Finite Map Spaces and Quarks: Algebras of Program Structure

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

Doug Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304
smith@kestrel.edu

We present two algebras that unify the disparate software composition models of Feature-Oriented Programming and Aspect-Oriented Programming. In one algebra, a *finite map space* underlies program synthesis, where adding finite maps and modifying their contents are fundamental operations. A second and more general algebra uses *quarks*, a construct that represents both expressions and their transformations. Special cases of our algebras correspond to existing systems and languages, and thus can serve as a foundation for next-generation tools and languages for feature-based program synthesis.

1 Introduction

Software engineers define structures called *programs* and use tools to manipulate, transform, and analyze them. Object-orientation uses packages, classes, and members to structure a program. Compilers transform source structures to bytecode structures. Refactoring tools transform source structures to refactored source structures. And *model-driven design (MDD)* transforms models of one program representation to models of another. Software engineering is replete with such examples.

Composing structures, transforming structures, and defining relationships between structures is the role of mathematics. Prior work in *software product lines (SPLs)* used features to informally describe program synthesis as vector composition [48]. In this paper, we formalize this result and unify it with other compositional models on *Feature Oriented Programming (FOP)* and *Aspect Oriented Programming (AOP)* [3][5][11][12][19][25][27][28][30][32][33][35][37][38][42][46][49]. We present two algebras that define a blueprint for tools that manipulate different program artifacts (e.g., source code, grammars, makefiles) and realize many different composition models. This is in contrast to the laborious, expensive, and error-prone ways that are now used to build separate tools and languages for each program representation and for each composition model [12][29][46]. Even more important is the long-term potential of connecting results in mathematics to feature-based program design.

Feature-based program synthesis is governed by simple mathematics. A *finite map* is a set of (name,value) pairs. In one algebra, a *finite map space* is shown to underlie program synthesis, where adding finite maps and modifying their contents are fundamental operations. A second and more general algebra represents features as *quarks*, a construct that represents expressions and their transformations.

The concrete justification, evaluation, and inspiration for our work are existing systems and languages that are special cases of our algebras. Our work can help others understand, characterize, and compare software compositional approaches, and see the fundamental mathematical abstractions that underlie this area of research.

2 Feature Oriented Programming

A *feature* is an increment in program functionality. A software product line is a family of programs where no two programs have the same combination of features. Every program in a product line has multiple representations (e.g., source, documentation). When a feature is added to a program, any or all of the program's representations may change. Below we informally sketch the first two generations of FOP, GenVoca and AHEAD [12], which have been used to build product-lines in many applications areas (e.g. [5][10][12]). A third generation based on MDD is discussed in Section 6.

2.1 GenVoca: The First Generation

A GenVoca model of an SPL represents base programs as values (0-ary functions):

```
f       // base program with feature f
h       // base program with feature h
```

Features are unary functions:

```
i•x     // adds feature i to program x
j•x     // adds feature j to program x
```

where the operator • denotes function composition.

The *design* of a program is a named expression:

```
p1 = j•f       // p1 has features j and f
p2 = j•h       // p2 has features j and h
p3 = i•j•f     // p3 has features i,j,f
```

The set of programs that can be defined by a GenVoca model is its product line. Expression optimization is program design optimization, and expression evaluation is program synthesis [10]. The correctness of feature compositions is addressed in [13][47].

2.2 AHEAD: The Second Generation

AHEAD advanced GenVoca by revealing the internal structure of values and unary functions as vectors and modifications to vectors. Every program has multiple representations, such as source, documentation, bytecode, and makefiles. A GenVoca value is a vector of representations of a program. For example, in a product line of parsers, a base parser `f` is defined by its grammar g_f, Java source s_f, and documentation d_f. Program `f`'s vector is $\mathbf{f}=[g_f,s_f,d_f]$. (We denote vectors in **bold** font).

Note: another name for vector is “record”, which is more in line with conventional programming language research [21][23]. However, we want to draw the connection of program synthesis to vector arithmetic in this paper, and hence stress the vector analogy.

A GenVoca unary function maps a vector of program representations to a vector of extended representations. Suppose feature `j` extends a grammar by Δg_j (new rules and tokens are added), extends source code by Δs_j (new classes and members are added and existing methods are modified), and extends documentation by Δd_j. The vector of deltas for feature `j` is $\mathbf{j}=[\Delta g_j,\Delta s_j,\Delta d_j]$, which we call a *delta vector*.

The representations of a program are computed by vector composition [48]. The representations for parser p_1, which is produced by composing features `j` and `f`, are:

```
p₁ = j•f                          ; GenVoca expression
   = [Δgⱼ,Δsⱼ,Δdⱼ]•[gf,sf,df]    ; substitution
   = [Δgⱼ•gf,Δsⱼ•sf,Δdⱼ•df]      ; composition
```

$$\begin{aligned}\mathbf{p}_1 &= \mathbf{j}\bullet\mathbf{f} && \text{; GenVoca expression}\\ &= [\Delta g_j,\Delta s_j,\Delta d_j]\bullet[g_f,s_f,d_f] && \text{; substitution}\\ &= [\Delta g_j\bullet g_f,\Delta s_j\bullet s_f,\Delta d_j\bullet d_f] && \text{; composition}\end{aligned}$$

That is, the grammar of $\mathbf{p}_1$ is the base grammar composed with its extension ($\Delta g_j\bullet g_f$), the source of $\mathbf{p}_1$ is the base source composed with its extension ($\Delta s_j\bullet s_f$), and so on.

Representations can have sub-representations, recursively. Every sub-representation can be modeled as a vector, and can be transformed by delta vectors. In general, GenVoca values are nested vectors and functions are nested delta vectors, where the • operator recursively composes nested vectors. This is the essence of AHEAD [7].

3 A Finite Map Space Algebra

A vector space is a fundamental mathematical structure; it is a collection of elements called *vectors* that can be added and scaled [44]. Feature-based program synthesis satisfies many properties of a vector space. In the following sections, we show the informal delta vectors of AHEAD are functions in a *Finite Map Space (FMS)*, where finite maps are added and their contents are extended or “scaled” by modifiers. Although we use Java examples, our algebra applies to non-Java artifacts as well — see [12] for grammars and makefiles and [2] for XML documents.

3.1 An Informal Overview of Key Concepts

Feature-based synthesis is an incremental process that adds new members and classes to a program and extends existing members. These actions are expressed by two operations: addition and modification.

3.1.1 Addition

Programs are values. Consider two values `C` and `D` that represent two differently named Java classes `C` and `D`. The sum `C+D` produces a program (a value) that has both classes. Conceptually, addition (+) is the union of these classes.

Now consider values `C1` and `C2` of Figure 1a-b, both representing different declarations of Java class `C`. The sum `C1+C2` yields a single class that is the union of the members of `C1` and `C2` (Figure 1c). That is, `+` produces the union of the contents of classes with the same name.

```
(a) C1:      class C { void foo() {...} }
(b) C2:      class C { int bar; }
(c) C1+C2:   class C { void foo() {...}; int bar; }
```

Figure 1. Class Declarations and Their Sum

Problems arise when adding classes that share a member. Consider the values `D1` and `D2` of Figure 2a-b. Both define a class `D` with conflicting `bar` declarations. We don't want `D1+D2` to yield a single class with both `bar` declarations (Figure 2c). Instead, we flag this as an *addition error* `A`. Both declarations are replaced with a single declaration with `A` indicating the error term in `D1+D2` (Figure 2d).

```
(a) D1:      class D { int bar; }
(b) D2:      class D { String bar="4"; }
(c)          class D { int bar; String bar="4"; }
(d) D1+D2:   class D { A bar; }
```

Figure 2. Inconsistent Term Error

Our choice of raising error `A`, as opposed to member replacement, stems from experience in feature-based development. Replacement occurs when a newer declaration replaces an existing declaration. While member replacement is useful in class inheritance, it is problematic in program synthesis. Replacement erases the semantics of a member on which other features depend. Consequently, these features may no longer work correctly [47]. Designs that avoid replacement avoid these problems.

Addition scales [12]. Packages can be added, which produces a package that is the sum of the classes in each input package; packages of packages can be added, and so on. [12] shows non-code artifacts also have hierarchical structures which can be added.

3.1.2 Modification

A member of a class (including initializers and constructors) is a term. Features modify terms by specializing or extending their definitions. An important class of modifications are method extensions (as in mixins) and AOP advice application. Although our model of modification is couched in terms of arbitrary modifications (and works with them), it also provides a disciplined way of working with these special cases.

Each term has a name and a value. The name of a term uniquely distinguishes it from other terms, and the value is the term's program source. Term values are changed by a *modifier,* which is a (`selector`, `rewrite`) pair. The `selector` identifies terms, and the `rewrite` updates the value of a selected term. Modifiers come in two flavors: universal and unique existential.

A *universal modifier (UM)* can change any or all terms of a program; it expresses a structural property of a program that universally holds after the modifier is applied. Suppose a `print` statement is to be appended to every `get` method in class `C`. To express this as a UM, a `selector` qualifies the `get` method terms of `C`. The `rewrite` appends a `print` statement to the body of each selected method. After this modifier is applied, we know that all `get` methods of class `C` end with a `print` statement.

In general, a UM is a classical pattern rewrite of a program transformation system [18]. Another way to write a UM is as advice in AOP [32]: an advice pointcut is a `selector` and an advice body is a `rewrite`. (Modifiers are a form of quantification [26]). Although the effects of advice are traditionally understood as alterations in execution flow, all advice, including `cflow`, can be implemented by static transformations. See [29][35] for explanations of how aspect compilers realize `cflow` transformationally.

In contrast, a *unique existential modifier (EM)*, expresses an existential property of a design — that a particular term in a program is defined. The selector of an EM identifies a term *on the basis of the term's name, not its value*. If the term is defined, the rewrite does nothing. If undefined, an *existence error* `E` is raised (i.e., `E` becomes the new value of the term).

EMs are common in feature development. The functionality of a feature is often invoked by adding method calls to existing hook methods. If these hook methods are not present, the feature will not work correctly. Feature designs demand that hook methods exist. Suppose a `print` statement is to be appended to the hook method `m()` of class `C`. To express this design, we use a UM whose `selector` identifies the `m()` method, and the `rewrite` appends the `print`. We also use an EM to ensure that `m()` is defined.

In mixin-supported languages, an EM never exists separately and is always paired with a UM that modifies the EM identified term. Such pairings are expressed by a variety of OO techniques, including mixins, traits, virtual classes, and nested inheritance [27]. The essential idea is this: mixin for a method is a UM that uses the signature of the method as its `selector`, and the body of the method as its `rewrite`. In AHEAD, a mixin for method `m()` in the above example is:

```
   void m() { Super.m(); print(“here”); }
```

where `Super.m()` is replaced with the original body of `m()`. An EM for this declaration is implicit, because if `m()` has no prior definition, an existence error is raised [12].

In the next sections, we formalize the above ideas. We define primitive terms, their addition and modification, and show how sets of terms, represented by finite maps, are added and modified. *The properties that we label in roman-numerals are those required by a vector space.* Later we extend our algebra to include EMs. We do so as algebras without EMs are easy to manipulate manually; EMs can be added later to express more precise feature designs.

3.2 Primitive Terms

Consider any primitive term (a method, initializer, constructor, etc.) of a class. Each term `t` has an identifying *name*, denoted `t.name`, and a *value*, denoted `t.value`. The value of a term is its program source. *Zero* (`0`) denotes the null or undefined value.

Figure 3 is the lattice `L` of all values that can be assigned to a term. Value `0` is the bottom (denoting null or undefined), `T` is top (the `E` or `A` error), and the x_i are non-zero elements between `0` and `T` ($0 < x_i < T$) called *normal values*.

> We simplify errors by unifying the `E` and `A` errors as `T`. In an implementation, `T` has secondary information indicating the source of the error (i.e., `E` or `A`). In a more elaborate version of our theory, `T` is replaced by `E` and `A`, where `A<E`.

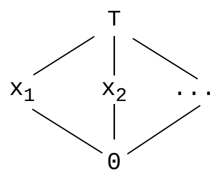


Figure 3. Value Lattice

For example, consider the term that represents method `m()` in the last section. The lattice for `m()` has value `0` (meaning `m()` is undefined), `T` (meaning `m()` has `E` or `A` errors), and a normal value for every source code specification that could define `m()`.

3.3 Term Addition

Term addition (`+`) is the join (least-upper-bound) operation of lattice `L` and has the following identities [24]:[1]

`(i)` *associative*: For all $x,y,z \in L$: $x+(y+z)=(x+y)+z$

`(ii)` *commutative*: For all $x,y \in L$: $x+y = y+x$

`(iii)` *additive identity*: For all $x \in L$: $x+0=x$

For example, adding two different normal values of `m()` yields `T`. Adding `0` to a normal value of `m()` yields that value. And the order in which additions occur does not matter.

A useful identity not required by a vector space is [24]:

$$x + x = x \quad ; \text{ for all } x \in L \tag{1}$$

3.4 Universal Modifiers

Let `D` denote the set of all terms, and `L` denote the value lattice. A universal modifier μ is an ordered pair (μ^s, μ^r), where $\mu^s: D \rightarrow \texttt{boolean}$ is a term selector and $\mu^r: L \rightarrow L$ is a function that changes the value of a selected term. (Again, informally, think of μ^s as a pointcut and μ^r as advice, but realize that the ideas of UMs are more general). Selectors qualify terms on their name, value, or both. The application of μ to term `t` is denoted $\mu \cdot t$. If the selector of μ does not qualify `t` (that is, $\mu^s(t)$ is false), `t` remains unchanged. Otherwise, the value of `t` is updated by μ^r.

1. `L` is a join semi-lattice with universal bounds `0` and `T` [24].

The updating rules of μ^r are simple: modification is *strict* — modifying an error remains an error:

$$\mu^r \cdot T = T \tag{2}$$

Modifying a normal value `x` yields another (usually different) normal value `y`:

$$\mu^r \cdot x=y \quad ; \; 0<x<T \text{ and } 0<y<T \tag{3}$$

And zero is unchanged — advising nothing yields nothing:

$$\mu^r \cdot 0 = 0 \tag{4}$$

Note the generality of μ^r permits virtually arbitrary rewrites of normal values, such as a rewrite that simplifies arithmetic expressions. *Specific rewrites of μ^r model mixins and AOP advice, but the generality of μ^r allows us to model almost any (future) value rewriting technology.*

Let `M` denote the set of all universal modifiers. Using `(2)-(4)` it is easy to prove for all modifier rewrites:

`(iv)` *distributivity over addition*: For all $\mu \in M$ and $v,w \in L$: $\mu \cdot (v+w) = \mu \cdot v + \mu \cdot w$

Modifiers are composed by function composition, which we denote by juxtaposition. The *composite modifier* $\mu_1\mu_2$ means apply modifier μ_2 then apply μ_1.[2]

3.5 Finite Maps

Let us now scale addition and modification of individual terms to a container of terms (i.e., a module). In general, a *module* is a hierarchy of nested containers [12]. The essential concept is a *container* — a *finite map* that is a set of terms that are `(name, value)` pairs [23]. The name of a term uniquely identifies the term in its container. In the following sections, we define the operations of adding and modifying finite maps. By doing so, we define a *finite map space (FMS)*.

Finite maps have many notations. AHEAD used sets; Apel et al. use trees (where a node is a container and its children are its terms) [6]. A *vector* is another, where the index of a component is a term's name, and the component value is the term's value. We list terms in alphabetical order of their name.[3] Vector `[a b c]` denotes a container with terms `a`, `b`, and `c`. When a term has an undefined value, we use `0` to indicate this. Thus vector `[a b 0]` denotes a container where term `c` has an undefined value.

2. There is also an *identity modifier* (`1`), which is the modifier that selects no terms. It leaves terms intact when applied, i.e., for all $v \in L$: $1 \cdot v = v$.

3. The order of components in a vector is the order in which its terms must appear in source. For example, the children of an XML node can be ordered. The XML node would be a vector and its components would be the children in order of their listing. As Java imposes no ordering of classes within a package or members within a class, we use an alphabetical ordering.

Note that a vector notation of finite maps is approximate: the number of terms in a classical vector is fixed and does not vary over time. However, the number of terms in a finite map can vary (as we will illustrate shortly). Again, we emphasize to readers that when we use the term "vector", we mean a notation for a finite map.

We use a compiler name-mangling technique to uniquely name Java class members [50]. The name of a Java member is its identifier followed by the signature of its arguments [50]. So field `"int a;"` has name `"a"`; and method `"bool b(int y){...}"` has name `"b(int)"`. Different naming schemes may be needed for non-Java artifacts.

A class is a container whose members are terms. Consider the following definition of Java class `X`:

```
class X { int a=5; bool b(int y){…}; }                          (5)
```

A vector for this class is $\mathbf{X}_1$=[a_1 b_1] with terms a_1 and b_1:

```
a₁.name = "a"
a₁.value = "int a=5;"
b₁.name = "b(int)"
b₁.value = "bool b(int y){…};"
```

Vector subscripts denote different definitions of a class. Our first definition of class `X` is $\mathbf{X}_1$.

A word on notation: if member `c` exists in class `X` in some program of a product line, we write vector $\mathbf{X}_1$ as [a_1 b_1 0], leaving room for term `c` to be added later. As we are using vectors to convey ideas, padding vectors with zeros is a notational issue, and not a limitation of finite maps.

Now consider a second definition of `X`:

```
class X { int c; }                                              (6)
```

Its vector is $\mathbf{X}_2$=[0 0 c_2] where:

```
c₂.name = "c"
c₂.value = "int c;"
```

The source code of a vector **v** is its *image*, denoted by `image(`**v**`)`. The image of $\mathbf{X}_1$ is `(5)` and the image of $\mathbf{X}_2$ is `(6)`.

Finally, the *zero vector (**0**)* has all terms with value `0`. The image of $\mathbf{X}_3$=**0** for class `X` is:

```
class X { }
```

3.6 Addition of Finite Maps

Vector addition sums corresponding components:

$$[x_1\ x_2\ \ldots\ x_m] + [y_1\ y_2\ \ldots\ y_m] = [x_1+y_1\ x_2+y_2\ \ldots\ x_m+y_m] \qquad (7)$$

Using the rules for adding primitive terms, the sum $\mathbf{X}_1+\mathbf{X}_2=\mathbf{X}_{12}=[a_1\ b_1\ c_2]$ has the image:

```
class X { int a=5; bool b(int y){…}; int c; }
```

A fourth definition of class `X` has vector $\mathbf{X}_4 = [a_4\ 0\ 0]$:

```
class X { int a=8; }
```

where:

```
a₄.name = "a"
a₄.value = "int a=8;"
```

The image of $\mathbf{X}_1+\mathbf{X}_4=\mathbf{X}_{14}=[a_1+a_4\ b_1\ 0]=[\top\ b_1\ 0]$ is:

```
class X { T a; float b(){…}; }
```

The error term (an `A` error, actually) arises because there are conflicting declarations of member `"a"` in both $\mathbf{X}_1$ and $\mathbf{X}_4$.[4]

Vector addition satisfies the theorems below, which follow directly from lifting primitive terms to vectors `(i)`-`(iii)`:

`(v)` *associative*: For all $\mathbf{u},\mathbf{v},\mathbf{w}\in V$: $\mathbf{u}+(\mathbf{v}+\mathbf{w})=(\mathbf{u}+\mathbf{v})+\mathbf{w}$

`(vi)` *commutative*: For all $\mathbf{v},\mathbf{w}\in V$: $\mathbf{v}+\mathbf{w} = \mathbf{w}+\mathbf{v}$

`(vii)` *identity element*: For all $\mathbf{v}\in V$ and $0\in V$: $\mathbf{v}+0=\mathbf{v}$

Any two vectors $\mathbf{u},\mathbf{w}\in V$ can be added. Some additions will produce vectors with errors. Just as programs can be created with errors, the same holds for features. Feature models are used to limit feature compositions to those that do not produce errors [13][47].

3.7 Universal Modification of Finite Maps

Applying a universal modifier μ to a vector $\mathbf{v}$, denoted $\mu\cdot\mathbf{v}$, is equivalent to applying μ to each term of $\mathbf{v}$. That is, μ distributes over the terms of $\mathbf{v}$:

$$\mu\cdot\mathbf{v} = \mu\cdot[t_1 \ldots t_n] = [\mu\cdot t_1 \ldots \mu\cdot t_n] \qquad (8)$$

Intuitively, the justification for `(8)` comes from pattern rewrites in program transformation systems. That is, given an input pattern, replace it with an output pattern. Such a pattern would be applied to all components of a program. (This is property `(8)`). The same idea arises as quantification in AOP where advising a program is the same as

4. Terms can have non-primitive values, which are vectors, and vectors *can* be added. Nested vectors model module hierarchies [12]. Class and member modifiers (`public`, `private`), class initializers, and `implements` clauses can also be modeled by specially-named terms of a class. These terms may have vectors as values.

advising each of its parts [8][35]. Our formalization is that a modifier can alter every term of a container.

The universal modifier rewrite of a vector satisfies the following theorem, whose proof follows from (8) and (iv).

(viii) *distributivity over vector addition*: For all $\mu \in M$ and $\mathbf{v},\mathbf{w} \in V$: $\mu \cdot (\mathbf{v}+\mathbf{w}) = \mu \cdot \mathbf{v} + \mu \cdot \mathbf{w}$

We will consider EMs shortly.

3.8 Finite Map Spaces

A *Finite Map Space (FMS)* is a collection of finite maps that can be added and modified, as defined in the previous sections. We identified key properties (in roman numerals) that an FMS has in common with a vector space. However, a vector space has additional requirements [44]. First, scalars can be both added and multiplied. The modifiers (scalars) of FMSs are composed only via one operation (function composition); a second operation is lacking. Second, vector addition requires additive inverses (i.e., negative values). Term values are program fragments — parse trees or strings: there are no “negative” parse trees or strings. Terms can be removed via subtraction, but this is not the same as an additive inverse. Third, multipliers and terms have the same data types in vector spaces; in FMSs, modifiers are different from terms.

Although an FMS is not a vector space, drawing an analogy is useful, as the following section demonstrates.

3.9 Perspective

3.9.1 AHEAD Features

One of the goals of this paper is to provide a precise definition of AHEAD features and their compositions. We can now do so. In Section 2.2 we saw that every AHEAD feature was either a vector of values or a delta vector (i.e., a vector of deltas). A vector of values is simply a vector in an FMS. A delta vector is a unary function $f: V \to V$ that transforms vectors. Its general form is:

$$f(\mathbf{x}) = \mathbf{I} + \mu \cdot \mathbf{x} \qquad (9)$$

where μ are the changes f makes to vector $\mathbf{x}$ and $\mathbf{I}$ is a vector of terms that f adds (introduces) to $\mathbf{x}$. FMSs allow us to “peer” inside features to reveal finer-grained structures of their organization and how these structures compose. For example, the AHEAD expression of a program $p = k \bullet j \bullet i$ expands into a sum of modified vectors:

$$\begin{aligned} \mathbf{p} &= k(j(i)) && \text{; same as } k \bullet j \bullet i \\ &= \mathbf{I}_k + \mu_k \cdot (\mathbf{I}_j + \mu_j \cdot (\mathbf{I}_i)) \\ &= \mathbf{I}_k + \mu_k \cdot \mathbf{I}_j + \mu_k \mu_j \cdot \mathbf{I}_i && (10) \end{aligned}$$

where vector $\mathbf{I}_j$ denotes the collection of terms that are introduced by feature j and μ_j are the modifications of j. Exposing additional internal structure makes it easier to explain complex tasks, such as in [15], and the topic of the next section.

3.9.2 Feature Oriented Refactoring

Feature-oriented refactoring (FOR) is the inverse of feature composition: the goal is to refactor a legacy application p into a composition of features, say p=k•j•i [33]. FOR is done in three steps. First, the terms of p are partitioned, assigning each term to a feature. This is modeled by the addition of vectors, one vector per feature:

$$\mathbf{p} = \mathbf{I}_k + \mathbf{I}_j' + \mathbf{I}_i'$$

Next feature-specific modifiers are factored out:

$$\mathbf{p} = \mathbf{I}_k + \mu_k \cdot \mathbf{I}_j + \mu_k\mu_j \cdot \mathbf{I}_i \quad ;\ \text{where } \mathbf{I}_j' = \mu_k \cdot \mathbf{I}_j \text{ and } \mathbf{I}_i' = \mu_k\mu_j \cdot \mathbf{I}_i$$

Finally, the expression is rewritten to define an FMS function for each feature:

$$\begin{aligned}\mathbf{p} &= \mathbf{I}_k + \mu_k \cdot (\mathbf{I}_j + \mu_j \cdot (\mathbf{I}_i)) \\ &= k(j(i)) \qquad ;\ \text{same as } k \bullet j \bullet i\end{aligned}$$

Note the above expression is identical to (10). In general, feature-refactoring code is conceptually equivalent to factoring expressions, thus providing a simple way to understand what is otherwise a complex and partially-automatable task. Interestingly, FMSs can be extended with an annihilating modifier 0, where for all $\mathbf{v} \in V$: $0 \cdot \mathbf{v} = \mathbf{0}$. Then the $\mathbf{I}$ vectors are analogs to basis vectors of a vector space, and the modifiers are vector coefficients.

3.9.3 Canonical Forms and Compositionality

Mathematics imposes constraints on the definition of features that are not immediately obvious from a software engineering viewpoint. For example, (9) is a canonical form of features, rather than:

$$F(\mathbf{x}) = \mu \cdot (\mathbf{I} + \mathbf{x}) \tag{11}$$

A software engineering argument for (11) is the following: if μ represents a program invariant, then μ must be applied to *all* introductions of a program, including the introductions $\mathbf{I}$ of F. However, the main reason to prefer (9) is the property of *compositionality*: a composition of features is another feature. Consider features H and G:

$$\begin{aligned}H(\mathbf{x}) &= \mathbf{I}_h + \mu_h \cdot \mathbf{x} \\ G(\mathbf{x}) &= \mathbf{I}_g + \mu_g \cdot \mathbf{x}\end{aligned}$$

Their composition can be factored into form (9):

$$\begin{aligned}H \bullet G(\mathbf{x}) &= \mathbf{I}_h + \mu_h \cdot (\mathbf{I}_g + \mu_g \cdot \mathbf{x}) \\ &= \mathbf{I}_h + \mu_h \cdot \mathbf{I}_g + \mu_h\mu_g \cdot \mathbf{x}\end{aligned}$$

$$= (\mathbf{I}_h + \mu_h \cdot \mathbf{I}_g) + \mu_h\mu_g \cdot \mathbf{x}$$
$$= \mathbf{I}_{hg} + \mu_{hg} \cdot \mathbf{x} \qquad (12)$$

In contrast, we have been unable to do the same for (11):

$$\begin{aligned} H(\mathbf{x}) &= \mu_h \cdot (\mathbf{I}_h + \mathbf{x}) \\ G(\mathbf{x}) &= \mu_g \cdot (\mathbf{I}_g + \mathbf{x}) \end{aligned}$$

$$\begin{aligned} H \bullet G(x) &= \mu_h \cdot (\mathbf{I}_h + \mu_g \cdot (\mathbf{I}_g + \mathbf{x})) \\ &= \mu_h \cdot \mathbf{I}_h + \mu_h\mu_g \cdot \mathbf{I}_g + \mu_h\mu_g \cdot \mathbf{x} \end{aligned}$$

(11) requires the *same* modifier to be applied to all summands, which is not the case above. (9) satisfies the property of compositionality, but (11) does not. We explore how advice can be applied program-wide in Section 4.

3.9.4 Hierarchical Scalability

A key strength of AHEAD is that containers (modules) at every level of abstraction are governed by the same laws. This is useful in creating product lines of product lines. Given a base product line M, we can define a feature G that extends M to yield an enhanced product line M1=G•M. In effect, G modifies existing features of M and adds new features. Given more features like G, a product line of product lines (i.e., a product line of M variations) is formed. FMSs have this powerful property. A GenVoca model can be expressed as a vector of values (V_1 … V_m) and unary functions (F_1 … F_p) with zero padding for new features that might be added subsequently:

$$[V_1 \dots V_m \; F_1 \dots F_p \; 0 \dots 0]$$

Let $\mathbf{x}=[x_1 \dots x_n]$ be such a vector. The general form of a feature G for product line $\mathbf{x}$ is:

$$\begin{aligned} G(\mathbf{x}) &= [\; g_1(\mathbf{x}_1) \;\dots\; g_n(\mathbf{x}_n) \;] \\ &= [\; i_1+\mu_1 \cdot \mathbf{x}_1 \;\dots\; i_n+\mu_n \cdot \mathbf{x}_n \;] \end{aligned} \qquad (13)$$

G has the canonical form (9) as we show below. Let:

$$\begin{aligned} \mathbf{i} &= [i_1 \dots i_n] \\ \underline{\mu} &= \mu_1 \dots \mu_n && \text{; any permutation as } \mu_i,\mu_j \\ & && \text{; are commutative for } i \neq j \end{aligned}$$

Form (9) follows:

$$\begin{aligned} G(\mathbf{x}) &= [i_1+\mu_1 \cdot x_1 \;\dots\; i_n+\mu_n \cdot x_n] && \text{; (13)} \\ &= [i_1 \dots i_n] + [\mu_1 \cdot x_1 \dots \mu_n \cdot x_n] && \text{; (7)} \\ &= \mathbf{i} + [\mu_1 \cdot x_1 \dots \mu_n \cdot x_n] && \text{; defn. } \mathbf{i} \\ &= \mathbf{i} + \mu_1 \dots \mu_n \cdot [x_1 \dots x_n] && \text{; (8)}^5 \end{aligned}$$

5. Each μ_i has selector that identifies a unique term of $\mathbf{x}$. Repeated applications of (8) removes each μ_i to apply to the entire vector.

$= \mathbf{i} + \mu_1 \ldots \mu_n \cdot \mathbf{x}$; defn. $\mathbf{x}$
$= \mathbf{i} + \underline{\mu} \cdot \mathbf{x}$; defn. $\underline{\mu}$

Applying the same laws at all levels of abstraction has practical value: it was fundamental in creating the AHEAD Tool Suite [12], implementations of multi-dimensional separation of concerns [11], meta-models and software service packs [12], and of course product lines of product lines [11].

3.9.5 Unique Existential Modifiers

Recall that an EM is a test for the existence of a particular term; the term's value is unchanged if it is not `0` (undefined), otherwise an error is produced. An EM is a special modifier, denoted `!`, that is an ordered pair of functions ($!^s$, $!^r$). Again let `D` denote the set of all terms and `L` denote the value lattice. Function $!^s$:D→boolean is a selector that identifies a single term of a vector and $!^r$:L→L is the term value rewriting function. (Informally, think of $!^s$ as a pointcut that identifies a single term by name and $!^r$ as an AspectJ `declare error` action, but again realize that the idea of EMs is more general). The application of `!` to term `x` is denoted `!·x`. If selector $!^s$ does not qualify `x` (that is, $!^s(x)$ is false), the value of `x` is unchanged. Otherwise, the value of `x` is updated by $!^r$. Like UMs, `!` applied to a vector equals the vector where `!` is applied to each term. Of course, only one term of the vector is actually selected:

$$!\cdot\mathbf{v} = !\cdot[t_1 \ldots t_n] = [!\cdot t_1 \ldots !\cdot t_n] \qquad (14)$$

The updating rules of $!^r$ are *strict* — modifying an error remains an error, and normal values are unchanged:

$$!^r\cdot T = T \qquad (15)$$

$$!^r\cdot v = v \quad ; \text{ for all } v \in L \text{ and } 0<v<T \qquad (16)$$

The key mapping is:

$$!^r\cdot 0 = T \qquad (17)$$

That is, if the term singled out by `!` has an undefined value, then an (existential) error is raised. Due to (17), ! *does not distribute* over addition (+). Basic identities involving $!^r$ are easy to prove, such as *idempotence*:

$$!_1^r !_1^r v = !_1^r v \quad ; \text{ for all } v \in L \qquad (18)$$

That is, multiple applications of the *same* $!^r$ function are equivalent to a single application, and EM and UM rewrites are commutative:

$$\mu^r !^r\cdot v = !^r \mu^r\cdot v \quad ; \text{ for all } v \in L \text{ and } \mu \in M \qquad (19)$$

The canonical form of feature functions generalizes to:

$$F(\mathbf{x}) = \mathbf{I} + [!]\mu\cdot\mathbf{x} \qquad (20)$$

where [!] means the composition of one or more EMs is optional and μ is a single or composite UM. It can be shown functions of the form (20) satisfy the property of compositionality [16] (see Appendix II).

In general, expressions involving ! are more complicated to manipulate *manually* although there is no problem for tools. FMS expressions can be developed without ! functions to explore a basic design concept (as we did with FOR in Section 3.9.2), and add ! afterwards to impose EM constraints. As an example, the UM below expresses a method extension in AOP:

```
void around():execution( void m()){ ..; proceed(); ..;}
```

and later it can be replaced by the method mixin:

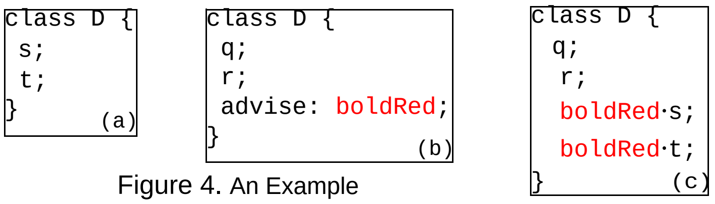
```
void m(){ ..; Super.m(); ..; }
```

to additionally specify an EM to test that method m() exists.

3.9.6 Implementation

We show in Section 5 that existing systems implement special cases of FMSs. We present an elementary example here to illustrate one possible concrete mapping of an FMS to an implementation. There are many possibilities, and choosing a “good” one is the subject of on-going work.

Figure 4a shows a class D with members s and t. D’s vector is **D**=[0 0 s t]. (Only stubs for s and t are present; their details are not significant). Figure 4b shows a refinement of class D that adds members q and r, and has a piece of advice that colors all existing members boldRed. This refinement is the function F(**x**)=[q r 0 0] + boldRed·**x**. That is, F boldRed-modifies its input program **x** and adds terms q and r. Figure 4c is the image of F(**D**): all members in **D** are advised in boldRed and the resulting vector is [q r boldRed·s boldRed·t].



```
class D {
  s;
  t;
}           (a)
```

```
class D {
  q;
  r;
  advise: boldRed;
}                (b)
```

```
class D {
  q;
  r;
  boldRed·s;
  boldRed·t;
}           (c)
```

Figure 4. An Example

This example scales to packages by replacing “class” with “package”, interpreting q-t as individual classes, and boldRed as an aspect file with advice. Although specific details are elided, the essential idea of composition is the same.

When AHEAD was implemented, virtually the same algorithm was used for composing (i.e., summing, modifying) different program representations. Because separate composition tools were written (sometimes in different languages) for different program representations, we could not reuse common code nor enforce a consistent definition of composition. Similar problems have occurred in other projects, e.g., Hyper/J [46] and aspect weavers for different languages [28]. FMS provides a blueprint for building a single tool to implement these operations once, thereby achieving code reuse and definition consistency.

Here is how we envision a tool that implements FMSs: there will be a class `G` whose objects are parse trees or XML trees, thus providing a standard way to represent all kinds of program representations. `G` methods are operations on `G` objects, such as sum and modify. Modifications are (selector, rewrite) pairs which would use standard transformation system techniques for searching trees and performing tree replacements [29]. Representation-specific twists on `G` methods will inevitably arise, and this would be handled by subclassing `G`. Parsers would map different representations (makefiles, Java programs, grammars, etc.) to `G` objects and then `G` methods are invoked. Unparsers would map transformed `G` objects back to their original representations. In effect, our tool would be an object-oriented framework for manipulating different program representations. Simonyi's Intentional Programming is a prototype of these ideas [42]. A possible type system for FMSs is gDeep [7].

4 Quarks

A characteristic of AspectJ is that advice is *always* applied to an entire program, which is the base program and every aspect [15]. To illustrate, let `B` denote the base program of Figure 5a and `A1` and `A2` be the aspects of Figure 5b-c. Both aspects introduce an increment method and advise the execution of increment methods.

```
class C {
  int a = 0;
  void inc() {a++;}
}
```
(a)

```
aspect A1 {
  void C.inc2() {a=a+2;}      // i1
  after(): execution(void C.inc*())
  { outA1(); }                // μ1
```
(b)

```
aspect A2 {
  void C.inc3() {a=a+3;}      // i2
  after(): execution(void C.inc*())
  { outA2(); }                // μ2
```
(c)

```
class C { // program P1
  int a = 0;
  void inc() {a++;   outA1();}
  void inc2(){a=a+2; outA1();}
}
```
(d)

```
class C { // program P2
  int a = 0;
  void inc() {a++; outA1();
              outA2();}
  void inc2(){a=a+2; outA1();
              outA2();}
  void inc3(){a=a+3; outA1();
              outA2();}
}
```
(e)

Figure 5. Aspects and Woven Aspects

Figure 5d shows the program `P1` that AspectJ produces when `A1` is applied to `B`; there are two increment methods and both are advised. Figure 5e shows the program `P2` when `A2` and `A1` are applied to `B`, where `A2` is applied after `A1`.[6] The resulting program has three increment methods, and all are advised by `A1` and `A2`. Again the idea is that advice is always applied to all introductions of the base program and its aspects.[7]

Applying advice after all introductions have been made is useful in implementing certain kinds of invariants [43], e.g. "every call to method `m()` must be followed by a call to method `q()`". Of course, the invariant holds immediately after the modifier is applied, but not necessarily after subsequent modifiers are applied. (This is the problem of *aspect interaction* [22]).

Here is a general mathematical description of what is happening. When aspect A_1, which has advice μ_1 and introduction i_1, is applied to base program `B` whose FMS expression is `b`, the resulting program `P1` has the expression $\mu_1 \cdot (i_1 + b)$. When aspect A_2, which has advice μ_2 and introduction i_2, is applied to `P1`, the resulting program `P2` has the expression $\mu_2 \mu_1 \cdot (i_2 + i_1 + b)$. That is, `A2` transforms expression $\mu_1 \cdot (i_1 + b)$ into $\mu_2 \mu_1 \cdot (i_2 + i_1 + b)$. In effect, aspects are an *expression rewrite*: introductions of the base program and all aspects are first added together and only then advice is applied. The shape of a program expression produced by AspectJ is distinctive [35]:

$$\mu_n \ldots \mu_1 \cdot (i_n + \ldots + i_1 + b) \tag{21}$$

In contrast, the feature functions of FMSs yield program expressions with a different shape (where feature F_j has modifier μ_j and introduction i_j):

$$i_n + \mu_n \cdot i_{n-1} + \mu_n \mu_{n-1} \cdot i_{n-2} + \ldots + \mu_n \mu_{n-1} \ldots \mu_1 \cdot b \tag{22}$$

The images (programs) of (21) and (22) can be quite different. Figure 6a shows the composition `A1(B)` and Figure 6b the composition `A2(A1(B))`, where $A1(\mathbf{x}) = \mathbf{i}_1 + \mu_1 \cdot \mathbf{x}$ and $A2(\mathbf{x}) = \mathbf{i}_2 + \mu_2 \cdot \mathbf{x}$ are FMS functions. Neither program matches any of the woven AspectJ programs of Figure 5d-e. Note only the `inc()` method in Figure 6a is advised by μ_1 as μ_1 modifies its input `b`. Similarly `inc()` and `inc2()` are advised by μ_2 in Figure 6b, as μ_2 modifies its input ($i_1 + \mu_1 \cdot b$). In general, the set of programs that can be produced by AHEAD and similar systems is different than the set producible by AspectJ when comparable features are used [35]. We want a technology that can synthesize both sets of programs, and more.

```
class C { // program A1(B)
  int a = 0;

  void inc() {a++; outA1();}
  void inc2(){a=a+2;}
}                                    (a)
```

```
class C { // program A2(A1(B))
  int a = 0;
  void inc() {a++; outA1();
              outA2(); }
  void inc2(){a=a+2; outA2();}
  void inc3(){a=a+3; }
}                                    (b)
```

Figure 6. FMS-Synthetic Programs

6. In AspectJ parlance, `A2` has *precedence* over `A1`. Note that AOP precedence has the opposite meaning in mathematics: AOP precedence means apply *later*, not apply *first*.
7. More generally, AspectJ advice can modify advice bodies. By mapping advice bodies to introduced methods [35][41], there is a simple and uniform rule for AspectJ advice: it applies to all introductions of the base program and all introductions of all aspects.

To unify both, we distinguish modifiers that are applied immediately as in (9) to *the current state of a program*, from modifiers whose application is delayed until all features have been composed. The former, called *immediate modification*, is basic to the incremental development of programs (e.g., extreme programming), and the latter called *delayed modification* is basic to AspectJ.

Quarks express these distinctions by generalizing FMS functions to admit delayed modifiers. The *vanilla quark* represents a feature as a triple $<\gamma,\mathbf{i},\lambda>$ where γ is a delayed modifier, λ is an immediate modifier, and $\mathbf{i}$ is an FMS vector. A base program $\mathbf{i}$ is represented by the quark $<1,\mathbf{i},1>$, where 1 is the identity modifier (see footnote 2). An FMS function $F(\mathbf{x})=\mathbf{I}+\mu\cdot\mathbf{x}$ is $<1,\mathbf{I},\mu>$, where the $\mathbf{I}$ and μ coefficients of F are the right-most components of the quark, and 1 is the delayed modifier.

Vanilla quarks are composed by the operation •:

$$<\gamma_2,\mathbf{i}_2,\lambda_2>\bullet<\gamma_1,\mathbf{i}_1,\lambda_1> = <\gamma_2\gamma_1,\mathbf{i}_2+\lambda_2\cdot\mathbf{i}_1,\lambda_2\lambda_1> \quad (23)$$

Intuitively, (23) encodes a generalization of FMS function composition. FMS functions $F(\mathbf{x})=\mathbf{i}_2+\lambda_2\cdot\mathbf{x}$ and $G(\mathbf{x})=\mathbf{i}_1+\lambda_1\cdot\mathbf{x}$ are the quarks $<1,\mathbf{i}_2,\lambda_2>$ and $<1,\mathbf{i}_1,\lambda_1>$. Their composition $G\bullet F(\mathbf{x})=\mathbf{i}_2+\lambda_2\cdot\mathbf{i}_1+\lambda_2\lambda_1\cdot\mathbf{x}$ is the quark $<1,\mathbf{i}_2+\lambda_2\cdot\mathbf{i}_1,\lambda_2\lambda_1>$. The generalization of (23) is that left-most term of a quark (which is “1” in the F and G functions) lists the delayed modifiers that are to be applied. So if feature F had delayed modifier γ_2 it would be represented by the quark $<\gamma_2,\mathbf{i}_2,\lambda_2>$.

Quarks are composed by •, like features in GenVoca models. The reason is that quarks are a more general way to implement features. A formal reason is that a GenVoca model is a *monoid*: features are composed (by function composition), composition is associative, and there is the unit element (identity function). Quarks are also monoids, identities (23)-(25), as we show below.

The identity quark is $<1,\mathbf{0},1>$, where $\mathbf{0}$ is the zero vector:

$$<1,\mathbf{0},1>\bullet<\gamma,\mathbf{i},\lambda> = <\gamma,\mathbf{i},\lambda>\bullet<1,\mathbf{0},1> = <\gamma,\mathbf{i},\lambda> \quad (24)$$

Quark composition is not commutative and is associative:

$$<\gamma_3,\mathbf{i}_3,\lambda_3>\bullet[<\gamma_2,\mathbf{i}_2,\lambda_2>\bullet<\gamma_1,\mathbf{i}_1,\lambda_1>] = [<\gamma_3,\mathbf{i}_3,\lambda_3>\bullet<\gamma_2,\mathbf{i}_2,\lambda_2>]\bullet<\gamma_1,\mathbf{i}_1,\lambda_1> \quad (25)$$

The proof of (25) is straightforward.

Finally, the *image* of a quark is the expression that it represents, namely the *introductions that have accumulated in the quark times the accumulated global modifiers*:

$$\text{image}(<\gamma,\mathbf{i},\lambda>) = \gamma\cdot\mathbf{i} \quad (26)$$

The `image()` operation is intended to be applied to a quark where all features have been composed. λ does not appear in the final expression $\gamma\cdot\mathbf{i}$, as it has been applied to the accumulated introductions represented by $\mathbf{i}$. The next section illustrates this idea.

Note that quarks are more abstract than the feature functions of a FMS `(9)`. Quarks require *two* applications of `image()` to produce a source code representation. The first application converts a quark into an expression, and the second application converts the expression into source.

5 Implemented Special Cases

Static composition models that are exemplified by different tools and their research communities are special cases of quarks. We denote arbitrary advice by α, and the advice of a mixin by δ. We exclude EMs for simplicity.

Immediate Mixins. The composition models of AHEAD, Jx [38], gbeta [25], Classbox/J [19], and Jiazzi [36] use immediate mixins and the delayed modifier `1` (identity):

$B = \langle 1,\mathbf{I}_b,1\rangle$
$F = \langle 1,\mathbf{I}_f,\delta_f\rangle$
$G = \langle 1,\mathbf{I}_g,\delta_g\rangle$

$\mathrm{image}(G\bullet F\bullet B)$
$= \mathrm{image}(\langle 1,\mathbf{I}_g,\delta_g\rangle\bullet\langle 1,\mathbf{I}_f,\delta_f\rangle\bullet\langle 1,\mathbf{I}_b,1\rangle)$
$= \mathrm{image}(\langle 1,\mathbf{I}_g+\delta_g\cdot\mathbf{I}_f+\delta_g\delta_f\cdot\mathbf{I}_b,\delta_g\delta_f\rangle)$

$= \mathbf{I}_g+\delta_g\cdot\mathbf{I}_f+\delta_g\delta_f\cdot\mathbf{I}_b$ `// compare with (22)`

Delayed Advice. The composition model of AspectJ uses delayed advice and `1` as the immediate modifier:

$B = \langle 1,\mathbf{I}_b,1\rangle$
$F = \langle \alpha_f,\mathbf{I}_f,1\rangle$
$G = \langle \alpha_g,\mathbf{I}_g,1\rangle$

$\mathrm{image}(G\bullet F\bullet B)$
$= \mathrm{image}(\langle \alpha_g,\mathbf{I}_g,1\rangle\bullet\langle \alpha_f,\mathbf{I}_f,1\rangle\bullet\langle 1,\mathbf{I}_b,1\rangle)$
$= \mathrm{image}(\langle \alpha_g\alpha_f,\mathbf{I}_g+\mathbf{I}_f+\mathbf{I}_b,1\rangle)$

$= \alpha_g\alpha_f\cdot(\mathbf{I}_g+\mathbf{I}_f+\mathbf{I}_b)$ `// compare with (21)`

Immediate Mixins and Delayed Advice. CaesarJ [37] and FeatureC++ [3] both use delayed advice and immediate mixins:

$B = \langle 1,\mathbf{I}_b,1\rangle$
$F = \langle \alpha_f,\mathbf{I}_f,\delta_f\rangle$
$G = \langle \alpha_g,\mathbf{I}_g,\delta_g\rangle$

$\mathrm{image}(G\bullet F\bullet B)$

$$= \text{image}(<\alpha_g, \mathbf{I}_g, \delta_g>\bullet<\alpha_f, \mathbf{I}_f, \delta_f>\bullet<1, \mathbf{I}_b, 1>)$$
$$= \text{image}(<\alpha_g\alpha_f, \mathbf{I}_g+\delta_g\cdot\mathbf{I}_f+\delta_g\delta_f\cdot\mathbf{I}_b, \delta_g\delta_f>)$$

$$= \alpha_g\alpha_f\cdot(\mathbf{I}_g+\delta_g\cdot\mathbf{I}_f+\delta_g\delta_f\cdot\mathbf{I}_b)$$

In general, quarks provide a compact way to express and differentiate a variety of well-known composition models.

5.1 Perspective

Quantification. We noted earlier that modifiers express universal and unique existential quantification. Quarks add an extra dimension to features: immediate modifiers express bounded quantification (i.e., quantifiers over a program's terms at an incomplete state of its design), and delayed modifiers express unbounded quantification (i.e., quantifiers over a program's terms at a complete state of its design) [35]. Both kinds of quantification arise in program development.

Implementation. Quarks extend an FMS implementation by queuing the application of delayed modifiers until an `image()` operation is invoked. Delayed and immediate modifiers can be differentiated syntactically in languages by the presence of a "`delayed`" or "`immediate`" keyword that prefaces a modifier declaration, e.g.:

```
immediate after(): pointcut { advice-body }
delayed void m() { Super.m(); print("hi"); }
```

The above `after` advice is applied immediately to an input program, while the delayed mixin is evaluated after all features have been composed. A delayed mixin with an identity rewrite ensures that a particular term exists in a program.

Families of Quarks. In addition to immediate and delayed advice, higher-order advice — or *modifiers of modifiers (MOMs)* — has been explored [30][49][5]. MOMs come in delayed or immediate flavors, allowing a family of quarks to be created. One quark might have immediate modifiers and MOMs. Another could also have delayed MOMs. Modifiers of MOMs could be also defined, creating even more sophisticated quarks. See Appendix I. Vanilla quarks will be sufficient for most applications.

6 Related and Future Work

We previously cited prior work on mixins and aspects that provided a concrete grounding for FMSs [3][5][11][12][19][25][27][28][30][32][33][35][37][38][42][46][49]. In this section, we review a broader context in which FMSs can be placed.

There is a vast literature on program algebras and design calculi (e.g. [1][20][9][40]) which are intended to help calculate correct programs from formal specifications. The rules of a calculus assure the preservation of specified functionality and properties. The programming model is to incrementally transform or refine a formal specification to code in such a way that the code is mathematically consistent with the specification. Our work is different in its focus on (a) feature and aspect-oriented development, (b) their role in creating software product-lines, and (c) relating existing systems that synthesis programs to vector arithmetic. There is no explicit formal specification of requirements. Instead, product requirements are specified by the user's choice of features, and an FMS expression represents both the feature choices and the metaprogram that generates the desired product. Although not a focus of this paper, the development of correct programs by composing features is reviewed in [17].

Writing programs that manipulate other programs as data is *metaprogramming*. Program manipulation of FMS expressions is an example. Applying an image operation to translate an expression to a program is staged generation or *staged metaprogramming — i.e., writing generators of generators* [45]. As quarks require two applications of `image()` to yield program source, mapping a quark to source is an example of 2-staged metaprogramming.

The origin of FMSs lies in [34][35], which suggested that step-wise development of programs could be expressed by simple algebras. This paper formalizes these algebras as FMSs and quarks. A next step is to support vector (finite map) subtraction. Subtraction arises in the refactoring of programs and product lines, where classes and class members can be deleted (see Appendix IV). Refactorings also appear to be structure-preserving maps between GenVoca models [15][16] (see Appendix III).

Understanding program synthesis as FMS arithmetic captures the lock-step refinement of program representations. Other functional relationships that are not captured are *derivations* — deriving one program representation (`.class`) from another (`.java`). Derivations are central to *model-driven design (MDD)*. Capturing both refinement and derivation relationships among artifacts can be compactly expressed by category theory, which is the basis for the third generation of FOP [14][48].

Apel et al. have developed a feature algebra that models features as trees, called *feature structure trees (FSTs)* [6]. Features are composed by tree superimposition and by tree traversals and rewrites. Unlike FMSs, the FST algebra abandons commutativity of summation in favor of replacement. Introductions and mixin modifiers are unified, and advice is treated distinctly from mixins. Also, a separate operation is proposed to modify advice.

7 Conclusions

Software engineers define structures called programs that are added and transformed during program development. Finite Map Spaces (FMSs) capture the elementary mathematics that lies behind feature-based program construction. The simplicity of FMSs allowed us to show how diverse languages and tools implement special cases of FMSs. Further, FMSs enabled us to demonstrate the usefulness of a concise and unified explanation of different composition models. Our work can help others understand, characterize, and compare software compositional approaches, and see the fundamental mathematical operations that underlie this area of research. It is not accidental that we see the same ideas invented over and over again in different software composition languages and tools; these ideas are part of a larger paradigm that we are only now beginning to understand.

The long-term contribution of this paper is to lay the groundwork for understanding the activities of program design and construction from a simple mathematical perspective. From a product line viewpoint (where program synthesis can be scripted), the mathematical nature of program development becomes evident. Although we focussed on one activity in this paper (feature-based program synthesis), our approach can be generalized to account for other activities (e.g., refactorings and MDD) by admitting other operations, such as subtraction and derivation, and relationships, such as homomorphisms [15]. This paper takes us a step forward in our quest to define a pragmatic mathematics of program construction.

Acknowledgements. We gratefully acknowledge comments from H. Ossher, J. Gray, S. Apel, C. Lengauer, W. Cook, M. Grechanik, and P. Kim. This work was support in part by NSF's Science of Design Program (#CCF-0438786, #CCF-0724979).

8 References

[1] J-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[2] F. I. Anfurrutia, O. Diaz, and S. Trujillo. "On the Refinement of XML". *ICWE 2007*.

[3] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. "FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming". *GPCE 2005*.

[4] S. Apel, "The Role of Features and Aspects in Software Development", Ph.D. Dissertation, Dept. of Tech. and Bus. Info. Sys., University of Magdeburg, Germany, March 2007.

[5] S. Apel, C. Kästner, T. Leich, and G. Saake. "Aspect Refinement — Unifying AOP and Stepwise Refinement". *TOOLS EUROPE 2007*.

[6] S. Apel, C. Lengauer, D. Batory, B. Möller, and C. Kästner. "An Algebra for Feature-Oriented Software Development". University of Passau, MIP-0706, 2007.

[7] S. Apel and D. Hutchins, "gDeep: A Calculus for Feature Composition". Submitted.

[8] AspectJ Manual. `www.eclipse.org/aspectj/doc/ progguide/language.html`

[9] R-J. Back and J. Von Wright, The Refinement Calculus: A Systematic Introduction, Springer-Verlag, 1998

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

[11] D. Batory, J. Liu, J.N. Sarvela. "Refinements and Multi-Dimensional Separation of Concerns". *ACM SIGSOFT 2003.*

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

[13] D. Batory. "Feature Models, Grammars, and Propositional Formulas". *SPLC 2005*.

[14] D. Batory. "From Implementation to Theory in Program Synthesis". Keynote at *POPL 2007*.

[15] D. Batory. "Program Refactorings, Program Synthesis, and Model-Driven Design". *ETAPS 2007*.

[16] D. Batory and D. Smith. "Finite Map Spaces and Quarks". University of Texas TR-07-66.

[17] D. Batory and E. Boerger, "Modularizing Theorems for Software Product Lines". to appear.

[18] I.D. Baxter. “Design Maintenance Systems”. *CACM*, April 1992.

[19] A. Bergel, S. Ducasse, and O. Nierstrasz. “Classbox/J: Controlling the Scope of Change in Java”. *OOPSLA 2005*.

[20] R.S. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[21] L. Cardelli. “A Semantics of Multiple Inheritance”, *Proc. Int. Symp. on Semantics of Data Types*, 1984.

[22] R. Chitchyan, et al. *Aspects, Dependencies, and Interactions Workshop*, ECOOP 2006.

[23] G. Collins and D. Syme, “A Theory of Finite Maps”, *Higher Order Logic Theorem Proving and Its Applications*, 1995.

[24] B.A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 2002.

[25] E. Ernst. “Higher-Order Hierarchies”, *ECOOP 2003*.

[26] R.E. Filman, T. Elrad, S. Clarke, M. Aksit. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[27] M. Flatt, S. Krishnamurthi and M. Felleisen. “Classes and Mixins”. *POPL 1998*.

[28] J. Gray, et al. “Handling Crosscutting Constraints in Domain-Specific Modeling”. *CACM* Oct. 2001.

[29] J. Gray and S. Roychoudhury. “A Technique for Constructing Aspect Weavers Using a Program Transformation Engine”. *AOSD 2004*.

[30] K. Gybels and J. Brichau. “Arranging Language Features for More Robust Pattern-based Crosscuts”. *AOSD 2003*.

[31] W. Harrison, H. Ossher, and P. Tarr, “General Composition of Software Artifacts”. *Symposium on Software Composition*, March 2006.

[32] G. Kiczales, et al. “An Overview of AspectJ”. *ECOOP 2001*.

[33] J. Liu, D. Batory, and C. Lengauer. “Feature Oriented Refactoring of Legacy Applications”. *ICSE 2006*.

[34] R. Lopez-Herrejon et al. “Evaluating Support for Features in Advanced Modularization Technologies”. *ECOOP* 2005.

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

[36] S. McDirmid, M. Flatt, and W. C. Hsieh. “Jiazzi: New-Age Components for Old-Fashioned Java”. *OOPSLA 2001*.

[37] M. Mezini, K. Ostermann. “Conquering Aspects with Caesar”. *AOSD 2003*.

[38] N. Nystrom, S. Chong, A.C. Myers. “Scalable Extensibility via Nested Inheritance”. *OOPSLA 2004*.

[39] H. Ossher and W. Harrison, “Combination of Inheritance Hierarchies”. *OOPSLA 1992*.

[40] D. Pavlovic and D.R. Smith, “Software Development by Refinement” in *Formal Methods at the Crossroads: From Panacea to Foundational Support*, UNU/IIST 10th Anniversary B. Aichernig and T. Maibaum, eds., LNCS 2757, 2003.

[41] H. Rajan, K.J. Sullivan, “Classpects: Unifying Aspect- andObject-Oriented Programming”, *ICSE 2005*.

[42] C. Simonyi. “The Death of Computer Languages, the Birth of Intentional Programming”. NATO Science Committee Conference 1995.

[43] D.R. Smith. “A Generative Approach to Aspect-Oriented Programming”, *GPCE 2004*.

[44] G. Strang. *Introduction to Linear Algebra*, Wellesley-Cambridge Press, 1998.

[45] W. Taha and T. Sheard. “Multi-Stage Programming With Explicit Annotations”. *PEPM 1997*.

[46] P. Tarr and H. Ossher. “Hyper/J User and Installation Manual”. IBM Corporation, 2001.

[47] S. Thaker, D. Batory, D. Kitchin, and W. Cook. “Safe Composition of Product Lines”. *GPCE 2007*.

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

[49] D.B. Tucker and S. Krishnamurthi. “Pointcuts and Advice in Higher-Order Languages”. *AOSD 2003*.

[50] Wikipedia. “Name Mangling”, `http://en.wikipedia.org/wiki/Name_mangling`

Appendix I. Modifiers of Modifiers

In addition to immediate and delayed advice, researchers have explored higher-order advice or modifiers of modifiers [30][49]. Suppose feature `B` advises the `set()` and `get()` methods of a `Buffer` to trap changes to its state:

```
pointcut changesState() :
    execution( void Buffer.put*(int) ) ||
    execution( void Buffer.get*() );
```

Now suppose another feature adds method `clear()` to reinitialize `Buffer` contents. To capture calls to it, the `changesState` pointcut should be updated to:

```
pointcut changesState() :
    execution( void Buffer.put*(int) ) ||
    execution( void Buffer.get*() ) ||
    execution( void Buffer.clear() );
```

FMSs have two primitives: terms and modifiers of terms. We need a third to modify advice: a *modifier of a modifier (MOM).* A MOM h is a (`selector`, `rewrite`) pair. Its `pointcut` h^s`:M`$\rightarrow$`boolean` selects modifiers of a quark’s image. Its `rewrite` h^r`:M`$\rightarrow$`M` is a function that maps a modifier to a new modifier. If the `selector` does not qualify a modifier, the modifier is unchanged.

Suppose the image of a quark is expression `e`:

$$e = \gamma_2\gamma_1\cdot(i_3+\lambda_3\cdot(i_2+\lambda_2\cdot i_1))$$

Let $h[\mu]$ denote the modifier that results in applying h to modifier μ. MOM h rewrites `e` to $h(e)$:

$$h(e) = h[\gamma_2]h[\gamma_1]\cdot(i_3+h[\lambda_3]\cdot(i_2+h[\lambda_2]\cdot i_1))$$

That is, all modifiers of `e` may be altered by h. (Readers will note that this is a form of distribution or quantification, where h is applied to all modifiers of an expression). Again, only if the modifier is selected by the MOM’s `selector` will the MOM’s `rewrite` be applied.

MOMs are functions that are composed by function composition, which we denote by juxtaposition. The composite MOM $\mathscr{h}_1\mathscr{h}_2$ means apply $\mathscr{h}_2$ first then $\mathscr{h}_1$. The set of MOMs $\mathscr{H}$ is a monoid:

- *associative*: For all $\mathscr{a},\mathscr{b},\mathscr{c}\in\mathscr{H}$, we have $(\mathscr{ab})\mathscr{c} = \mathscr{a}(\mathscr{bc})$
- *identity element*: there exists an element $\mathscr{1}\in\mathscr{H}$, such that for all $\mathscr{h}\in\mathscr{H}$, $\mathscr{h1} = \mathscr{1h} = \mathscr{h}$.

To see how MOMs fit into quarks, we have to step back and recognize that a quark is an abstract representation of an expression, the quark's image. Taking the image of a quark composition `A•B` is equivalent to a transformation τ that maps the image of `B` to the image of `A•B`:

$$\texttt{A=<}\gamma_a\texttt{,}i_a\texttt{,}\lambda_a\texttt{>}$$
$$\texttt{B=<}\gamma_b\texttt{,}i_b\texttt{,}\lambda_b\texttt{>}$$

$$\begin{aligned}\texttt{image(A•B)} &= \gamma_a\gamma_b\cdot(i_a+\lambda_a\cdot i_b) \\ &= \tau(\gamma_b\cdot i_b) \qquad = \tau(\texttt{image(B)})\end{aligned}$$

Transformations like τ can be hard to write; composing quarks and projecting an image is easier. *But if* τ_a *could be defined, it would be a tree (expression) rewrite.*

The grammar below defines the structure of all quark images:

```
I   :  I + I     // vector addition
    |  i         // primitive vector
    |  M·I       // scalar multiplied vector
    ;

M   :  MM        // compound modifier
    |  μ         // primitive modifier
    ;
```

Let $\mathscr{R}^{\mathscr{h}}$ be the rewrite that is performed by applying a MOM $\mathscr{h}$ to a quark's image. The expression rewrite rules are:

$$\begin{aligned}
&\mathscr{R}^{\mathscr{h}}(\texttt{I+I}) \rightarrow \mathscr{R}^{\mathscr{h}}(\texttt{I}) + \mathscr{R}^{\mathscr{h}}(\texttt{I})\\
&\mathscr{R}^{\mathscr{h}}(\texttt{i}) \rightarrow \texttt{i}\\
&\mathscr{R}^{\mathscr{h}}(\texttt{M·I}) \rightarrow \mathscr{R}^{\mathscr{h}}(\texttt{M})\cdot\mathscr{R}^{\mathscr{h}}(\texttt{I})\\
&\mathscr{R}^{\mathscr{h}}(\texttt{MM}) \rightarrow \mathscr{R}^{\mathscr{h}}(\texttt{M})\mathscr{R}^{\mathscr{h}}(\texttt{M})\\
&\mathscr{R}^{\mathscr{h}}(\mu) \rightarrow \mathscr{h}[\mu]
\end{aligned}$$

That is, the rewriting of a quark image is a function that maps images to images. Composition of rewrites is denoted by juxtaposition. The composite rewrite $\mathscr{R}^{\mathscr{h}_1}\mathscr{R}^{\mathscr{h}_2}$ means apply rewrite $\mathscr{R}^{\mathscr{h}_2}$ first then $\mathscr{R}^{\mathscr{h}_1}$.

The *strawberry quark* has introductions ($\mathtt{i}$), immediate modifiers (λ), delayed modifiers (γ), and delayed MOMs ($\mathscr{h}$): $\langle\mathscr{h},\gamma,\mathtt{i},\lambda\rangle$. This means that MOMs are applied when the image of the program's quark is computed. The image of a strawberry quark is:

$$\mathtt{image}(\langle\mathscr{h},\gamma,\mathtt{i},\lambda\rangle) = \mathscr{R}^{\mathscr{h}}(\gamma\cdot\mathtt{i})$$

Strawberry quarks are composed by the operation •:

$$\langle\mathscr{h}_2,\gamma_2,\mathtt{i}_2,\lambda_2\rangle\bullet\langle\mathscr{h}_1,\lambda_1,\mathtt{i}_1\gamma_1,\rangle = \langle\mathscr{h}_2\mathscr{h}_1,\lambda_2\lambda_1,\mathtt{i}_2+\lambda_2\cdot\mathtt{i}_1,\gamma_2\gamma_1\rangle$$

The justification for `()` is the same as `(23)`.

The identity strawberry quark is `<1,1,0,1>`:

$$\langle 1,1,0,1\rangle\bullet\langle\mathscr{h},\gamma,\mathtt{i},\lambda\rangle = \langle\mathscr{h},\gamma,\mathtt{i},\lambda\rangle\bullet\langle 1,1,0,1\rangle = \langle\mathscr{h},\gamma,\mathtt{i},\lambda\rangle$$

Composition of strawberry quarks is associative:

$$\begin{aligned}&\langle\mathscr{h}_3,\gamma_3,\mathtt{i}_3\lambda_3,\rangle\bullet[\langle\mathscr{h}_2,\gamma_2,\mathtt{i}_2\lambda_2,\rangle\bullet\langle\mathscr{h}_1,\gamma_1,\mathtt{i}_1,\lambda_1\rangle] =\\ &[\langle\mathscr{h}_3,\gamma_3,\mathtt{i}_3,\lambda_3\rangle\bullet\langle\mathscr{h}_2,\gamma_2,\mathtt{i}_2,\lambda_2\rangle]\bullet\langle\mathscr{h}_1,\gamma_1,\mathtt{i}_1,\lambda_1\rangle\end{aligned} \tag{27}$$

To prove `(27)`, we compose the left hand side of `(27)`:

$$\begin{aligned}&\langle\mathscr{h}_3,\gamma_3,\mathtt{i}_3,\lambda_3\rangle\bullet[\langle\mathscr{h}_2,\gamma_2,\mathtt{i}_2,\lambda_2\rangle\bullet\langle\mathscr{h}_1,\gamma_1,\mathtt{i}_1,\lambda_1\rangle]\\ &= \langle\mathscr{h}_3,\gamma_3,\mathtt{i}_3,\lambda_3\rangle\bullet\langle\mathscr{h}_2\mathscr{h}_1,\delta_2\lambda_1,\mathtt{i}_2+\lambda_2\cdot\mathtt{i}_1,\lambda_2\lambda_1\rangle\\ &= \langle\mathscr{h}_3\mathscr{h}_2\mathscr{h}_1,\gamma_3\gamma_2\gamma_1,\mathtt{i}_3+\lambda_3\cdot\mathtt{i}_2+\lambda_3\lambda_2\cdot\mathtt{i}_1,\lambda_3\lambda_2\lambda_1\rangle\end{aligned} \tag{28}$$

Now we compose the right side:

$$\begin{aligned}&[\langle\mathscr{h}_3,\gamma_3,\mathtt{i}_3,\lambda_3\rangle\bullet\langle\mathscr{h}_2,\gamma_2,\mathtt{i}_2,\lambda_2\rangle]\bullet\langle\mathscr{h}_1,\gamma_1,\mathtt{i}_1,\lambda_1\rangle\\ &= \langle\mathscr{h}_3\mathscr{h}_2,\gamma_3\gamma_2,\mathtt{i}_3+\lambda_3\cdot\mathtt{i}_2,\lambda_3\lambda_2\rangle\bullet\langle\mathscr{h}_1,\gamma_1,\mathtt{i}_1,\lambda_1\rangle\\ &= \langle\mathscr{h}_3\mathscr{h}_2\mathscr{h}_1,\gamma_3\gamma_2\gamma_1,\mathtt{i}_3+\lambda_3\cdot\mathtt{i}_2+\lambda_3\lambda_2\cdot\mathtt{i}_1,\lambda_3\lambda_2\lambda_1\rangle\end{aligned} \tag{29}$$

`(28)` and `(29)` are equal, q.e.d.

AFM. The *Aspect Feature Modules* [4] use strawberry quarks, where immediate modifiers are mixins, delayed modifiers are advice, and MOMs are mixin modifiers ($\mathscr{d}$) that individually target one piece of advice for extension:

$$\begin{aligned}&\mathtt{B} = \langle 1,1,\mathtt{I}_b,1\rangle\\ &\mathtt{F} = \langle\mathscr{d}_f,\alpha_f,\mathtt{I}_f,\delta_f\rangle\\ &\mathtt{G} = \langle\mathscr{d}_g,\alpha_g,\mathtt{I}_g,\delta_g\rangle\end{aligned}$$

$$\begin{aligned}&\mathtt{image}(\mathtt{G}\bullet\mathtt{F}\bullet\mathtt{B})\\ &= \mathtt{image}(\langle\mathscr{d}_g,\mu_g,\mathtt{I}_g,\delta_g\rangle\bullet\langle\mathscr{d}_f,\mu_f,\mathtt{I}_f,\delta_f\rangle\bullet\langle 1,1,\mathtt{I}_b,1\rangle)\\ &= \mathtt{image}(\langle\mathscr{d}_g\mathscr{d}_f,\mu_g\mu_f,\mathtt{I}_g+\delta_g\cdot\mathtt{I}_f+\delta_g\delta_f\cdot\mathtt{I}_b,\delta_g\delta_f\rangle)\end{aligned}$$

$$\begin{aligned}&= \mathscr{R}^{hg}\mathtt{R}^{hf}(\mu_g\mu_f\cdot(\mathtt{I}_g + \delta_g\cdot\mathtt{I}_f + \delta_g\delta_f\cdot\mathtt{I}_b))\\ &= \mathscr{R}^{hg}(\mathscr{d}_f[\mu_g]\mathscr{d}_f[\mu_f]\cdot(\mathtt{I}_g + \mathscr{d}_f[\delta_g]\cdot\mathtt{I}_f + \mathscr{d}_f[\delta_g]\mathscr{d}_f[\delta_f]\cdot\mathtt{I}_b))\\ &= \mathscr{d}_g\mathscr{d}_f[\mu_g]\mathscr{d}_g\mathscr{d}_f[\mu_f]\cdot(\mathtt{I}_g\ \mathscr{d}_g\mathscr{d}_f[\delta_g]\cdot\mathtt{I}_f+\mathscr{d}_g\mathscr{d}_f[\delta_g]\mathscr{d}_g\mathscr{d}_f[\delta_f]\cdot\mathtt{I}_b)\end{aligned}$$

Appendix II. Canonical Forms Using !

In this section, we show that the composition of two functions in form (20) yields another function in the same form. Without loss of generality, consider the following feature functions F, G, and H that modify the value of single term. This allows us to drop the ^{r} superscripts from ! and μ, as we assume the selectors of ! and μ have qualified the term.

$$\begin{aligned}
F(x) &= i_f + !\mu_f \cdot x && \text{; uses !}\\
G(x) &= i_g + !\mu_g \cdot x && \text{; uses !}\\
H(x) &= i_h + \mu_h \cdot x && \text{; does not use !}
\end{aligned}$$

We need the following identities, whose proof follows a case analysis:

$$!(x+!y) = x + !y \qquad \text{; for all } x,y \in L \tag{30}$$

$$!(x+y) = x + y \qquad \text{; for all } x\text{: } 0<x<T \text{ and ; for all } y \in L \tag{31}$$

Consider the composition of functions F and G that use !:

$$\begin{aligned}
F \bullet G &= i_f + !\mu_f \cdot (i_g + !\mu_g \cdot x) && \text{; defn of F,G}\\
&= i_f + !(\mu_f \cdot i_g + \mu_f !\mu_g \cdot x) && \text{; (8)}\\
&= i_f + !(\mu_f \cdot i_g + !\mu_f \mu_g \cdot x) && \text{; (19)}\\
&= i_f + \mu_f \cdot i_g + !\mu_f \mu_g \cdot x && \text{; (30)}\\
&= (i_f + \mu_f \cdot i_g) + !\mu_f \mu_g \cdot x && \text{; (i)}\\
&= i_{fg} + !\mu_{fg} \cdot x && \text{; form (20)}
\end{aligned}$$

Now consider the composition of functions where only the inner function uses !:

$$\begin{aligned}
H \bullet G &= i_h + \mu_h \cdot (i_g + !\mu_g \cdot x) && \text{; defn of H,G}\\
&= i_g + (\mu_h \cdot i_g + \mu_h !\mu_g \cdot x) && \text{; (8)}\\
&= i_h + (\mu_h \cdot i_g + !\mu_h \mu_g \cdot x) && \text{; (19)}\\
&= (i_h + \mu_h \cdot i_g) + !\mu_h \mu_g \cdot x && \text{; (i)}\\
&= i_{hg} + !\mu_{hg} \cdot x && \text{; form (20)}
\end{aligned}$$

Lastly consider composing functions where only the outer function uses !:

$$\begin{aligned}
G \bullet H &= i_g + !\mu_g \cdot (i_h + \mu_h \cdot x) && \text{; defn of G,H}\\
&= i_g + !(\mu_g \cdot i_h + \mu_h \mu_g \cdot x) && \text{; (8)} \qquad (32)
\end{aligned}$$

Expression (32) contains a subexpression of the form $!(m+n)$, (where $m=\mu_g \cdot i_h$ and $n=\mu_h \mu_g \cdot x$), which cannot be simplified without more information. For any implementation of function H, we will know the value of i_h: it will either be zero, a normal value, or an error. A case analysis shows that such information is sufficient to guarantee that (32) can be rewritten into form (20).

Case 0: $i_h=0$

$$\begin{aligned}
G\bullet H &= \mathbf{i}_g + \mathbf{!}(\mu_g\cdot\mathbf{i}_h + \mu_h\mu_g\cdot\mathbf{x}) &&; (32)\\
&= \mathbf{i}_g + \mathbf{!}(\mu_g\cdot\mathbf{0} + \mu_h\mu_g\cdot\mathbf{x}) &&; \mathbf{i}_h=0\\
&= \mathbf{i}_g + \mathbf{!}(\mathbf{0} + \mu_h\mu_g\cdot\mathbf{x}) &&; (4)\\
&= \mathbf{i}_g + \mathbf{!}\mu_h\mu_g\cdot\mathbf{x} &&; \text{(iii) and (20)}
\end{aligned}$$

Case 1: $0<\mathbf{i}_h<T$

$$\begin{aligned}
G\bullet H &= i_g + !(\mu_g\cdot i_h + \mu_h\mu_g\cdot x) &&; (32)\\
&= i_g + \mu_g\cdot i_h + \mu_h\mu_g\cdot x &&; (31)\\
&= (i_g + \mu_g\cdot i_h) + \mu_h\mu_g\cdot x &&; (20)
\end{aligned}$$

Case 2: $i_h=T$

$$\begin{aligned}
G\bullet H &= i_g + !(\mu_g\cdot i_h + \mu_h\mu_g\cdot x) &&; (32)\\
&= i_g + !(\mu_g\cdot T + \mu_h\mu_g\cdot x) &&; i_h=T\\
&= i_g + !(T + \mu_h\mu_g\cdot x) &&; (2)\\
&= i_g + !(T) &&; \text{lattice } L\\
&= i_g + T &&; (15)\\
&= T &&; (20)
\end{aligned}$$

Note in Case 2, we could define $G\bullet H=T+x$, or $\mathbf{G}\bullet H=T+0\cdot x$ if we had annihilators, to match the exact form of (20).

We know that the composition of two functions that do not use ! can be written into the form (20). Thus, functions that use ! preserve the compositionality property — the result is always in the form of (20). q.e.d.

Appendix III. FMS Homomorphisms

A *homomorphism* is a map from one structure to another that preserves meaning. Common object-oriented refactorings, like rename member, rename class, and move member, preserve the meaning of a program and intuitively define a homomorphism between the original program structure and its refactored structure.[8]

Let $FMS_1=<V_1,M_1>$ and $FMS_2=<V_2,M_2>$ where V_1, V_2 are sets of vectors, and M_1, M_2 are sets of modifiers. Let: hV: $V_1\rightarrow V_2$ and $hM:M_1\rightarrow M_2$. A property of hV is that it preserves the underlying value lattice L. $h=<hV,hM>:FMS_1\rightarrow FMS_2$ is a *FMS-Homomorphism* if for all $\mu\in M_1$ and $v\in V_1$:

$$hV(\mu\cdot v) = hM(\mu)\cdot hV(v) \qquad (33)$$

and for all vectors $v_1,v_2\in V_1$:

$$hV(v_1+v_2) = hV(v_1)+hV(v_2) \qquad (34)$$

8. Commonly refactorings are called meaning preserving and “structure” altering. In mathematics, “structure” denotes meaning, so there is a misalignment of conventional and mathematical use of the term “structure”.

We conjecture that common object-oriented refactorings are FMS homomorphisms (see [15] for details).

Appendix IV. Subtraction

Let x, y, z be values of lattice L. Term subtraction is governed by the following rules:

$$x - 0 = x \tag{35}$$

$$x - y = 0 \quad ;\ \text{if } y>0 \tag{36}$$

That is, subtracting 0 does not alter a value, and subtracting a non-zero value yields 0.

Subtraction is not commutative and is left-associative (i.e., $x-y-z = ((x-y)-z)$). It is easy to prove the following identity:

$$x - (y+z) = x - y - z \tag{37}$$

and that UMs distribute over subtraction:

$$\mu\cdot(x-y) = \mu\cdot x - \mu\cdot y \tag{38}$$

(38) can be simplified since modifiers do not effect the result of subtraction:

$$-\mu\cdot y = -y \tag{39}$$

Lifting term subtraction to vector subtraction means that the difference of two vectors equals the differences between their respective components:

$$[x_1\ x_2\ \ldots\ x_m] - [y_1\ y_2\ \ldots\ y_m] = [x_1-y_1\ x_2-y_2\ \ldots\ x_m-y_m] \tag{40}$$

Subtraction permits features to delete or replace terms. The general form of a feature F that adds vector **I**, subtracts vector **S**, and modifies existing vectors by μ is:

$$F(\mathbf{x}) = \mathbf{I} + \mu\cdot(\mathbf{x}-\mathbf{S}) \tag{41}$$

(41) satisfies the notion of compositionality in Section 3.9.3. That is, compositions of functions of form (41) are also of form (41).

To allow quarks to admit subtraction, another term needs to be added. A vanilla quark generalizes to a 4-tuple $<\gamma,\mathbf{i},\mathbf{s},\lambda>$, here called a *chocolate quark*, where γ is the global modifier, **i** is the vector of introductions to be added, **s** is the vector of introductions to be subtracted, and λ is the local modifier. Defining chocolate quark composition, chocolate quark identity, and associativity is left as an exercise.