented by the form **(leaf *AtomicItem* (tags …))** to provide a place to hang the tags list for such nodes. Thus, the left hand side (lhs) of **RefineComposite** will have to match an AST node of the form **(leaf a (tags (itype image)))**.

By the same token, the implementation form of the transformed AST node **a[i,j]** will be represented for the convenience of the localization machinery. Rather than encoding **a[i,j]** as an inline syntactic expression that will have to be recognized and deconstructed with every use, it is represented by a translator-generated temporary symbol (e.g., **bwp27**) of type **bwpixel**. Similarly, the other translator-generated variables shown as **i** and **j** in the idealized example, will be translator-generated variables with forms more like **idx28** and **idx29**. Further, rather than encoding the loop information (e.g., **(∀i,j: ...)**) in terms of AST syntax that will have to be recognized and deconstructed by every subsequent rule, it is stored in a canonical form on the tags list, thereby allowing it to be ignored by all rules for which it is not relevant. This canonical form will be defined later.

Additionally, the **RefineComposite** rule will:

- Create a record of the correspondence relationship between the large-grain composite **a** and the component **bwp27** computed from it (e.g., the expression **(_mappings (bwp27) (a))**), which will be needed to determine what can be shared between various loops and how loops nest, and

- Generate a rule containing the details of the refinement relationship (e.g., a rule like **bwp27 => a[idx28, idx29]**), which will be needed to (eventually) re-express translator symbols in terms of the original data structures and the generated control variables.

How would one formulate the **RefineComposite** rule in AOG? Given a routine to generate symbols (**gensym**), a first approximation of this rule might be:

```
  (=> RefineComposite Localize Image `?op (gensym ‘bwp))
```

But this form of the rule does not do quite enough. An image instance should be represented in the AST in the leaf form – e.g., **(leaf a …)**. Thus, the rule will have to deal with a structure like **(leaf a (tags Prop1 Prop2 …))**. For robustness, the rule will allow the naked atomic image **a** as well. To accommodate this case, the rule pattern will have to use AOG's "*or*" pattern operator, **$(por pat1 pat2 …)**, which allows alternative sub-patterns (e.g., **pat1 pat2…**) to be matched.

```
  (=> RefineComposite Localize Image
      `$(por (leaf ?op) ?op) (gensym ‘bwp))
```

Now, **(leaf a …)** will get translated to some black and white pixel symbol such as **bwp27** with **?op** bound[14] to **a** (i.e., **((?op a))**). However, the rule does not yet record the relationship among the image **a**, the **bwpixel bwp27**, and some yet-to-be-generated index variables (e.g., **idx28** and **idx29**) that will be needed to loop over **a** to compute the various values of **bwp27**. So, the next iteration of the rule adds the name of a pre-routine (say **RCChores**) that will do the translator chores of **gensym**-ing the **bwpixel** object **(bwp27)**, binding it to a new pattern variable (say **?bwp**), and while it is at it, **gensym**-ing a couple of index objects and binding them to **?idx1** and **?idx2**. The next iteration of the rule looks like:

```
  (=> RefineComposite Localize Image
      `$(por (leaf ?op) ?op) `(leaf ?bwp) `RCChores)
```

Executing this rule on the AST structure **(leaf a …)** will create the binding list **((?op a) (?bwp bwp27) (?idx1 idx28) (?idx2 idx29))** and rewrite **(leaf a …)** to **(leaf bwp27)**. However, it does not yet record the relationship among **a**, **bwp27**, **idx28**, and **idx29**. Other references to images in the example expression will create analogous sets of **image**, **bwpixel**, and **index** objects, some of which will end up being redundant. In particular, new loop index variables will get generated at each image reference in the AST expression. Most of these will be redundant and other rules will be added that merge away these redundancies by discarding redundant **bwpixel**s and

[14] A binding list is defined as a list of ***(variable value)*** pairs and is written as ***((vbl1 val1) (vbl2 val2) ...)***. Instantiation of an expression with a binding list rewrites the expression substituting each ***valn*** for the corresponding ***vbln*** in the expression.

indexes. So, the next version of the rule will create a shorthand form expressing the relationship among these items and add it to the tags list. The shorthand will have the form[15]

```
   (_forall (idx28 idx29)
           (_suchthat (_member idx28 (_range minrow maxrow))
                      (_member idx29 (_range mincol maxcol))
                      (_mappings (bwp27) (a))))
```

The **idx** variable names will become loop control variables that will be used to iterate over the image **a** generating pixels like **bwp27**, which will eventually be refined into array references such as **(aref a idx28 idx29)**. The **_suchthat** sub-expression captures all of the relationships that will be needed to perform loop localization steps and final code generation for the loop. The **_member** clauses define the ranges of the index variables. The lists in the **_mappings** clause establish the correspondences between elements (e.g., **bwp27, bwp41,** etc.) and the composites from which they are derived (e.g., **a, b,** etc.), thereby enabling the finding and elimination of redundant elements and loop indexes.

The final form of the **RefineComposite** rule (annotated with *explanatory comments*) is:

```
  (=> RefineComposite Localize Image
      `$(por (leaf ?op)  Pattern to match an image leaf structure
             ?op)        or just an image atom. Bind it to ?op
      `(leaf ?bwp        Rewrite image as the bwpixel bound to ?bwp.
          (tags         Add a property list to bwpixel structure.
             (_forall (?idx1 ?idx2)     Add a loop shorthand of indexes,
                                         ranges, and
               (_suchthat   (_member ?idx1 (_range minrow maxrow))
                            (_member ?idx2 (_range mincol maxcol))
                            (_mappings (?bwp) (?op))))    DS relations.
             (itype bwpixel)))   Add new type expression.
      `RCChores)  Name the pre-routine that creates bwpixel & indexes.
```

Follow-on phases (**CodeGen** and **SpecRefine** respectively) will cast the resulting shorthand(s) into more conventional loop forms and refine intermediate symbols like **bwp27** into a computation expressed in terms of the source data, e.g., **a[idx32,idx33]**. But this refinement presents a constraint coordination problem to be solved. How will the later phases know to refine **bwp27** into **a[idx32,idx33]** since when it was first generated, the original relationship suggested it would be refined into **a[idx27,idx28]**? In other words, along the way, **idx27** and **idx28** became redundant and were replaced with **idx32** and **idx33**. But just replacing **bwp27** with **bwp31** in the

[15] The AST is constructed using AST structures such as: **_forall** and **_sum** for expressing iterations; **_suchthat** as a holder of restrictive clauses defining the iterations; **_member** and **_range** as boolean operators used for expressing those restrictions; and **_mappings** for expressing the relationships between larger grain data structures (e.g., an image) and their smaller grain components (e.g., a black and white pixel).

AST at the point the redundancy is discovered does not work because the redundant indexes (e.g., **bwp27**) may occur in multiple places in the expression due to previously executed rules. Worse yet, there may be instances of **bwp27** that are yet to appear in the expression tree due to deferred rules that are pending. Other complexities arise when only the indexes are shared (e.g., between different images such as **a** and **b**). Finally, since the replacement of **bwp27** is, in theory, recursive to an indefinite depth, there may be several related abstractions simultaneously undergoing localization combination and coordination. For example, a color pixel abstraction, say **cp27**, may represent a call to the red method of the pixel class – say **(red pixel26)** – and the **pixel26** abstraction may represent an access to the image – say **a[idx64, idx65].** Each of these abstractions can potentially change through combination during the localization process. So, how is this problem handled in AOG?

3.3.3 Speculative Refinements Propagate Constraints

This coordination problem is solved in AOG by the Speculative Refinement (SR) process, which dynamically builds refinement rules and stores them on the relevant translator generated objects, e.g., **bwp27**. In effect, the resultant set of rules propagates and coordinates constraints and translation decisions over a DSL expression. These rules are “speculative” in the sense that a rule may be altered or eliminated by subsequent localization decisions. For example, the combination process seen in the previous section incrementally modifies these rules to properly reflect the incremental removal of redundancies. Once all rules are coordinated, they will be applied in a follow-on phase called the **SpecRefine** phase.

As an example of how SR rules are created, consider the **RefineComposite** rule shown earlier. Its pre-routine, **RCChores**, will create the several SR rules while processing various sub-expressions. Among them are:

```
(=> SpecRule89 SpecRefine bwp27 `bwp27 `(aref a idx27 idx28))
(=> SpecRule90 SpecRefine bwp31 `bwp31 `(aref a idx32 idx33))
```

Later, the pre-routine of the **FunctionalOpsOnParallelComposites** rule makes the decision to replace **bwp27** with **bwp31**. The SR rule **SpecRule89** gets changed to:

```
(=> SpecRule89 SpecRefine bwp27 `bwp27 `bwp31)
```

Thus, at the end of the loop localization phase all speculative refinement rules are coordinated to reflect the current state of localization combinations. The follow-on speculative refinement phase recursively applies any **SpecRefine** rules (e.g., **SpecRule89**) that are attached to abstractions (e.g., **bwp27**) in the AST tree. The result is a consistent and coordinated expression of references to common indexes, pixels, structure field names (e.g., **`red`**), and so forth.

3.3.4 Domain-specific Optimizations

Domain-specific optimizations are opportunistic rules that run between refinement stages and attempt to improve the resultant code through use of domain knowledge. A simple example of one such DS optimization that will be triggered for the example expression is the *reduction in strength* (RIS) optimization rule[16] that replaces the square operation[17] with multiplication. Since the item to be squared (e.g., the expression bound to **`?expr`**) is an expression as opposed to a variable or constant, the rule must avoid performing computation of the expression twice. The rule's pre-routine does this by inventing a temporary variable (e.g., **`t1`**) to hold the value of the expression to be squared and binding it to a pattern variable (**`?tmpvbl`**). The DS optimization transformation has the form:

```
(=> RuleName PhaseName TypeName (** ?expr 2) (* ?tmpvbl ?tmpvbl)
    Pre Post)
```

The pre-routine also has to generate an assignment statement of the form

```
(= ?tmpvbl ?expr)
```

to be placed into a yet-to-be-created context that dominates (i.e., precedes on all data flow paths) the statement containing the **`?expr`** expression. The placement of such assignment statements is handled by rules (called *deferred rules*) that the pre-routine dynamically creates. The lhs pattern of such a rule describes the expected context and the rhs rewrites the context to add the assignment statement. Deferred rules will be triggered when their target context is eventually created. The expected context for this assignment statement will be created during the **`CodeGen`** phase when the localization tags that evolved during the localization phase are finally converted into conventional loops. The next section describes the eventual generation of this context and the results produced by the pending deferred rules.

[16] Most compilers perform optimizations like this. However, if one waits until compile time to perform the RIS optimization, many opportunities for architectural shaping optimizations will be lost because this optimization may be a prepatory optimization that will establish some of the enabling conditions for a later TD-transform. AOG uses mostly conventional optimizations. [1] AOG's contribution is in the way in which it orchestrates a variety of complementary optimizations (some conventional, some not) to achieve a global architecture that optimizes the computation as a whole. For this example, because of the characteristics of the neighborhoods **s** and **sp** and the nature of a CPU without parallel instructions, AOG's goal is to setup the computation so that the two inner (neighborhood) loops can be unrolled and simplified into arithmetic expressions. By contrast, for a CPU with parallel instructions, AOG attempts to create an architecture that turns the inner loops into an expression that processes each neighborhood row in parallel. Section 4 treats this parallel architecture case.

[17] Represented as a superscript 2 in the idealized representation and as "**" in the AST.

3.3.5 Localization Results

Upon completion of the speculative refinement phase, a follow-on phase (**CodeGen**) converts localization tags into AST loop forms. At this point in the generation process, the Sobel edge detection expression will be converted into the AST expression:

```
(_forall (idx32 idx33)
   (_suchthat  (_member idx32 (_range 0 (- m 1)))
               (_member idx33 (_range 0 (- n 1))))
   (=    (aref b idx32 idx33)
         (sqrt (+ (* t1 t1) (* t2 t2)))))
```

This form is a context that will trigger the two still pending but deferred transforms that were created by the RIS optimization. They will insert the temporary variable assignment statements at the beginning of the loop body, resulting in the new form:

```
(_forall (idx32 idx33)
   (_suchthat  (_member idx32 (_range 0 (- m 1)))
               (_member idx33 (_range 0 (- n 1))))
   (= t1 (⊕ (aref a idx32 idx33) s))
   (= t2 (⊕ (aref a idx32 idx33) sp))
   (= (aref b idx32 idx33)
      (sqrt (+ (* t1 t1) (* t2 t2)))))
```

Subsequent phases will further refine this form by in-lining definitions[18] for the convolution operator (⊕) inner loop, which is expressed in terms of the variables **idx32** and **idx33** as well as calls to methods of **s** and **sp** (e.g., the **w** method of **s** and **sp**). Recursive in-lining will further replace these method calls by their definitions. The inlining phase is followed by a series of phases that apply TD transformations to architecturally shape the code so that its operational behavior is better tailored to its computational environment. (See the next section.) The final code produced for a CPU without parallel instructions is:

```
for (idx32=0; idx32 < m; idx32++)                    /* No parallelism*/
    {im1=idx32-1; ip1= idx32+1;
     for (idx33=0; idx33 < n; idx33++)
         { if (idx32==0 || idx33==0 ||
                idx32==m-1 || idx33==n-1)    /*Neighborhood Off edge?*/
               then b[idx32, idx33] = 0;                /* Off edge */
               else {jm1= idx33-1; jp1 = idx33+1;   /* Not off edge */
                     t1 = a[im1,jm1]*(-1)+a[im1,idx33]*(-2) +
                          a[im1,jp1]*(-1)+a[ip1,jm1]*1 +
                          a[ip1,idx33]*2+a[ip1,jp1]*1;
                     t2 = a[im1,jm1]*(-1)+a[idx32,jm1]*(-2) +
                          a[ip1,jm1]*(-1)+a[im1,jp1]*1 +
```

[18] Operator and method definitions are expressed as pure functions to simplify their manipulation.

```
                        a[idx32,jp1]*2+a[ip1,jp1]*1;
                  b[idx32,idx33] = sqrt(t1*t1 + t2*t2 )}}}
```

This result requires about 340 total transformation applications.

4 Architectural Shaping

4.1 Exogenous Constraints on Code Form

Up to this point, the paper has addressed the search space explosions that arise from the interactions of domain properties and refinement constraints (e.g., the refinement choice to implement a container as a linked list constrains the choices of search algorithms). These are endogenous constraints in that they arise from the essence of the computation itself and not from the nature or constraints of the external environment. Endogenous constraints act like a set of simultaneous logical equations, the solution of which is the code that achieves the DSL specification while simultaneously obeying all logical constraints. Computational correctness dictates that these constraints must be met. However, there are other kinds of less rigid requirements, influences and opportunities that suggest but do not require changes to the code's operational properties (e.g., the opportunity for parallel computations). Could the code be reorganized to better interact with another piece of software such as network software, middleware, user interface, data base management system, etc? Could the code be reorganized to exploit hardware parallelism? These are all "constraints" in the broadest sense and since they arise mostly because of the computational environment, they are called exogenous constraints. Like endogenous constraints, the exogenous constraints introduce search space explosions because there are so many alternative ways in which a computation can be reorganized and so many constraints among the individual reorganization steps. However, before examining ways to control and limit this explosion, consider the concrete result of AOG reorganizing the Sobel example to exploit hardware parallelism.

In contrast to the target code produced by AOG for a CPU without parallel instructions (above), consider how AOG alters this code to exploit parallel instructions such as the MMX instructions of the Pentium^{TM} processor. For this case, AOG will produce code[19] that looks quite different:

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

[19] In the name of compactness, the examples from here on will revert to the short idealized names for generated variables, e.g., **i** and **j** rather than **idx32** and **idx33**.

```
                                  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)**, **padd2 ((x0, x1) , (x2, x3)) = (x0+x2, x1+x3)**, and **unpackadd((x0, x1)) = (x0+x1).** These routines lend themselves to direct translation into MMX instruction sequences. In this example, the neighborhood objects **s** and **sp** have become pure data arrays to exploit the MMX instructions. Notice that the special case that tests to see if the template is hanging over the edge of the image (i.e., "**if (i==0 || j==0 || i==m-1 || j==n-1)…** ") has completely disappeared. Transformations have split the main loop on that test, turning the single loop of the previous version into five loops by 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-directed optimization phases. Up to that stage, the transformations that fire are the same, resulting in two interim program forms that are the same except for the tags. This version requires about 310 transformation applications.

4.2 Using Domain Knowledge in Architectural Shaping

So, how can AOG accomplish such a significant difference? The short answer is that AOG retains domain knowledge in the form of tags attached to the component parts and applies that domain knowledge by invoking the transformations named in those tags. To understand this strategy, consider the desired architecture and the domain knowledge that can be brought to bear to arrive at that architecture.

In order to exploit MMX instructions, the generated code needs to have some important architectural properties. First, the weights need to be formed into a vector to exploit the vector processing of the MMX instructions. Second, the neighborhood loop body needs to be branch free so that the vector processing instructions will not be interrupted by branch instructions.

Now, consider the domain knowledge that is available for accomplishing these architectural goals. The person who defines the neighborhood **s** knows that it will be used in some DSL expression containing a convolution operation, for example

```
(t1 = (a ⊕ s)).
```

This expression will translate into some form that is conceptually equivalent to

```
{∀i,j: (t1[i,j]: t1[i,j] =
          (∑p,q: (a[i+p,j+q] * w.s(a[i,j],m,n,p,q))))}
```

where the loop over the **m** by **n** image **a** (i.e., ∀**i,j**) is introduced by the control localization rules, the loop over the neighborhood (i.e., ∑**p,q**) is introduced by the definition of the ⊕ operator, and the temporary image variable **t1** is introduced by the RIS optimization as a temporary holder of the computation result. When the **w.s** (weight) method is authored, neither the ∀**i,j** loop nor the ∑**p,q** loop have been generated but the author of the method knows that loops of this form will exist even though their detail structure will not be known until some specific DSL expression (e.g., **(a ⊕ s)**) has been translated. Such knowledge is highly domain-specific and, in the context of the **w.s** method, suggests how to reshape the **w.s** definition and its future context to exploit an MMX architecture.

The body of the **w.s** definition has the conceptual form

```
if the neighborhood is hanging off the image's edge
      then 0
      else compute w as a function of p and q
```

The off-edge test depends on the indexes **i** and **j** but not on **p** or **q** and therefore, the test should be moved outside of the **p** and **q** loop (by distributing the loop over the **if** statement) to establish some of the enabling conditions for a later transform that splits up the **i**, **j** loop to avoid bus stalls (i.e., **_SplitLoopOnCases**). The author of the method will add a tag to the **if** statement that will schedule a transformation named **_PromoteConditionAboveLoop**, which distributes the **p** and **q** loop over the **if** to help enable **_SplitLoopOnCases**.

Promoting the condition above the loop is motivated by the desire to eliminate the off-edge test altogether. The author knows that if the off-edge condition predicates can be incorporated into the loop control logic for the **i**, **j** loop, the special case logic will become a separate set of loops and the branching logic will disappear from the body of the **i**, **j** loop. This will eliminate branch induced bus stalls as the data flows onto the CPU bus. All of this will increase the parallelism in the computation. Thus, the author of the method also adds a tag that will trigger the **_SplitLoopOnCases** transformation. It will attempt to incorporate the off-edge condition into the loop control logic thereby forming separate loops to perform the special case computations.

Finally, the method author recognizes that the **w** values need to be formed into an array so that a whole row of neighborhood weights can be used as input to a parallel computation. Thus, the **else** branch is tagged to invoke the **_MapToArray** transformation, which will form such an array of values and change the code using **w** accordingly. Finally, the author of the inner convolution loop (the **p**, **q** loop) needs to tag that loop so that it is reshaped to exploit the MMX instructions. This is accomplished by a tag on the definition of the convolution's inner loop. The tag will invoke the **_MMXLoop** transformation to do the reshaping.

Thus, this strategy produces a set of definitions tagged with cooperating transformations that will reshape the code into an MMX form, and do so without deep analysis or search. These tags capture the domain knowledge that is available at the time the components are authored. But what about a non-MMX contexts? How does the AOG system deal with differing contexts (e.g., MMX vs non-MMX)? Simply put, it allows choice among separately tagged component versions for different contexts by using an analog of C's **ifdef**. Once the contextual constraints are chosen, the **ifdef** analog chooses the definitions that are specific to those contextual constraints. The tags on those definitions will then shape the generated code to fit the selected context.

However, one issue remains. How are the transformations invoked so that the individual steps in the reshaping process happen in the proper order? The answer is that TD tags are triggered based on events. The TD tag format is **(_on *event TDTransformCall*)**. Each TD tag contains an event expression (i.e., ***event***) that tells it when to trigger the call to the transformation (i.e., ***TDTransformCall***). The events can be preplanned, named stages that may sequence transformations according to an abstract script. Alternatively, the events may be unscriptable opportunistic events caused by other transformation or generator actions (e.g., substitution of an expression). The next section illustrates both kinds.

4.3 Example Tag-Directed Transformations Exploiting Parallelism

Now, the steps in refining **(= t1 (a[i,j] ⊕ s))**[20] and its role in the global optimizations will be sketched in a bit more detail. A refinement phase named **Formals** will inline definitions for ⊕ and the methods of **s** that ⊕ uses. The definition for the MMX version of the ⊕ operator with a type signature of "**(⊕ image [iterator, iterator], neighborhood)**" is defined as:

```
(DefComponent BConvXImageArrayXNeighborhood
      (⊕ ?a[?i ?j] ?s "bind ?m & ?n to dimension fields of ?a"
                      "generate ?p and ?q names from ?s")
      (_sum (?p ?q)
              (_suchthat (_member ?p (prange[21] ?s ?a[?i ?j]))
                         (_member ?q (qrange  ?s ?a[?i ?j])))
              (*   ?a[(row ?s ?a[?i ?j] ?p ?q)
                       (col ?s ?a[?i ?j] ?p ?q)]
                  (w ?s ?a[?i ?j] ?m ?n ?p ?q))
              (tags (_on (RestructLoop 1) (_MMXLoop ?p ?q))))[22]
```

This definition is simplified to eliminate many of the variations and details (e.g., type checking) of the real component, thereby making it more intuitive. The parameter list of a component is a pattern and in reality, more complex than is shown here. In particular, the parameter pattern portion that binds **?m** ,**?n**, ?**p** and **?q** is expressed in the example in English pseudo-code that eliminates some of the complexity of the actual pattern expression. To make the connections even more obvious, the example will use pattern variables that directly correspond to the target program variables that they will be bound to, e.g., the pattern variable **?a** will be bound to the target program variable **a**. Further, the definition uses a bit of syntactic sugar to make some of the pattern elements more intuitive (e.g., we use **a[i j]** for array references instead of the actual AST representation of **(aref a i j)**). However, in a concession to the true AST format, the example expresses the operator and method expressions in a LISP-like, space-delimited prefix form, i.e., **(⊕ a[i j] s)** instead of **(a[i,j] ⊕ s)**.

[20] Since **(= t2 (a[i,j] ⊕ sp))** behaves analogously, its final form is analogous to **(= t2 (a[i,j] ⊕ s))** but differs in the weight constants, the generated variable names and some structural differences due to differing partial evaluation results induced by differing constants.

[21] In the general case, **prange** and **qrange** computations may depend upon the properties of **a** and **s** (e.g., **m, n, p, q**) in addition to the neighborhood indexes (e.g., **i** and **j**), for this example the superfluous arguments are dropped to keep it simple.

[22] Defcomponent is AOG's way of defining method-like entities. It is an expression of the form **(DefComponent *MethodName* (*Object* . *ParameterPat*) *[*Pre: *PreName] [*Post: *PostName] Body*)** and is converted into a transformation equivalent to **(⇒ *MethodName* Formals *Object EnhancedPattern Body PreName PostName*)** where **Formals** is the phase where operator and method inlining occurs, and ***EnhancedPattern*** is a pattern automatically derived from **(*MethodName Object . ParameterPat*)**. The enhancements are added for use by the AOG system.

The definition of ⊕ creates the **(?p ?q)** summation loop using **?s**'s methods (i.e., **prange**, **qrange**, **w**, **row** and **col**) to compute the neighborhood values: 1) the ranges of **?p** and **?q**, 2) the weights of neighborhood positions, and 3) the row and column positions with respect to **?a**. The **_MMXLoop** transformation will eventually reshape the loop body into an MMX friendly form. It will be triggered during the **RestructLoop** phase after any other **RestructLoop** transforms with indexes that are less than 1 (e.g., 0).

When this component is applied to **(⊕ a[i j] s)**, it will produce the binding list **((?s s) (?p p) (?q q) (?a a) (?m m) (?n n) (?i i) (?j j))** and rewrite **(⊕ a[i j] s)** as

```
(_sum (p q)
       (_suchthat (_member p (prange s a[i j]))
                  (_member q (qrange s a[i j])))
                  (*   a[(row s a[i j] p q)
                         (col s a[i j] p q)]
                       (w s a[i j] m n p q))
       (tags (_on (RestructLoop 1) (_MMXLoop p q)))
```

The methods of **s** will be recursively inlined. Both range methods reduce to **(_range −1 1)**, the **row** method expression reduces to **(+ i p)** and the **col** method expression reduces to **(+ j q)**.

The **w** method is more interesting. Its definition (slightly simplified) is:

```
(Defcomponent W (s ?a[?i ?j] ?m ?n ?p ?q)
  :pre gensignal

  (if
     (||   (== ?i 0) (== ?j 0)
           (== ?i (- ?m 1)) (== ?j (- ?n 1))) /*  Off Edge? */
     (then 0)                                  /*Special Case*/
     (else                                     /*Default Case*/
        (if (&& (!= ?p 0) (!= ?q 0)))
```

```
       (then ?q)
       (else
         (if (&& (== ?p 0) (!= ?q  0)) (then (* 2 ?q)) (else 0)))
       (tags
         (_on MigrationOfMe (_MapToArray ?p ?q)
                            (_Post ?signal1)
                            (_Post ?signal2))))))
(tags (_on ?signal1 (__PromoteConditionAboveLoop ?p ?q))
      (_on ?signal2 (_MergeCommonCondition))
      (_on (RestructLoop 0) (_SplitLoopOnCases))))
```

When this component's parameter pattern is matched against **(w s a[i j] m n p q)** from the **p** and **q** loop body, it will produce the binding list **((?p p) (?q q) (?a a) (?m m) (?n n) (?i i)) (?j j) (?signal1 signal84) (?signal2 signal85))** where the two unique signal names are generated by the pre-routine **gensignal**. They will be used to sequence the transforms.

These signals are introduced because of a little wrinkle – an ordering dependency. **_MapToArray** must execute before **_PromoteConditionAboveLoop** lest it cause mischief to **_MapToArray** 's enabling conditions in an attempt to set up its own enabling conditions. For the same reason, the transform **_MergeCommonCondition**, which tries to establish additional enabling conditions for **_SplitLoopOnCases** (by combining common off-edge tests), must also execute after both **_MapToArray** and **_PromoteConditionAboveLoop**. To assure the proper order of execution, the designer schedules these two transforms on the signals posted after completion of **_MapToArray** by the two **_Post** transforms.

The act of inlining the definition of **w** generates a **MigrationOfMe** event, which indicates subtree movement, for any tag in **w**'s definition subtree that is waiting on this event. This event will cause **_MapToArray** to be scheduled as the first TD transform. **_MapToArray** replaces the **if** expression to which it is attached with a reference to a newly created vector name **s[p,q]**, which it then sets about defining. First, it formulates the data declaration using the subtree to which it is attached as the body of a loop that will generate values for the newly created vector:

```
int s[(prange s a[i j]) (qrange s a[i j])] =
(_forall (p q) (_suchthat (_member p (prange s a[i j]))
                          (_member q (qrange s a[i j])))
         (if  (&& (!= p 0) (!= q 0))
              (then q)
```

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

After substitution of the definitions of `prange.s` and `qrange.s` and partial evaluation of the whole expression, it simplifies to

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

This is incorporated into a pending scope that will hold this definition (and perhaps others) until a later phase when they will be inserted at the proper place in the program.

Upon completion of **_MapToArray,** the two _**Post** transformations run, posting signals **signal84** and **signal85**, which will cause scheduling of the transformations that are waiting on those signals. The **(p q)** loop now has the form

```
  (_sum (p q)
   (_suchthat (_member p (_range -1 1)) (_member q (_range -1 1)))
   (* a[(+ i p),(+ j q)]
      (if (|| (== i 0) (== j 0)
              (== i (- m 1)) (== j (- n 1)))          /*Off Edge?*/
          (then 0)                                    /*Special Case*/
          (else s[p q])                               /*Default Case*/
             (tags (_on signal84 (_PromoteConditionAboveLoop p q))
                   (_on signal85 (_MergeCommonCondition))
                   (_on (RestructLoop 0) (_SplitLoopOnCases))))
     (tags (_on (RestructLoop 1) (_MMXLoop p q)))))
```

The **signal84** event will cause _**PromoteConditionAboveLoop** to run. In order to establish its own enabling conditions, which require the **if** statement be the first statement or operator in the body of the **(p q)** loop, it will call another transformation that distributes the **(* a[i+p,j+q]...)** expression over the **if**. This produces a **then** clause of **(* a[(+ i p) (+ j q)] 0[23])** which with partial evaluation becomes 0 and an **else** clause of **(* a[(+ i p) (+ j q)] s[p q])**. It next finds the **(= t1 ...)** assignment surrounding it and also distributes that over the **if**. After these transforms have run, the expression is reduced to

[23] This zero started out as a summation loop just after distribution but partial evaluation simplified it.

```
(if (|| (== i 0) (== j 0)
         (== i (- m 1)) (== j (- n 1)))      /*Off Edge?*/
    (then (= t1 0))                          /*Special Case*/
    (else                                    /*Default Case*/
      (= t1
         (_sum (p q)
               (_suchthat (_member p (_range -1 1))
                          (_member q (_range -1 1)))
               (* a[(+ i p) (+ j q)] s[p q])
               (tags (_on (RestructLoop 1) (_MMXLoop p q))))))
    (tags (_on (RestructLoop 0) (_SplitLoopOnCases))))
```

Next, **_MergeCommonCondition** runs to establish some more enabling conditions for loop splitting. A discussion of these details is beyond the space available in this paper. Once the array has been created and the edge test promoted above the **(p q)** loop, the scheduling queue is empty so, the next phase in the phase list – **RestructLoop** – is posted. It will trigger **_SplitLoopOnCases**, which will restructure the **(i j)** loop that surrounds the expression shown above.

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 separate version of the **(i j)** loop, thereby producing the five loops in the MMX code shown earlier. **_SplitLoopOnCases** checks the enabling conditions, deconstructs both the loop control information and the branching test, and reformulates the single loop structure into five loops:

```
(_forall (i j) (_suchthat (_member i (_range 0 (- m 1)))
                          (_member j (_range 0 (- n 1)))
                          (== i 0))
               (= b[i,j] 0))
(_forall (i j) (_suchthat (_member i (_range 0 (- m 1)))
                          (_member j (_range 0 (- n 1)))
                          (== j 0))
               (= b[i,j] 0))
(_forall (i j) (_suchthat (_member i (_range 0 (- m 1)))
                          (_member j (_range 0 (- n 1)))
                          (== i (- m 1)))
```

```
                (= b[i,j] 0))
  (_forall (i j) (_suchthat (_member i (_range 0 (- m 1)))
                            (_member j (_range 0 (- n 1)))
                            (== j (- n 1)))
                (= b[i,j] 0))
  (_forall (i j) (_suchthat (_member i (_range 0 (- m 1)))
                            (_member j (_range 0 (- n 1)))
                            (!= i 0) (!= j 0)
                            (!= i (- m 1))(!= j (- n 1)))
                ... default case (p q) loop ...)
```

To simplify the generated control expressions, `_SplitLoopOnCases` invokes some lightweight inference using AOG's built-in pattern language. The inference step uses a set of rules that recognize the idiomatic iteration patterns associated with specific simplication strategies. For example, suppose that the control variable `i` is really a fixed constant (i.e.,

`(_Suchthat (_member i (_range 0 (- m 1))) ... (== i 0))`). This engenders elimination of the control variable `i` from the loop control altogether and the substitution of `0` everywhere `i` appears in the loop body. That is, `(= b[i,j] 0)` would become `(= b[0,j] 0)`. The overall result is the form shown at the start of section 4.

It is important to observe that this overall shaping process is largely search and analysis free because domain-specific information is used to plan the global optimizations. What transforms to run, when to run them, and what other prepatory transforms are required are all details that are mostly determined at the time that components are entered into the reusable library. Any conventional optimization methodology would have to do a substantial amount of analysis and search to determine which transforms to run and what order to run them in.

5 Related Research

Good general sources for some topics in this paper include the following: generative programming [5-6, 17]; transformations and meta-programming [17, 19, 35, 37, 44]; pattern matching [23, 43]; and LISP, CLOS and Prolog [14, 23, 24, 28, 31, 40].

Related areas of work include compiler building and language processing systems. Some examples are the ASF+SDF compiler system [41, 42], the DMS maintenance system [4], the Stratego system [43], and the TXL transformation system [15]. As a general characterization, such systems desire to specify languages and their processing with various domain-specific abstractions (e.g., syntactic or semantic domain languages) and from those generate working language processors (e.g., compilers, analyzers, maintainers, etc.). The most striking difference between these systems as a group and AOG is AOG's emphasis on preserving and using problem domain knowledge in translation and optimization. Indeed, AOG provides a specialized control structure (i.e., AOG's unique tag-directed transformations) and machinery specifically designed for this job. By contrast, this group of systems tends to provide the most tools and domain languages for more conventional translation and transformation jobs, e.g., parsing and expression rewriting. AOG has been used for but does not strongly emphasize parsing. While AOG has been used to parse large programs (e.g., AOG parses itself, its domain abstractions and some DSLs), the parsing of text-based programs is a minor goal for AOG. Its parsing machinery is based on AOG's general purpose pattern engine and language, which is not highly specialized to parsing. AOG's pattern language has a procedural orientation (i.e., it's Prolog-like) and has the ability to express an arbitrary algorithms (i.e., it is Turing complete). Architecturally, it is an interpretive, user-extensible pattern matcher with backtracking. When AOG requires highly domain-specific languages (e.g., BNF-based grammar specifications or type inference rules), they are built as distinct domain languages that are translated into this general purpose pattern language.

A final difference is that AOG's PD transformations can be inherited. For example, some PD transformations are stored on types, and more specifically, on the most general type that subsumes all subtypes to which the transformation may be applied. In summary, the major differences between these systems as a group and AOG are: 1) AOG's general emphasis on preserving and applying domain knowledge to translation and optimization, 2) AOG's unique tag-directed control structure, 3) AOG's reduced emphasis on parsing, 4) AOG's general purpose pattern language and matcher, and 5) AOG's use of PD transformations with inheritance.

AOG bears a strong relation to and uses ideas from Neighbors work (e.g., DSL to DSL refinement and intra-DSL optimization phases). [32-34] The main differences are: 1) AOG's use of metaprograms aimed at narrowly specific generation problems (e.g., localization), 2) the fact that the AOG pattern-directed transformations are organized into a two-dimensional space of ***object*** and ***phase***, which determines which transformations are candidates for execution (i.e., which ones are visible), 3) AOG's use of transformations with inheritance, 4) AOG's control regime that provides scripts of explicitly named phases each of which defines a narrow translation job, and 5) the tag-directed control regime for architectural shaping optimizations.

The work bears a conceptual relationship to Kiczales' Aspect Oriented Programming (AOP) in the common emphasis on non-conventional architectural structures (e.g., aspects) but the translation machinery appears to be different. [18, 29] AOP's translation mechanism does not use a tag-directed control regime. In contrast, the AOG's tags retain domain knowledge, anticipate optimizations, are distributed over the program, are triggered by events, and may 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. [2, 3, 6, 17] Batory optimizes type equations to choose components from which to assemble custom classes and methods. AOG inlines and interweaves the bodies of methods invoked by compositions of method calls (i.e., DSL expressions). Thus, Batory's generation focus is at the class creation level and AOG's is at the instance application level.

AOG and Doug Smith's work are similar in that they make heavy use of domain-specific information in the course of generation. [38, 17] 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 does highly specialized reasoning about loop limits). However, partial evaluation (a form of inference) [17, 25, 44] is heavily used in AOG, which is how three level if-then-else expressions (which are interweavings of several neighborhood and operator definitions) get simplified to expressions like "**`a[im1,j]*(-2)`**".

The organization of the transformations into goal driven stages is conceptually similar to the work of Boyle, et al [13, 21]. However, Boyle's phases are implicit and built into the transformation metaprogram. By contrast, AOG uses lists of explicitly named phases that act like scripts and can be altered or extended by the user for different contexts. Further, the AOG work differs in that it uses domain-specific information to associate TD transformations tags with reusable components as early in their life as possible to eliminate search for transformations during the reshaping phases.

The pattern language is similar to the work of Wile [46, 47] and Crew [16]. 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 pattern language in that it is influenced by logic programming [14, 31]. However, ASTLOG's architecture is driven by program analysis objectives. It is a batch-oriented model that operates on a set of object files created by compile and link operations. Such a model is not well suited to dynamic manipulation and change of the AST under the control of a transformation-based generator. In addition, AOG's pattern language is distinguished from both ASTLOG and classic Prolog [14, 31] in that it does *mostly local reasoning* on the AST. That is, rather than operating on a large global data base all of which is always accessible, AOG's "data base" (or focus) is some specified locale within the AST.

There are a variety of other connections that are beyond the space limitations of the paper. For example, there are relations to other generators like SciComp's [26], Intentional Programming [17], meta-programming and reflection [17, 37], formal synthesis systems (e.g., Specware) [17, 39], deforestation [45], the connection of goals and strategies to transformations [20] and other procedural transformation systems [30] (e.g., Refine). The differences are greater or lesser across this group and broad generalization is hard. However, the most obvious broad difference between AOG and most of these systems is AOG's use of tag-directed transformations, which operate in the optimization domain and are triggered based on optimization-specific events. This makes the AOG control structure unusual, allowing planned, key optimizations to be attached to the reusable components they will optimize. Their effect is interleaved with opportunistic optimizations and partial evaluation simplifications. The overall optimization process behaves like an abstract algorithm where the algorithmic steps are phases 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 partly determined by tags on the AST itself. From a different point of view, the event driven transforms behave like interrupts that allow for operations whose invocation order cannot be planned in advance and whose effect is largely reorganization, architectural shaping, and simplification.

6 Evaluation of AOG

AOG is its own best customer. The main strategy for evaluating the AOG system is the use of AOG in the building of AOG. The pattern language -- the key DSL building block of both PD and TD transformations -- is used throughout AOG. At last count, there are more than 230 instances of its use within AOG. This is the deepest, most fundamental kind of reuse in AOG. Specifically, reuse of the pattern language occurs in:

- The utility routines (e.g., tag list management);

- The partial evaluator, which simplifies and canonicalizes newly created AST structures;

- The type inference rules, which generate patterns for inferring the type of an AST subtree (e.g., 90+ rules are required for basic programming language structures and the Image Algebra domain and another 14 are required for a *container domain* used by a GenVoca [2, 6] demonstration example discussed below);

- AOG's source navigator and cross reference tool[24] uses the pattern engine to parse all of AOG (20-30KLOC) for the purpose of 1) computing and caching cross references, 2) recording the kind of each item (e.g., defun, transformation, defcomponent, pattern, etc.) along with its file

[24] Used for source code segmenting, searching, navigating, analyzing and managing.

location, 3) computing an index of all items, 4) searching for individual items included in the index, and 5) rereading the source text for the navigator to display; and finally,

- The pretty printer/source code generator uses about 30 or so PD transformations to generate C from the AST.

A second dimension of evaluation addresses the issue of whether or not AOG can be applied to a wide variety of domains or is it just tailored for IA-like domains. The answer is that that it can be applied to any domain. To address this question of universality, I am implementing other DSLs for domains that are fundamentally different from IA. The first such implementation was the development of a GenVoca-like [2, 6] system on top of AOG. This required less than 40 lines of code to implement a general mechanism to manage GenVoca's *type vector* (which is a list of layered, aspect-like components that control the choice among alternative definitions applied within those layers) and two lines of code to implement GenVoca's AST traversal strategy. Of course, the definition of any domain components (i.e., the reusable parts) to be translated by GenVoca requires additional work regardless of which GenVoca is being used. The amount of work required to define those domain components (e.g., container domain components) for the experimental AOG-GenVoca is roughly comparable to that required for the original GenVoca system. Specifically, the container domain data types are defined as AOG class instances (one line of code to define each new class) and each GenVoca *component* is defined as a AOG Defcomponent that is roughly the same size as the original GenVoca components. After that, 14 type inference rules were needed to define type inference in the container domain. In addition, for each domain component that needs to introduce new C data declarations into a scope that is some indefinite distance up the AST from the point at which the component is generated, one deferred transformation rule per new data definition is needed to dynamically create a deferred transformation. These deferred transformations eventually move the data declarations and field declarations (e.g., C typedefs and C fields in a typedef) into the correct scope and into the correct definition structure, once those scopes and definition structures are created. In the latest version of AOG, these deferred transforms could be eliminated because definition generation into non-existent but planned scopes is handled by a different, automatic mechanism.

In summary, the creation of AOG-GenVoca on top of AOG was a few hours of work and the creation of the domain components for AOG-GenVoca was a couple of days or so (much of which was time spent understanding the domain). Further, the work to create new GenVoca target domains and new, reusable domain components is roughly comparable to that required by the original GenVoca. All of the domain components produced are reusable and therefore, could be applied to any target application development that used the target domain (e.g., containers). Since this exercise did not require all of AOG's capabilities (e.g., neither partial evaluation nor TD transforms were needed), the opportunity exists to create an extended GenVoca with the capability to further tune the generated code to various computing platforms by appropriately tagging the existing reusable GenVoca components with TD transform expressions. This would allow additional reuse mileage from existing GenVoca components.

Other similar experiments focusing on different transformation-based domain translators are in work.

7 Conclusions

The key ideas embodied in AOG are summarized below:

- **Control localization** contributes a generalized framework for integrating and optimizing implicit, delocalized control structures. This generalizes the loop and language specific techniques of APL [22] in that it is user extensible (e.g., to new domains) and can coordinate a range of different kinds of interdependent controls.

- **Speculative Refinement** contributes a way to incrementally propagate constraints and optimization decisions over a specific DSL expression via the dynamic creation of set of refinement transforms that are customized to the specific DSL expression.

- Dynamically created, **deferred transformations** contribute a general method for putting generated code into contexts that have yet to be generated.

- Organizing transformations into a **two dimensional memory space** (***object*** and ***phase***) reduces the number of transformations that have to be tried for each AST subtree. For the IA domain, it is generally zero, one or two. The maximum in that domain is less than ten. Allowing transformations to be stored under various kinds of objects (e.g., a translator generated symbol or a type) opens the door for inheritance of transformations as well as strategies for coordinating related design decisions among separated portions of a target program (e.g., Speculative Refinement).

- Allowing **PD transformations to be inherited** from super classes (e.g., in the type hierarchy) raises the level of abstraction of transformations.

- **TD transformations preserve and exploit domain knowledge** (e.g., the existence and relationships between image loops and neighborhood loops) in the form of tags that will orchestrate cooperating transformations, which taken as a whole, will derive desired, global architectural properties. This technique eliminates most of the computation required by more conventional optimization strategies that must discover what transformations are possible, what ordering constraints exist among them and what preparatory transformations are required. TD transformations allow a nearly search-free, distributed optimization plan to be laid out mostly in advance. The plan exploits all available knowledge (including domain knowledge) to reduce analysis and eliminate search.

- **TD transforms** usually incorporate few newly invented optimizations (they reuse known optimizations) and taken as a group, perform no overall optimization process that, in theory, could not be performed by an appropriately integrated optimization system. However, they are unique in that they allow the user to assemble **a customized set of cooperating optimizations** that are tailored to *a specific kind of DSL expression* set in the context of *a specific hardware platform*. In other words, they provide the user the ability to fill in the many optimization gaps that evolving DSLs and evolving hardware platforms are constantly introducing. The user does not have to wait until someone builds that exact new optimization system or compiler that deals with the exact DSL and the exact hardware platform. He can take matters into his own hands by writing PD transforms to translate his DSL into any general purpose language supported by the hardware (e.g., C) or even translate his DSL into a special purpose DSL provided by the target hardware (e.g., a display processor DSL). Further, by adding TD transforms, he provides a mechanism to shape the general purpose or special purpose output code to the peculiarities of the target hardware. AOG's contribution is in the search-free reuse, assembly and orchestration of well-known transformations to achieve desirable global architectures for specific computational environments.

AOG is being developed to study of the effects of new generator architectures 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.

Table 1 summarizes the problems, strategies and techniques discussed in this paper.

Table 1: Explosion Control Strategies and Techniques

Source of Explosion	**Explosion Control Strategy**	**Techniques**
Numerous **refinements** and **constraints** explode derivation pathways	**Phased DSL to DSL Refinement** – Incrementally translate from higher to lower level DSLs	Mutual exclusion of DSLs produces a **subsetting of refinement rules** that hides irrelevant ones
Complexity of generated code explodes code reorganization choices ▪ **Overly complex generated code** explodes complexities of lhs patterns ▪ **Domain-specific Optimizations** done at code level may explode search space	**Inter-Refinement Optimization Phases** – Apply specialized rules to simplify DS forms by mapping a DSL domain to itself: ▪ **Simplification** – Direct reduction of the code by removing inefficiencies without reorganizing the code ▪ **Domain-specific Optimizations** done at correct domain level may use DSL knowledge to reduce search and analysis	Mutual exclusion of domains implies a **subsetting of optimization rules** **Partial evaluation** is a metaprogram specialized to inefficiency removal, e.g., unrolling a loop over variable i may allow simplifications such as (x + i) =>(x + 0) => x **Use of domain knowledge leverages optimizations** (e.g., knowledge of ATN domain leverages ATN state removal optimization)
Implied related control structures spread across DSL expressions explode the choices in integrating those controls	**Localization** – Generate customized, integrated control expressions from the implied control structures that are dispersed across DSL expressions	**Rule-based metaprogram specialized** to merge and coordinate implied controls **Speculative Refinement** to produce customized refinements that coordinate localization constraints across expression **Explicit grouping of localization rules** by object (e.g., data type) and optimization goal (i.e., phase) to hide irrelevant rules
Global constraints require coordinated global optimizations that explode constraint propagation choices	**Architectural Shaping** – Metaprogram optimizations that reshape the computation to better fit global constraints while preserving its computational function	**Event-triggered Tag-Directed Rules** preserve domain-specific knowledge (via AST tags containing event-based rule invocations) for use in the code domain

8 References

1. David F. Bacon, Susan L. Graham, and Oliver J. Sharp, “Compiler Transformations for High-Performance Computing,” *ACM Surveys*, vol. 26, no. 4, December, 1994.

2. Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas, “Scalable Software Libraries,” *Symposium on the Foundations of Software Engineering*, Los Angeles, California, 1993.

3. Don Batory, Marty Sirkin, and Jeff Thomas, “Reengineering a Complex Application Using a Scalable Data Structure Compiler,” *ACM Sigsoft International Symposium on the Foundations of Software Engineering (FSE)*, pp. 110-120, 1994.

4. I. D. Baxter, Christopher Pidgeon, and Michael Mehlich, “DMS: Program Transformations for Practical Scalable Software Evolution,” *International Conference on Software Engineering*, 2004.

5. "The Library Scaling Problem and the Limits of Concrete Component Reuse," *International Conference on Software Reuse*, Rio de Janeiro, Brazil, November, 1994.

6. Ted J. Biggerstaff, “A Perspective of Generative Reuse, Annals of Software Engineering,” Osman Balci, ed., *Baltzer Science Publishers*, AE Bussum, The Netherlands, 1998a.

7. Ted J. Biggerstaff, “Anticipatory Optimization in Domain Specific Translation,” *International Conference on Software Reuse*, Victoria, B. C., Canada, pp. 124-133, 1998b.

8. Ted J. Biggerstaff, “Composite Folding in Anticipatory Optimization,” *Microsoft Research Technical Report*, MSR-TR-98-22, 1998c.

9. Ted J. Biggerstaff, “Pattern Matching for Program Generation: A User Manual,” *Microsoft Research Technical Report*, MSR-TR-98-55, 1998e.

10. Ted J. Biggerstaff, “Fixing Some Transformation Problems,” *Automated Software Engineering Conference*, Cocoa Beach, Florida, 1999.

11. Ted J. Biggerstaff, “A New Control Structure for Transformation-Based Generators,” *Software Reuse: Advances in Software Reusability*, Vienna, Austria, Springer , June, 2000.

12. Ted J. Biggerstaff, “Control Localization in Domain Specific Translation,” *International Conference on Software Reuse* ,2002.

13. James M. Boyle, “Abstract Programming and Program Transformation—An Approach to Reusing Programs,” *Software Reusability*, T. Biggerstaff and A. Perlis, eds., Addison-Wesley/ACM Press, pp. 361-413 , 1989.

14. W. F. Clocksin, and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, 1987.

15. James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider, “Source Transformation in Software Engineering using the TXL Transformation System,” *Journal of Information and Software Technology*, vol. 44, no. 13, pp. 827-837, October, 2002.

16. Roger F. Crew, “ASTLOG: A Language for Examining Abstract Syntax Trees” *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, 1997.

17. Krzysztof Czarnecki and Ulrich W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

18. Tzilla Elrad, Robert E. Filman, Atef Bader, (Eds.), “Special Issue on Aspect-Oriented Programming,” *Communications of the ACM*, vol. 44, no. 10, pp. 28-97, 2001.

19. Martin Feather, “A Survey and Classification of some Program Transformation Approaches and Techniques,” *Program Specification and Transformation*, Elsevier (North-Holland), IFIP, 1987.

20. Stephen F. Fickas, “Automating the Transformational Development of Software,” *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, pp. 1268-1277, November, 1985.

21. Stephen Fitzpatrick, Terence J. Harmer, Alan Stewart, Maurice Clint, and James M. Boyle, “The Automated Transformation of Abstract Specifications of Numerical Algorithms into Efficient Array Processor Implementations,” *Science of Computer Programming*, vol. 28, no. 1, pp. 1-41, 1997.

22. L. J. Guibas and D. K. Wyatt, “Compilation and Delayed Evaluation in APL,” *Fifth Annual ACM Symposium Principles of Programming Languages*, pp. 1-8, 1978.

23. Paul Graham, *On Lisp: Advanced Techniques for Common Lisp*, Prentice-Hall, 1994.

24. Paul Graham, *The ANSI Common Lisp*, Prentice-Hall, 1996.

25. Neil D. Jones, “An Introduction to Partial Evaluation,” *ACM Computing Surveys*, vol. 28, no. 3, 1996.

26. Elaine Kant, “Synthesis of Mathematical Modeling Software,” *IEEE Software*, May, 1993.

27. M. D. Katz and D. Volper, “Constraint Propagation in Software Libraries of Transformation Systems,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 3, 1992.

28. Sonya E. Keene, *Object Oriented Programming in Common Lisp: A Programmers Guide to the Common Lisp Object System*, Addison-Wesley, 1989.

29. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maede, Cristina Lopes, Jean-Marc Loingtier and , John Irwin, “Aspect Oriented Programming,” *Tech. Report* SPL97-08 P9710042, Xerox PARC, 1997.

30. Gordon B. Kotik, A. Joseph Rockmore, and Douglas R. Smith, “Use of Refine for Knowledge-Based Software Development,” *Western Conference on Knowledge-Based Engineering and Expert Systems*, 1986.

31. John Malpas, *Prolog: A Relational Language and it Applications*, Prentice-Hall, 1987.

32. James M. Neighbors, “Software Construction Using Components,” PhD dissertation, University of California at Irvine, 1980.

33. James M. Neighbors, “The Draco Approach to Constructing Software From Reusable Components,” *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp 564-573, September, 1984.

34. James M. Neighbors, “Draco: A Method for Engineering Reusable Software Systems,” *Software Reusability*, T. Biggerstaff and A. Perlis eds., Addison-Wesley/ACM Press, pp. 295-319, 1989.

35. Helmut A. Partsch, *Specification and Transformation of Programs*, Springer-Verlag, 1990.

36. Gerhard X. Ritter and Joseph N. Wilson, *Handbook of Computer Vision Algorithms in the Image Algebra*, CRC Press, 1996.

37. Tim Sheard, “Accomplishments and Research Challenges in Meta-Programming,” *SAIG 2001 Workshop*, Florence, Italy, September, 2001.

38. Douglas R. Smith, “KIDS-A Knowledge-Based Software Development System,” *Automating Software Design*, M. Lowry and R. McCartney, eds., AAAI/MIT Press, pp. 483-514, 1991.

39. Y. V. Srinivas, “Refinement of Parameterized Algebraic Specifications,” *Proceedings of a Workshop on Algorithmic Languages and Calculii*, R. Bird and L. Meertens, eds., Alsac FR. Chapman and Hill, pp. 164-186, 1997.

40. Guy L. Steele Jr., *Common Lisp: The Language (Second Edition)*, Digital Press, 1990.

41. M. G. J. Van Den Brand, J. Heering, P. Klint and P. A. Olivier, “Compiling Language Definitions: The ASF+SDF Compiler,” *ACM TOPLAS*, vol. 24, no. 4, pp. 334-368, July, 2002

42. M. G. J. Van Den Brand, P. Klint, and Jurgen J. Vinju, “Term Rewriting with Traversal Functions,” *ACM TOSEM*, vol. 12, no. 2, pp. 152-190, April, 2003.

43. Eelco Visser, “Strategic Pattern Matching. In: Rewriting Techniques and Applications,” *RTA ’99*, Trento, Italy, Springer-Verlag, July, 1999.

44. Eelco Visser, “A Survey of Strategies in Program Transformation Systems,” *Workshop on Reduction Strategies in Rewriting and Programming (WRS ’01)*, B. Gramlich and S. L. Alba, eds., Utrecht, The Netherlands, May, 2001.

45. Philip Wadler, “Deforestation: Transforming Programs to Eliminate Trees,” *Journal of Theoretical Computer Science*, vol. 73, pp. 231-248, 1990.

46. David S. Wile, “Popart: Producer of Parsers and Related Tools,” USC/Information Sciences Institute Technical Report, Marina del Rey, California, 1994. (http://mr.teknowledge.com/wile/popart.html)

47. David S. Wile, “Toward a Calculus for Abstract Syntax Trees” *Proceedings of a Workshop on Algorithmic Languages and Calculii*, R. Bird and L. Meertens, eds., Alsac FR. Chapman and Hill, pp. 324-352, 1997.