Control Localization in Domain Specific Translation

Ted J. Biggerstaff

tbiggerstaff@austin.rr.com

Abstract. Domain specific languages (DSLs) excel at programming productivity because they provide large-grain composite data structures (e.g., a graphics image) and large-grain operators for composition (e.g., image addition or convolution). As a result, extensive computations can be written as APL-like one-liners that are equivalent to tens or hundreds of lines of code (LOC) when written in a conventional language like Java. The problem with DSL specifications is that they de-localize the control components making un-optimized machine translations significantly slower than for the human optimized equivalent. Specifically, operations on DSL composites imply multiple control structures (e.g., loops) that process the individual elements of large-grain composites and those multiple, implicit control structures are distributed (i.e., de-localized) across the expression of operators and operands. Human programmers recognize the relation among these distributed control structures and merge them to minimize the redundancy of control. For example, merged control structures may perform several operations on several large-grain data structures in a single pass. This merging is the process of *control localization*. This paper discusses strategies for automating localization without large search spaces and outlines a domain specific example of transformation rules for localizing control. The example is based on the localization method in the *Anticipatory Optimization Generator* (AOG) system [3-8].

1 Introduction

1.1 The General 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 (e.g., field, integer, real, character, etc.) with respect to some conventional programming language. Thus, operators on such large-grain data structures imply some kind of extended control structure such as a loop, a sequence of statements, a recursive function, or other. As one composes large-grain operators and operands together into longer expressions, each subexpression implies not only some atomic computations (e.g., pixel addition) that will eventually be expressed in terms of atomic operators (e.g., +) and data (e.g., integers), but it also implies some control structure to sequence through those atomic computations. Those implied control structures are typically distributed (i.e., de-localized) across the whole expression.

For example, if one defines an addition operator[1] 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, suppose that the pixels are integers (i.e., `A` and `B` are grayscale images). Then the expression `(A + B)` implies a two dimensional (2D) loop over `A` and `B`. Subsequent squaring of each pixel in `(A + B)` implies a second 2D loop. Human programmers easily identify this case and merge the two loops into a single 2D loop.

This 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 design tricks to avoid the generator's search space from becoming intractably large. More complex operators hint at some of this complexity. For example, consider a *convolution operator*[2] $\oplus$, which performs a sum of products of pixels and weights. The pixels come from image neighborhoods, each of which is centered on one of the image's pixels, and the weights come from the neighborhood definition, which associates a weight value with each relative pixel position in the neighborhood. The weights are defined separately from $\oplus$. Suppose the weights are defined in a domain object `S`, which is called a *neighborhood* of a pixel, where the actual pixel position defining the center of the 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 provide, among other data, the neighborhood size and the definition of the method for computing the weights. The $\oplus$ 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 $\oplus$ 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 localization. In fact, localization is a subproblem of the more general problem of *constraint propagation* in domain specific translation, which is NP Complete [15]. 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 dramatically. The

[1] The meta-language definitions used are: 1) `courier for code`, 2) *`courier italics for meta-code`*, and 3) <u>*`courier italics underlined for comments`*</u>.

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

downside of using domain specific knowledge in localization is that localization in different domains may have different rules. But that is a small price to pay because the strategy transforms a general problem that has the potential of an exploding solution space into a specialized problem that does not.

While this paper will focus on the Image Algebra (IA) domain [22], the localization problem is universal over most complex domains. Localization is required when the domain's language separates 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), 4) data aggregations (e.g., fields aggregated into records) and others. AOG implements localization rules for two control domains: 1) loops over images and similar large grain data structures and 2) iterations induced by fields within records.

2 The Technology

2.1 An Example Mini-Domain

To provide a concrete context for discussing the issues of localization, this section will define a subset of the Image Algebra [22] as a mini-DSL for writing program specifications.

Domain Entity[3]	**Description**	**Definition**	**Comments**
Image	A 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 size [(imax – imin +1) by (jmax – jmin +1)] s.t. $imin \leq i \leq imax$ and $jmin \leq j \leq jmax$	Subclasses include images with gray-scale or color pixels. To simplify the discussion, assume all images have the same size.
Neighborhood	A matrix overlaying a region of an image centered on	$W(S) = \{ w_{p,q} : w_{p,q}$ is a numerical weight$\}$	Neighborhoods are objects with methods. The methods

[3] See 22 for a more complete and more formal definition of this domain.

	an image pixel such that the matrix associates a numerical weight with each overlay position	weight} where S is a neighborhood of size [(pmax – pmin + 1) by (qmax – qmin + 1)] s.t. pmin ≤ p ≤ pmax and qmin ≤ q ≤ qmax	define the weights computation, neighborhood size, special case behaviors, and methods to compute a neighborhood position in terms of image coordinates.
Convolution	The convolution (A ⊕ S) applies the neighborhood S to each pixel in A to produce a new image B = (A ⊕ s)	$(A \oplus S) = \{\forall_{i,j} (b_{i,j} : b_{i,j} = (\Sigma_{p,q} (w_{p,q} * a_{i+p,j+q}))\}$ where $w_{p,q} \in W(S)$, p and q range over the neighborhood S; i and j range over the images A and B	Variants of the convolution operator are produced by replacing the $\Sigma_{p,q}$ operation with $\Pi_{p,q}$, $Min_{p,q}$, $Max_{p,q}$, & others and the * operation with +, max, min & 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., $(A+B) = \{\forall_{i,j} (a_{i,j} + b_{i,j})\}$	

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

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

if it is not. Given these definitions, one can write an expression for the Sobel edge detection method [22] that has the following form:

$$B = [(A \oplus S)^2 + (A \oplus SP)^2]^{1/2}$$

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). What technology will be employed to specify localization? AOG uses a pattern-directed transformation regime.

2.2 Pattern-Directed Control Regimes

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

`Pattern` $\Rightarrow$ *`RewrittenExpression`*

The left hand side of the rule (*`Pattern`*) matches a subtree of the Abstract Syntax Tree (AST) specification of the target program and binds matching elements to transformation variables (e.g., `?operator`) in `Pattern`. If that is successful, then the right hand side of the rule (*`RewrittenExpression`*) is instantiated with the variable bindings and replaces the matched portion of the AST. Operationally, rules are chosen (i.e., triggered) based largely on the syntactic pattern of the left hand side thereby motivating the moniker "Pattern-Directed" for such systems. Beyond syntactic forms, *`Pattern`*-s may also include 1) semantic constraints (e.g., type restrictions), and 2) other constraints (often called *enabling conditions*) that must be true before the rule can be fully executed. In addition, algorithmic checks on enabling conditions and bookkeeping chores (e.g., inventing translator variables and objects) are often handled by a separate procedure associated with the rule.

One of the key questions with transformation systems is what is the control regime underlying the system. That is, what is the storage organization of rules and how are the transformations chosen or triggered? We will consider the question of storage organization in a minute but first we will look at triggering strategies. In general, control regimes are some mixture of two kinds of triggering strategies: pattern-directed (PD) triggering and metaprogram controlled triggering. PD triggering produces a control regime that looks like a search process directed mostly by syntactic or semantic information local to AST subtrees. PD searches are bias toward programming syntax and semantics mainly because the tools used are biased toward such information.

The problem with pure PD control strategies is that rule choice is largely local to an AST subtree and therefore, often leads to a strategically blind search in a huge search space. In contrast, the triggering choices may be made by a goal-driven metaprogram. Metaprograms are algorithms that operate on programs and therefore, save state information, which allows them to make design choices based on earlier successes or failures. This purposefulness and use of state information tends to reduce the search space over that of PD control. However, case combinations can still

[4] Most non-toy transformation systems use various control machinery to attempt to overcome the inefficiencies of the "global soup" model of transformations while retaining the convenience of viewing the set of transformations more or less as a set of separate transformations.

explode the search space. This paper will look at the techniques that are used to overcome these problems.

3 Localization Technology

3.1 Defusing Search Space Explosions

Since program generation is NP Complete, it will often produce large and intractable search spaces. How does AOG overcome this? AOG uses several techniques, one of which is localization. More narrowly, AOG localization reduces the search space by: 1) defining localization as a specialized optimization process with a narrow specific goal, 2) grouping the localization rules in ways that make irrelevant rules invisible, and 3) using domain knowledge (e.g., knowledge about the general design of the code to be generated) to further prune the search space.

One way to reduce the search space is by grouping transformations so that at each decision point only a small number of relevant transformations need to be tried. AOG implements this idea by grouping the rules in two dimensions: 1) first under a relevant object (e.g., a type object) and 2) then under a *phase* name. The meta-program goal being pursued at a given moment determines which objects and which phases are in scope at that moment. The phase name captures the strategic goal or job that those rules as a group are intended to accomplish (e.g., the `LoopLocalize` phase is the localization phase for the IA domain). In addition, the object under which the rules are stored 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 deducible from the resulting type of the expression plus the details of the expression. In AOG, an expression type combined with the result of a specific rule's pattern match provides the needed data flow knowledge. Thus, the localization process for a specific expression of type *X* is a matter of trying all rules in the `LoopLocalize` group of the type *X* object and of types that *X* inherits from. AOG rules provide this organizational information by the following rule format:

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

The transform's name is *XformName* and it 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. *XformName* is enabled only during the *PhaseName* phase (e.g., `LoopLocalize`). *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 rule is named `FoldZeroXform`, is stored in the `dsnumber` type structure, is enabled only in phase *`SomePhaseName`*, and rewrites an expression like `(+ 27 0)` to `27`. In the pattern, 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.

3.2 RefineComposite Rule

The IA mini-DSL will need a refinement rule (`RefineComposite`) to refine a composite image like `a` to a unique black and white pixel object, say `bwp27`, and simultaneously, introduce the elements of a loop control structure to iteratively generate all values of `bwp27`. These will include some target program loop index variables (e.g., `idx28` and `idx29`), some shorthand for the putative target program loop control to be generated (e.g., an expression `(forall (idx28 idx29) ...)`), a record of the correspondence relationship between the large-grain composite `a` and the component `bwp27` (e.g., the expression `(mappings (bwp27) (a))`, and the details of the refinement relationship (e.g., some rule `bwp27 => a[idx28, idx29]`). How would one formulate `RefineComposite` in AOG?

Given a routine (e.g., `gensym`) to generate symbols (e.g., `bwp27`), an overly simple form of this rule might look like:

```
(=> RefineComposite LoopLocalize Image `?op
    (gensym 'bwp))
```

where `?op` matches any expression of type `Image` and rewrites it as some `gensym`-ed black and white pixel `bwp27`. But this form of the rule does not do quite enough. An image instance may be represented in the AST in an alternative form – e.g., `(leaf a ...)`. The leaf structure provides a place to hang an AST node property list, which in AOG is called a *tags* list. Thus, the rule will have to deal with a structure like `(leaf a (tags` *`Prop1 Prop2`* `...))`. To accommodate this case, the rule pattern will have to be extended using AOG's "*`or`*" pattern operator, `$(por` *`pat1 pat2`* `...)`, which allows alternative sub-patterns (e.g., *`pat1 pat2`*...) to be matched.

```
(=> RefineComposite LoopLocalize Image
    `$(por (leaf ?op) ?op) (gensym 'bwp))
```

Now, `(leaf a ...)` will get translated to some pixel symbol `bwp27` with `?op` bound[5] to `a` (i.e., `{{a/?op}}`). However, the rule does not record the relationship among the image `a`, the pixel `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 pixel 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 LoopLocalize Image
    `$(por (leaf ?op) ?op) `(leaf ?bwp) `RCChores)
```

Executing this rule on the AST structure `(leaf a ...)` will create the binding list `{{a/?op} {bwp27/?bwp} {idx28/?idx1} {idx29/?idx2}}` and rewrite `(leaf a ...)` to `(leaf bwp27)`. However, it does not yet record the relationship among `a`, `bwp27`, `idx28`, and `idx29`. Other instances of images in the expressions will create analogous sets of image, pixel, 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. Most of these will be redundant. Other rules will be added that merge away these redundancies by discarding redundant pixels and indexes. To supply the data for these merging rules, the next version of the rule will need to create a shorthand form expressing the relationship among these items and add it to the tags list. The shorthand will have the form

```
(_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`. `bwp27` will eventually be refined into some array reference such as `(aref`[6] `a idx28 idx29)`. The `_member` clauses define the ranges of these control 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.

---

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

[6] `(aref a idx28 idx29)` is the AST representation of the code `a[idx28,idx29]`.

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

```
(=> RefineComposite LoopLocalize Image
    `$(por (leaf ?op) Pattern to match an image leaf
           ?op)       or image atom. Bind image to ?op.

    `(leaf ?bwp       Rewrite image as ?bwp pixel.
        (tags          Add a property list to pixel.

            (_forall    Loop shorthand introducing
              (?idx1 ?idx2) loop indexes and
                (_suchthat   loop ranges & relations.
                  (_member ?idx1 (_range minrow maxrow))
                  (_member ?idx2 (_range mincol maxcol))
                  (mappings (?bwp) (?op))))

            (itype bwpixel))) Add new type expression.

   `RCChores)          Name the pre-routine that
                        creates pixel & indexes.
```

3.3 Combining Loop Shorthands

After `RefineComposite` has executed, the loop shorthand information will be residing on the tags list of the `bwp27` leaf in the AST. But this loop shorthand is only one of a set of incipient loops that are dispersed over the expression. These incipient loops must be moved up the expression tree and combined in order to reduce redundant looping. AOG will require a set of rules to do this job. For example, one rule (`ConvolutionOnLeaves`[7]) will move the loop shorthand up to the convolution operator and another (`FunctionalOpsOnComposites`[8]) will move it up to the mathematical square function. These two rules rewrite the AST subtree[9] from

```
(** (⊕ (leaf bwp27
             (tags (forall (idx28 idx29) ...)... )) s)
    2)
```

to

```
(** (⊕ (leaf bwp27 ...) s
```

[7] Definition not shown due to space limitations.
[8] Definition not shown.
[9] These examples omit much of the actual complexity and enabling condition checking but capture the essence of the rewrites.

```
         (tags (forall (idx28 idx29) ...))) 2)
```

and then to

```
  (** (⊕ (leaf bwp27 ...) s) 2
      (tags (forall (idx28 idx29) ...)... )).
```

Eventually, the process will get to a level in the expression tree where two of these incipient loops will have to be combined to share some portions of the loop. In this case, it will share the index variables, preserving one set and throwing the other set away. For example, the rule `FunctionalOpsOnParallelComposites` will perform such a combination by rewriting[10] the AST subtree

```
  (+  (**(⊕ bwp27 s) 2
         (tags (forall (idx28 idx29) ...)... ))
      (**(⊕ bwp31 s) 2
         (tags (forall (idx32 idx33) ...)... )))
```

to

```
  (+ (** (⊕  bwp27 s) 2)
     (** (⊕  bwp31 s) 2)
     (tags (forall (idx32 idx33) ...)...)))
```

throwing away idx28 and idx29 and retaining idx32 and idx33.

This movement and combination process continues until the dispersed, incipient loop structures are localized to minimize the number of passes over images. Follow-on phases (`CodeGen` and `SpecRefine` respectively) will cast the resulting short-hand(s) into more conventional loop forms and refine intermediate symbols like `bwp27 and` bwp31 into a computation expressed in terms of the source data, e.g., `a[idx32,idx33]`. But this refinement presents a coordination problem to be solved. How will `SpecRefine` know to refine `bwp27` into `a[idx32,idx33]` instead of its original refinement into `a[idx27,idx28]`? Just replacing `bwp27` with `bwp31` in the AST during the combination rule (`FunctionalOpsOnParallelComposites`) does not work because `bwp27` may occur in multiple places in the expression due to previously executed rules. Worst yet, there may be instances of `bwp27` that are yet to appear in the expression due to deferred rules that are pending. Other complexities arise when only the indexes are shared (e.g., for 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 undergoing localization combination and coordination. For example, an RGB 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 –

[10] These examples omit details (e.g., leaf structures) when they are not essential to understanding the example.

say `a[idx64, idx65].` Each of these abstractions can potentially change through combination during the localization process. So, how does AOG assure that all of these generated symbols get mapped into a set of correctly coordinated target code symbols?

3.4 Speculative Refinements

Speculative refinement is a process of dynamically generating and modifying rules to create the correct final mapping. When executed in a follow-on phase called `SpecRefine`, these generated rules map away discarded symbols and map the surviving symbols to the properly coordinated target code symbols (e.g., array names and loop indexes).

As an example, consider the `RefineComposite` rule shown earlier. Its pre-routine, `RCChores`, will create several speculative refinement rules at various times while processing various sub-expressions. Among them are:

```
(=> SpecRule89 SpecRefine bwp27[11] `bwp27
     `(aref a idx27 idx28))
(=> SpecRule90 SpecRefine bwp31 `bwp31
     `(aref a idx32 idx33))
```

Later, the pre-routine of the `FunctionalOpsOnParallelComposites` rule chose `bwp31` to replace the equivalent `bwp27`. It recorded this decision by replacing the right hand side of rule `SpecRule89` with `bwp31`. `SpecRule89` will now map all `bwp27` references to `bwp31` which `SpecRule90` will then map to `(aref a idx32 idx33)`.

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 rules (e.g., `SpecRule89`) that are attached to AST abstractions (e.g., `bwp27`). The result is a consistent and coordinated expression of references to common indexes, images, field names (e.g., `red`), etc.

4 Other Explosion Control Strategies

AOG uses other control regimes and strategies that are beyond the scope of this paper. These include an event driven triggering of transformations called *Tag-Directed (TD)* transformation control[12] that allows cross-component and cross-domain optimizations (code reweavings) to occur in the programming language domain while re-

[11] Notice that these rules are stored on the gensym-ed objects.
[12] Patent number 6,314,562.

taining and using domain specific knowledge to reduce the search space. TD rules perform architectural reshaping of the code to accommodate external, global constraints such as interaction protocols or parallel CPU architectures. See 3 and 5-9.

5 Related Research

Central sources for many of the topics in this paper are found in [4, 11, 21, 23, 26].

This work bears the strongest relation to Neighbor’s work [18-19]. The main differences are: 1) the AOG PD rules are distributed over the two dimensional space of objects and phases, 2) the use of the specialized control regimes (including TD control) for specialized program optimization strategies, and 3) the inclusion of cross-domain optimization machinery.

The work bears a strong relationship to Kiczales' Aspect Oriented Programming [12, 16] at least in terms of its objectives, but the optimization machinery appears to be quite different.

This work is largely orthogonal but complementary to the work of Batory [1,4, 11] with Batory focusing on generating class and method definitions and AOG focusing on optimizing expressions of calls to such methods.

AOG and Doug Smith's work are similar in that they make heavy use of domain specific information [24, 11]. However, Smith relies more on generalized inference and AOG relies more on partial evaluation [11, 26].

The organization of the transformations into goal driven stages is similar to Boyle’s TAMPR [9]. However, Boyle’s stages are implicit and they do not perform localization in the AOG sense.

The pattern language [6] is most similar to the work of Wile [28,29], Visser [25] and Crew [10]. Wile and Visser lean toward an architecture driven by compiling and parsing notions, though Visser’s pattern matching notation parallels much of the AOG pattern notation. Both Wile and Visser are influenced less by logic programming than AOG. On the other hand, logic programming influences both ASTLOG and the AOG pattern language. However, ASTLOG’s architecture is driven by program analysis objectives and is not really designed for dynamic change and manipulation of the AST. In addition, AOG’s pattern language is distinguished from both ASTLOG and classic Prolog in that it does *mostly local reasoning* with a distributed rule base.

There are a variety of other connections that are beyond the space limitations of this paper. For example, there are relations to Intentional Programming [11], metaprogramming and reflection [11, 23], logic programming based generation, formal synthesis systems (e.g., Specware) [11], deforestation [27], transformation replay [2], other generator designs [13, 14] and procedural transformation systems [17]. The differences are greater or lesser across this group and broad generalization is hard. However, the most obvious general difference between AOG and most of these systems is AOG’s use of specialized control regimes to limit the search space.

References

1. Batory, Don, Singhal, Vivek, Sirkin, Marty, and Thomas, Jeff: Scalable Software Libraries. Symposium on the Foundations of Software Engineering. Los Angeles, California (1993)
2. Baxter, I. D.: Design Maintenance Systems. Communications of the ACM, Vol. 55, No. 4 (1992) 73-89
3. Biggerstaff, Ted J.: Anticipatory Optimization in Domain Specific Translation. International Conference on Software Reuse Victoria, B. C., Canada (1998a) 124-133
4. Biggerstaff, Ted J.: A Perspective of Generative Reuse. Annals of Software Engineering, Baltzer Science Publishers, AE Bussum, The Netherlands (1998b)
5. Biggerstaff, Ted J.: Composite Folding in Anticipatory Optimization. Microsoft Research Technical Report, MSR-TR-98-22 (1998c)
6. Biggerstaff, Ted J.: Pattern Matching for Program Generation: A User Manual. Microsoft Research Technical Report MSR-TR-98-55 (1998d)
7. Biggerstaff, Ted J.: Fixing Some Transformation Problems. Automated Software Engineering Conference, Cocoa Beach, Florida (1999)
8. Biggerstaff, Ted J.: A New Control Structure for Transformation-Based Generators. In: Frakes, William B. (ed.): Software Reuse: Advances in Software Reusability, Vienna, Austria, Springer (June, 2000)
9. Boyle, James M.: Abstract Programming and Program Transformation—An Approach to Reusing Programs. In: Biggerstaff, Ted and Perlis, Alan (eds.): Software Reusability, Addison-Wesley/ACM Press (1989) 361-413
10. Crew, R. F.: ASTLOG: A Language for Examining Abstract Syntax Trees. Proceedings of the USENIX Conference on Domain-Specific Languages, Santa Barbara, California (1997)
11. Czarnecki, Krzysztof and Eisenecker, Ulrich W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
12. Elrad, Tzilla, Filman, Robert E., Bader, Atef (Eds.): Special Issue on Aspect-Oriented Programming. Communications of the ACM, Vol. 44, No. 10 (2001) 28-97
13. Fickas, Stephen F.: Automating the Transformational Development of Software. IEEE Transactions on Software Engineering, SE-11 (11), (Nov. 1985) 1286-1277
14. Kant, Elaine: Synthesis of Mathematical Modeling Software. IEEE Software, (May, 1993)
15. Katz, M. D. and Volper, D.: Constraint Propagation in Software Libraries of Transformation Systems. International Journal of Software Engineering and Knowledge Engineering, 2, 3 (1992)
16. Kiczales, Gregor, Lamping, John, Mendhekar, Anurag, Maede, Chris, Lopes, Cristina, Loingtier, Jean-Marc and Irwin, John: Aspect Oriented Programming. Tech. Report SPL97-08 P9710042, Xerox PARC (1997)
17. Kotik, Gordon B., Rockmore, A. Joseph, and Smith, Douglas R.: Use of Refine for Knowledge-Based Software Development. Western Conference on Knowledge-Based Engineering and Expert Systems (1986)
18. Neighbors, James M.: Software Construction Using Components. PhD Thesis, University of California at Irvine, (1980)
19. Neighbors, James M.: The Draco Approach to Constructing Software From Reusable Components. IEEE Transactions on Software Engineering, SE-10 (5), (Sept. 1984) 564-573

20. Neighbors, James M.: Draco: A Method for Engineering Reusable Software Systems. In: Biggerstaff, Ted and Perlis, Alan (eds.): Software Reusability, Addison-Wesley/ACM Press (1989) 295-319
21. Partsch, Helmut A.: Specification and Transformation of Programs. Springer-Verlag (1990)
22. Ritter, Gerhard X. and Wilson, Joseph N.: Handbook of Computer Vision Algorithms in the Image Algebra. CRC Press, (1996)
23. Sheard, Tim: Accomplishments and Research Challenges in Meta-Programming. SAIG 2001 Workshop, Florence, Italy (Sept., 2001)
24. Smith, Douglas R.: KIDS-A Knowledge-Based Software Development System. In: Lowry, M. & McCartney, R., (eds.): Automating Software Design, AAAI/MIT Press (1991) 483-514
25. Visser, Eclo: Strategic Pattern Matching. In: Rewriting Techniques and Applications (RTA '99), Trento, Italy. Springer-Verlag (July, 1999)
26. Visser, Eclo: A Survey of Strategies in Program Transformation Systems. In: Gramlich, B. and Alba, S. L. (eds.): Workshop on Reduction Strategies in Rewriting and Programming (WRS '01), Utrecht, The Netherlands (May 2001)
27. Wadler, Philip: Deforestation: Transforming Programs to Eliminate Trees. Journal of Theoretical Computer Science, Vol. 73 (1990) 231-248
28. Wile, David S.: Popart: Producer of Parsers and Related Tools. USC/Information Sciences Institute Technical Report, Marina del Rey, California (1994) (http:// www.isi.edu/software-sciences/wile/Popart/ popart.html)
29. Wile, David S.: Toward a Calculus for Abstract Syntax Trees. In: Bird, R. and Meertens, L. (eds.): Proceedings of a Workshop on Algorithmic Languages and Calculii. Alsac FR. Chapman and Hill (1997) 324-352