Invited Presentation, at European Joint Conferences on Theory and Practice of Software (ETAPS) Compiler Construction Conference (April 2007)

Program Refactoring, Program Synthesis, and Model-Driven Development

Don Batory
Department of Computer Sciences
University of Texas at Austin

Abstract. Program refactoring, feature-based and aspect-oriented software synthesis, and model-driven development are disjoint research areas. However, they are all architectural metaprogramming technologies as they treat programs as values and use functions (a.k.a. *transformations*) to map programs to other programs. In this paper, I explore their underlying connections by reviewing recent advances in each area from an architectural metaprogramming perspective. I conjecture how these areas can converge and outline a theory that may unify them.

1 Introduction

Among the greatest challenges that we face today is dealing with the alarming complexity of software, and the alarming rate at which software complexity is increasing. Brooks observed 20 years ago that programmers spent a majority of their time on accidental complexity, rather than essential complexity [12]. Unfortunately, we often can't tell the difference between the two.

Complexity is controlled by imposing structure. This paper is about the essential complexity of software structure. There are increasingly overlapping ideas in the areas of program refactoring, program synthesis, and model-driven development, all of which deal with program structure and maintenance. I conjecture how these areas can converge and outline a theory that may unify them.

I have long believed there is a common conceptual foundation for what we do in programming, software design, and maintenance. The results I expect from making these foundations explicit are increasing automation, building better tools, writing better code, and reducing program development and maintenance costs. All are worthy goals. But there may be an even bigger prize: discovering the material that will be taught to future graduates and undergraduates.

By, say, 2020 (which I hope will be a good year), programmers will be writing functions, objects, classes, and methods just as they do today. But there will be a difference in the level of abstraction at which programs are written. I expect the rise of *architectural metaprogramming*: the idea that programming and design is a computation, where programs are values and functions (a.k.a. *transformations*) map programs to programs.

In the following sections, I sketch the ideas architectural metaprogramming, and then reflect on recent advances in program refactoring [17][18], program synthesis [26], and model-driven development [7][37] from its perspective.

2 Basics of Architectural Metaprogramming

Programs are values. Figure 1a shows a value `C` that is the Java definition of a class `c`. Figure 1b shows another value `D`; it is the Java definition of a class `d`. (I use Java, but any language could be used provided that one shows how concepts translate).

```
class c {
  int x;
  void inc() {x++;}
}
```

(a) value `C`

```
class d {
  int compute() {}
}
```

(b) value `D`

Figure 1 Values `C` and `D`

Values can be added. The sum `C+D` is a program with classes `c` and `d`. As another example: let `C1` be the definition of a class `c` with a `comp()` method (Figure 2a), and let `C2` be another definition of class `c` which has an `x` field and an `inc()` method (Figure 2b). `C1+C2` yields a single definition of class `c` (Figure 2c), formed by the disjoint union of the members in `C1` and `C2`.[1]

```
class c {
  int comp() {}
}
```

(a) value `C1`

```
class c {
  int x;
  void inc() {x++;}
}
```

(b) value `C2`

```
class c {
  int comp() {}
  int x;
  void inc() {x++;}
}
```

(c) value `C1+C2`

Figure 2 Sum

S*ummation* (or simply "sum") is disjoint set union [26] with the properties:

- Sum identity `0` is the *null program* or *null value*. For any program `P`: `P+0=P`
- Sum is commutative (as disjoint set union is commutative): `A+P=P+A`
- Sum is associative (as disjoint set union is associative): `(A+B)+C=A+(B+C)`

Values can be subtracted. *Subtraction* is set difference; if a program is formed by `C+D`, and `C` is subtracted, the result is `D`: `(D+C)-C=D.` Subtraction has the properties:

- `0` is the identity: `P-0=P` and `P-P=0`
- Subtraction is left associative: `P-C-D=((P-C)-D)`
- Subtraction is not commutative: `P-C` ≠ `C-P`

```
class c {
  int z;
  void inc(){z++;}
}
```

Figure 3 Value `C3`

A third operation is really a menagerie of operations called *distributive transformations (DTs)*. (The reason for the name will become clear shortly). One is `rename(p,q,r)`: in program `p`, replace name `q` with name `r`. Recall program `C2` (Figure 2b). Suppose we want to replace name "`x`" with name "`z`". The computation `rename(C2,x,z)` does this replacement and yields value `C3`, shown in Figure 3.

---

1. Think of a primitive value as an aspect-oriented introduction that defines a member and its class (e.g., "`int c.x;`") [22]. The same for other declarations such as initialization blocks, `extends` clauses, etc. When values are converted to source code, their members are collected into classes, and classes into packages, to show their hierarchical modularities.

Consider another computation: `rename(D,x,z)` which equals `D`. That is, `rename` leaves `D` unaltered as `D` does not reference `x`. In mathematics this is a *fixed point*, i.e., a value `x` such that `f(x)=x`. DTs usually have many fixed points.

The key property of DTs is that they distribute over `+` and `-` (hence their name). That is, a DT `f` of a sum equals the sum of the transformed values. The same for subtraction:

```
f(A+B) =  f(A) + f(B)
f(A-B) =  f(A) - f(B)                                      (1)
```

As an illustration, consider renaming "`x`" to "`z`" in program `C2+D`. This computation, `rename(C2+D,x,z)`, is performed by applying `rename` to programs `C2` and `D` individually, and summing their results:

```
rename(C2+D,x,z)
= rename(C2,x,z) + rename(D,x,z)    // distribution
= C3 + D                            // evaluation
```

DTs have other properties. Transforming a non-null value yields a non-null value:

```
f(x) ≠ 0 ; where x ≠ 0                                     (2)
```

That is, applying a DT to a non-null value will not nullify (erase, delete) that value, but may alter it. If a value is to be deleted, subtraction should be used. And the null value cannot be transformed:

```
f(0) = 0                                                   (3)
```

Another property is composition. DTs are functions, and thus compose like functions. If `f1` and `f2` are DTs, `f1•f2` denotes their composition. Function composition is not commutative (`f1•f2 ≠ f2•f1`) and is associative (`(f1•f2)•f3 = f1•(f2•f3)`).

Expressions that are formed by adding, subtracting, and transforming programs are *architectural meta-expressions* or simply *meta-expressions*.

Before proceeding further, I use the term "*structure*" to mean what are the parts and how are they connected? The structure of a cube, for example, is a solid bounded by six equal squares, where two adjacent faces meet at a right angle. The term "*property*" is an attribute that is given or is derivable from a structure. If `E` is the length of an edge (a given property), derivable properties of a cube are its surface area ($6*E^2$) and volume (E^3). The software analog: the structure of a program is its architectural meta-expression. Compilers prove properties of a program by analyzing its structure, such as the property of type correctness. In this paper, I focus solely on program structure. Now let's look at some applications of architectural metaprogramming.

3 Recent Advances in Program Refactoring

A *refactoring* is a transformation that changes the structure of a program, but not its behavior [20]. Classic examples include rename method and move a method from a subclass to a superclass. Common *Integrated Development Environments (IDEs)*, such as Eclipse, Visual Studio, and IntelliJ, have built-in or plug-in refactoring tools. Discussed below is an interesting problem in program refactoring.

The use of components (e.g., frameworks and libraries) is common in contemporary software development. Components enable software to be built quickly and in a cost-effective way. The *Application Program Interface (API)* of a component is a set of Java interfaces and classes that are exported to application developers. Whenever an API changes, client code that invokes the API must also change. Such changes are performed manually and are disruptive events in program development. Programmers want an easy (push-button) and safe (behavior-preserving) way to update their applications when a component API changes [17][18].

Figure 4 illustrates an API change called "move method". An instance method **`m`** of a **`home`** class (Figure 4a) becomes a static method **`m`** of a **`host`** class (Figure 4b). The moved method takes an instance of the **`home`** class as an extra argument, and all calls to the old method are replaced with calls to the new method.

Figure 4 shows the essence of the problem: above the dashed lines is component code, and below is client code. When the API is refactored, the client code changes. As component developers do not have access to client code, the client programmer must manually update his/her own code.

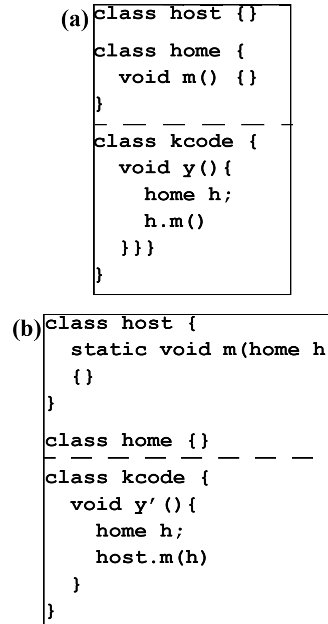

Figure 4 Move Method

This API change can be written as an architectural meta-expression. Let value **`home.m`** denote the **`home`** method **`m()`**, and let **`host.m`** denote the **`host`** method **`m()`**. Let μ be the DT that transforms **`home.m`** to **`host.m`**, and otherwise leaves all other primitive values unchanged. That is, μ**`(home.m)=host.m`** and for all **`x`**$\neq$**`home.m`**: μ**`(x)=x`**. Let ϕ be the DT that renames all calls to **`home.m`** to calls to **`host.m`**, and otherwise leaves primitive values unchanged. That is, ϕ**`(kcode.y)=kcode.y'`** and for all primitives **`x`** that do not call **`home.m`**: ϕ**`(x)=x`**. The meta-expression that relates the updated program (P_{new}) to the original program (P_{old}) is:

$$P_{new} = \phi\bullet\mu(P_{old}) = \mu\bullet\phi(P_{old}) \qquad (4)$$

In this particular case, the order in which μ and ϕ are composed does not matter. The reason is that each transformation changes different code fragments (much like two pieces of aspect-oriented advice advising different join points [22]).

To see how computation **(4)** proceeds, let P_{old}**`=home.m+kcode.y+..`**:

$$\begin{aligned}
&\phi\bullet\mu(P_{old}) \\
&= \phi\bullet\mu(\texttt{home.m + kcode.y + ..}) && \texttt{// substitution} \\
&= \phi\bullet\mu(\texttt{home.m}) + \phi\bullet\mu(\texttt{kcode.y}) + \texttt{..} && \texttt{// distribution} \\
&= \phi(\texttt{host.m}) + \phi(\texttt{kcode.y}) + \texttt{..} && \texttt{// evaluation of } \mu \\
&= \texttt{host.m + kcode.y' + ..} && \texttt{// evaluation of } \phi \\
&= P_{new}
\end{aligned}$$

Other API changes (refactorings) besides move method include: move field, delete method (which is usually done after a method is renamed or moved), change argument type (replace an argument type with its supertype), and replace method call (with another that is semantically equivalent and in the same class) [17]. My preliminary work suggests that these and other refactorings can be written as meta-expressions.

In a recent paper, Dig and Johnson explored how APIs evolve [18]. They manually analyzed the change logs, release notes and documentation of different versions of five medium to large systems (e.g., 50K to 2M LOC), including Eclipse, Struts, and JHotDraw. They discovered that over 80% of the API changes were due to refactorings. This means that a large fraction of API changes can be fully automated.

By 2020, programmers will use advanced IDEs that will "mark" API interfaces, classes, methods, and fields. The only way marked elements can change is by refactorings. When a new version of a component is released, the refactorings of its API are also released. These refactorings are applied automatically to the client code whenever a client installs a new version of a component, thereby avoiding the tedious and error prone changes that are now performed manually. In this way, the disruptive effects of updating component versions are minimized.

Underneath the covers, future IDEs will use architectural meta-expressions to perform these updates. Assume that DTs θ are sufficient to express API refactorings. Further assume that private edits to a component, which change component internals and are invisible to clients, are also modeled by transformations ε. Updating component V_0 to version V_1 is an interleaved sequence of refactorings and private edits, such as:

$$V_1 = \varepsilon_6 \bullet \varepsilon_5 \bullet \varepsilon_4 \bullet \theta_3 \bullet \varepsilon_3 \bullet \varepsilon_2 \bullet \theta_2 \bullet \theta_1 \bullet \varepsilon_1 (V_0) \quad (5)$$

The IDE will keep a history of these changes. The modifications $\underline{\theta}$ of V_0 that may alter client code are the API refactorings, which is the projection of the changes of (5) with private edits removed:

$$\underline{\theta} = \theta_3 \bullet \theta_2 \bullet \theta_1 \quad (6)$$

The metaprogramming function U automatically updates a client program P_0 that uses V_0 to a program P_1 that uses V_1, where $P_1 = U(P_0)$:

$$U(x) = \underline{\theta}(x - V_0) + V_1 \quad (7)$$

To see a computation, let client program $P_0 = C + V_0$, where C is the client code to be updated. Applying U to P_0 updates P_0's code (transforming C to $\underline{\theta}(C)$) and replaces V_0 with V_1. This is the essential idea behind [18].

$$\begin{aligned}
& U(P_0) \\
&= \underline{\theta}(P_0 - V_0) + V_1 && \text{// substitution of } U \\
&= \underline{\theta}(C + V_0 - V_0) + V_1 && \text{// substitution of } P_0 \\
&= \underline{\theta}(C) + V_1 && \text{// subtraction} \\
&= P_1
\end{aligned}$$

Note that U can be applied to any program P_0, whose size can be arbitrarily large. One of the benefits of architectural metaprogramming is that its concepts scale to large programs.

Perspective. By 2020, IDEs will be *component evolution calculators*. They will allow programmers to edit components, and perhaps invisible to programmer actions, IDEs will create metaprogramming update functions like **`U`** for distribution. When a client wants a new version of a component, s/he will download a metaprogramming function **`U`** rather than the new version itself. The client's IDE will then apply **`U`** to the client's code base, automatically and safely updating the client's program.[1] An interesting research problem is to generalize the above analysis to deal with refactorings that involve value additions and subtractions, and to develop in detail an algebra for refactorings in conjunction with a refactoring tool to show the connection between theory and practice.

4 Recent Advances in Program Synthesis

Declarative languages will be used to specify programs in 2020. Unlike past work that relied on formal logic specifications (and compilers to derive program implementations from such specifications), the languages I envision will be much simpler. They will exploit results from *Software Product Lines (SPLs)*, an area of research that focuses on designs for a family of systems and on automating system construction. A fundamental idea in SPL is using features to describe and differentiate programs within a family, where a *feature* is an increment in functionality [21][14].

Features are used in many engineering disciplines for product specification. At the Dell web site, customers configure a personal computer (i.e., a product in a Dell product line) by selecting optional hardware and software features listed on a web page [16]. Such pages are *Declarative Domain-Specific Languages (DDSLs)* for Dell products. Another example is BMW's web site to customize an automobile [11].

Software can be specified in the same way. Figure 5 shows an elementary DDSL for a product-line of Java programs. Called the *Graph Product Line (GPL)*, each program implements a unique combination of graph algorithms [25]. A particular program is specified by selecting a set of features. The program specified in Figure 5 (reading selected features from left to right) implements vertex numbering, strongly connected components, and cycle checking using a depth first search (**`DFS`**) on a weighted, directed graph. More generally, each feature can be customized via parameters (much like GUI components have customizable property lists [2]), but the essential idea of declarative feature selections remains.

Figure 5 DDSL for the Graph Product Line

1. There is a database transaction-like quality to this update. If any refactoring of $\underline{\theta}$ fails, then the all changes are rolled back, and client-programmer intervention is needed to repair the program for subsequent **`U`** application.

The compiler for the GPL DDSL outputs a meta-expression, shown below:

```
Number•StrongC•Cycle•DFS•Weighted(Directed)
```

As users select GPL features, terms are inserted into this expression. Evaluating the expression synthesizes the specified program. My students and I have built many examples of more realistic applications using this technology, ranging from customized or extensible database systems twenty years ago [3], to extensible Java preprocessors ten years ago [4], to web portlets [37] (which we'll consider later). We call this technology *Feature Oriented Programming (FOP)*, where features are either metaprogramming constants or functions [6]. A model of a product-line is an algebra: constants represent base programs (e.g., `Directed`), and functions add features (`Weighted`, `DFS`, etc.) to programs. Each domain has its own algebra, and different meta-expressions synthesize different programs of that domain (product-line). How are features expressed by architectural metaprogramming? This is the topic of the next subsections.

4.1 A Look Inside Features

If we peer inside implementations of FOP functions and constants, we find two ideas that have been popularized by *Aspect Oriented Programming (AOP)* [22]. (I will use the *ideas* of AOP, rather than their AspectJ semantics which has problems [26].) The first is introduction, also known as inter-type declarations. An *introduction* adds a new member to an existing class, or more generally adds a new class or package to a program. Introduction is metaprogramming addition.

The second idea is *advice*, which is the execution of additional code at points called join points. Although it is not obvious, advice is a distributive transformation (see [26] for an explanation, including examples of how complex pointcuts like `cflow` [22] are expressed transformationally). That is, applying advice `A` to a program `P` is the same as applying `A` to each component of `P` and summing the results. Advice or the act of advising is quite different from a refactoring even though both are transformations: refactoring is behavior preserving, whereas advise is behavior-extending. Neither AOP or FOP support subtraction.

Here's how introduction works. Start with a simple program `P` consisting of a single class `r` with field `b` (Figure 6a), and incrementally add or introduce method `foo` (Figure 6b), integer `i` (Figure 6c), and class `t` (Figure 6d). From a metaprogramming viewpoint, the original program in Figure 5a is `P=r.b`. That is, program `P` consists of a single member `b` in class `r`. Introducing method `foo` adds another term to `P`'s meta-expression: (`P=r.b+r.foo`). Introducing field `i` adds yet another term (`P=r.b+r.foo+r.i`). And

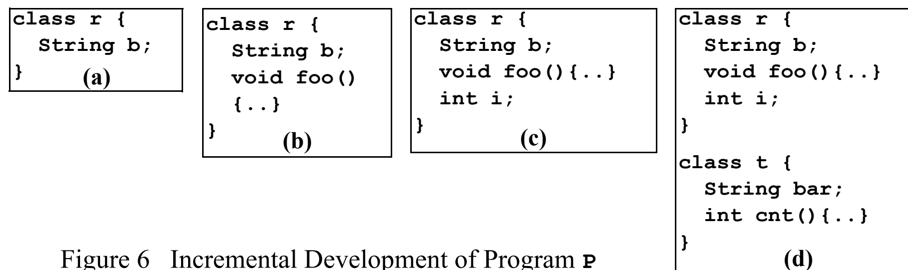

Figure 6 Incremental Development of Program `P`

introducing class **t** adds even more terms (**P=r.b+r.foo+r.i+t.bar+t.cnt**). Evaluating the meta-expression for **P** in each figure synthesizes the listed program.

Now consider advice. *Join points* are events that occur during program execution, such as when a method is called, or when a method is executed, or when a field is updated [21]. Advice is a piece of code that is executed when designated join points occur. Although advice is usually given a dynamic interpretation (i.e., when an event occurs at run-time), it is also possible to give it a static metaprogramming interpretation "at this point in the program, insert this code" [26]. The latter interpretation is common for implementations of many aspect compilers, including the AspectJ compiler **ajc** [21].

Here's how advice works. Consider program **P** of Figure 7a. It consists of a single class **c** and an aspect with a single piece of advice. The advice extends each **set** method of **c** by printing "**hi**". A program that an aspect compiler synthesizes or weaves is shown in Figure 7b. A metaprogramming explanation of weaving is that an aspect compiler inhales the program's source, creates a meta-expression that sums all base code and introductions, and then applies advice. The meta-expression for program **P** in Figure 7a is:

```
P = hi( i + j + setI + setJ ) (8)
```

where values **i**, **j**, **setI**, and **setJ** correspond to the members of class **c**, and function **hi** is the advice. Evaluation of **(8)** proceeds incrementally. First **hi** distributes over each term:

```
P = hi(i) + hi(j) + hi(setI) +
hi(setJ)                      (9)
```

and then each term is evaluated:

```
P =  i  +  j  + setI′ + setJ′(10)
```

Some terms are fixed points (e.g., **hi(i)=i** and **hi(j)=j**, meaning that **hi** does not advise join points in **i** and **j**), while others transform values (**hi(setI)=setI′** and **hi(setJ)=setJ′**). Again, the view from an aspect compiler is to inhale aspect files and base Java files, construct a meta-expression, and evaluate the expression to synthesize the specified program [26].

(a)
```
class c {
  int i,j;
  void setI( int x )
  { i=x; }
  void setJ( int x )
  { j=x; }
}

aspect asp {
  after(): execution(
    void c.set*(..))
    { print("hi"); }
}
```

(b)
```
class c {
  int i,j;
  void setI′( int x )
  { i=x; print("hi"); }
  void setJ′( int x )
  { j=x; print("hi"); }
}
```

Figure 7 Compiling Advice

4.2 Architectural Metaprogramming Implementation of Features

A base program in FOP is a constant, which is a sum of introductions. An FOP function **F(x)** is a feature that advises or modifies (a_f) its input program **x** and introduces new terms (i_f). In other words, **F(x)** adds new members, classes, packages to an input program **x**, and integrates this new functionality by modifying or advising **x**. A general form of all FOP features **F** is:

$$F(x) = i_f + a_f(x)$$

Given a base program **B** and features **F** and **G**, their composition expands into architectural meta-expressions. The FOP expressions **B**, **F(B)**, and **G•F(B)** expand to:

$$\mathtt{B = b}$$
$$\mathtt{F(B) = i_f + a_f(b)}$$
$$\mathtt{G \bullet F(B) = i_g + a_g(i_f + a_f(b)) = i_g + a_g(i_f) + a_g \bullet a_f(b)}$$

A program's code is synthesized by evaluating its meta-expression. This is how GenVoca **[3]** and AHEAD **[6]**, two different implementations of FOP, work.

Perspective. By 2020, many narrow domains will be well-understood, and whose programs are prime candidates for automated construction from declarative specs. The complexity of these programs will be controlled by standardization, where programs will be specified declaratively using "standardized" features, much like personal computers are customized on Dell web pages. Programming languages will have constructs to define features and their compositions (e.g. [31][29]). Compilers will become *program calculators*: they will inhale source code, produce a meta-expression, perhaps even optimize the meta-expression [5], and evaluate the expression to synthesize the target program. Architectural metaprogramming will be at the core of this technology.

5 How are Advice and Refactorings Related?

Program refactorings and advice are transformations. What does it mean to compose them? There is a lot of work on refactoring object-oriented code into aspects (e.g., [10][39]), but less work on refactoring programs that have *both* object-oriented code and aspect code (e.g., [19][15]). Refactorings are not language constructs; they are transformations that are defined and implemented by tools that are "outside" of a target language. Thus, refactorings can modify both object-oriented code and aspect code. In contrast, advice only applies to constructs within a host language, i.e. object-oriented code and other aspect code, but not to refactorings.

To illustrate, let **`P`** be the program of Figure 7a. Applying the refactoring **`rename(P,set*,SET*)`** renames all lowercase **`set`** methods to uppercase **`SET`** methods, we obtain the program **`P'`** of Figure 8. Programs **`P`** and **`P'`** have the same behavior. The **`rename`** refactoring alters the Java source (by renaming **`setI`** to **`SETI`** and **`setJ`** to **`SETJ`**), and alters the advice declaration (by renaming **`set*`** to **`SET*`**).

```
class c {
  int i,j;
  void SETI(int x)
   {i=x;}
  void SETJ(int x)
   {j=x;}
}
aspect adv {
  after():
  execution
   (void c.SET*(..))
   { print("hi"); }
}
```

Figure 8 Program **`P'`**

How can this be explained in terms of architectural metaprogramming? Recall differential operators in calculus: they transform expressions. The differential with respect to **`x`** of a summation is straightforward: every term is transformed:

$$\frac{\partial}{\partial x}(a+b) = \frac{\partial a}{\partial x} + \frac{\partial b}{\partial x}$$

`rename` is similar: it transforms each term of a meta-expression. Let β be a DT of **`rename`**, **`i`** and **`j`** be introductions, and **`a`** and **`b`** be advice. Beyond the distributivity of β over **`+`** and **`-`** in **`(1)`**, the β refactoring also distributes over advice application and composition:

$$\beta(\mathbf{a(i)}) = \beta(\mathbf{a})(\beta(\mathbf{b})) \tag{11}$$

$$\beta(\mathbf{a \bullet b}) = \beta(\mathbf{a}) \bullet \beta(\mathbf{b}) \tag{12}$$

A β transformation of expression **i+b•a(x)** is **β(i)+β(b)•β(a)(β(x))**. Mapping terms of an expression in this manner is called a *catamorphism* [27], a generalization of folds on lists in functional programming. Catamorphisms are grounded in *category theory*, the theory of mathematical structures and their relationships [32]. More later.

Here's how a meta-calculation proceeds. Given program **P** of Figure 7a, a compiler creates its meta-expression. The **rename** refactoring β is then applied to **P**; β distributes over each term of **P**, and then each term is evaluated: **β(hi)=HI**, **β(setI)=SETI**, and **β(setJ)=SETJ**. The terms **β(i)=i** and **β(j)=j** are fixed points. The result is the meta-expression for program **P′**:

```
β(P)
= β( hi( i + j + setI + setJ ) )              // substitution
= β(hi)( β(i) + β(j) + β(setI) + β(setJ) )    // distribution
= HI( i + j + SETI + SETJ )                   // evaluation
= P′                                                    (13)
```

Perspective. Refactorings are operators on meta-expressions that have higher precedence than advice. Interesting research problems are to determine if (a) all common refactorings can be expressed as meta-expressions, and (b) the exact relationship between refactorings and advice, and (c) to show under what circumstances the relationship is (or is not) a catamorphism. Catamorphisms are particularly simple mappings, and knowing when they can (or cannot) be applied may be very useful when building tools.

Note that refactorings, advice, and introductions modify the structure of a program's code, but they could also be used to express and modify the structure of grammars, makefiles, XML documents, and other non-code artifacts. We are now ready to make a conceptual leap to generalize architectural metaprogramming to non-code structures.

6 Recent Advances in Model-Driven Development

Model-Driven Development (MDD) is an emerging paradigm for software creation. It advocates the use of *Domain Specific Languages (DSLs)*, encourages the use of automation, and exploits data exchange standards [13][33]. An MDD model is written in a DSL to capture the details of a slice of a program's design. Several models are typically needed to specify a program completely. Program synthesis is the process of transforming high-level models into executables, which are also considered models [9].

There are many MDD technologies. The most well-known is OMG's *Model-Driven Architecture*, where models are defined in terms of UML and are manipulated by graph transformations [23]. Vanderbilt's *Model Integrated Computing* [35] and Tata's Mastercraft [24] are pioneering examples of MDD. More recently, other groups have offered their own MDD technologies (see [30] for a recent list).

MDD is an architectural metaprogramming paradigm. Models are values and transformations map models to models. To illustrate, consider two models: the Java source of a program and its bytecode. The transformation that maps Java source to Java bytecodes

is **javac**, the Java compiler. If **javac** is a transformation, an interesting question to ask if it is distributive. That is, can each Java file be compiled separately from other files, and the bytecodes added? Does **javac(C+D)=javac(C)+javac(D)**? Unfortunately, the answer is no: **javac** is not distributive. I note that research by Ancona, et al. on separate class compilation may lead to a future version of **javac** that is distributive [1].

A more conventional example of MDD is PinkCreek. It is an MDD case study for synthesizing portlets, which are web components [37]. Transformations map an annotated state chart to a series of different platform-specific models. Figure 9 shows a graph where models are nodes and arrows are transformations; the most abstract model in a PinkCreek specification is a state chart (**sc**), and the most implementation-specific is Java source (**code**) and JSP code (**jsp**). The graph is created by a metaprogram that takes a state chart (**sc**) and applies transformations successively to derive each representation. (That is, a transformation maps an **sc** model to a **ctrl** model, another transformation maps a **ctrl** model to an **act_sk** model, etc.).

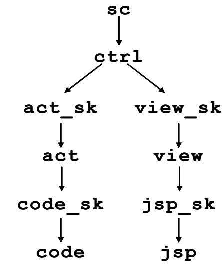

Figure 9 PinkCreek Models

As FOP and MDD are both metaprogramming paradigms, how can they be combined? Recall that features extend the functionality of a program or a model. Let **S0** and **S1** be the source code representations of programs **P0** and **P1**. And let feature **F(x)** relate **S0** and **S1** by **S1=F(S0)**. Let **B0** and **B1** be the bytecode representations of **S0** and **S1**, and let **G(x)** be the bytecode feature that relates **B0** to **B1**, i.e., **B1=G(B0)**. These relationships are captured by the *commuting diagram* of Figure 10. It expresses a fundamental relationship in MDD between features (model-extension transformations) and derivations (model-conversion transformations) [37]. Bytecode **B1** can be synthesized from **S0** in two different ways: either derive **B0** from **S0** using **javac** and then apply feature **G**, or extend **S0** to **S1** by applying feature **F** and then derive **B1** using **javac**. Their equivalence is expressed compositionally as:

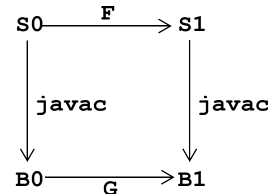

Figure 10 Commuting Diagram

$$\texttt{javac} \bullet \texttt{F} = \texttt{G} \bullet \texttt{javac} \quad (14)$$

Another interesting point is the relationship between functions **F** and **G**. We know that **F** and **G** are features of the form: $F(x)=i_f+a_f(x)$ and $G(x)=i_g+a_g(x)$. In effect, **G** is a compiled version of **F**: both **F** and **G** advise their input programs **x** in equivalent ways (**F** advises source and **G** performs the corresponding advise in bytecodes) and both add equivalent introductions (**F** adds source members and **G** adds the corresponding members in bytecode). We have seen this correspondence before. The relationship between **F** and **G** appears to be a catamorphism: each source term of function **F** is mapped to a corresponding bytecode term of function **G**. Exploring this connection may be an interesting research problem.

Let's now return to commuting diagrams. An important property of commuting diagrams is that they can be pasted together. Given a model in the upper-left corner, we often want to compute the model in the lower right. Any path from the upper-left corner

to the lower right produces the same result [32]. Three different paths are indicated in Figure 11.

To make this idea concrete, consider how features alter state charts in PinkCreek. In general, a feature extends a state chart by (a) adding new states, (b) adding new transitions, and (c) altering existing annotations. Figure 12a depicts a state chart of a base portlet. Figure 12b shows the result of a feature that adds a new state and transitions to Figure 12a.

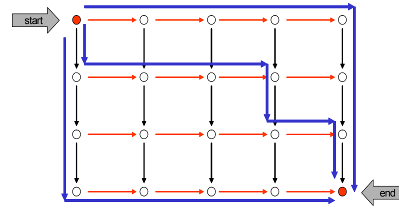

Figure 11 Commuting Paths

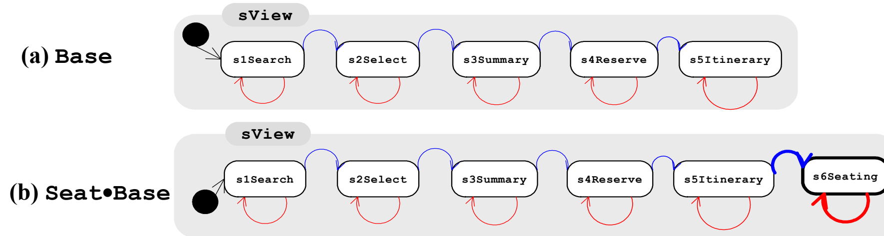

Figure 12 State Chart Extensions

When a feature extends one representation, it may extend derived representations as well. In the case of PinkCreek, all of the models in Figure 9 may be modified when the state chart is extended. That is, if the state chart **sc** is extended, so too must its controller **ctrl**, and its action skeleton (**act_sk**), etc. (Figure 13a). PinkCreek has a metaprogram that translates a state chart feature into a feature of each lower-level representation; as a rule, the ability to translate features of one model to features of another is not always possible or practical. For PinkCreek, it was both possible and practical.

As features are composed, a multi-pleated commuting diagram is swept out (Figure 13b). Traversing this diagram synthesizes the representations of a target portlet. Synthesis begins at the root of the base diagram and ends at the target models which are produced by the last feature. Although all traversals produce the same results, not all traversals are equally efficient. Diagram traversal is an interesting optimization problem. Finding the cheapest traversal is equivalent to finding the most efficient metaprogram that will synthesize the target portlet. This is a form of *multi-stage programming* (i.e., writing programs that write other programs) and *multi-stage* optimization [36].

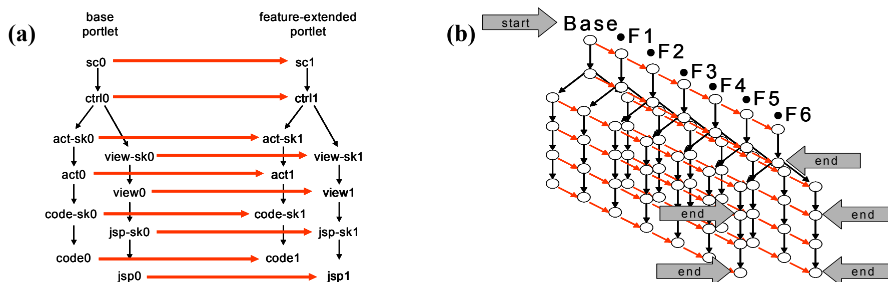

Figure 13 PinkCreek Diagrams

Perspective. Initially PinkCreek tools did not satisfy the properties of commuting diagrams: synthesizing via different paths yielded different results. This exposed previously unrecognized errors in PinkCreek tools and specifications. The significance of commuting diagrams became immediately clear: they provided validity checks on the correctness of model abstractions, portlet specifications, and tools. They offered constraints on both individual transformations and compositions of transformations. In short, commuting diagrams are very useful as they provided a better understanding of the portlet domain and the PinkCreek model.

PinkCreek also revealed a theoretical backbone of architectural metaprogramming: category theory, where catamorphisms and commuting diagrams arise. As mentioned earlier, category theory is a theory of mathematical structures and relationships between these structures. As we are studying the structure of software, and mathematics is the science of structure, architectural metaprogramming may be a direct connection.

Although this connection is preliminary, I have already found that category theory unifies several previously disconnected results in metaprogramming and software design in a surprisingly simple and elegant way [8]. It points to an interesting and very different way of teaching and understanding software design and construction with an emphasis on science, and less on ad hoc techniques. Of course, much more work needs to be done to confirm this conjecture, but so far results are encouraging.

And finally, refactorings are not limited to the restructuring of source code; they apply to models and features as well (e.g. [34][38][40]), where the results of Section 3 and Section 5 should be directly applicable. Demonstrating this unity should be both an interesting and important research topic as it will further underscore the importance of architectural metaprogramming in software design and maintenance.

7 Conclusions

Just as the structure of matter is fundamental to chemistry and physics, so too is the structure of software fundamental to computer science. By structure, I mean what are modules and how do they compose? Today, the structure of software is not well-understood. Software design is an art form. As long as it remains so, our abilities to automate key tasks in program design, synthesis, and maintenance will be limited.

Recent work in program refactoring, program synthesis, and model-driven design are raising the level of abstraction in programming. Their individual successes are not accidental; I contend they focused on the essential complexities of software structure, and not on accidental complexities. Like other results, they are examples of a general programming paradigm that we are only now beginning to recognize. As is evident from the discussions in this paper, many details of architectural metaprogramming are not well understood and it is an open problem to nail them down precisely.

By 2020 the purview of software engineering, as before, will be to manage complexity. Embracing the ideas of architectural metaprogramming offers an appealing future: they will enable us to automate what is well-understood, to customize programs for performance, capability, or both, and to reduce maintenance and development costs, all on a principled basis. It will lead to higher-level programming languages, declarative lan-

guages for specifying programs in narrow domains, IDEs as program evolution calculators, and compilers as program calculators. Our understanding of programs, their representation and manipulation will be greatly expanded beyond code. But again, the grand prize is discovering the material that we will be teaching our future graduates and undergraduates that ties together these areas in an elegant way. An exciting future awaits us.

Acknowledgments. I gratefully acknowledge the helpful comments of S. Apel, O. Diaz, D. Dig, C. Kaestner, V. Kulkarni, C. Lengauer, R. Lopez-Herrejon, and S. Trujillo on earlier drafts of this paper.

8 References

[1] D. Ancona, F. Damiani, and S. Drossopoulou. "Polymorphic Bytecode: Compositional Compilation for Java-like Languages", *POPL 2005*.

[2] M. Antkiewicz and K. Czarnecki. "FeaturePlugin: Feature Modeling Plug-In for Eclipse", *OOPSLA Eclipse Technology eXchange (ETX) Workshop*, 2004.

[3] D. Batory. "Concepts for a Database System Compiler", *ACM PODS 1988*.

[4] D. Batory, B. Lofaso, and Y. Smaragdakis. "JTS: Tools for Implementing Domain-Specific Languages". *International Conference on Software Reuse*, 1998.

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

[7] D. Batory. "Multi-Level Models in Model-Driven Development, Product-Lines, and Metaprogramming", *IBM Systems Journal*, 45#3, 2006.

[8] D. Batory. "From Implementation to Theory in Product Synthesis", *POPL 2007* keynote.

[9] J. Bezivin. "From Object Composition to Model Transformation with the MDA", *TOOLS'USA*, August 2001.

[10] D. Binkley, et al. "Automated Refactoring of Object Oriented Code into Aspects", *ICSM 2005*.

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

[12] F.P. Brookes. "No Silver bullet: Essence and Accidents of Software Engineering", *IEEE Computer*, April 1987.

[13] A. W. Brown, G. Booch, S. Iyengar, J. Rumbaugh, and B. Selic. "An MDA Manifesto", Chapter 11 in *Model-Driven Architecture Straight from the Masters*, D. S. Frankel and J. Parodi, Editors, Meghan-Kiffer Press, Tampa, FL, 2004.

[14] P. Clements and L. Northrup. *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.

[15] L. Cole and P. Borba. "Deriving Refactorings for AspectJ", *AOSD 2005*.

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

[17] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. "Automated Detection of Refactorings in Evolving Components", *ECOOP 2006*.

[18] D. Dig and R. Johnson. "How do APIs Evolve? A Story of Refactoring", *Journal of Software Maintenance and Evolution*, 18#2, 2006.

[19] S. Hanenberg, C. Oberschulte, and R. Unland. “Refactoring of Aspect-Oriented Software”. *Net.ObjectDays 2003*.

[20] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.

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

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kirsten, J. Palm, and W.G. Griswold. “An overview of AspectJ”, *ECOOP 2001*.

[23] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model-Driven Architecture -- Practice and Promise*, Addison-Wesley, 2003.

[24] V. Kulkarni and S. Reddy. “Model-Driven Development of Enterprise Applications”, in *UML Modeling Languages and Applications*, *Springer LNCS 3297, 2005*.

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

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

[27] E. Meijer, M. Fokkinga, and R. Paterson. “Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire”, *FPCA 1991*.

[28] M.P. Monteiro and J.M. Fernandes. “Towards a Catalog of Aspect-Oriented Refactorings”, *AOSD 2005*.

[29] N. Nystrom, X. Qi, A.C. Myers. “J&: Nested Intersection for Scalable Software Composition”, *OOPSLA 2006*.

[30] Object Management Group. **www.omg.org/mda/committed-products.htm**

[31] Odersky, M., et al. “An Overview of the Scala Programming Language”. September 2004, **scala.epfl.ch**

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

[33] D.C. Schmidt. “Model-Driven Engineering”. *IEEE Computer* 39(2), 2006.

[34] G. Sunyé, D. Pollet, Y. Le Traon, J-M. Jézéquel. “Refactoring UML Models”. *Int Conf. UML*, LNCS 2185, Springer-Verlag 2001.

[35] J. Sztipanovits and G. Karsai. “Model Integrated Computing”, *IEEE Computer*, April 1997.

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

[37] S. Trujillo, D. Batory, and O. Diaz. “Feature Oriented Model-Driven Development: A Case Study for Portlets”, *ICSE 2007*.

[38] R. Van Der Straeten, V. Jonckers, and T. Mens. “Supporting Model Refactorings through Behaviour Inheritance Consistencies”, *Int Conf. UML*, LNCS 3273, Springer-Verlag 2004.

[39] C. Zhang and H.-A. Jacobsen. “Resolving Feature Convolution in Middleware Systems”, *OOPSLA 2004*.

[40] J. Zhang, Y. Lin, and J. Gray. “Generic and Domain-Specific Model Refactoring using a Model Transformation Engine”, in *Model-driven Software Development*, (S. Beydeda, M. Book, and V. Gruhn, eds.), Springer 2005.