Lecture Notes in Computer Science
Publisher: Springer-Verlag GmbH ISSN: 0302-9743
Volume 3016 / 2004
Title: Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003.

The Road to Utopia: A Future for Generative Programming[1]

Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
batory@cs.utexas.edu

Abstract. The future of software engineering lies in automation and will exploit the combined strengths of generative programming, domain-specific languages, and automatic programming. While each of these areas is still in its infancy, a spectacularly successful example of their combination was realized twenty-five years ago: relational query optimization. In this paper, I chart the successes and mindset used by database researchers to generate efficient query processing programs automatically. I argue that the road that they have so successfully followed is the same road that the generative programming, domain-specific languages, and automatic programming communities are now traversing.

1 Introduction

Just as the structure of matter is fundamental to chemistry and physics, so too is the structure of software fundamental to computer science. By the term 'structure' I mean what are modules, and how are modules composed to build programs? Unfortunately, the structure of software is not well-understood. *Software design*, which is the process by which the structure of an application is defined, is an art form. And as long as it remains so, our abilities to automate software development and make software engineering a true engineering discipline will be limited.

Our goal should be to create a mathematical science of software design. That is, we need to create general purpose theories of how customized software can be synthesized automatically. Object-oriented models are adequate if we implement programs manually; higher-level representations of programs are required for program synthesis. These theories will embody advances in *generative programming (GP)*. That is, we want to understand the programs in a domain so well that they can be generated automatically. We want generators to synthesize these programs and do the hard technical work for us. This is a shared goal of the generative programming, metaprogramming, and skeleton

1. Author's note: This is the text of a keynote presentation at the *Dagstuhl Seminar for Domain-Specific Program Generation*, March 2003. The quotations within this paper are taken from the pre-seminar surveys that invitees identified as key research issues; the quotations from Jim Neighbors are from his review of this paper.

communities. Program generation should *not* be an ad-hoc set of implementation techniques; rather, it is essential that we develop practical theories that integrate programming concepts and domain knowledge to automate software development.

We also need advances in *domain-specific languages (DSLs)*, which are special-purpose programming languages (or extensions to general-purpose languages) that allow programmers to more easily express programs in terms of domain-specific abstractions (e.g., state machines, EJB declarations). We do not want to be programming in Java and C# twenty years from now. Rather, we want to elevate program specifications to compact domain-specific notations that are easier to write, understand, and maintain.

And finally, we need advances in *automatic programming (AP)*. This is the extreme of GP and DSLs. Namely, the challenge of AP is to synthesize an efficient program from a declarative specification. This is a very hard problem; in the early 1980s, researchers abandoned AP as existing techniques simply did not scale to programs beyond a few hundred lines [1]. Now AP is undergoing a renaissance, and its need (e.g., for fault tolerance) is even more critical than ever.

To make advances simultaneously on *all* these fronts seems impossible. Yet, there exists a spectacular example of GP, DSLs, and AP in a fundamental area of computer science. And ironically, it was achieved about the same time that others were giving up on AP. The area is databases; the result is relational query optimizers.

In this paper, I review the successes and mindset that database researchers used to generate efficient query processing programs automatically and explain that the road that they followed so successfully is the same road that the GP, DSL, and AP communities are now traversing. I cite lessons that should be learned and chart a road-map that others could follow to achieve comparable successes in other domains. I use "Utopia" as the name of the objective that lies at the end of this road.

2 Lessons Learned and Lessons To Be Learned

2.1 Relational Query Processing

Relational queries are expressed as SQL SELECT statements. A parser translates a SELECT statement into an inefficient relational algebra expression, a query optimizer rewrites this expression into an equivalent expression that has better (or optimal) performance properties, and a code generator translates the optimized expression into an executable program (Figure 1).

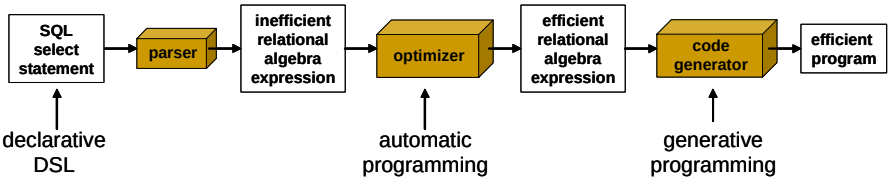

Figure 1 Relational Query Optimization

SQL is a classic example of a declarative DSL. It is a language that is specific to tabular representations of data. The code generator, which maps a relational algebra expression

to an executable program, is an example of GP. The query optimizer is the key to AP: it searches the space of semantically equivalent expressions to locate an expression which has good (or optimal) performance characteristics.

Relational query processing is an instance of a very powerful and very successful paradigm that researchers in GP should strive to replicate in other domains. If this could be achieved, would this be Utopia? No, but it would be on the road…

2.2 Cognitive Systems

> "The world changes quickly, and our applications need to adapt to these changes quickly. Program generation (as useful as it is) is but a link in a long chain" — Calton Pu.

Software evolution is inevitable, and evolution is part of maintenance. Maintenance is the most expensive part of a program's life cycle. To minimize costs, we ideally would like to automate as many maintenance tasks as possible.

Cognitive systems is an exciting area of contemporary research [8]. A *cognitive system* is a program whose performance improves as it gains knowledge and experience. In effect, it is a program that automates some of its maintenance tasks. So how have relational optimizers faired?

It turns out that relational optimizers are cognitive systems! Query optimization relies on cost models that are driven by database statistics. Example statistics include the number of tuples in a relation, the selectivity of predicates (e.g., what fraction of a relation's tuples satisfy a given predicate), the length of attribute values, etc. [22]. These statistics change over time as data is inserted and removed from tables. Thus, keeping statistics up-to-date is a key problem for optimizer maintenance. Interestingly, most optimizers — even simple ones — refresh their database statistics automatically. As generated query evaluation programs execute, statistics are gathered on data that is retrieved, and are used to refresh or update previously stored statistics. This allows the optimizer to improve the programs that it subsequently generates. In this way, optimizers learn new behaviors automatically and thus are cognitive systems.

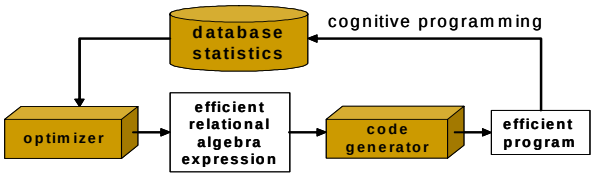

Figure 2 Cognitive Programming in Relational Query

So if this also can be replicated in other domains, would this be Utopia? No, but it is on the road…

2.3 Generating Non-Code Artifacts

> "How do we cope with non-binary non-code artifacts, e.g., manuals, tutorials, help files etc. that are specific for a generated product?" — Ulrich Eisenecker

Generating non-code artifacts is an important problem, for no other reason than generating source code for a target program is always insufficient. Architects routinely use multiple representations of programs to express a system's design, using representations such as process models, UML models, makefiles, and documents [13]. The productivity gains of generators that automatically synthesize source code is negated if other representations (which might be needed for integrating this program into a larger system) must be written manually. Clearly, *all* needed representations of a program should be generated. What are the experiences of database researchers in this regard?

Interestingly, relational optimizers use multiple representations of programs. In fact, two different representations are maintained for each relational algebra operator. One is the source code representation of the operator, and the second is a cost model representation, which is needed for optimization. Thus, for a relational algebra expression **E**:

```
E = join( select(...), select(...))
```

an optimizer derives a cost model of that program by composing cost model representations of the operators:

$$E_{cost} = join_{cost}(select_{cost}(...), select_{cost}(...))$$

In a similar manner, an optimizer can also derive source code representation of an expression by composing the code representations of these same operators:

$$E_{code} = join_{code}(select_{code}(...), select_{code}(...))$$

That is, the modularity and structure imposed on code (or code generators) is exactly the same as that for cost models: they all align along operator boundaries.

During optimization, optimizers synthesize cost models for each program they evaluate. They search the space of equivalent programs, synthesize the cost model of each program, use the cost model to estimate the performance of that program, and identify the program that will be the most efficient. Code is synthesized only for the most efficient program.

So the need for multiple representations is indeed present, but the approach is not sufficiently developed for general purpose needs. Recent work suggests it is indeed possible to synthesize arbitrary representations of programs using an algebraic approach [5].

2.4 The Quality of Performance (or Cost Model) Estimates

> "A performance estimate of good quality, especially for modern hierarchical parallel systems, is needed." — Holger Bischof

> "Especially important are rules and methods for composing skeletons in large-scale applications with reliable performance prediction." — Sergei Gorlatch

The quality of performance estimates has long been known to be critical for identifying good access plans in query optimization. Interestingly, cost estimates used by query optimizers have historically been simple and crude, based on averages. For *n*-way joins (for large *n*) estimates are known to be very poor [15]. Further, performance estimates that are based on page caching – that is, knowing what pages are on disk and which are in the cache, are highly unreliable. Despite these limited capabilities, relational optimiz-

ers have done quite well. I am sure there are domains other than query processing that require more precise estimates.

In any case, if these problems are solved, would this be Utopia? No, its on the road…

2.5 Extensibility of Domain Algebras

> "A key technical problem is expressiveness: can we fix a general purpose set of skeletons (read: operators) that covers all the interesting structures? Should we have several different skeleton sets for different application domains?"— Susanna Pelagatti
>
> "Expandability of skeleton frameworks must be studied. The programming model of skeletons must provide open source skeleton libraries/frameworks." — Marco Danelutto
>
> "DSLs are best understood in terms of their 'negative space' – what they don't do is just as important as what they do... How to avoid 'mission creep' for languages?" — Shriram Krishnamurthi

Are domain algebras closed, meaning do they have a fixed set of operators, or are domain algebras open, allowing new operators to be added subsequently? This is a fundamental question whose answer is not immediately obvious. The experience of database researchers is quite interesting with respect to this topic. To appreciate the lessons that they learned (and the lessons that we should learn), it is instructive to see what database researchers did "right".

The success of relational query optimization hinged on the creation of a science to specify and optimize query evaluation programs. Specifically researchers:

- identified the fundamental operators of this domain, which was relational algebra,
- represented programs as equations (or expressions) which were compositions of these operators, and
- defined algebraic relationships among these operators to optimize equations.

Compositional programming is a holy grail for programming paradigms: it defines a set of building-blocks or "legos" that can be snapped together to build different programs. The key property of *compositionality* is made possible by algebraic models. Compositionality is the hallmark of great engineering, of which relational query optimization is an example.

Now, let's return to the open or closed nature of domain algebras. Relational algebra was originally defined by the project, select, join, and cross-product operators. For years, *by definition* it was closed[1]. During this time, people were trying to understand the implications of the classic 1979 Selinger paper on System R optimizer [22], which revolutionized query evaluation and set the standard relational optimizers for the next two decades. This paper dealt only with queries formed from compositions of the basic relational operators. But from 1985 onward, there was a series of papers approximately

1. Not "closed" in a mathematical sense, such as addition is closed in integers but not in subranges. By "closed" I mean a social club: no new members were thought to be needed.

titled “I found yet another useful operator to add to relational algebra”. Among these operators are data cube (aggregation) [10], transitive closure [16], parallelization of query evaluation programs (that map a sequential program to a parallel program) [9], and new operators for time series [24], just to mention a few.

So is relational algebra complete? No! It is obvious now that it will never be closed, and will never be complete. There will always be something more. And this will be true for most domains. Database systems now deal with open algebras, where new operators are added as needed; they are more type extensible than before; hooks are provided into the optimizer to account for the peculiarities of new operators, such as Hellerstein’s work [14] on user-defined queries.

But the core of model remains fixed: query optimization is still based on an algebra and programs are still represented algebraically, because the algebraic paradigm is simply too powerful to abandon.

2.6 Implications with Open Algebras

> “Once a synthesis system solves non-trivial problems, it usually gets lost in vast search spaces which is not only spanned by the different ways to derive a specific program, but also the set of all possible programs satisfying the specification. Control of the search is thus a major problem, specifically the comparison and choice between different ‘equivalent’ programs.” — Bernd Fischer

> “How to represent the space of possible optimization alternatives for a component (read: operator), so that the best combination of optimizations can be chosen when the component is used?” — Paul Kelly

Given the fact that open algebras will be common, how has this impacted query processing research? My guess is that the database community was lucky. The original optimizers supported only the initial set of relational operators. This constraint made query optimization amenable to a dynamic programming solution that admitted reasonable heuristics [22]. The end result was that database people could legitimately claim that their optimization algorithms were guaranteed to find the “best” query evaluation program. And it was this guarantee that was absolutely crucial for early acceptance. Prior work on query optimization used only heuristics, and the results were both unsatisfying and unconvincing. Providing hard guarantees made all the difference in the world to the acceptance of relational optimizers.

Ironically, the most advanced databases today use rule-based optimizers that offer many fewer guarantees. But by now, database people are willing to live with this. So is this Utopia? No, its on the road…

2.7 What is Domain Analysis?

> “A key problem is what exactly are the common algorithmic structures which underpin ‘enough’ parallel algorithms to be interesting? Is it reasonable to expect that such a generic collection exists, or is it more appropriate to look in domain specific ways?” — Murray Cole

"What is the target domain?" is a core question of generator technologies. An analysis of a domain, called *domain analysis*, identifies the building blocks of programs. Generators implement the output of domain analysis. But what exactly is "domain analysis" and what should be its output? Today, "domain analysis" is almost a meaningless term. But oddly enough, whatever it is, we all agree domain analysis is important! For instance:

> "On domain engineering – any synthesis system which is useful to users must be able to generate a large number of non-trivial programs which implies it must cover a substantial part of the domain of interest. Formalizing and organizing this domain is the major effort in building a synthesis system." — Bernd Fischer
>
> "We need a methodology to systematically map domain analysis into a DSL design and implementation." — Charles Consel
>
> "We need a systematic way to design DSLs." — Krzysztof Czarnecki
>
> "Future work should improve existing methodologies for DSLs" — Laurent Reveillere

So what did database people do? They had two different outputs of domain analysis. First, they defined relational algebra, which is the set of operators whose compositions defined the domain of query evaluation programs. (So defining a domain algebra is equivalent to defining the domain of programs to generate). Another related analysis produced the SQL language, which defined declarative specifications of data retrieval that hid its relational algebra underpinnings. So database people created both and integrated both.

In general, however, these are separable tasks. You can define a DSL and map it to a program directly, introducing optimizations along the way. Or you can define a DSL and map it to an algebra whose expressions you can optimize.

This brings up a fundamental result on hierarchies of DSLs and optimizations. The first time I saw this result was in Jim Neighbor's 1984 thesis on DRACO [19]. The idea is simple: programs are written in DSLs. You can transform (map) an unoptimized DSL program to an optimized DSL program because the domain abstractions are still visible. Stated another way, *you can not optimize abstractions that have been compiled away*.

Given an optimized DSL program, you translate it to an unoptimized program in a lower-level abstraction DSL and repeat the same process until you get to machine code. So it is this "zig-zag" series of translations that characterize hierarchies of DSLs (or hierarchies of languages, in general) and their optimizations (Figure 3a).

Figure 3b depicts a DRACO view of relational query optimization. You begin with an SQL SELECT statement. A parser produces an unoptimized relational algebra expression. A query optimizer optimizes this expression and produces an optimized relational algebra expression. A code generator translates this to an unoptimized Java program, and the Java compiler applies its optimization techniques to produce an optimized Java program.

GP occurs when mapping between levels of abstraction, and AP occurs when optimizing within a level of abstraction. More commonly, optimizations are done internally by

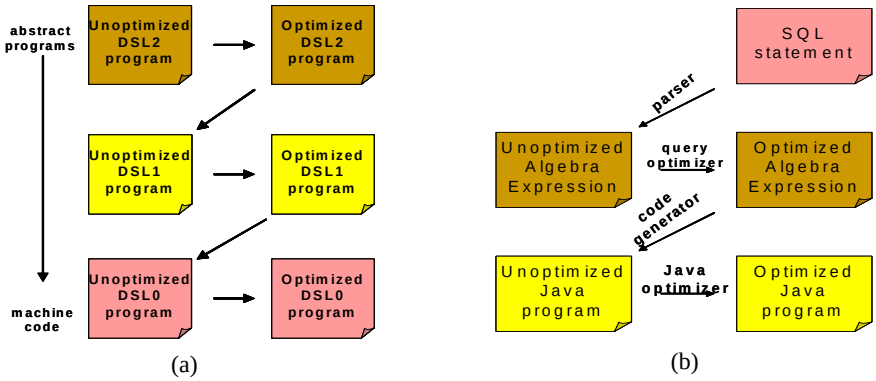

Figure 3 DRACO DSL Hierarchies and Optimizations

DSL compilers. That is, a DSL program is mapped to an unoptimized internal representation, and then this representation is optimized before translating to an executable.

The point is that there can be different outputs of domain analysis, and that different optimizations occur between various translations. The most significant optimizations, I assert, occur at the "architectural" or "algebraic" level.

So is this Utopia? No, its on the road…

2.8 Software Design

> "Software technology in domain-specific programming involves getting the interface right, getting the split right (how to separate the domain-specific from the domain-independent)" — Chris Lengauer

> "It is most essential for component software to standardize well-defined interfaces." — Wolfgang Weck

These are fundamental problems of software design. I'll sketch a common problem, and then show how a database approach solves it. But first, we need to understand the relationship between operator compositions and layered designs.

Figure 4 depicts a layered design, where layer **a** is on the bottom, layer **b** sits atop **a**, and layer **c** is atop **b**. Operator implementations often correspond to layers or layers of abstraction. The design in Figure 4 corresponds to the composition **`c(b(a))`**, where layers **a**, **b**, and **c** implement their respective operators.

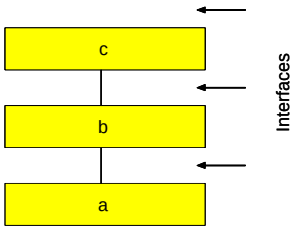

Figure 4 Layering Interpretation of Composition **`c(b(a))`**

In general, systems are conceptually, but *not* physically, layered [12]. Interfaces delineate the boundaries of operators/layers **a**, **b**, and **c**. These interfaces might be Java interfaces or *they might be DSL specifications!*

Now to an example. Suppose a program maintains a set of records of form (**age**, **tag**) and these records are stored on an ascending age-ordered linked list (Figure 5). Here the first record has **age=1**, **tag=A**, the next record **age=10**, **tag=B** and so on.

Figure 5 A Linked List for our Example

Periodically, we want to count all records that satisfy the predicate **age>n** and **tag==t**, for some **n**, **t**. What is the code for this retrieval? Here's our first try: we write an ordered list data type. Our retrieval is simple: we examine every record, and apply the full predicate. This program is easy to write. Unfortunately it is inefficient.

```
int count = 0;
Node node = container.first;
while (node != null) {
   if (node.tag == t && node.age > n)
       count++;
   node = node.next;
}
```

Our next try exploits a property of ordered lists. The observation is that we can skip over records that don't satisfy the key predicate, **age>n**. As soon as we find the first record that satisfies the predicate, we know that all records past this point also satisfy the predicate, so all we need to do is to apply the residual predicate, **tag==t**, to the remaining records. This leads to the following 2-loop program. It takes longer to write, longer to debug, but the result is more efficient.

```
int count = 0;
Node node = container.first;

while (node != null && node.age <= n)
       node = node.next;

while (node != null) {
   if (node.tag == t)

       count++;
   node = node.next;
}
```

There is yet another alternative: a Java programmer would ask: Why not use the Java library? With library classes, you would have to write even less code! In the J2SDK, **TreeSet** implements **SortedCollection**. With **TreeSet**, the desired subcollection can be extracted in one operation (called **tail()**). By iterating over the extracted elements and applying the residual predicate as before we can produce the desired result. The code is indeed shorter:

```
int count = 0;
// ts is TreeSet with elements

TreeSet onlyOld = ts.tail(n);
Iterator i = onlyOld.iterator();
```

```
while (i.hasNext()) {
   Node node = (Node) i.next();
   if (node.tag == t)
      count++;
}
```

Unfortunately, the **`TreeSet`** code is *much* slower (maybe even slower than original list implementation). Why? The reason is **`TreeSet`** creates an index over all the records that it sorts. Applying the **`tail`** operation creates an index that references the extracted elements. Index construction can be very slow. This raises a classical dilemma: if you want execution speed, stay with customized code. If you want to write the code fast, use libraries.

This problem has everything to do with selecting the right interfaces and right abstractions and is a classical situation for the use of DSLs and GP. Let me review a database solution to this problem [3]; a more general formulation is presented in [21].

Figure 6 depicts the code we wrote. Users would write the code above the left horizontal line, which is the user program and traversal loop. But look what happens when we swap data structures (or change the record ordering). Our program breaks. The reason is that implementation details — specifically the record storage order — of the data structure leaked into our program. So if these details change, our program has to change too.

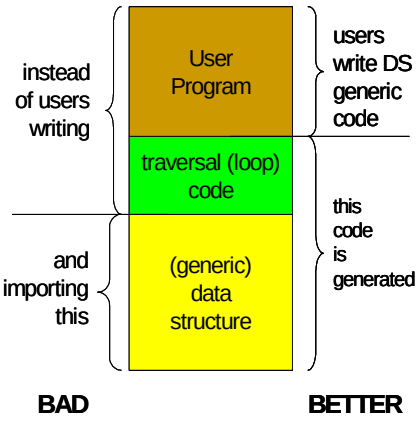

Figure 6 What To Generate

What we need are higher-level abstractions (e.g., a DSL) to specify a data structure and its traversals. That means that users should write data structure generic code, and the rest (loop, data structure itself) is generated. (That is, users should write the code above the right horizontal line in Figure 6). A database solution is to use a SQL to specify retrieval predicates and declarative relation implementation specifications to implement the container. That is, a container is a relation. Predicates for cursors (iterators) are declaratively specified in SQL and the SQL compiler generates the cursor/iterator class that is specific to this retrieval. This makes the user program data-structure (relation implementation) independent! Further, it exploits DSLs, algebraic optimizations of relational optimizers and amazingly, the code a user has to write is even simpler than using generics! And the code is efficient [4], too!

```
cursor1 c = new cursor1(n,t);
int count = 0;

for (c.first(); c.more(); c.next())
   count++;
```

So is this Utopia? No, its on the road…

2.9 Scalability of Algebraic Approaches

Do algebraic approaches scale? Let's face it, query evaluation programs and data structures are tiny. Can an algebraic approach synthesize large systems? If it can't, then we should pack our bags and try something else.

Interestingly, it does scale. And this brings us to a key result about scaling. *Feature Oriented Domain Analysis (FODA)* is pioneering work by Kyo Kang, et al [17]. His work deals with product-lines and producing a family of related applications by composing primitives.

So the goal is to synthesize a large application from "primitives". But what are these primitives? Consider the following thought experiment. Suppose you have programs that you want others to use. How would you describe them? Well, you shouldn't say what DLLs or object-oriented classes each uses. No one will care. Instead, you are more likely to describe the program by the features it has. (A *feature* is a characteristic that is useful in distinguishing programs within a family of related programs [11]). For example, you might say **`Program1`** has features **`X`**, **`Y`**, and **`Z`**. But **`Program2`** is better because it has features **`X`**, **`Q`**, and **`R`**. The reason is that clients have an understanding of their requirements and can see how features relate to requirements.

A common way to specify products is by its set of features. While this is almost unheard of in software, it is indeed common in many other engineering disciplines. For example, go to the Dell Web site. You'll find lots of web pages that provide declarative DSL specifications (e.g. menu lists) from which you can specify the features that you want on your customized PC. After completing this specification, you can initiate its order. We want to do the same for software.

Here is a program synthesis vision that has evolved concurrently with FODA [2]. Program **`P`** is a package of classes (**`class1`**—**`class4`**). **`P`** will have an algebraic definition as a composition of features (or rather feature *operators*). Consider Figure 7. **`P`** starts with **`featureX`**, which encapsulates fragments of **`class1`**—**`class3`**. **`featureY`** is added, which extends **`class1`**—**`class3`** and introduces **`class4`**, and **`featureZ`** extends all four classes. Thus, by composing features which encapsulate fragments of classes, a package of fully formed classes is synthesized.

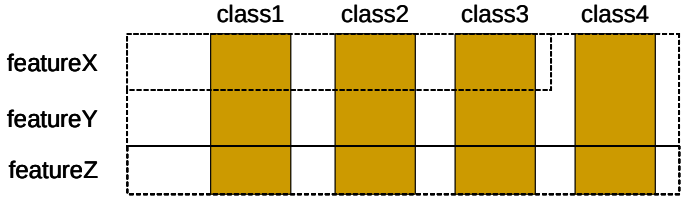

Figure 7 Program `P = featureZ( featureY( featureX ) )`

This algebraic approach is related to the age-old concept of *step-wise refinement* which asserts that a complex program can be constructed from a simple program by adding details (in this case, features), one at a time. This means that feature operators are implemented by program refinements or program extensions.[1] Program **`P`** is created by

starting with a simple program, **`featureX`**. This program is extended by **`featureY`** and then by **`featureZ`** — a classic example of step-wise development.

Here is an example problem that illustrates the scalability of algebraic approaches. I and my students are now building customized tools for processing programs written in extensible-Java languages. These tools belong to *Integrated Development Environments (IDEs)*. The GUI shown in Figure 8 is a declarative DSL. We allow architects to select the set of optional Java extensions that they want (in the left-most panel), the set of optional tools that they want (in the middle panel), and by pressing the Generate button, the selected tools will be synthesized and will work for that specified dialect of Java. We are now generating over 200K Java LOC from such specifications, all of which are (internally) driven by equations. So algebraic approaches do indeed scale, and there seems to be no limit to the size of a system that can be produced [6].

Figure 8 A Declarative GUI Interface for Customized IDE

So is this Utopia? No, its on the road…

2.10 Tool Support

"One of the biggest technical problems is that metaprogramming and generative programming are not directly supported by programming languages. A cleaner and safer mechanism (other than (ab)using the template mechanism and type system in C++) is clearly needed." — Andrew Lumsdaine

"The 'big' technical challenge is getting a good infrastructure for developing generators that includes extensible programming languages transformation systems, and metaprogramming libraries." — Yannis Smaragdakis

"Type safe program generation and efficient implementation of multi-stage languages are important." — Walid Taha

---

1. The term "refinement" is often used in the context of adding implementation details *without* modifying the original specification. "Extensions" add details to implementations and can enhance the original specification. The definition of these terms is not agreed upon in the OO and programming methodology communities, and my informal use of these terms reflects this lack of consistency. Still, it remains is an open problem to relate notions of program synthesis using features to concepts in algebraic program specifications. Defining these relationships clearly should lead to a valuable advance in both areas: a formal theory for practical generative programming, and a practical outlet for theory.

Long ago, I realized that the scale of programs will have an impact on the tools that are used. Some domains will have operators that can be implemented by comparatively simple macro expanding techniques. Figure 9 shows a pair of classes that implement a bare-bones singly-linked list program shown in black, non-italic font. By applying an doubly-linked list operator, this program is transformed into a program that is a doubly-linked list. The transformation-added code is shown in *red, italic* font. Relatively simple tools can be built to achieve this kind of program rewriting.

```
class list {
  Node first;
  Node last;

  void insert( Node x ) {
    if (last == null)
      last = x;
    x.next = first;
    first = x;
    x.prior = null;
  }
}
```

```
class Node {
   Node next;
   String value;
   Node prior;
}
```

Figure 9 Transformation of a Singly-Linked List into a Doubly-

Surprisingly, this simple approach has worked well for *large* applications. However, for smaller programs or algorithm synthesis, much more is needed. The work at Kestrel on synthesizing scheduling algorithms [7] and the synthesis and optimization of orbital algorithms at NASA Ames [23] are really impressive. They require nontrivial “domain theories” and a non-trivial programming infrastructure. Even the simple data structures domain requires more sophistication than macros.

One reason for this is that domain-specific optimizations are below the level of relational algebraic rewrites; one has to break encapsulation of abstractions to achieve better performance. And the operations (functions) seem much more complicated.

Oddly, the synthesis of large systems has different requirements. Architects generally don’t care about low-level optimization issues. The problem is more of gluing operators together; breaking encapsulations to optimize is rarely done. For years I thought lisp-quote-unquote features were critical for all generative tools. I now think that most domains don’t need such sophistication. Tools for synthesis-in-the-large will be very different than those for synthesis-in-the-small.

There seems, however, to be a pleasing result: there is one way to conceptualize program families using features, but how operators are implemented is domain-dependent. There are there are lots of ways to implement operators: as macros, lisp-quote-unquote, program transformation systems, objects, etc. It is a matter of choosing the right implementation technology. However, the process by which one identifies the fundamental operators in a domain is largely independent of operator implementation.

So is this Utopia? No, its on the road…

2.11 Verification

> "What kinds of verifications are appropriate/feasible for what kinds of languages, and what approaches are appropriate to carry out these verifications? How can languages be designed to facilitate verification?" — Julia Lawall

We want guarantees about generated programs. We want proofs that properties of algorithms that implement operators are not violated when operators are composed. What are the lessons learned by database researchers?

As far as I can tell, verification has never been an issue. And it is not surprising either. Historically there are no (or trivial) synchronization issues, no real-time issues in query evaluation programs. Query evaluation programs in database systems wrestle with scale and performance, not correctness issues.

I want to point out that there is important work on verifying programs using features. Most properties that are to be preserved in compositions are local properties of features. You want to prove that feature properties are not violated by composition. I recommend reading the Li, Krishnamurthi, and Fisler paper [18] to get a flavor of this line of work.

So is this Utopia? No, its on the road…

2.12 Technology Transfer

> "How do we make our concepts accessible to 'Joe Parallel Programmer', who knows Fortran/C+MPI/Threads and is no so unhappy with these?" — Murray Cole

> "We need to reduce the necessary expertise to use generative programming and metaprogramming tools for DSL definition and implementation. The use of current systems is very much a craft."' — David Wile

Technology transfer are tough issues indeed. By far, technology transfer is the hardest problem. Education is the key. We must demonstrate over and over again where GP, DSLs, and AP are relevant and beneficial. We must be constantly looking for new applications to demonstrate their value. Sadly, I fear, not until large companies like Microsoft see the advantage, progress will be glacial. You have heard of the 17 year lag between the discovery of ideas and practice; I think things are much longer for software engineering simply because the inertia is so great.

So is this Utopia? No, its on the road…

2.13 And More!

> "What are the relative merits of different programming models?" — Prem Devanbu

> "Language design and hiding the meta-level is an important problem." — Joerg Striegnitz

> "The economic issues (cost, time-to-market, maintenance, flexibility) are not well understood." — Chris Ramming

There is no lack of other issues. Every issue raised above is indeed important. Often the progress of a field hinges on economics. And until we understand the economic ramifications (i.e., benefits), transfer of our ideas to industry will be slow.

3 Epilog

So if we solved all of the previously mentioned problems, would this be Utopia? It might be. But let's put this in perspective: Did database people know they were on the road to Utopia? Hardly. Let's start with Codd's 1970 seminal paper on the Relational Model. Its first public review in Computing Surveys panned the idea [20]. And it is easy to forget that the appreciation of the Relational Model grew over time.

> "It isn't like someone sat down in the early 1980's to do domain analysis. No — we had trial and error as briefly outlined:
>
> (1) CODASYL 1960s - every update type and every query type requires a custom program to be written,
>
> (2) Codd 1970 - relational algebra - no keys but in theory no custom programs,
>
> (3) IBM & researchers (late) 1970s - compelling business issues press development at business labs and universities. Query languages, schema languages, normal forms, keys, etc.
>
> (4) Oracle early 1980s - and they are off...
>
> Now, which domain analysis methodology shall we assert could have achieved this in a shorter term? It takes time and experience on the road to a solution; it also takes the patience to continually abstract from the problem at hand until you recognize you already have a solution to the immediate problem." — Jim Neighbors

In short, it takes time and clarity of hindsight to find Utopia. Utopia is a small place and is easy to miss.

> "People of different backgrounds have very different opinions on fundamental problems and principles of software engineering, amazingly." — Stan Jarzabek
>
> "I am not convinced that any of these problems is concretely enough defined that if solved I will recognize the solution as a big advance. I believe the area is in a state where there is no agreement as to what will constitute the next big step." — Yannis Smaragdakis
>
> "Mindset is a very important issue. How can researchers find Utopia if they are not trying to get there? How can they be trying to get there if they are not solving a specific problem? Without a problem, they are on the road to where?" — Jim Neighbors

My response: this is Science. The signs along the road to scientific advancement are strange, if not obscure. But what did you expect? Some observations and results will be difficult, if not impossible to explain. But eventually they will all make sense. However, if you don't look, you'll just drive right past Utopia, never knowing what you missed.

My parting message is simple: database researchers got it right; they understood the significance of generative programming, domain-specific languages, automatic programming and lots of other concepts and their relationships, and they made it all work.

Software engineering is about the challenges of designing and building large-scale programs. The future of software engineering will require making programs first-class ob-

jects and using algebras and operators to manipulate these programs. Until these ideas are in place, we are unlikely to reach Utopia. Our challenge is to replicate the success of database researchers in other domains. I believe that our respective communities — generative programming, metaprogramming, and the skeleton communities — represent the future of what software engineering will become, not what it is today.

I hope to see you on the road!

Acknowledgements. I am grateful to Chris Lengauer and Jim Neighbors for their comments and insights on an earlier draft of this paper.

1 References

[1] R. Balzer, "A Fifteen-Year Perspective on Automatic Programming", *IEEE Transactions on Software Engineering*, November 1985.

[2] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, October 1992.

[3] D. Batory, V. Singhal, J. Thomas, and M. Sirkin, "Scalable Software Libraries", *ACM SIGSOFT 1993*.

[4] D. Batory, G. Chen, E. Robertson, and T. Wang, "Design Wizards and Visual Programming Environments for GenVoca Generators", *IEEE Transactions on Software Engineering*, May 2000, 441-452.

[5] D. Batory, J.N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement", *International Conference on Software Engineering (ICSE-2003)*.

[6] D. Batory, R. Lopez-Herrejon, J.P. Martin, "Generating Product-Lines of Product-Families", *Automated Software Engineering Conference*, 2002.

[7] L. Blaine, et al., "Planware: Domain-Specific Synthesis of High-performance Schedulers", *Automated Software Engineering Conference 1998*, 270-280.

[8] R.J. Brachman, "Systems That Know What They're Doing", *IEEE Intelligent Systems*, Vol. 17#6, 67-71 (Nov. 2002).

[9] D. DeWitt, et al., The Gamma Database Machine Project, *IEEE Transactions on Data and Knowledge Engineering*, March 1990.

[10] J. Gray, et al. "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals", *Data Mining and Knowledge Discovery* 1, 29-53, 1997.

[11] M. Griss, "Implementing Product-Line Features by Composing Component Aspects", *Software Product-Line Conference*, Denver, August 2000.

[12] A.N. Habermann, L. Flon, and L. Cooprider, "Modularization and Hierarchy in a Family of Operating Systems", *CACM*, 19 #5, May 1976.

[13] A. Hein, M. Schlick, R. Vinga-Martins, "Applying Feature Models in Industrial Settings", *Software Product Line Conference (SPLC1)*, August 2000.

[14] J. Hellerstein, "Predicate Migration: Optimizing Queries with Expensive Predicates", *SIGMOD 1993*.

[15] Y.E. Ioannidis and S. Christodoulakis, "On the Propagation of Errors in the Size of Join Results", *ACM SIGMOD 1991*.

[16] Y. E. Ioannidis, "On the Computation of the Transitive Closure of Relational Operators", *Very Large Database Conference 1986*, 403-411.

[17] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. “Feature-Oriented Domain Analysis (FODA) Feasibility Study”. Tech. Rep. CMU/SEI-90-TR-21, Soft. Eng. Institute, Carnegie Mellon Univ., Pittsburgh, PA, Nov. 1990.

[18] H.C. Li, S. Krishnamurthi, and K. Fisler, “Interfaces for Modular Feature Verification”, *Automated Software Engineering Conference 2002*, 195-204.

[19] J. Neighbors, “Software construction using components”. Ph. D. Thesis, (Technical Report TR-160), University of California, Irvine, 1980.

[20] J.E. Sammet and R.W. Rector, “In Recognition of the 25th Anniversary of Computing Reviews: Selected Reviews 1960-1984”, *Communications of the ACM*, January 1985, 53-68.

[21] U.P. Schultz, J.L. Lawall, and C. Consel, “Specialization Patterns”, Research Report #3835, January 1999.

[22] P. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, “Access Path Selection in a Relational Database System”, *ACM SIGMOD 1979*, 23-34.

[23] M. Stickel, et al., “Deductive Composition of Astronomical Software from Subroutine Libraries”, In *Automated Deduction*, A. Bundy, ed., Springer-Verlag Lecture Notes in Computer Science, Vol. 814.

[24] M. Stonebraker, “Inclusion of New Types in Relational Data Base Systems”, *IEEE Data Engineering Conference*, Los Angeles, CA, February 1986.