

New Structures for Transformation-Based Generators

Ted J. Biggerstaff¹
tbiggerstaff@austin.rr.com

The 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 control and storage structure deficiencies of conventional transformation-based generators. While pattern-directed (PD) transformations (also called *rules*) make the specification of the transformations easy and, in fact, mostly do a pretty good job of refining (i.e., translating) domain specific abstractions into code, they often explode the search space when one is later trying to produce highly optimized code from deeply factored, inter-constrained 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.

The general form of the *constraint propagation problem*² in program generation is NP complete [3]. What can one do? Don't solve the general problem! Instead, solve specialized sub-problems that avoid search space explosions via metaprograms that use specialized storage and control structures as well as specialized strategies. Some such strategies and mechanisms have been employed in earlier generators (e.g., Draco) and some are introduced by the Anticipatory Optimization Generator (AOG) generator (e.g., Localization and Tag-Directed rules) [1-2]. Table 1 summarizes these sub-problems, the strategies used and the mechanisms that they employ. The remainder of the paper examines these strategies and their implementation structures.

Phased Translation: Draco [4] attacks the problem of the large search spaces by defining distinct Domain Specific Languages (DSLs) and performing program generation by phased translation of higher level DSLs into successively lower level DSLs until a conventional programming language is reached. Since the operators and operands of the DSLs are distinct, this has the effect of implicitly grouping the translation rules so that only those rules relevant to the current DSL construct being translated are tried. In effect, this reduces one very large search space to a series of much smaller search spaces.

Domain Specific Optimization: However, this strategy introduces another problem – complex generated code that explodes the number of cases needed to recognize each conceptual construct (e.g., $(x + 0)$ and $(0 + x)$ and x are all really the same). To solve this problem, Draco introduces an optimization phase between each DSL to DSL translation phase. This optimization phase converts the generated DSL code into a simpler, canonical form using rules that map from a domain (i.e., a DSL) to itself. These optimization rules use domain knowledge to perform optimizations. These optimizations are relatively simple using domain knowledge but are often extremely hard to do in a programming language because key domain knowledge has been translated away. Domain knowledge is the key leverage that makes the optimizations easy.

Localization: AOG adopts these same strategies as a starting point but adds new strategies. 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 operators applied to large-grain data structures imply some kind of extended control structure such as a loop, a sequence of statements, a recursive function, or other structure. 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 and data items (e.g., integer addition), but it also implies

¹ Address: 3801 Far View Drive, Austin, Texas 78730; email: tbiggerstaff@austin.rr.com.

² This is the program generation problem of coordinating the derivation of separate but related components. Constraints introduced by the choice of one reusable component (e.g., a data implementation) affect or limit the choices of separate components (e.g., a search operator). For example, a doubly linked list implementation of a container prevents generating a search method for that container that uses the Boyer-Moore algorithm.

some control structure to sequence through those atomic computations. Those implied (and often redundant) control structures are typically distributed (i.e., *de-localized*) across the whole expression. *Localization* is the metaprocess that creates localized, merged forms of the implied, interdependent control structure pieces (e.g., loop controls) that are spread across (i.e., de-localized in) the DSL specifications. Localization is propagating and coordinating constraints across a small area of the program, i.e., across a single DSL expression, thereby reducing the overall search space.

Table 1: Search Space Explosion Control Strategies and Techniques

Source of Explosion	Explosion Control Strategy	Mechanism
Numerous refinements and constraints explode derivation pathways	Phased DSL to DSL Refinement – incrementally translating to lower level DSLs	Implicit Refinement Rule Grouping induced by mutual exclusion of the operators and operands in differing DSLs
Overly complex target code explodes pattern cases	Inter-Refinement Optimization Phases – Apply domain specific optimization rules to simplify DS forms: <ul style="list-style-type: none"> ▪ Domain Specific Optimizations – Optimizations of DS forms through use of domain specific knowledge ▪ Simplification – direct reduction of the form by removing redundancies without reorganizing the basic form 	Implicit Optimization Rule Grouping by domain <ul style="list-style-type: none"> ▪ Mapping from a domain to itself to simplify DS expressions (e.g., ATN state removal) ▪ Partial evaluation employs an implicit grouping of rules specialized to redundancy removal (e.g., $(X + 0) => X$).
De-localization of (interrelated) implied control structures in specification explodes constraint combinations	Localization – Rules to introduce, migrate and merge implied, interrelated control structures that dispersed over expressions	Explicit grouping of rules by object (e.g., type) and optimization goal (i.e., phase) to hide irrelevant rules
Cross-domain & cross-component optimizations explodes constraint patterns	Architectural Shaping – Inter-component and inter-domain optimizations (metaprograms) that reshape the computation to better fit global constraints without altering its computational function	Even-Driven Tag-Directed Rules preserve domain specific knowledge (via AST tags containing event-based invocations of the rules) for use in the context of the program language domain

AOG performs localization (and other specialized optimizations) using rules that are explicitly organized in two dimensions: 1) An arbitrary DSL entity (e.g., type), and 2) An explicit phase. The first dimension encodes useful domain specific knowledge and the second encodes the kind of job being accomplished (e.g., localization). By sequencing through a number of explicit phases, AOG exposes only the rules that are relevant to both a specific AST subtree and a specific generation job, and hides the rest. This reduces the search space.

Architectural Shaping: AOG uses other control regimes and strategies. These include an event driven triggering of transformations called Tag-Directed (TD) transformation control that allows cross-component and cross-domain optimizations (code reweavings) to occur in the programming language domain while retaining 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 those introduced by interaction protocols or parallel CPU architectures [1-2].

References:

1. Biggerstaff, Ted J.: Fixing Some Transformation Problems. Automated Software Engineering Conference, Cocoa Beach, Florida (1999)
2. Biggerstaff, Ted J.: A New Control Structure for Transformation-Based Generators. In: Software Reuse: Advances in Software Reusability, Vienna, Austria, Springer (June, 2000)
3. 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)
4. Neighbors, James M.: Software Construction Using Components. PhD Thesis, University of California at Irvine, (1980)