From Implementation To Theory in Product Synthesis

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

Abstract

Future software development will rely on *product* synthesis, i.e., the synthesis of code and non-code artifacts for a target component or application. Prior work on feature-based product synthesis can be unified by applying elementary ideas from category theory. Doing so reveals (a) important and previously unrecognized properties that product synthesis tools must satisfy, and (b) non-obvious generalizations of current techniques that will guide future research efforts in automated product development.

1 Introduction

Program synthesis is not currently, but will be, a foundation of software development. Today's software design methodologies aim at repeatable craftsmanship. Transitioning from craftsmanship to automation will reveal new ways to think about program design and implementation; new paradigms and languages will emerge. But what might they be like?

My research has focussed on this question. My earliest years were spent on exploring what was necessary, what would work, and what would scale. The most recent were marked by an increasing realization that *structure* and the *manipulation of structure* are the core problems. Program synthesis is a subproblem of *product synthesis*, where code is but one of many artifacts to be produced. All program representations (code and non-code) can be and should be treated uniformly.

My work starts with practice, and then I try to fit practice (or practical experience) to abstract structures. My work is distinguished from others in informal software design by my use of algebra, as algebra is *by far* the simplest way to explain the structure manipulations that I need. The uniformity and scale that algebra imposes on structures is astonishing, and is in stark contrast to ad hoc programming techniques. I believe that typical software designers will find simple algebraic approaches appealing in software design and development.

A long-standing challenge in Computer Science is how to implement large and efficient programs from declarative specifications automatically [12][25]. Unlike classical work on this problem which starts with a formal logic specification (e.g., [27]), I use techniques pioneered in software product lines and other engineering disciplines that use declarative descriptions of products based on *features* (i.e., increments in product functionality) [26].

The first two generations of my work, GenVoca and AHEAD, have a simple and informal algebraic description. The newest generation is based on category theory. Although category theory was related to formal software development years ago [20][24][41], I am approaching it from practice.

Very few ideas from category theory are used in this paper. This may be explained, in part, by the fact that this work is still in its infancy. However, there is another possible explanation: a fundamental advance was made in databases almost 40 years ago when Codd noticed the connection between set theory and databases [16]. Little of set theory was used, but even this small amount was extremely useful. I believe there is a parallel with category theory and product synthesis.

My research was strongly influenced by relational query optimization [44] and parameterized programming [23], and shares common goals with algebraic specifications and refinements [20][41], and work in programming languages on virtual classes [35], mixins [11][43], and higher-order hierarchies [21].

2 GenVoca

A common way to depict the structure of software is by a UML class diagram, which defines a space of configurations and relationships among program classes. Figure 1 is an example.

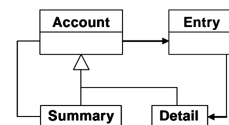


Figure 1. A Design

Designs are not created spontaneously, but rather are the result of hard work, starting from a simple design and progressively adding details. The structure of software has an additional dimension of time that we routinely hide. If this dimension were exposed (by rotating Figure 1 in 3-space), we could see how this design incrementally evolved from simpler designs to its current state. Figure 2 shows the design had only two classes at time `t0`, another class was added at time `t1`, and yet another at `t2`.

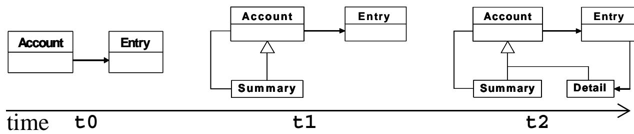


Figure 2. Design Evolution

Design evolution can be modeled by functions that map an input design to an output design. In Figure 2, a function maps the design at `t0` to the design at `t1`, and a second function maps the design at `t1` to that at `t2`. These functions implement *program deltas* or *structural transformations*; applying them allows us to move forward in time w.r.t. the evolution of that program's design. Note: deltas are sometimes program *refinements* (semantics-preserving transformations) or *extensions* (semantics-altering transformations). As these terms have different meanings, I use the more neutral term 'delta'.

This description of design evolution is very close to what is needed for product synthesis: Start with a simple product and apply deltas (functions) to progressively elaborate its structure. *Product synthesis rests in the ability to implement and compose deltas.*

On closer inspection, there is something odd about the deltas (functions) that arise in traditional software development. Namely, they can only be used in a very restricted context: for that program and only at a given point in time. Such deltas are not reusable and are a consequence of 1-of-a-kind design methodologies which produce 1-of-a-kind deltas. There is little or no benefit for automating the changes that these deltas make; it is cheaper to have programmers manually achieve their effects by ad hoc modifications to code. This is largely what we see today.

Program design, implementation, and maintenance is enormously expensive. Parnas argued that if a program is at all useful, variants of it will arise. Instead of creating each from scratch, a more economical way is to create a design for a family of programs, so that the costs of program development are amortized [40]. This is the idea underlying *software product lines (SPLs)*. Figure 3 shows an elementary SPL that has six programs or products `P0…P5`. Arrows denote functions (deltas). Function `A` maps `P0` to `P2`, `B` maps `P0` to `P1`, etc. Note that designs are written as expressions, such as `P5=G•A•P0`, where `P0` is modeled by a constant function and • denotes function composition.

Unlike traditional software development, it is common in SPL to reuse deltas. For example, delta `B` is used in the synthesis of two products `P3=D•B•P0` and `P4=A•B•P0`, and `A` is reused in three products `P4`, `P2=A•P0`, and `P5=G•A•P0`. In these circumstances, mechanizing deltas (i.e., automating the changes that a delta makes) becomes economical.

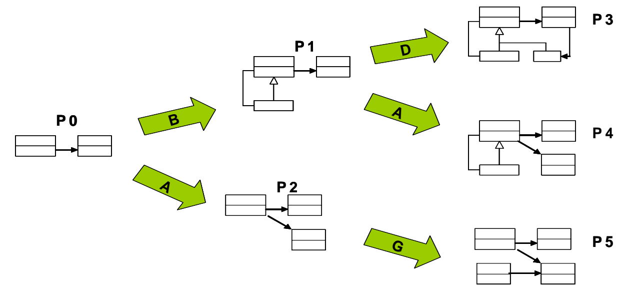


Figure 3. A Product Line

An elementary consequence of engineering SPLs is discovering the meaning of a delta: it is a stereotypical increment in product functionality called a *feature*. A fundamental concept of SPLs is that products of a product line are differentiated by the features that they have; different products have different features [26].

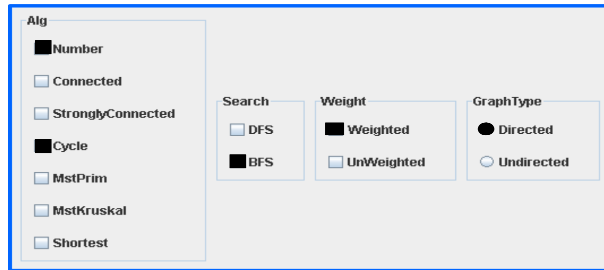
Features are a very general concept. They are used in many engineering disciplines to specify products using *declarative domain-specific languages (DDSLs)*. PCs can be configured via Dell web pages, which are DDSLs for Dell product lines [18]. A client specifies a customized PC by selecting desired features (CPU, disks, etc.). Another example is the web site to design your own BMW where automotive features can be selected [13].

The same holds for software. An elementary example is the *Graph Product Line (GPL)*, which is an SPL of programs that implement different graph algorithms [32]. Figure 4 shows a DDSL for GPL. GPL programs are specified declaratively by selecting graph features, e.g., directed vs. undirected graphs, weighted vs. unweighted graphs, search algorithms (breath-first or depth-first), and graph algorithms (vertex numbering, cycle checking, etc.). As users select features (denoted by darkened options in Figure 4), an expression is created that composes the deltas of these features. By evaluating this expression, the specified program is synthesized. The program declared in Figure 4 implements vertex numbering and cycle checking using a breadth-first search on a weighted, directed graph.[1]

These are the ideas of GenVoca [4]. A model of a SPL is an *algebra* — a set of constants (base programs) and functions that modify programs by adding a particular feature. By composing features, expressions are produced that represent the designs of different programs in a product line.

Although GenVoca hardly qualifies as a rigorous mathematical description of software, it is none-the-less useful.

1. More generally, features can have what Goguen calls *horizontal parameters* [23], much like GUI components have property sheets. This would elaborate, but not invalidate, the simplicity of the ideas described here.




```
program = Cycle•Number•BFS•Weighted•Directed
```

Figure 4. Declarative Program Specifications

Recall the challenge problem of Section 1 (which is more commonly known as the *automatic programming problem* [12][25]): how can an efficient program be implemented automatically from a high level specification (not necessarily a logic specification)? GenVoca represents a program's design as an expression. Expressions can be optimized automatically by using algebraic identities as rewrite rules. This means that program designs can be optimized automatically.

An example is relational query optimization [44] (Figure 5). A database retrieval program is specified declaratively as an SQL SELECT statement. A parser maps the statement to an unoptimized relational algebra expression. An optimizer optimizes the expression by rewriting it using algebraic identities. And a code generator translates the optimized expression into an efficient program. The keys to solving the automatic programming problem are: modeling program designs as expressions and optimizing expressions. Relational algebra is an example of GenVoca. Others are [5][10][52].

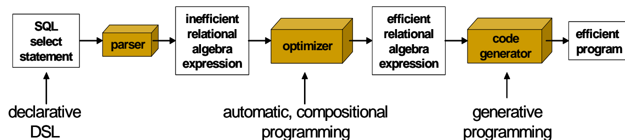


Figure 5. The Relational Query Optimization Paradigm

To recap:

- a program design is an expression,
- a product line is modeled by different expressions,
- expression evaluation is program synthesis, and
- expression optimization is design optimization.

Now, let's look more closely at how constants and functions are implemented.

3 AHEAD

Generating code for an individual program is useful, but not sufficient. Large programs today are rarely defined solely by source code. Architects routinely use many different representations of a program to express its design, such as process models, UML models, makefiles, documents, performance models, etc. For synthetic programs to work in a broader software engineering context, all relevant representations of a program must be generated. This lead me to consider more general notions of modularity — modules are not limited to code.

A *module* is a containment hierarchy of related artifacts. Figure 6a shows several modules. Referring to levels of a tree and counting from bottom up, a class is a 2-level module that contains a set of methods and a set of fields. An interface is also a 2-level module that contains a set of method signatures and a set of constants. A package is a 3-level module that contains a set of classes and interfaces. A J2EE EAR file is a 4-level module containing a set of packages, deployment descriptors, and HTML files.

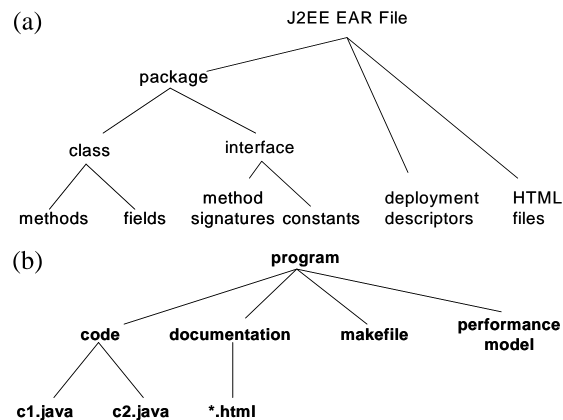


Figure 6. Module Hierarchies

A program is a module (Figure 6b). It could contain code, documentation, a makefile, and a performance model. Program code could be source and binaries, and program documentation could be a set of HTML files. In general, a module can have arbitrary depth and arbitrary contents.

The connection with GenVoca is straightforward: a program (module) is a GenVoca constant. A feature is a function that maps modules to modules.

When a feature is added to a program, any or all of the program's representations may be extended. Adding a feature to the program in Figure 6b extends its code (to implement the feature), extends its documentation (to document the feature), extends its makefile (to build the feature), extends its performance model (to profile the feature), etc. This idea recurses: the code of a program is a set of Java files. Each of these files may be extended, and new files added. The same holds for program bytecodes and documentation.

The key to understanding GenVoca functions is knowing how terminal artifacts are transformed, as the delta of a

non-terminal reduces to adding new artifacts (which is easy) and extending existing artifacts. I'll explain the evolution of my thoughts on extending Java classes, and then generalize to the extension of noncode artifacts.

3.1 Deltas of Artifacts

Many technologies can encode deltas. The simplest are preprocessor **`#IFDEF`** statements that surround code to be included only if particular features are selected. I found such code to be unmaintainable and too restrictive to compose deltas flexibly. Transformation systems offer another extreme: they match well with GenVoca, but I found them too hard to use. As I was using Java, I wanted a language construct that would give me most of what I wanted, but would also be easy for Java programmers to learn and use.

A critical property of this language construct is *monotonicity*: program extension is accomplished by *addition*. That is, a feature could *add* new code to existing methods, it could *add* new methods and fields to existing classes, and could *add* new classes. Removing, deleting, or merging classes, methods, etc. requires non-monotonic reasoning, which can be very complicated [15]. My experience is that architects reason monotonically with features — program **`P`** has features **`a`** and **`b`**, and now it has feature **`c`**, where the key properties of **`a`**, **`b`**, **`c`** are preserved *or appropriately transformed* [30]. This form of program development was embodied in the ideas of stepwise refinement by Wirth [51] and Dijkstra [19], and widely used in practice as *object-oriented (OO)* frameworks [22]. In fact, a *vast* majority of features are OO collaborations (i.e., role-based designs) [49][45]. Class inheritance, which is used in implementing OO frameworks and OO collaborations, was an obvious place to start [6].

Class inheritance serves two distinct purposes in OO programming languages: (1) to express subtyping (e.g., a Secretary "is a" Person), and (2) to express code reuse. It is this latter use that I exploited.

A class is extended by adding new fields, new methods, and extending existing methods (by overriding and invoking superclass methods). The evolution of a class can be modeled by a linear inheritance hierarchy called a *delta chain*. Figure 7 illustrates a chain of three classes. The top represents the definition of a class at time **`t0`**, the middle represents the definition at time **`t1`** (as new fields were added and existing methods are modified), and the bottom defines the class at time **`t2`**. Mixins (i.e., a class whose superclass is specified by a parameter) were used to create customized chains [45]. Eventually I realized that only the terminal class of a delta chain was instantiated; non-terminal classes were abstract and represented intermediate derivations of the terminal class.

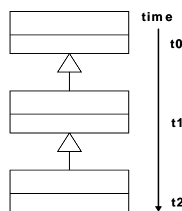


Figure 7. Delta Chain

As my experience increased with this technique, I noticed that I needed more than Java inheritance could offer: the name of a class should remain unchanged after applying a delta, and I wanted to add new constructors to an existing class and extend static methods. Java did not support these basic needs. It is at this time that I abandoned inheritance — *subclassing is the wrong paradigm for deltas* — and began to think in terms of functions that mapped class definitions to extended class definitions.

Using preprocessors, I added a "**`refines class`**" declaration to Java [1]. Other than not having an "**`extends`**" clause, it was syntactically identical to a Java subclass declaration. Every new field, method, and constructor that was listed would be added to a designated class. Further, deltas of methods, constructors, and class initializations have the same syntax in Java and are applied to existing definitions of a designated class.[2] Figure 8b defines a delta that can be applied only to class **`foo`**. That is, it is a function whose domain contains different definitions of class **`foo`** and whose codomain consists of extended **`foo`** definitions. The delta of Figure 8b adds field **`y`**, a single-parameter constructor, and extends the **`inc()`** method.

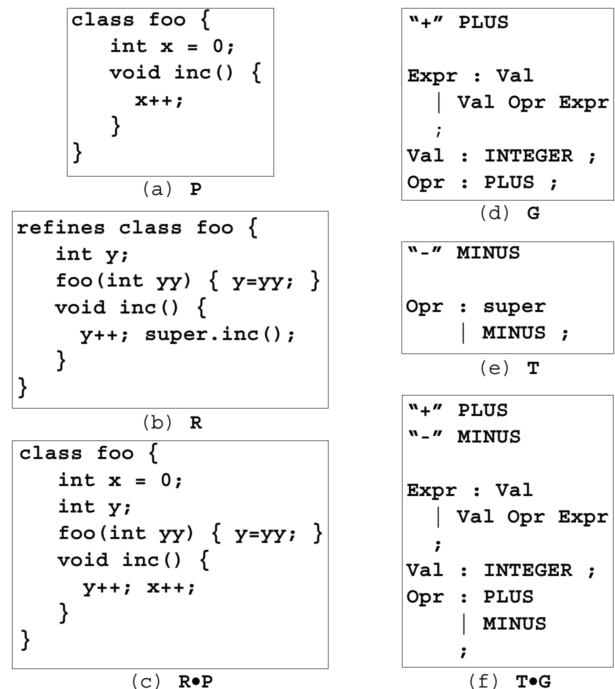


Figure 8. Deltas and Composition

2. **`super`** invocations were used in static methods to express changes.

The **`refines class`** construct enabled me to consider Java class definitions as constants, and **`refines class`** definitions as functions. Figure 8a is a constant (labeled **`P`**), Figure 8b is a function (labeled **`R`**), and their composition **`R•P`** is Figure 8c. In effect, these ideas elevated my thinking to an algebra by which I could specify the synthesis of individual class definitions by composing constants and functions a la GenVoca, *but now on a miniature scale*.

As an aside, AspectJ was being developed concurrently and independently of this work. Aspects can offer much greater power than **`refines class`**, but aspects do not behave like functions, they compose in complicated ways, and can produce unpredictable results. With basic changes, they could be functions (deltas) with elegant properties [34].

The "**`refines class`**" construct modified a target module by adding new elements and extending existing elements. Exactly the same ideas could be applied to other program representations. For example, a grammar **`G`** is a set of token and production definitions (Figure 8d). A grammar delta **`T`** adds new tokens, new productions, and extends existing productions by adding new right-hand sides. Figure 8e shows a delta of a production **`Opr`**, where "**`super`**" means the prior right-hand side of **`Opr`**. The composition **`T•G`** is shown in Figure 8f.

In general, any artifact can be given a hierarchical structure, and a standard way to extend such artifact can be defined. In effect, a delta is purely a structural operation: the same abstract algorithms were used to extend and compose source code, bytecode, grammars, XML files, etc.

3.2 Theory and Implementation[3]

A *constant module* is a vector, whose elements are primitive artifacts (e.g., files) or are themselves constant modules. The constant module of Figure 9a, for example, is the nested vector of Figure 9b.

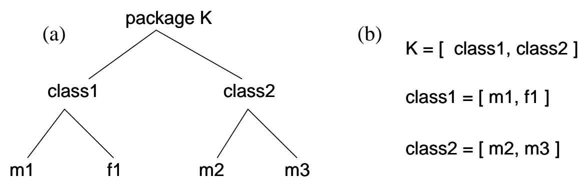


Figure 9. Encoding Modules as Vectors

Each vector element is indexed by its name. The value of a primitive element is an artifact (e.g., a code file, HTML file, etc.), and the value of a module is its vector. Elements can be null (∅), meaning that this particular artifact is not present in the module.

3. This description of AHEAD is due to C. Lengauer. The original description used nested sets, not nested vectors [9].

A *function module* is also a vector, whose elements are deltas that extend primitive artifacts or are module deltas (i.e., function modules). Functions that map a null artifact to a non-null artifact model artifact addition. The identity function (ι) models deltas that make no modification.

Module composition is vector composition, where elements are composed pairwise. Let **`A`** and **`B`** be vectors (modules) of length **`n`**; the composition of **`A•B`** is:

$$
\begin{aligned}
&\mathtt{A \bullet B} \\
&= [a_1 \ldots a_n] \bullet [b_1 \ldots b_n] \\
&= [a_1 \bullet b_1, \ \ldots \ a_n \bullet b_n] \qquad (1)
\end{aligned}
$$

When modules are nested, **`(1)`** is applied recursively. Note that for **`(1)`** to be type correct, composed elements must be of the same type (i.e., code deltas compose with code, HTML deltas compose with HTML, etc.).

`(1)` is the *Law of Composition*. A GenVoca expression is evaluated by recursively expanding modules and applying **`(1)`**. As an example, let **`A`** be a constant module and **`B`** be a function module (see Figure 10):

$$
\begin{aligned}
&A = [\ Code_a,\ R.txt_a,\ Doc_a\ ] \\
&Code_a = [\ X.java_a,\ Y.java_a\ ] \\
&Doc_a = [\ W.html_a,\ \varnothing\ ] \qquad (2)
\end{aligned}
$$

$$
\begin{aligned}
&B = [\ Code_b,\ R.txt_b,\ Doc_b\ ] \\
&Code_b = [\ X.java_b,\ Y.java_b\ ] \\
&Doc_b = [\ \iota,\ Z.html_b\ ] \qquad (3)
\end{aligned}
$$

The composition **`C = B•A`** is:

$$
\begin{aligned}
C &= B \bullet A \\
&= [Code_b, R.txt_b, Doc_b] \bullet [Code_a, R.txt_a, Doc_a] \\
&= [Code_b \bullet Code_a,\ R.txt_b \bullet R.txt_a,\ Doc_b \bullet Doc_a] \\
&= [\ [X.java_b, Y.java_b] \bullet [X.java_a, Y.java_a], \\
&\quad R.txt_b \bullet R.txt_a,\ [\iota, Z.html_b] \bullet [W.html_a, \varnothing]\ ] \\
&= [\ [X.java_b \bullet X.java_a,\ Y.java_b \bullet Y.java_a], \\
&\quad R.txt_b \bullet R.txt_a,\ [W.html_b,\ Z.html_a]\ ] \qquad (4)
\end{aligned}
$$

The result is a nested vector of expressions. Each expression defines a terminal artifact that is to be synthesized; by evaluating each expression, a hierarchical module that defines the target program is generated.

A module is implemented by a file system directory. Primitive artifacts are files and nonprimitives are directories. Figure 10 depicts **`(2)-(4)`**. The composition operator • is

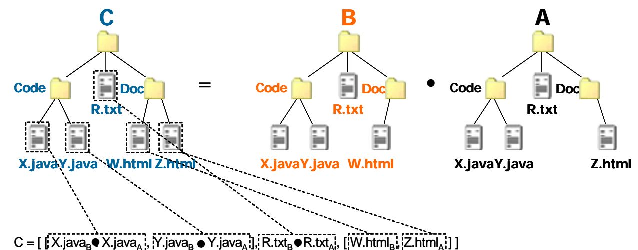


Figure 10. Composition

polymorphic and its implementation relies on ad hoc polymorphism, i.e., a distinct tool implements • for each type of artifact. That is, one tool composes Java files, a different tool composes grammar files, a third tool composes XML files, etc.

These are the ideas of AHEAD [9]. AHEAD has been used to synthesize its tool suite (i.e., AHEAD tools are bootstrapped, where the size of the tool suite exceeds 200K Java LOC), simulators for the U.S. Army [9], and product lines of portlets [48] and overlay networks [3], among other applications. Experience with AHEAD suggests that it is simple to use and understand. To recap:

- a constant module is a containment hierarchy of related code and non-code artifacts and is modeled by a vector,
- a delta is also a containment hierarchy of related code and non-code deltas, and it too is modeled by a vector,
- feature/module composition is vector composition, and
- a delta is a structural operation that adds new elements and extends existing elements.

AHEAD focussed exclusively on synthesizing artifacts by composing deltas. How artifact derivation (e.g., mapping source code to bytecode) is expressed and is related to deltas was not explored. This is the next topic.

4 Category Theory

4.1 Preliminaries

A *category* is a directed graph with special properties. Nodes are called *objects* and edges are *arrows*. An arrow drawn from object X to object Y behaves like a function with X as its domain and Y as its codomain. Arrows compose like functions, and arrow composition is associative. Also, there are identity arrows (identity functions) for each object, indicated by loops.

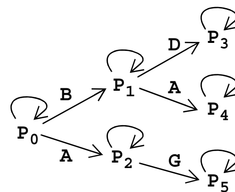


Figure 11. A Category

A product line is a category: Figure 3 is identical to Figure 11, minus identity arrows (which I henceforth omit for readability). Each object P_i in Figure 11 is a domain with one element — the *i*th program in Figure 3[4]. Arrows (deltas) are total functions.

Not part of category theory is the concept of a *synthesis line*. It is a traversal of a category from an initial object to a target object and corresponds to a GenVoca expression that synthesizes a program. The synthesis line of program P_4 is $P_0 \rightarrow P_1 \rightarrow P_4$.

A central concept in AHEAD is that it hierarchically modularizes multiple program representations. Consider the following structurally isomorphic categories $\mathcal{P}$, $\mathcal{J}$, and $\mathcal{B}$ of Figure 12. $\mathcal{P}$ is a product line of programs, $\mathcal{J}$ denotes the Java source code of these programs, $\mathcal{B}$ denotes their bytecodes. Category $\mathcal{P}$ abstracts the other two: a projection of $\mathcal{P}$ yields $\mathcal{J}$, and another projection yields $\mathcal{B}$. Further, program P_i has J_i as its source and B_i as its bytecode. When a synthesis line is drawn in $\mathcal{P}$, corresponding synthesis lines are drawn simultaneously in $\mathcal{J}$ and $\mathcal{B}$ to model the synthesis of that program's source and bytecode. That is, line $P_0 \rightarrow P_1 \rightarrow P_4$ is mapped to lines $J_0 \rightarrow J_1 \rightarrow J_4$ and $B_0 \rightarrow B_1 \rightarrow B_4$.

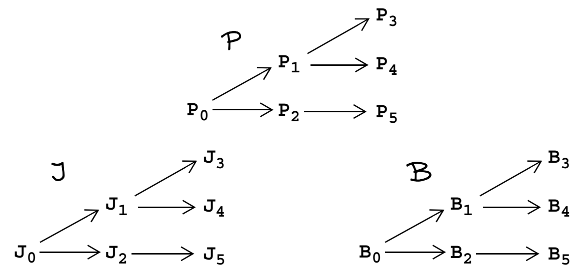


Figure 12. Categories of Program Representations

A mapping from $\mathcal{P}$ to $\mathcal{J}$ and $\mathcal{B}$ can be expressed by two ideas from category theory. The *product* of categories $\mathcal{J}$ and $\mathcal{B}$, denoted $\mathcal{J} \times \mathcal{B}$, is formed by pairing each object in $\mathcal{J}$ with each object in $\mathcal{B}$. Figure 13 displays three such pairs $[J_0, B_0]$, $[J_1, B_1]$, and $[J_4, B_4]$. An arrow between $[J_i, B_j]$ to $[J_k, B_l]$ corresponds to two arrows, one $J_i \rightarrow J_k$ in category $\mathcal{J}$ and another $B_j \rightarrow B_l$ in $\mathcal{B}$ [42].

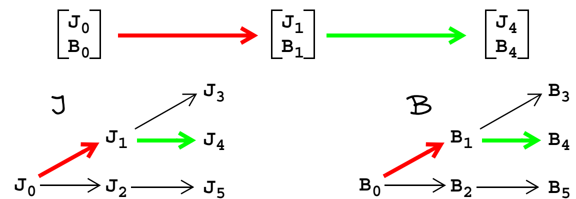


Figure 13. Product of Categories

We are not interested in *all* pairs of objects from $\mathcal{J}$ and $\mathcal{B}$, but only those that have the same subscript, namely $[J_i, B_i]$. A *subcategory* $\mathcal{D}$ of a category $\mathcal{C}$ is formed by discarding unneeded objects and arrows of $\mathcal{C}$, such that $\mathcal{D}$ remains a category [42]. Category $\mathcal{P}$ of Figure 12 is a subcategory of $\mathcal{J} \times \mathcal{B}$. As this operation is so common, we call it an *AHEAD product of isomorphic categories*, and denote it by $\mathcal{J} \otimes \mathcal{B}$. Each object P_i in category $\mathcal{P} = \mathcal{J} \otimes \mathcal{B}$ maps to a pair of objects $[J_i, B_i]$, and each arrow $P_i \rightarrow P_j$ in $\mathcal{P}$ maps to a pair of arrows $J_i \rightarrow J_j$ and $B_i \rightarrow B_j$.

4. Again, programs can have horizontal parameters [23], thus enabling domains to have multiple programs (footnote 1).

4.2 A Categorical Interpretation of AHEAD

A product line of programs is formed by the AHEAD product of categories of different program representations (Figure 14a). Each program representation category is itself an AHEAD product of lower-level isomorphic categories, recursively.

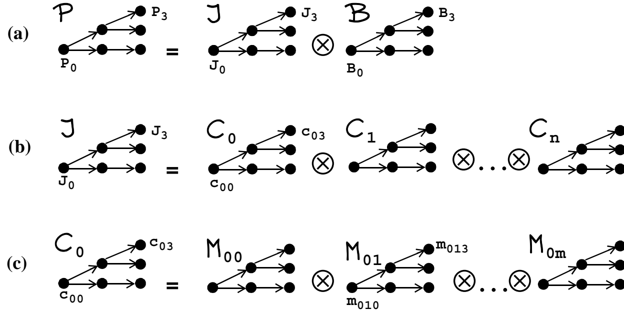


Figure 14. AHEAD Product Nesting of Categories

For example, the source code category $\mathcal{J}$ is an AHEAD product of isomorphic categories, $C_0 \ldots C_n$, one category for each class that can appear in a program. In category C_0 of Figure 14b, c_{00} denotes class c_0 in program P_0, c_{03} denotes class c_0 in program P_3, etc. The same applies to the remaining C_i categories. Arrows in C_i are class deltas. For example, the arrow in C_0 from c_{00} to c_{03} is the delta of class c_0 that occurs in the mapping of program P_0 to P_3.

Each C_i category is itself an AHEAD product of isomorphic categories $M_{i0} \ldots M_{im}$, one category for each member that can appear in class c_i.[5] In category M_{01} of Figure 14c, m_{010} denotes class c_0's member m_1 in program P_0, m_{013} denotes class c_0's member m_1 in program P_3, etc. The same applies for the remaining M_{0i} categories.

Note the scale on which AHEAD works. A typical AHEAD tool has 500 classes and well over 2000 total members. A categorical interpretation of it is a recursive nesting of AHEAD products of over 2500 different categories. Although this seems complex, it remains understandable as each category has a clear physical meaning (as a class or class member) and all categories have identical shapes.

An interesting detail is how class or member additions are modeled. Suppose class c_0 appears only in program P_3. This means that c_0 is represented by a null or nonexistent class (∅) in all other programs. Only when the $P_1 \rightarrow P_3$ arrow is traversed will a non-null class be synthesized, i.e., a class addition is modeled by a function that maps a null class to a non-null class. Interestingly, the AHEAD tool for browsing a feature code base shows nulls and identity maps between nulls of different objects; so the categorical description of AHEAD in previous paragraphs is faithful [1].[6]

4.3 Functors

Consider categories $\mathcal{J}$ and $\mathcal{B}$. A functor $F{:}\mathcal{J} \rightarrow \mathcal{B}$ is a map that takes each $\mathcal{J}$ object j to a $\mathcal{B}$ object $F(j)$ and each $\mathcal{J}$ arrow $f{:}j \rightarrow r$ to a $\mathcal{B}$ arrow $F(f){:}F(j) \rightarrow F(r)$, such that for all $\mathcal{J}$ objects j and composable $\mathcal{J}$ arrows f and g [42]:[7]

- $F(id_j) = id_{F(j)}$
- $F(g \bullet f) = F(g) \bullet F(f)$

As the categories that arise in AHEAD are structurally isomorphic, the functors between categories are particularly simple: they map corresponding objects and arrows. An interesting question is how are such functors implemented? Let $\mathcal{J}$ denote the category of Java source representations of a software product line, and let $\mathcal{B}$ denote its corresponding category of Java bytecode representations. Objects in $\mathcal{J}$ are mapped to objects in $\mathcal{B}$ by the Java compiler. That is, `javac` is a function that maps Java source to Java bytecodes. As another example, let $\mathcal{D}$ denote the category of Javadoc representations for the product line of $\mathcal{J}$. The mapping of $\mathcal{J}$ objects to $\mathcal{D}$ objects is accomplished by the `javadoc` tool.

As a general rule, *common tools used by software engineers implement object-to-object mappings of functors*.[8] This is important, as it supports the belief that common tools correspond to operations that transform program representations and thus fit naturally into algebraic models of large scale program construction at an appropriate level of abstraction.

It is interesting to note that AHEAD provides a tool for implementing source-code-arrow-to-bytecode-arrow mappings (which we'll discuss in Section 4.4). But this raises an even more basic question of how objects and arrows of a category are defined. *Today's languages provide great support for implementing objects (e.g., source code), but virtually no help in defining and composing arrows.* That is, there are no language constructs like AHEAD `refines` that define functions to map class declarations and that allow such functions to be composed [29]. It is as if one-half of a fundamental picture is missing: *language support is needed to manipulate and extend class definitions in an algebraic way.* Work on virtual classes and mixins is a good start [11][35], so too is work on Scala [39], higher-order hierarchies [21], and giving aspects "functional semantics" [34]; all go a long way to help product line development. More in Section 5.

5. Categories could also be defined to represent class initializers, implements clauses, etc.

6. The elimination of multiple null classes can be expressed categorically using functors. The explanation given above is the simplest to see a categorical connection to AHEAD.

7. The identity arrow for object s is id_s.

8. Tools like `javac` often correspond to natural transformations [42].

4.4 Commutativity Diagrams

A fundamental relationship exists between artifact derivation and artifact deltas. It can be expressed by the *commutativity diagram* [42] of Figure 15. Horizontal arrows are artifact deltas (a.k.a. *endogenous transformations* [37]) and vertical arrows are artifact derivations (a.k.a. *exogenous transformations* [37]).

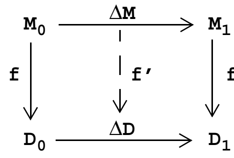


Figure 15. Commutativity Diagram

A fundamental property of commutativity diagrams is that any path from the upper left object to the lower right object produces the same result. In the case of Figure 15, a higher-order function or *operator* `f'` maps function `ΔM` to function `ΔD`. The general relationship is:

$$f \bullet \Delta M = f'(\Delta M) \bullet f \quad (5)$$

Functors give rise to commutativity diagrams. Figure 16 shows how the Java source of programs P_0 and P_1 relate to their corresponding bytecodes B_0 and B_1.

The horizontal arrow $J_0 \rightarrow J_1$ is a source code delta ΔJ (i.e., a set of AHEAD class additions and class deltas), and the $B_0 \rightarrow B_1$ arrow is the corresponding bytecode delta ΔB. The vertical arrows are the mappings of a functor from $\mathfrak{J}$ to $\mathcal{B}$. `javac` implements the object-to-object mappings. AHEAD has a special tool to implement the arrow-to-arrow mappings (i.e., `ΔB = mapdelta(ΔJ)` indicated in Figure 16 by the dashed arrow). This commutativity relationship:

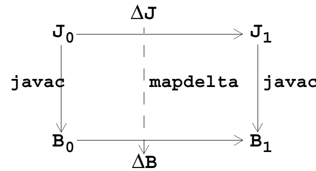


Figure 16. Source-Bytecode Diagram

$$javac \bullet \Delta J = mapdelta(\Delta J) \bullet javac \quad (6)$$

is an instance of (5). That is, extending source of J_0 by ΔJ and compiling equals compiling the source of J_0 and extending its bytecode by ΔB.

Proofs should be offered to justify commutativity diagrams. But the scale of programs in AHEAD makes this difficult. I believe that `javac` preserves the class structure of a Java source program in its translation to bytecodes, but I know of no formal proof of this. Similarly, I believe that the source or bytecode deltas that I use preserve or elaborate the original source or bytecode structure, but again I have no formal proof. More important, however, is proving that *properties* of features are preserved (or correctly transformed) by artifact derivation and composing artifact deltas. Proving properties of arrows on the scale of Figure 16 seems appropriate for the recently proposed Verified Software Grand Challenge of Hoare, Misra, and Shankar [29], which seeks scalable technologies for program verification.

In the absence of proofs, forms of equivalence can be empirically demonstrated by building both ways. That is, start with program $j_0 \in J_0$, and produce programs $b_1 = javac(\Delta J \bullet j_0)$ and $b_1' = mapdelta(\Delta J) \bullet javac(j_0)$ and test their equivalence. In the case of bytecodes, b_1 and b_1' are subjected to the same set of system or integration tests; if both have the same responses, these systems can be considered behaviorally identical (for those tests). In the case that a commutativity diagram yields a source document, "diffs" can be used to test for source equivalence. (*Source equivalence* is syntactic equivalence with two relaxations: it allows permutations of members when member ordering is not significant and it allows white space to differ when white space is unimportant).

Experience to date is that commutativity diagrams expose a *large* number of relationships in SPL models, and impose a correspondingly large number of constraints on tools. (Without commutativity diagrams, we were unaware of these constraints). Not surprisingly, our tools initially did not satisfy these constraints, and consequently had to be repaired. By fixing our tools, we have greater confidence in them because they implement explicit relationships in categorical models of product lines, and that tools can be treated as operations of an algebra. Both are big wins from an engineering perspective because we can reason algebraically about our designs, rather than hacking code to implement imprecise designs. This is a good example where category theory exposes important and previously unrecognized properties that program synthesis tools must satisfy.

4.5 Synthesis Geometries

Suppose programs in R_0 have specifications from which multiple representations can be derived. Figure 17a shows a configuration space for R_0 consisting of three representations that can be derived from the topmost object. Moving forward in time by drawing horizontally the synthesis line $R_0 \rightarrow R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow R_4$, we sweep out a mesh of commutativity diagrams here called a *synthesis geometry*. The geometry of Figure 17b is regular, although it could just as easily be ragged (Figure 17c). Meshes are created by translating delta arrows that connect the topmost objects into delta arrows of lower-rung objects. Ragged geometries arise when delta arrows cannot be translated.[9]

9. In principle, such arrows exist, but there may not be a tool to compute them. For example, until we built the `mapdelta` tool, we could not implement the $B_0 \rightarrow B_1$ arrow of Figure 16, even though we knew such an arrow should exist. In general, mapping arrows is non-trivial.

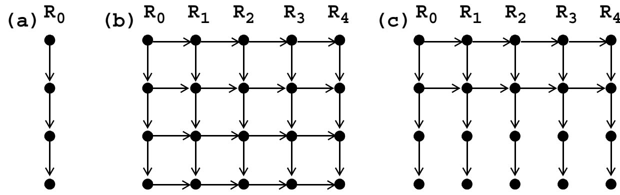


Figure 17. Synthesis Geometries

Given an object in the upper left corner, we generally want to compute the object in the lower right. Any path from the upper left to the lower right will produce the desired result. For a rectangular $m \times n$ mesh, there are $\binom{m+n}{m}$ such paths.

Given a metric that defines the cost of traversing (synthesizing and composing) an arrow, synthesis geometries warp (Figure 18). No longer are all paths equidistant. It becomes an interesting problem to determine the shortest path, called a *geodesic,* to synthesize a target result.

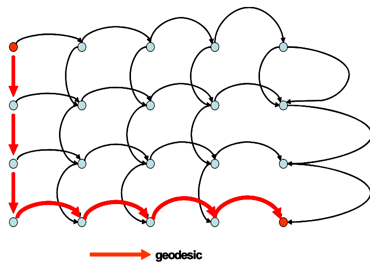


Figure 18. Geodesic

The following describes the *PinkCreek* project on which I worked with S. Trujillo and O. Diaz [48]. PinkCreek is a product-line of web portlets (i.e., web components). Our work combined notions of model driven architectures with feature oriented designs. An initial configuration space of objects (a.k.a. *models*) is displayed in Figure 19a, where a series of different artifacts were derived from the topmost object. As we moved forward in time, a multi-pleated synthesis geometry was created (Figure 19b).

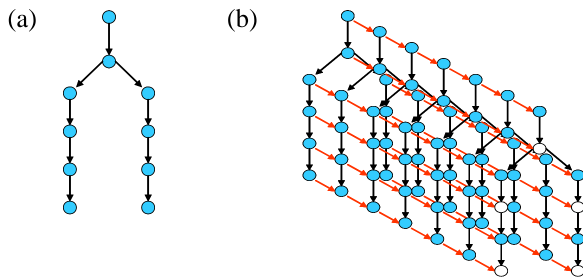


Figure 19. PinkCreek Geometries

PinkCreek geodesics are not lines. Starting at the upper left object (which corresponds to a base statechart), there is a set of terminal objects on the final plane that are to be computed (indicated as white circles in Figure 19b). Initially, my thought was to traverse the spine of the geometry and follow the arrows downward on the final plane. "Traversing the spine" means refining the statechart into its final form, and then deriving its representations. This, as it turns out, was not a geodesic. A more efficient path was discovered experimentally: start from the *initial* statechart, derive its representations, and then proceed forward in time for each desired representation. This second approach was 2-3 times faster than my path and was a consequence of special optimizations that were possible in the PinkCreek design. Without these optimizations, there would have been a factor of 10 difference in favor of my path.

When there is only one terminal object, a geodesic can be computed by Dijkstra's shortest path algorithm [2]. In PinkCreek, there is one initial object and *n* terminal objects and computing a geodesic requires solving the Directed Steiner Tree Problem, which is NP-hard [14]. More generally, a geodesic can have *m* initial objects and *n* terminal objects. While I suspect that some simple heuristics for computing geodesics may suffice, there may be some interesting optimization problems to be addressed.

4.6 Additional Dimensions of Time

There are many other connections of AHEAD to category theory. Perhaps the most intriguing is multiple dimensions of time in software design. Here is the idea: suppose Figure 20a displays a configuration space of objects. Figure 20b shows a geometry that is swept as we move forward in one dimension of time. Figure 20c shows the geometry going forward along a second dimension of time. Figure 20d shows a third dimension. Each dimension is described by a distinct product line (i.e., a distinct set of composable features). A particular product is described by an expression (feature composition) along each dimension. That is, a program is represented by *n* expressions, one per dimension, in an *n*-dimensional model. Categorically this space is represented by product of non-isomorphic categories. Pushouts are used to compute arrows.

A classic example is Cook's *Extensibility Problem* [17], now known as the *Expression Problem* [50], which is a fundamental problem of software design: the focus is achieving data type and operation extensibility in a type-safe manner. Incrementally extending a data type is one dimension; adding operations is a second [33]. We rediscovered these ideas in the context of synthesizing large programs, and in fact, use three dimensions of time when synthesizing AHEAD tools [7][8].

4.7 Perspective

Category theory has an unfortunate reputation in Software Engineering. The most common refrain I hear is "category

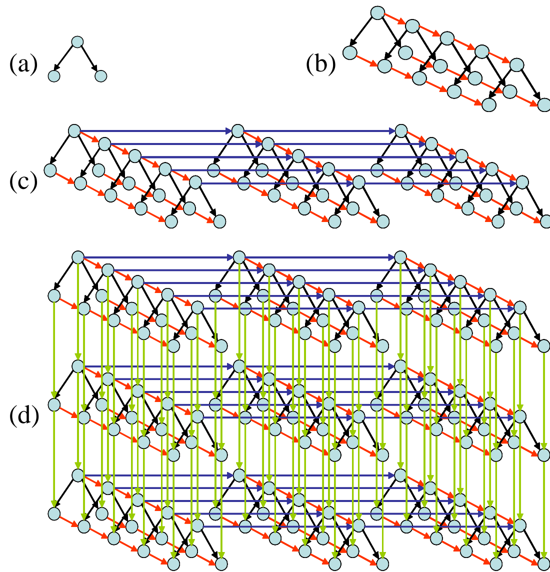


Figure 20. Geometries with Multiple Dimensions of Time

theory is so general, it can only be descriptive, not prescriptive", meaning it doesn't tell you what to do.

My experience is different. Commutativity diagrams expose many relationships in SPL models that must be satisfied by synthesis tools. This, as noted earlier, is a big win. Misra once noted "A theory is a tool that reduces experimentation, often by an infinite amount [38]." That statement applies here: category theory has saved me years of experimentation. I would have figured out some (not all) of these ideas eventually, but the real problem would have been trying to convince others of a general paradigm without firm theoretical backing. I'm not sure it would be possible.

Here is another example where I found category theory to be prescriptive. The synthesis geometries of AHEAD are straight lines (start with a base artifact and progressively extend it). By definition, such lines *are* geodesics, so it was OK to compose arrows immediately. Category theory exposes a much more general approach: collect the arrows, determine the synthesis geometry, and then compute a geodesic. This generalization is not obvious, but now forms a basis for my future work in program synthesis.

5 Related and Future Work

AHEAD is a model of *metaprogramming*, i.e., that program design and implementation is a computation. What we described in Section 4.5 is a form of *staged programming*, where computing a geodesic synthesizes a metaprogram, which when evaluated, synthesizes a target result. Staged programming was introduced in [47], but it is different from our usage.

Object-oriented collaborations (or role-based designs) were perhaps the first recognizable way to express features [43]. The ideas of collaborations could be implemented by virtual classes [35] and mixins [11], and have been the basis of several feature-based design methodologies [49][45]. Unfortunately, support for collaborations has not found its way into main-stream programming languages.

AHEAD has similar goals to Goguen's parameterized programming. His work offers two distinct forms of parameterization, horizontal and vertical, and uses views to define morphisms (mappings) between module interfaces that would otherwise not be composable. AHEAD uses vertical parameters to express deltas, although features (as mentioned in footnote 1) can indeed have horizontal (e.g., performance) parameters. AHEAD does not use views. As programs get larger, the likelihood that features will have compatible semantics but not have syntactically identical interfaces is unlikely. Views can be used in circumstances as indicated in Goguen's work.

Ernst's *Higher-Order Hierarchies (HOH*) has much of the flavor of AHEAD, where any number of (virtual) classes in an inheritance hierarchy can be extended lock-step. Different extensions can be composed, i.e., there is an "algebra" to specify extension compositions. Implementations of HOH exist in gbeta and CaesarJ [21]. There are two basic distinctions between AHEAD and HOH. First, AHEAD deals with hierarchies of noncode artifacts, as well as code artifacts. Second, HOH uses virtual classes (i.e., classes that are nested inside other classes and that can be extended), and can instantiate the enclosing classes. AHEAD simplifies HOH by eliminating enclosing classes and the ability to instantiate them. Further, analysis of product lines is simplified in AHEAD as there are models that define all programs and all legal feature combinations. HOH has no product line models.

Scala is another language that can express program code deltas [39]. Scala is general, and requires programmers to express "type plumbing", i.e., type bindings in deltas. AHEAD's `refines class` construct is much more limited, and hides (or assumes) type bindings. Consequently, `refines class` declarations are more compact and may be easier for typical programmers to use [33]. A major difference is Scala has a type system; AHEAD does not.

In the algebraic specifications community, the terms refinement and extension have different meanings. Consider a 2-space, where points along the X-axis are specifications, and points along the Y-axis are implementations. A standard paradigm (e.g., Z [46]) builds a specification incrementally by extensions. Once completed, the specification is refined progressively into an implementation. The horizontal

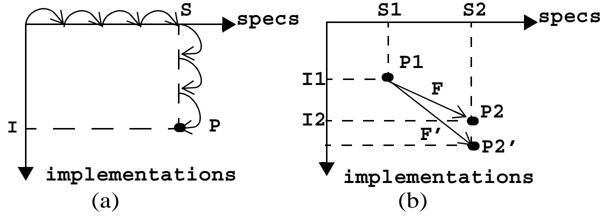


Figure 21. Extension and Refinement

arrows below denote specification extensions, and the vertical arrows are refinements. Program `P` has specification `S` and implementation `I` in Figure 21a.

AHEAD is different. Following the ideas in Section 3, a feature both extends a specification and refines an implementation, i.e., features move diagonally through this 2-space. In Figure 21b, program `P1=(S1,I1)` is mapped to program `P2=(S2,I2)` by feature `F`. It is possible for features to share the same specification extension (e.g., `F'` in Figure 21b), which leads to optimizations (e.g., which feature produces the most efficient program?) [44]. As best as I can tell, features implement a form of constrained subtypes [31], where type specifications are very weak.

Specware uses category theory as a formal foundation for program synthesis [41]. Specifications are composed by pushouts, which is inherently a commutative operation. Pushouts are appropriate as the specifications that are composed in Specware are orthogonal (i.e., pushouts effectively compute specification union). AHEAD uses a different model: feature composition is function composition, which is not commutative. To illustrate, suppose feature `F1` increments field `V`, and feature `F2` doubles `V`. The order in which `F1` and `F2` are composed matters. This example would lead to inconsistencies in Specware. Of course, a major advantage of Specware are guarantees of correctness in the code that it synthesizes; AHEAD offers no such assurances. Despite these differences, much could be gained by a closer comparison of both approaches.

Research in algebraic specification uses category theory and commutativity diagrams to express ideas similar to those in this paper [20][41]. In fact, the basic notion that refinements affect different representations of a module (e.g., its interface, implementation, parameters) is clearly present. In a similar vein, there is an enormous literature on category theory (e.g., [28][42]). It is an interesting and open problem to see how these different areas of research can cross-fertilize each other: results in theory finding practical outlets, and practical examples to illuminate theory.

Finally, the ideas presented in this paper are not restricted to the topics covered. *Model driven architectures (MDA)* is an emerging paradigm where models (i.e., different representations of programs) are extended and derived from other models by transformations. The work on PinkCreek in Section 4.5 is an example that combines ideas of MDA and features. Autonomic computing deals with the dynamic restructuring of applications (conceptually similar to automatic programming discussed in Section 2). Service oriented architectures deals with the modularization of services, increments of program functionality that have distributed implementations. And finally, program refactorings can be understood as program transformations. The essence of all these topics can be understood in terms of functions that derive or modify programs.

6 Conclusions

Product synthesis will become increasingly important to future software development. The challenge in scaling synthesizers is not one of possibility: there are lots of ad hoc ways of doing this now. Rather, the challenge is to show how scaling can be accomplished in a principled manner so that synthesizers are not just ad hoc collections of tools using an incomprehensible patchwork of techniques. Product synthesis is a technological statement that the development of products (programs) in a domain is understood well enough to be automated. However, we must make the same claim for synthesizers themselves: their complexity must also be controlled and must remain low as product complexity scales; otherwise, synthesis technologies will be gruesome to use. Product synthesis must be placed on a more formal basis.

Programming languages will be vital to this effort. We need languages that will enable programmers to define both objects and arrows. Work on mixins and virtual classes, for example, is a good start. The key step in advancing programming languages is to think in terms of functions that map (class) structures, and to abandon the use of inheritance for this purpose. Doing so, it will be much easier to recognize the mathematical elegance of product synthesis.

It will take many years to sort out all of these issues, and will take the effort of many communities to do so. As programmers, we are geniuses at making the simplest things look complicated; the hard part is recognizing the underlying simplicity. Scalability of product synthesis demands simplicity, uniformity, and regularity: algebra offers this, and this is the reason why I use it.

Acknowledgements. I greatly appreciate the comments from M. Mehlich, G. Lavender, M. Poppleton, S. Nedunuri, W. Cook, S. Apel, S. Trujillo, O. Diaz, A. Rauschmayer, M. Wirsing, E. Boerger, and J. Misra on earlier drafts of this paper. Also, I thank C. Lengauer for recognizing the vector description of AHEAD, and V. Ramachandran and R. Chowdhury for their help in connecting my work with the

Directed Steiner Tree Problem, and discussions with J. McGregor on product line testing. This work was supported by NSF's Science of Design Project #CCF-0438786.

7 References

[1] AHEAD Tool Suite, `www.cs.utexas.edu/users/schwartz/index.html`

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1975.

[3] S. Apel and D. Batory. "When to Use Features and Aspects? A Case Study", *GPCE 2006*.

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

[5] D. Batory, G. Chen, E. Robertson, and T. Wang. "Design Wizards and Visual Programming Environments for GenVoca Generators", *IEEE TSE*, May 2000.

[6] D. Batory, R. Cardone, and Y. Smaragdakis. "Object-Oriented Frameworks and Product-Lines", *SPLC 2000*.

[7] D. Batory, R. Lopez-Herrejon, and J.P. Martin. "Generating Product-Lines of Product-Families", *ASE 2002*.

[8] D. Batory, J. Liu, J.N. Sarvela. "Refinements and Multidimensional Separation of Concerns", *ACM SIGSOFT 2003*.

[9] D. Batory, J.N. Sarvela, and A. Rauschmayer. "Scaling Step-Wise Refinement", *IEEE TSE*, June 2004.

[10] D. Benavides, P. Trinidad, and A. Ruiz-Cortes, "Automated Reasoning on Feature Models", *CAISE 2005*.

[11] G. Bracha and W. Cook. "Mixin-Based Inheritance". *OOPSLA and ECOOP 1990*.

[12] T. Biggerstaff and A. Perlis, *Software Reusability Volume II: Applications and Experiences*, Addison-Wesley, 1990.

[13] BMW. `www.bmwusa.com`

[14] M. Charikar, et al., "Approximation Algorithms for Directed Steiner Tree Problems, *ACM-SIAM Symposium on Discrete Algorithms (SODA) 1998*.

[15] P. Clark and B. Porter. "Building Concept Representations from Components", National Conference on Artificial Intelligence, 1997.

[16] E.F. Codd. "A Relational Model of Data for Large Shared Data Banks", *CACM 13 (6)*, 1970.

[17] W.R. Cook. "Object-Oriented Programming versus Abstract Data Types". *Workshop on FOOL*, Lecture Notes in Computer Science, Vol. 173. Spring-Verlag (1990).

[18] Dell Computers. `www.dell.com`

[19] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[20] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, Springer-Verlag, 1990.

[21] E. Ernst, "Higher Order Hierarchies", *ECOOP 2003*.

[22] M. Fayad and D.C. Schmidt, "Object-Oriented Application Frameworks", *CACM*, Oct. 1997.

[23] J. Goguen. "Principles of Parameterized Programming" in [12].

[24] J. Goguen. "A Categorical Manifesto". *Mathematical Structures in Computer Science*, 1991.

[25] J. Gray, "What Next? A Dozen Information-Technology Research Goals", Microsoft Research MSR-TR-99-50, 1999.

[26] K. Kang, et al., "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Tech Report CMU/SEI-90-TR-21.

[27] K.K. Lau. "Top-down Synthesis of Sorting Algorithms", *The Computer Journal*, 1992.

[28] F.W. Lawvere and S.H. Schanuel, *Conceptual Mathematics: A First Introduction To Categories*, Cambridge University Press, 1997.

[29] G. Leavens, et al. "Roadmap for Enhanced Languages and Methods to Aid Verification". Dept. Comp. Sci., Iowa State Univ., TR #06-21, July 2006. `ftp://ftp.cs.iastate.edu/pub/techreports/TR06-21/TR.pdf`

[30] H.C. Li, S. Krishnamurthi, and K. Fisler. "Modular Verification of Open Features Through Three-Valued Model Checking", *Automated Software Engineering Journal*, 2005.

[31] B. Liskov and J.M. Wing, "A Behavioral Notion of Subtyping", *ACM TOPLAS 1994*.

[32] R.E. Lopez-Herrejon and D. Batory. "A Standard Problem for Evaluating Product-Line Methodologies", *GCSE 2001*.

[33] R. Lopez-Herrejon, D. Batory, and W. Cook. "Evaluating Support for Features in Advanced Modularization Technologies", *ECOOP 2005*.

[34] R. Lopez-Herrejon, D. Batory, and C. Lengauer. "A Disciplined Approach to Aspect Composition", *PEPM 2006*.

[35] O.L. Madsen and B. Møller-Pedersen, "Virtual Classes: A Powerful Mechanism in Object-Oriented Programming", *OOPSLA 1989*.

[36] D. MacQueen. "An Implementation of Standard ML Modules", *ACM Conference on LISP and Functional Programming*, 1988.

[37] T. Mens, K. Czarnecki, and P. van Gorp. "A Taxonomy of Model Transformations", Dagstuhl Seminar Proceedings 04101 `http://drops.dagstuhl.de/opus/volltexte/2005/11`.

[38] J. Misra, "Inaugural Lecture for the Schlumberger Centennial Chair", 2002. `http://www.cs.utexas.edu/users/misra/Speeches.dir/Schlumberger.Jan02.pdf`

[39] M. Odersky, et al. "An Overview of the Scala Programming Language". September 2004, `scala.epfl.ch`

[40] D. L. Parnas, "On the Design and Development of Program Families", *IEEE TSE*, vol. SE-2#1 1976.

[41] D. Pavlovic and D.R. Smith. "Software Development by Refinement", UNU/IIST 10th Anniversary Colloquium, Formal Methods at the Crossroads: From Panaea to Foundational Support, Springer-Verlag LNCS 2757, 2003.

[42] B. Pierce. *Basic Category Theory for Computer Scientists*, MIT Press, 1991.

[43] T. Reenskaug, et al. “OORASS: Seamless support for the creation and maintenance of object-oriented systems”. *Journal of Object-Oriented Programming*, October 1992.

[44] 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*.

[45] Y. Smaragdakis and D. Batory, “Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs”, *ACM TOSEM*, April 2002.

[46] J.M. Spivey, *The Z Notation: A Reference Manual*, Oxford University Press, 1998.

[47] W. Taha and T. Sheard. “Multi-Stage Programming with Explicit Annotations”, *PEPM 1997*.

[48] S. Trujillo, D. Batory, and O. Diaz. “Feature Oriented Model Driven Development: A Case Study for Portlets”, submitted 2006.

[49] M. VanHilst and D. Notkin. “Using role components to implement collaboration-based designs”. *OOPSLA 1996.*

[50] P. Wadler. “The Expression Problem”. Posted on the Java Genericity mailing list (1998).

[51] N. Wirth. “Program Development by Stepwise Refinement”, *CACM*, April 1971.

[52] C. Zhang, G. Gao, and H.-A. Jacobsen. “Towards Just-in-time Middleware Architectures”, *AOSD 2005.*