An Algebraic Foundation for Automatic Feature-Based Program Synthesis

Sven Apel[a], Christian Lengauer[a], Bernhard Möller[b], Christian Kästner[c]

[a]*Department of Informatics and Mathematics, University of Passau,*
{apel,lengauer}@uni-passau.de

[b]*Institute of Computer Science, University of Augsburg,*
moeller@informatik.uni-augsburg.de

[c]*School of Computer Science, University of Magdeburg,*
kaestner@iti.cs.uni-magdeburg.de

Abstract

Feature-Oriented Software Development provides a multitude of formalisms, methods, languages, and tools for building variable, customizable, and extensible software. Along different lines of research, different notions of a feature have been developed. Although these notions have similar goals, no common basis for evaluation, comparison, and integration exists. We present a feature algebra that captures the key ideas of feature orientation and that provides a common ground for current and future research in this field, on which also alternative options can be explored. Furthermore, our algebraic framework is meant to serve as a basis for the development of the technology of automatic feature-based program synthesis and architectural metaprogramming.

Key words: Feature-oriented software development, automatic feature-based program synthesis, architectural metaprogramming, feature structure tree, feature composition, superimposition, quantification, weaving, feature algebra, quark model

1 Introduction

Feature-Oriented Software Development (*FOSD*) is a paradigm that provides formalisms, methods, languages, and tools for building variable, customizable, and extensible software. The main abstraction mechanism of FOSD is the *feature*. A feature reflects a stakeholder's requirement and is typically an increment in functionality; features are used to distinguish between different variants of a program or software system [42]. *Feature composition* is the process of composing code associated with features in a consistent way [6].
Research along different lines has been undertaken to realize the vision of FOSD [12, 17, 29, 40, 42, 61, 71]. While there are the common notions of a feature and feature

composition, present approaches use different techniques, representations, and formalisms. For example, AspectJ [1] and AHEAD [2] can both be used to implement features, but they provide different language constructs: AspectJ offers pointcuts, advice, and inter-type declarations, whereas AHEAD furnishes collaborations and refinements [12]. A promising way of integrating separate lines of research is to provide an encompassing formal framework that captures many of the common ideas such as introductions, refinements, or quantification and that hides (what we feel are) distracting differences.

We propose such a framework for FOSD: a *feature algebra*. First, the feature algebra abstracts from the details of different programming languages and environments used in FOSD. Second, alternative design decisions in the algebra reflect variants and alternatives in concrete programming language mechanisms; for example, certain kinds of feature composition may be allowed or disallowed. Third, the algebra is useful for describing, beside composition, also other operations on features formally and independently of the language, e.g., type checking [69] and interaction analysis [51]. Fourth, the algebraic description can be taken as an architectural view of a software system. External tools can use the algebra as a basis for optimizing feature expressions [17, 29].

The big picture of our endeavor is that the feature algebra serves as a formal foundation for automatic feature-based program synthesis [16, 29] and architectural metaprogramming [15]. Both paradigms emerged from FOSD and facilitate the treatment of programs as values manipulated by metaprograms, e.g., in order to add a feature to a program. This requires a formal theory that describes precisely which manipulations are allowed. Architectural metaprogramming applies this principle at the level of a software architecture. The algebra provides a formalism to express the necessary abstraction from the implementation level and is a means of reasoning about and manipulating software architectures. Metaprograms operate on feature-algebraic expressions to synthesize programs at the architectural level efficiently and consistently.

We introduce a uniform representation of features, outline the properties of the algebra, explain how the algebra models the key concepts of FOSD, and discuss alternative configurations of the algebra and their implications for the properties of feature composition. We have implemented all axioms, lemmas, and theorems in Prover9 [3] so that we can derive proofs of fundamental properties of feature composition fully automatically.

[1] `http://www.eclipse.org/aspectj/`
[2] `http://www.cs.utexas.edu/~schwartz/ATS.html`
[3] `http://prover9.org/`

2 A Language-Independent Representation of Features

2.1 Features and Their Composition

Different researchers have proposed different views of what a feature is or should be. A characterization that is common to most (if not all) work on FOSD is: *a feature is a structure that extends and modifies a given program in order to satisfy a stakeholder's requirement, to implement a design decision, and to offer a configuration option* [6]. This informal characterization guides our work towards a formal framework of FOSD.
Mathematically, we assume an abstract set F of features and describe feature composition by an operator $\bullet : F \times F \to F$. This allows us to combine elementary features to more complex ones, but also to recombine these further. A program p (which can itself be viewed as a feature) is composed of a series of features:

$$p = f_n \bullet (f_{n-1} \bullet (\ldots \bullet (f_2 \bullet f_1)))$$

The order of features in a composition may matter since feature composition is not generally commutative, and parenthesization may matter since feature composition is not in every case associative, as we will show. For simplicity, we define feature composition such that each feature can appear only once in a composition. Allowing multiple instances of one and the same feature in a composition would be possible, but this would only complicate the algebraic framework and does not provide any new insights.

2.2 The Structure of Features

We develop our model of features in several steps. Even though the algebra is language-independent, we explain its details and their implications by means of Java code. First, we consider a feature to be ultimately composed of elemental structures such as single fields or methods arranged in the form of a tree. (Sec. 2.2–3.1). More generally, we will need to consider forests of such trees to make the algebra work; we call these forests *basic features*.
Basic features are composed by superimposition of their tree structures. In a next step, we introduce the concept of a *modification* that acts as a rewrite on basic features (Sec. 3.2). Finally, a *full feature* takes the form of a triple, called a *quark*, consisting of a basic feature and two kinds of modifications (Sec. 4).
The tree structures in a basic feature are called *feature structure trees* (*FSTs*), while the forest itself is called a *feature structure forest* (*FSF*). As a borderline case, we also admit the empty FSF. An FST organizes the feature's structural elements, e.g., files, classes, fields, or methods, hierarchically. Figure 1 depicts an excerpt of the Java implementation of a feature BASE and its representation in form of an FST. One can think of an FST as a stripped-down abstract syntax tree that contains only the essential information. The nature of this information depends on the degree of granularity at which software artifacts are to be composed, as we discuss below.

```
1 package util;
2 class Calc {
3   int e0=0, e1=0, e2=0;
4   void enter(int v) { e2=e1; e1=e0; e0=v; }
5   void clear() { e0=e1=e2=0; }
6   String top() { return String.valueOf(e0); }
7 }
```

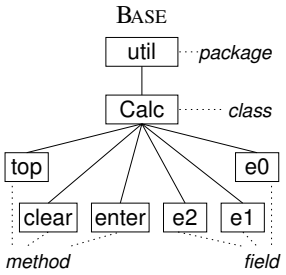

Fig. 1. Implementation and FST of the feature BASE.

For example, the FSTs we use to represent Java code contain nodes that denote packages, classes, interfaces, fields, methods, etc. They do not contain information about the internal structure of methods and so on. A different granularity would be to represent only packages and classes but not methods or fields as FST nodes (coarse granularity), or to represent additionally statements or expressions (fine granularity) [43]. However, the choice of the level of granularity does not affect our description of the algebra.

Each node of an FST is labeled with a name and a type. We call two nodes *compatible* when they have the same name and type and compatible parents. A node's name [4] corresponds to the name of the artifact's structural element it represents and a node's type corresponds to the syntactic category to which the element belongs. For example, the class `Calc` is represented by a node `Calc` of type `class`. We must consider both name and type to prevent the combination of incompatible nodes during feature composition, e.g., the composition of two classes with different names, or of a field with a method of the same name.

For the purposes of the present paper, we consider FSTs and FSFs to be ordered; there is an analogous model for the case of unordered trees. The rightmost child of a node represents the topmost element in the textual order of an artifact, e.g., the first member in a class is represented by the rightmost child node. Note that, at the granularity we chose for Java, the order of nodes could be arbitrary, but this may be different at a finer granularity (e.g., the order of statements matters) and it may also differ in other languages (e.g., the order of most XHTML elements matters) [10].

2.3 Feature Composition

How does the abstract description of a composition of basic features map to the concrete composition at the structural level? That is, how are FSTs composed in order to obtain new FSTs? Our answer is: by *FST superimposition* [10,17,23,27,60].

2.3.1 Superimposition

Superimposition has been applied successfully to the composition of class hierarchies in multi-team software development [60], the extension of distributed programs [24], the implementation of collaboration-based designs [66], feature-oriented programming [17,61], subject-oriented programming [35,68], aspect-oriented pro-

[4] Depending on the language, a name could be a simple identifier, a signature, etc.

gramming [55,56], and software component adaptation [23]. Although very different, all these applications superimpose hierarchically organized program constructs by matching their relative positions, names, and types in the hierarchy.
Two trees are superimposed by superimposing their subtrees, starting from their roots and descending recursively. [5] To keep the algebra simple, we want superimposition to be a total operation. If the roots of the two trees under consideration are not compatible, we just combine these trees into a two-element FSF. Otherwise, the two nodes are merged by superimposing recursively the FSF F_l of children of the left tree onto the FSF F_r of children of the right tree. In analogy to our conventions for FSTs (see Sec. 2.2), the children in F_l are superimposed onto those in F_r beginning with the rightmost node of F_l, thus preserving the order in the resulting FST; nodes in F_l that cannot be merged with nodes in F_r are added to the left; the nodes in F_r remain at their original positions.
In Figure 2, we illustrate the process of FST superimposition for trees with compatible roots. Superimposition is denoted by the operator • . In Figure 3, we depict the corresponding Java code. Our feature BASE is extended by superimposing a feature ADD onto it. The result is a new feature, which we call ADDBASE. We will present more complex examples later.

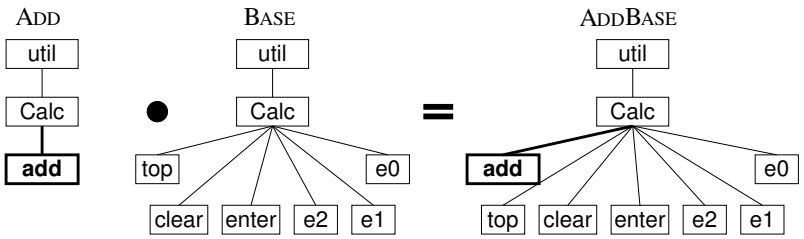

Fig. 2. An example of FST superimposition (ADD • BASE = ADDBASE).

Superimposition is orthogonal to the extension mechanisms provided by Java. Whereas existing classes of a program can be extended by inheritance or delegation creating new derived classes, with superimposition, existing classes can be extended without creating new classes.

2.3.2 Terminal and Non-Terminal Nodes

Independently of any particular language, an FST is made up of two different kinds of nodes:

Non-terminal nodes are the inner nodes of an FST, including the root. The subtree rooted at a non-terminal node reflects the structure of some implementation artifact of a feature. The artifact structure is regarded as *transparent* (substructures are represented by child nodes) and is subject to the recursive superimposition process. A non-terminal node has only a name and a type, i.e., no superimposition of additional content is necessary.

[5] Conceptually, FST superimposition is a form of *tree amalgamation* that considers inner nodes and that starts from a common root when there is one [21].

```
1 package util;
2 class Calc {
3   void add() { e0=e1+e0; e1=e2; }
4 }
```

•

```
1 package util;
2 class Calc {
3   int e0=0, e1=0, e2=0;
4   void enter(int val) { e2=e1; e1=e0; e0=val; }
5   void clear() { e0=e1=e2=0; }
6   String top() { return String.valueOf(e0); }
7 }
```

=

```
1 package util;
2 class Calc {
3   int e0=0, e1=0, e2=0;
4   void enter(int val) { e2=e1; e1=e0; e0=val; }
5   void clear() { e0=e1=e2=0; }
6   String top() { return String.valueOf(e0); }
7   void add() { e0=e1+e0; e1=e2; }
8 }
```

Fig. 3. Java code for the superimposition ADD • BASE = ADDBASE.

Terminal nodes are the leaves of an FST. A terminal node has a name, a type, and usually some content. Conceptually, a terminal node may also be the root of some substructure which, however, is regarded as *opaque* in our model (substructures are not represented by child nodes). The content of a terminal is not shown in the FST.

While the superimposition of two non-terminals continues the recursive descent in the FSTs, the superimposition of two terminals terminates the recursion and requires a special treatment that may differ for each type of node.

Let us illustrate these concepts for Java. In Java, we choose to represent packages, classes, and interfaces by non-terminals. The implementation artifacts they contain are represented by child nodes, e.g., a package contains several classes and classes contain inner classes, methods, and fields. Two compatible non-terminals are superimposed by superimposing their child nodes, e.g., two packages with equal names are merged to one package with the same name that contains the superimposition of the child elements (classes, interfaces, subpackages) of the two original packages. In contrast, Java methods, fields, imports, modifier lists, and `extends` and `implements` clauses are represented by terminals (the leaves of an FST), at which the recursion terminates. For each type of terminal node, there needs to be a rule for superimposing their contents.

2.3.3 Superimposition of Terminals

We now turn to the question of superimposing terminals. Each terminal type has to provide its own rule for superimposition. Here are four examples for Java and similar languages:

- Two methods can be superimposed if it is specified how the method bodies

are merged, e.g., by overriding and calling the original method by using the keywords `original` [19] or `Super` [17] inside a method body.
- Two field declarations with equal name and type are superimposed by replacing one initial value (if any) by the other.
- Two `implements` clauses are superimposed by taking the union of their elements and removing duplicates.
- Two modifier lists are superimposed by a specific policy, e.g., `public` replaces `private`, but not vice versa; the superimposition of a modifier list containing `static` with one not containing `static` is undefined, i.e., causes an error, and so on.

In other languages, such as XML or BNF grammars, similar rules based on overriding, replacement, or concatenation are useful [10, 17]. If the language does not provide a rule, the superimposition of the left onto the right terminal just leaves the right one unchanged (to preserve totality of superimposition). In the implementation, a further possibility is to disallow the composition of two terminals of a certain type and to throw an exception if it is being attempted. However, in our case studies, this phenomenon never occurred [10].

Figure 4 and Figure 5 depict how Java methods are superimposed during the composition of the two basic features COUNT and BASE. The two methods `enter` of COUNT and BASE are superimposed by inlining one method into the other. The keyword `original` specifies how the method bodies are merged by inlining (without knowledge of their source code). [6] The two `clear` methods are superimposed analogously. The semantics of this composition is similar to method overriding in Java, except that, in Java, the method of a subclass overrides the corresponding method of a superclass and, with superimposition, the method of the superimposing class overrides the method of the superimposed one.

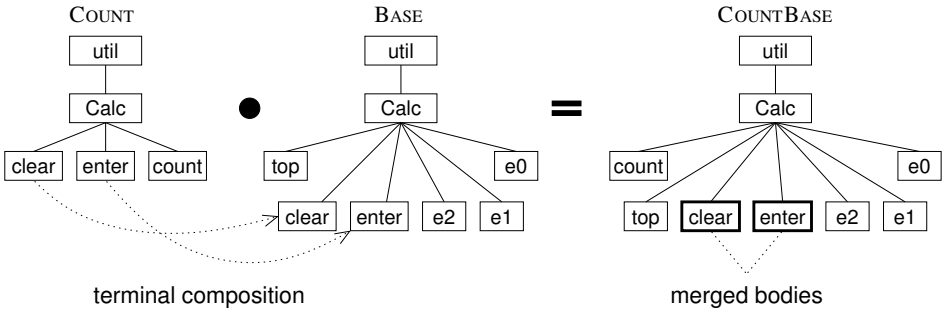

Fig. 4. Superimposing Java methods in COUNT • BASE = COUNTBASE.

Beside inlining, alternative superimposition rules for terminals such as wrapping are possible [17, 36].

[6] Technically, if a node `n_new` with the keyword `original` in a method `m` is superimposed onto a node `n_old` that also contains a declaration of method `m`, then the declaration of `m` in `n_old` is replaced by that of `m` in `n_new`, while replacing the pseudo-call to `original` by the original body of `m` taken from `n_old`; parameters and local variables receive fresh names if necessary.

```
1 package util;
2 class Calc {
3   int count=0;
4   void enter(int val) { original(val); count++; }
5   void clear() { original(); if(count > 0) count--; }
6 }
```

●

```
1 package util;
2 class Calc {
3   int e0=0, e1=0, e2=0;
4   void enter(int val) { e2=e1; e1=e0; e0=val; }
5   void clear() { e0=e1=e2=0; }
6   String top() { return String.valueOf(e0); }
7 }
```

=

```
1 package util;
2 class Calc {
3   int e0=0, e1=0, e2=0;
4   void enter(int val) { e2=e1; e1=e0; e0=val; count++; }
5   void clear() { e0=e1=e2=0; if(count > 0) count--; }
6   String top() { return String.valueOf(e0); }
7   int count=0;
8 }
```

Fig. 5. Superimposing Java methods by inlining.

2.4 Discussion

The superimposition of FSFs requires several properties of the language in which the elements of a feature are expressed:

(1) The structure of a feature must be hierarchical, i.e., a forest.
(2) Every structural element of a feature must have a name and type that become the name and type of the node in the FST.
(3) An element must not contain two or more direct child elements with the same name and type.
(4) The language must provide superimposition rules for elements that do not have a hierarchical substructure (terminals), or their superimposition leaves the program unchanged.

These constraints are satisfied by most contemporary programming languages. We have developed a tool, called *FeatureHouse*, that implements feature composition based on the feature algebra [10]. Using *FeatureHouse*, we have been able to compose features written in various languages such as Java, C#, C, Haskell, and JavaCC.[7] Other researchers have shown that also other (non-code) languages such as grammar or markup languages align well with the constraints above [3, 17]. Languages that do not satisfy these constraints are not "feature-ready", since they do not provide sufficient structural information. For example, XHTML is not feature-ready since most elements of an XHTML document do not have unique names. However, it may be possible to make such languages feature-ready by assigning

[7] `http://www.fosd.de/fh/`

names to elements explicitly or by extending the languages with an overlaying module structure, as Xak does for XML languages [3].

3 Feature Algebra

The feature algebra provides a formal foundation for FOSD. It abstracts from the concrete case of FSFs by listing the essential operators together with the algebraic laws, formulated as axioms, that we deem reasonable in some concrete setting of FOSD. All axioms hold in the concrete algebra of ordered FSFs as well as in the mentioned algebra of unordered FSFs, but also in many others, since they are not very restrictive. A manipulation of an algebraic expression induces a corresponding manipulation of an FSF.
As an important design decision, the set of operators of the algebra must be rich enough to admit, for every FSF, an expression that corresponds to it, in the ideal case even uniquely. [8] This way, the algebraic expressions facilitate formal reasoning about the corresponding FSFs. Thus, FSFs can be converted, without information loss, to algebraic expressions and vice versa. Our laws for algebraic expressions describe what is allowed and disallowed when manipulating FSFs.
Since the algebra uses only axioms in the shape of (implicitly universally quantified) equations, it lends itself to automated theorem proving with off-the-shelf first-order provers. We have implemented all axioms, lemmas, and theorems of the algebra in Prover9, to which we refer in the relevant paragraphs. In Appendix A, we list the source code of the corresponding Prover9 scripts.

3.1 Introductions

Introductions are the abstract counterparts of FSFs. Therefore, every feature algebra has to comprise a set I of introductions. Among them one usually distinguishes a subset of *atomic introductions*. In the concrete algebra of FSFs these correspond to leaf nodes, characterized by the *unique maximal paths* from the respective roots to them. A basic feature can also be represented as the superimposition of all paths resp. atomic introductions in its FSF. Hence, an abstract superimposition operator is the second main ingredient of a feature algebra; it is called *introduction sum*. First, we explain the properties of paths and, then, we introduce introduction sum.

3.1.1 Paths

To obtain an algebraic representation of FSFs, we first flatten their hierarchical structure and convert every FSF into a list of its paths.
Specifically, we use a simple prefix notation to identify paths, which is similar to fully qualified names in Java: the *path name* of an FSF node n consists of the names

[8] This requirement may necessitate restrictions on a given language. For example, to enforce it for Java, we require the textual order of the subelements of an element to be the order of the children of the node that represents the element (see Sec. 2.2).

of all the nodes along the path from the root of the tree in the FSF to which n belongs, separated by dots. [9]

Often we will need additional information about the genesis of a certain FSF. To this end, we use the concept of a *feature path name* which consists of a path name prefixed with the identifier of a feature in which the introduction occurs and a double colon '::'; for brevity, this is not depicted in the FSFs. As a notational convention, the identifier of a feature is its name written in italics rather than in small capitals. For example, the feature path name of the method `clear` in our feature BASE (from Fig. 2) is $\textit{Base}::util.Calc.clear$.

Let us now represent FSFs by lists of feature path names. To obtain a normalization, we do not use arbitrary lists of such paths. For every path name, also every non-empty prefix denotes a valid path to some ancestor node; the same holds for feature path names. For example, the prefix $\textit{Base}::util$ of $\textit{Base}::util.Calc$ denotes a valid path, too. We use only prefix-closed lists and arrange them in a particular order. The rationale behind this restriction is a possible extension of the algebra to be explained in Section 3.2.2. The paths belonging to an FST in an FSF all start with the name of the respective root node. They are grouped together in the corresponding list and ordered according to a generalized postfix traversal of the respective tree. In particular, paths leading to children reside left of the paths to their parents. According to our conventions, upon superimposition, this achieves that "missing" inner nodes are added to the partner tree before their children.

Every single path in an FSF, in particular, every maximal path to a leaf, is represented by the (repetition-free) list of all its prefixes. For example, the path $\textit{Base}::util.Calc$ corresponds to the list

$$[\textit{Base}::util.Calc,\ \textit{Base}::util,\ \textit{Base}]$$

3.1.2 Introduction Sum

Introduction sum $\oplus$ is a binary operation defined over the set I of introductions; it is the abstract counterpart of FSF superimposition denoted by $\bullet$ in Section 2.3. To emphasize that we are now working towards a more formal setting, we will, from now on, write $\oplus$ instead of $\bullet$. The result of an introduction sum is again an introduction:

Definition 1 (Introduction sum $\oplus$)

$$\oplus : I \times I \to I$$

We choose as the above-mentioned atomic introductions the lists for the maximal paths that can occur. Since, notationally, this would quickly become excessive, we allow single paths as shorthands for the lists that represent them. Thus, an FSF is denoted algebraically as the sum of all paths that correspond to its structural

[9] To be specific, the fully qualified name of an atomic introduction would also need to include the type of each path element. For brevity and because there are no ambiguities in our examples, we omit the type information here.

elements. For instance, our feature BASE (from Fig. 2) is expressed as the sum

$$\begin{aligned}\text{BASE} = \; & \textit{Base}::util.Calc.top \oplus \textit{Base}::util.Calc.clear \\ & \oplus \textit{Base}::util.Calc.enter \oplus \textit{Base}::util.Calc.e2 \\ & \oplus \textit{Base}::util.Calc.e1 \oplus \textit{Base}::util.Calc.e0\end{aligned}$$

Since the paths of an FSF are unique, the set of these summands is unique as well. Two features are composed by adding their atomic introductions. By our conventions, FSF list F_l is superimposed onto FSF list F_r by traversing F_l from right to left and successively adding the paths that are not yet in F_r to F_r such that the ordering scheme is respected. This way, adding a path p to one of its prefixes (both viewed as shorthands for the respective lists) yields p again. Therefore, we may, for clarity, add prefixes to a sum without changing the FSF that is being denoted. For example, the previous sum is equivalent to

$$\begin{aligned}\text{BASE} = \; & \textit{Base}::util.Calc.top \oplus \textit{Base}::util.Calc.clear \\ & \oplus \textit{Base}::util.Calc.enter \oplus \textit{Base}::util.Calc.e2 \\ & \oplus \textit{Base}::util.Calc.e1 \oplus \textit{Base}::util.Calc.e0 \\ & \oplus \textit{Base}::util.Calc \oplus \textit{Base}::util\end{aligned}$$

Note that the common prefixes of the paths at the beginning of the list have been factored out and are mentioned only once. From now on, we will predominantly use this form of representation.
Since each atomic introduction preserves the feature path name of corresponding FSF node, the source feature of an introduction remains known during the manipulation of an algebraic expression, e.g., BASE in $\textit{Base}::util.Calc$. This allows us to convert each algebraic expression (containing a sum of introductions with prefixes) straightforwardly back to an FSF, either to the original FSF or to a new composed FSF. When converting an introduction sum to a composed FSF, it is associated with a new (composed) feature. Two atomic introductions with the same path name that belong to *different* features, are composed via superimposition, as explained informally in Section 2.3. Of two atomic introductions with the same fully qualified name that belong to the *same* feature only the rightmost is effective, e.g.,

$$\textit{Foo}::i \oplus Bar::j \oplus \textit{Foo}::i \;=\; Bar::j \oplus \textit{Foo}::i$$

but

$$\textit{BarFoo}::i \oplus \textit{Bar}::j \oplus \textit{Foo}::i \;\neq\; \textit{Bar}::j \oplus \textit{Foo}::i$$

For example, the introduction sum representing the superimposition of Figure 2 is

$$\begin{aligned}\text{ADDBASE} = \; & \textit{Add}::util.Calc.add \oplus \textit{Add}::util.Calc \oplus \textit{Add}::util \\ & \oplus \textit{Base}::util.Calc.top \oplus \textit{Base}::util.Calc.clear \\ & \oplus \textit{Base}::util.Calc.enter \oplus \textit{Base}::util.Calc.e2 \\ & \oplus \textit{Base}::util.Calc.e1 \oplus \textit{Base}::util.Calc.e0 \\ & \oplus \textit{Base}::util.Calc \oplus \textit{Base}::util\end{aligned}$$

This sum represents a composed FSF consisting of a package `util` with a class `Calc` that contains four methods (including `add`) and three fields.
As a second example, the introduction sum that represents the superimposition of Figure 4 is

$$\begin{aligned}\textsc{CountBase} = \; & \textit{Count}::util.Calc.clear \oplus \textit{Count}::util.Calc.enter \\ & \oplus \textit{Count}::util.Calc.count \oplus \textit{Count}::util.Calc \oplus \textit{Count}::util \\ & \oplus \textit{Base}::util.Calc.top \oplus \textit{Base}::util.Calc.clear \\ & \oplus \textit{Base}::util.Calc.enter \oplus \textit{Base}::util.Calc.e2 \\ & \oplus \textit{Base}::util.Calc.e1 \oplus \textit{Base}::util.Calc.e0 \\ & \oplus \textit{Base}::util.Calc \oplus \textit{Base}::util\end{aligned}$$

This example differs from the previous one since it involves terminal superimposition. The sum represents a composed FSF consisting of a package `util` with a class `Calc` that contains three methods and three fields, and the bodies of the two `enter` methods are merged using a composition rule for method bodies (similarly for `clear`).

3.1.3 Axiomatization

We assume an abstract set I of *introductions*. Introduction sum $\oplus$ over I is assumed to induce an *idempotent monoid* $(I, \oplus, \mathbf{0})$, where $\mathbf{0} \in I$: [10]

Axiom 1 (Associativity of $\oplus$)

$$(i_3 \oplus i_2) \oplus i_1 = i_3 \oplus (i_2 \oplus i_1)$$

Introduction sum is associative like FSF superimposition is associative. This applies for terminals and non-terminals.

Axiom 2 (Neutrality of $\mathbf{0}$)

$$\mathbf{0} \oplus i = i \oplus \mathbf{0} = i$$

$\mathbf{0}$ is the empty introduction, i.e., an FSF without nodes.

Axiom 3 (Distant idempotence of $\oplus$)

$$i_2 \oplus i_1 \oplus i_2 = i_1 \oplus i_2$$

Only the rightmost occurrence of an introduction i is effective in a sum, because it has been introduced first. That is, duplicates of i have no effect, as pointed out at the end of Section 2.1.

Lemma 1 (Direct idempotence of $\oplus$)

$$i \oplus i = i$$

[10] All standard definitions of algebraic structures and properties are according to Hebisch and Weinert [37].

For $i_1 = \mathbf{0}$, direct idempotence follows from distant idempotence.

3.1.4 Discussion

Associativity is crucial for the practicality of the algebra and for the languages and tools that implement feature composition on the basis of the algebra. It ensures that the history of the introduction of structural elements is irrelevant. That is, associativity guarantees a pleasant and useful flexibility of feature composition. Although some languages for feature composition lack this property, e.g., AspectJ [52], it is definitely desirable, so that we choose to include a corresponding axiom in our algebra.

A further pleasant property of feature composition would be commutativity. It would ensure that all composition orders (permutations) of a set of features are behaviorally equivalent. However, this is not guaranteed in most languages and tools for feature composition – so we cannot require it. For example, while in Java the addition of new packages, classes, interfaces, and methods, etc. is commutative, the overriding of methods is not. Similar examples can be found for other languages. We decided not to make the commutativity of introduction sum a general postulate of our algebra, although it may hold in some cases.

Another issue is the relevance or irrelevance of distant (and direct) idempotence. In Section 2.1, we have motivated distant idempotence (i.e., the fact that the repeated addition of a fixed feature has no effect) with simplicity. However, there is a further reason: typically, languages and tools for feature composition have the idempotence property. This is easy to see since, usually, a single feature cannot introduce a member twice to a single class, e.g., a feature A can add a field `f` to a class `C` only once. Of course, one can imagine allowing multiple instances of a feature but, in that case, the problem of idempotence is relegated to the instances, i.e., feature instance composition is distantly idempotent. We refrain from distinguishing between different instances of a single feature since most languages and tools for feature composition do so as well and because nothing is gained.

3.1.5 Consequences of Distant Idempotence

Distant idempotence has some interesting consequences. It allows us to define an introduction inclusion relation:

Definition 2 (Introduction inclusion $\leq$)

$$i_2 \leq i_1 \quad \Leftrightarrow \quad i_2 \oplus i_1 = i_1$$

This means that all atomic introductions of i_2 are also present in i_1.[11] This relation is closely connected to the subtype relation in the Deep calculus [40].

From the definition we obtain the two following laws.

[11] This particular definition, rather than the symmetric $i_2 \leq i_1 \Leftrightarrow i_1 \oplus i_2 = i_1$ has been chosen, since it reflects our idea that, in $i_2 \oplus i_1$, the left introduction i_2 is superimposed onto i_1.

Lemma 2 (Reflexivity of $\leq$)

$$i \leq i$$

Lemma 3 (Transitivity of $\leq$)

$$i_3 \leq i_2 \wedge i_2 \leq i_1 \quad \Rightarrow \quad i_3 \leq i_1$$

Mathematically, a reflexive and transitive relation is known as a *preorder*. In preorders, least/greatest elements are defined as usual, but need not be unique.

Lemma 4 (Least element $\mathbf{0}$)

$$\mathbf{0} \leq i$$

Lemma 5 (Least element $\mathbf{0}$ is unique)

$$i \leq \mathbf{0} \quad \Rightarrow \quad i = \mathbf{0}$$

Lemma 6 (Upper bounds)

$$i_2 \leq i_2 \oplus i_1 \quad \text{and} \quad i_1 \leq i_2 \oplus i_1$$

In fact, by the definition of introduction inclusion, distant idempotence is equivalent to the property on the right.
The sum of two elements is even a least upper bound.

Lemma 7 (Least upper bound)

$$i_1 \leq i \wedge i_2 \leq i \quad \Rightarrow \quad i_1 \oplus i_2 \leq i$$

Based on the inclusion relation, we can define an equivalence relation between introduction sums:

Definition 3 (Introduction equivalence $\sim$)

$$i_2 \sim i_1 \quad \Leftrightarrow \quad i_2 \leq i_1 \wedge i_1 \leq i_2$$

This means that two sums of introductions are considered equivalent if they have the same atomic introductions.
Introduction inclusion and equivalence are useful for the comparison of different algebraic expressions and, consequently, of different programs composed of features [4, 5]. Moreover, least and greatest elements are unique up to introduction equivalence.

Lemma 8 (Quasi-commutativity w.r.t. $\sim$)

$$i_2 \oplus i_1 \sim i_1 \oplus i_2$$

In Appendix A.1, we provide an implementation in Prover9, with which the proofs of Lemmas 1–8 can be generated automatically.

3.2 Modifications

Besides superimposition, also other techniques for feature composition have been proposed, most notably *composition by quantification and weaving* [12, 25, 57]. The idea is that, when expressing the changes that a feature causes to another feature, we specify the points at which the two features are supposed to be composed. This idea has been explored in depth in work on multi-dimensional separation of concerns [68], aspect-oriented programming [53], adaptive programming [49], and strategic programming [48]. The process of determining the location of the composition is called *quantification* and the process of effecting the changes is called *weaving* [31]. In the remainder, we distinguish between two approaches of composition: *composition by superimposition* and *composition by quantification and weaving*. Our final definition of feature composition incorporates both (see Sec. 4).
In order to model composition by quantification and weaving, we introduce the concept of a modification. A *modification* consists of two parts:

(1) A characterization of the FSF nodes at which it will affect a feature during composition.
(2) A specification of how it affects these nodes.

In the context of our concrete model, a modification is performed by an FSF traversal that, at the same time, determines the nodes to be modified and applies the necessary changes to them. We take a declarative view of composition by quantification and weaving. Querying an FSF can yield more than one node at a time. This allows us to specify the modification of an entire set of nodes at once without having to reiterate it for every set member.
In the practice of programming, many different forms of changes of nodes are possible, e.g.,

(1) add a new child node (e.g., add a method to a class);
(2) alter a terminal's content (e.g., override and extend a method's body);
(3) delete a node (e.g., remove a method);
(4) rename a node (e.g., rename a class);
(5) alter a node's type (e.g., transform an interface to a class).

We concentrate on definitions of change that add new children or modify a terminal's content, i.e., the first two of the options above. The last three options are interesting as well, but are not common practice in feature-oriented languages and tools. Typically, these kinds of changes are supported in refactoring tools and transformation systems that have other properties than languages and tools for feature composition [34]. Common feature-oriented languages omit the last three options not without reason. On the one hand, their omittance simplifies the implementation of the language, especially, type checking and, on the other hand, it keeps the language simple, so that the programmer can comprehend the intention of the program better [47]. Next, allowing deletion of a feature may introduce inconsistency if parts of a program rely on its presence. Furthermore, if we included the last three options, we could not attain certain important properties of feature composition such as associativity.
Composition by superimposition and composition by quantification and weaving are very similar. But quantification enables us to address parts of a program more

specifically than superimposition, which always is applied at a root of an FSF. That is, we can locate the places of change by a pattern on FSFs (e.g., "all methods in package `util` whose name begins with `set`") that the structural elements of a program have to satisfy in order to be affected by a modification. For example, a feature could add a new field to *every* Java class of a package, regardless of the name of the class. Naturally, applying such a modification to different programs may lead to different results. Nevertheless, once the points of change are known, the two kinds of composition become equivalent. That is, once we decide on a program, we can find an equivalent introduction sum for every modification. Figure 6 illustrates this similarity.

We have observed the duality of composition by superimposition and composition by quantification and weaving before, but at the level of concrete programming techniques [12]. The feature algebra makes it explicit at a more abstract level.

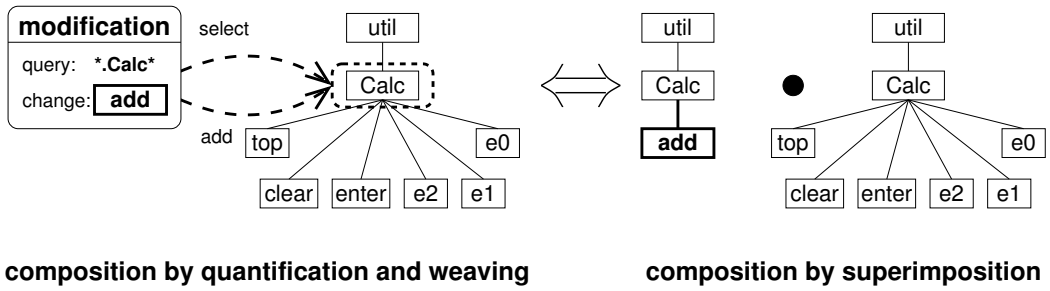

Fig. 6. Two dual notions of composition.

3.2.1 Semantics of Modification

In our concrete FSF model, a modification m consists of a query q, which selects a subset of the atomic introductions of an introduction sum, and a definition of change c that will be used to effect the desired changes:

$$m = (q, c)$$

A simple query can be represented by a path expression in which the node names may contain wildcards. [12] For example, the query q with the path expression $util.Calc.*$, applied to our example, would return the sum of all introductions that are members of the class `Calc`. However, we do not explore further how to express query expressions concretely. This is an issue of tool support that is being addressed in ongoing work [25].

For simplicity, we make the steps of querying and applying the changes transparent. We assume an abstract set M of *modifications* and define an operator *modification application* over M and the set I of introductions:

[12] In practice, queries with regular expressions or queries over types, or tools like XPath or XQuery might be useful.

Definition 4 (Modification application $\odot$)

$$\odot : M \times I \to I$$

A modification applied to an atomic introduction returns either that introduction again or the changed one:

$$m \odot i = (q, c) \odot i = \begin{cases} c(i), & \text{when } i \text{ is matched by } q \\ i, & \text{otherwise} \end{cases}$$

In terms of FSF expressions, our example of Figure 6 can be written as follows. Assume we have a base feature BASE:

$$\begin{aligned} \text{BASE} = \; & \textit{Base} :: util.Calc.top \oplus \textit{Base} :: util.Calc.clear \\ & \oplus \textit{Base} :: util.Calc.enter \oplus \textit{Base} :: util.Calc.e2 \\ & \oplus \textit{Base} :: util.Calc.e1 \oplus \textit{Base} :: util.Calc.e0 \\ & \oplus \textit{Base} :: util.Calc \oplus \textit{Base} :: util \end{aligned}$$

Furthermore, assume that we have a modification m_{Add} that adds method `add` to any class whose name begins with `Calc`: [13]

$$m_{Add} = (*.Calc*, c_{add})$$

Applying modification m_{Add} to BASE yields the new program:

$$\begin{aligned} m_{Add} \odot \text{BASE} = \; & \textit{Add} :: util.Calc.add \oplus \textit{Add} :: util.Calc \oplus \textit{Add} :: util \\ & \oplus \textit{Base} :: util.Calc.top \oplus \textit{Base} :: util.Calc.clear \\ & \oplus \textit{Base} :: util.Calc.enter \oplus \textit{Base} :: util.Calc.e2 \\ & \oplus \textit{Base} :: util.Calc.e1 \oplus \textit{Base} :: util.Calc.e0 \\ & \oplus \textit{Base} :: util.Calc \oplus \textit{Base} :: util \end{aligned}$$

3.2.2 Axiomatization

A modification is applied to a sum of introductions by applying it to each introduction in turn and summing the results:

Axiom 4 (Distributivity of $\odot$ over $\oplus$)

$$m \odot (i_2 \oplus i_1) = (m \odot i_2) \oplus (m \odot i_1)$$

The successive application of changes of a modification to an introduction sum implies the left distributivity of $\odot$ over $\oplus$. The distributivity law captures precisely the intention of quantification, i.e., that a modification visits each node of an FSF.

[13] As mentioned before, the representation of queries and changes is a matter of tool support and is not formalized in the algebra but illustrated only for presentation purposes; c_{add} represents the definition of change that adds method `add`.

Now it should become clear why we use only prefix-closed lists: A prefix-closed list contains all intermediate nodes in the structure of a software artifact, e.g., packages and classes. This circumstance allows us to define modifications that change also these inner nodes and not only the leaves of FSTs.
We define two modifications m_1 and m_2 to be *equivalent* if they behave identically on all introductions, i.e., if $m_1 \odot i = m_2 \odot i$ for all i. In the following, we write M also for the set of equivalence classes of modifications and $\odot$ for the corresponding induced operation on them.

Axiom 5 (Identical modification $\mathbf{1}$)

$$\mathbf{1} \odot i = i$$

The empty modification is denoted by $\mathbf{1} \in M$. It is the modification (actually an equivalence class) that does not change any introduction.

3.2.3 Modification Product

We define an operator *modification product* over the set M of modifications:

Definition 5 (Modification product $\circledast$)

$$\circledast : M \times M \to M$$

Applying a product of two modifications to an introduction means to apply first the right factor and then the left factor to the result:

Axiom 6 (Iterative application $\circledast$)

$$(m_2 \circledast m_1) \odot i = m_2 \odot (m_1 \odot i)$$

Note that the left modification may affect the extensions made by the right modification, i.e., m_2 may affect the introductions added by m_1.

3.2.4 Algebraic Structure

We begin with the algebraic properties of modification product. The discussion of modification application follows in Section 3.3. In Appendix A.2, we provide Prover9 scripts by which proofs of the lemmas regarding modifications can be generated automatically.
Up to modification equivalence, the modification product induces again a *monoid* $(M, \circledast, \mathbf{1})$:

Axiom 7 (Identity $\mathbf{1}$)

$$\mathbf{1} \circledast m = m \circledast \mathbf{1} = m$$

Lemma 9 (Associativity of $\circledast$)

$$(m_3 \circledast m_2) \circledast m_1 = m_3 \circledast (m_2 \circledast m_1)$$

As for introduction sum, for modification product, we do not require commutativity since it would limit the power of modification and is typically not seen in languages supporting modification in practice. In Section 4.4.2, we discuss commutativity as an optional axiom.

3.3 Introductions and Modifications in Concert

In order to describe feature composition, our algebra uses the operation of modification application $\odot$. Modification application integrates our two kinds of algebraic structure: $(I, \oplus, \mathbf{0})$ and $(M, \circledast, \mathbf{1})$.
A notable property of $(I, \oplus, \mathbf{0})$ is that it induces a *semimodule over the monoid* $(M, \circledast, \mathbf{1})$. This is due to the definition of the operation of modification application and the distributive and associative laws (Axiom 4 and Lemma 9).
A semimodule over a monoid is related to a vector space but weaker (modification application plays the role of the scalar product) [37]. In a vector space, the additive and multiplicative operations are commutative and there are inverse elements with respect to addition and multiplication. Nevertheless, the properties of a semimodule guarantee a pleasant and useful flexibility of feature composition, which is manifested in the associativity and distributivity laws.

4 The Quark Model

So far, we have introduced two sets (I and M) and three operations ($\oplus : I \times I \to I$, $\circledast : M \times M \to M$, and $\odot : M \times I \to I$) for composition; basic features have been identified with FSFs whose abstract counterpart is the introduction. Now, we integrate them into a notion of *full features* or *quarks* that involve both introductions and modifications.
The goal of the quark model is to provide a concise notation and formalism to represent features consisting of introductions and modifications. There are several options to integrate the three operations and, interestingly, they map to different choices in programming language design, as we will explain. We begin with a formalization of different forms of quarks and discuss their mandatory and optional algebraic properties and their relationship to existing and future programming languages.

4.1 Simple Quarks

For the purpose of integrating introductions and modifications, we introduce the *quark model*.[14] In a first step, we define a *quark* as a pair that represents a feature

[14] The idea and name of the quark model are due to Don Batory. Subsequently, the model was developed further in cooperation with us. The term 'quark' was chosen as an analogy to the physical particles in quantum chromodynamics. Originally, quarks have been considered to be atomic, but newer theories, e.g., preon or string theory, predict a further substructure.

consisting of an introduction and a modification:

$$f = \langle i, m \rangle$$

The introduction i of feature f represents an FSF; m is the modification that feature f applies.
In a second step, we introduce an operator for *quark composition* over the set Q of quarks:

Definition 6 (Quark composition ⧫)

$$\blacklozenge : Q \times Q \to Q$$

The composition of two quarks results again in a quark, and there are several options of how the elements of the two input quarks are combined to yield the output quark, as we will explain in Section 4.2.
A basic feature is represented in the quark model as a pair $\langle i, \mathbf{1} \rangle$ where $\mathbf{1}$ is the empty modification. The empty feature is represented as the pair of the empty introduction and the empty modification: $\langle \mathbf{0}, \mathbf{1} \rangle$. With the quark composition ⧫ to be discussed in the next subsection, the application of quark f to introduction i is defined as the composition $f \blacklozenge \langle i, \mathbf{1} \rangle$.

4.2 Local and Global Quarks

There are two options of applying modifications while composing a sequence of features:

(1) The modifications of a feature are applied to and may affect only the features that have been composed before, i.e., that are to the right in the sequence. In this case, we speak of *local modifications*.
(2) The modifications of a feature are applied to and may affect all features of the sequence. In this case, we speak of *global modifications*.

The distinction between local and global modifications is motivated by the different implementations of modification in languages and tools for feature composition. On the one hand, languages and tools that support local modification usually interpret a feature composition as a stepwise process in which features on the right are developed and composed earlier than features to their left [17]. The intention in disallowing that features affect features to be developed and composed subsequently is to reduce program complexity and prevent inadvertent interactions [7, 52].
On the other hand, there are languages and tools for feature composition that do not support stepwise development. With them, features are composed in one step and can affect each other without limitations. AspectJ is such a language. This is the reason why we include both local and global modification in the quark model. Quarks that apply modifications locally are called *local quarks* and quarks that apply modifications globally are called *global quarks*.
Quarks with local modifications l_i are composed as follows:

Axiom 8 (Local quark composition)

$$f_2 \blacklozenge f_1 = \langle i_2, l_2 \rangle \blacklozenge \langle i_1, l_1 \rangle = \langle \underline{i_2 \oplus (l_2 \odot i_1)}, l_2 \circledast l_1 \rangle$$

Local modifications can affect only introductions of features that have been constructed earlier. In our example, l_2 affects only i_1 and not i_2 (underlined term). For a composition $f_n \blacklozenge (f_{n-1} \blacklozenge (\ldots (f_2 \blacklozenge f_1)))$ of n features, a modification l_i of feature f_i can affect only the introductions of a feature f_j if $i > j$.
In our calculator example, suppose we would like to add a new class `Counter` to package `util` and a new field `count` to every class in package `util`. We can use a local quark combining introduction sum and modification application to achieve the desired composition result. Assume a base feature QUARK_1 in the form of a quark that consists of an introduction sum and the empty modification:

$$\begin{aligned}\text{QUARK}_1 = \langle\, (\, &\textit{Base}::util.Calc.top \oplus \textit{Base}::util.Calc.clear \\ &\oplus \textit{Base}::util.Calc.enter \oplus \textit{Base}::util.Calc.e2 \\ &\oplus \textit{Base}::util.Calc.e1 \oplus \textit{Base}::util.Calc.e0 \\ &\oplus \textit{Base}::util.Calc \oplus \textit{Base}::util\,), \mathbf{1} \rangle\end{aligned}$$

Furthermore, assume a feature QUARK_2 that adds class `Counter` by introduction sum and injects field `count` by modification application:

$$\text{QUARK}_2 = \langle\, (\, \textit{Counter}::util.Counter \oplus \textit{Counter}::util\,), (util.*, count)\,\rangle$$

Composing the two quarks yields the desired result:

$$\begin{aligned}\text{QUARK}_2 \blacklozenge \text{QUARK}_1 = \langle\, (\, &\textit{Counter}::util.Counter \oplus \textit{Counter}::util \\ &\oplus \textit{Counter}::util.Calc.count \\ &\oplus \textit{Base}::util.Calc.top \oplus \textit{Base}::util.Calc.clear \\ &\oplus \textit{Base}::util.Calc.enter \oplus \textit{Base}::util.Calc.e2 \\ &\oplus \textit{Base}::util.Calc.e1 \oplus \textit{Base}::util.Calc.e0 \\ &\oplus \textit{Base}::util.Calc \oplus \textit{Base}::util\,), (util.*, c_{count})\rangle\end{aligned}$$

The important point to understand is that the local modification of QUARK_2 does not add field `count` to class `Counter`, but only to class `Calc`. A local modification of a feature cannot affect program elements of the same feature and elements of features that are composed subsequently (i.e., that are to the left in the feature composition expression). With global modification, this is different.
Quarks with global modifications g_i are applied as follows:

Axiom 9 (Global quark composition)

$$f_2 \blacklozenge f_1 = \langle i_2, g_2 \rangle \blacklozenge \langle i_1, g_1 \rangle = \langle \underline{(g_2 \circledast g_1) \odot (i_2 \oplus i_1)}, g_2 \circledast g_1 \rangle$$

A global modification can affect also the introduction that is just being added during the composition. In our example, both, g_2 and g_1 may affect i_2 and i_1 (underlined

terms). For a composition $f_n \blacklozenge (f_{n-1} \blacklozenge (\ldots (f_2 \blacklozenge f_1)))$ of n features, a modification g_i of feature f_i can affect the introductions of a feature f_j for any pair (i, j).
In our calculator example, suppose we compose again QUARK_1 and QUARK_2 but now with global quark composition (for disambiguation we use QUARK'_2, which is identical to QUARK_2 except that its modification is a global modification). The result differs from local quark composition in that the modification of QUARK'_2 is now global and adds field `count` not only to class `Calc` but also to class `Counter`:

$$\begin{aligned}\text{QUARK}'_2 \blacklozenge \text{QUARK}_1 = \langle\, (\, &\textit{Counter}::util.Counter.count \\ &\oplus \textit{Counter}::util.Counter \oplus \textit{Counter}::util \\ &\oplus \textit{Counter}::util.Calc.count \\ &\oplus \textit{Base}::util.Calc.top \oplus \textit{Base}::util.Calc.clear \\ &\oplus \textit{Base}::util.Calc.enter \oplus \textit{Base}::util.Calc.e2 \\ &\oplus \textit{Base}::util.Calc.e1 \oplus \textit{Base}::util.Calc.e0 \\ &\oplus \textit{Base}::util.Calc \oplus \textit{Base}::util\,), (util.*, c_{\textit{count}})\rangle\end{aligned}$$

4.3 Full Quarks

Given the alternatives of local and global modification discussed in Section 4.2, we define a *full quark* as a triple consisting of a global modification, an introduction, and a local modification:

$$f = \langle g, i, l \rangle$$

The composition of two quarks results in a quark that is constructed by the following rules. The new introduction part of the quark is constructed using modification application and introduction sum, while the new modification parts result by modification product.
Axioms 8 and 9 lead to the following composition scheme results:

Axiom 10 (Full quark composition)

$$\begin{aligned} f_2 \blacklozenge f_1 &= \langle g_2, i_2, l_2 \rangle \blacklozenge \langle g_1, i_1, l_1 \rangle \\ &= \langle g_2 \circledast g_1, (g_2 \circledast g_1) \odot (i_2 \oplus (l_2 \odot i_1)), l_2 \circledast l_1 \rangle \end{aligned}$$

Note that Axioms 8 and 9 are special cases of Axiom 10 with $g_i = \mathbf{1}$ and $l_i = \mathbf{1}$, respectively, and projecting onto the non-$\mathbf{1}$ components.
Moreover, an introduction i can be embedded into a full quark of the form $\langle \mathbf{1}, i, \mathbf{1} \rangle$. For such quarks, our axioms yield:

$$\langle \mathbf{1}, i_2, \mathbf{1} \rangle \blacklozenge \langle \mathbf{1}, i_1, \mathbf{1} \rangle = \langle \mathbf{1}, i_2 \oplus i_1, \mathbf{1} \rangle$$

4.4 Algebraic Structure

We begin with an examination of the properties of the composition of simple, local, global, and full quarks assuming only the axioms given so far; we call the

set of these axioms the *standard configuration*. Then, we discuss a set of further, optional axioms, which we call henceforth *optional configurations*, and which limit the expressibility of modifications in order to improve the flexibility of quark composition. In Appendix A.3, we provide the Prover9 scripts by which proofs of the lemmas and theorems regarding quarks in the standard and optional configurations can be generated automatically.

4.4.1 Standard Configuration

In the column "standard configuration" of Table 1, we list the properties of the composition of simple, local, global, and full quarks that follow from the axioms that we have postulated so far. In particular, we are interested in associativity, identity, and direct and distant idempotence of quark composition.
Table 1 reveals that the composition of simple quarks is most flexible. It is associative, has the identity element $\langle \mathbf{1}, \mathbf{0}, \mathbf{1} \rangle$, and it is directly and distantly idempotent. The composition of local quarks is associative and $\langle \mathbf{1}, \mathbf{0}, \mathbf{1} \rangle$ is the identity element; but it is not idempotent. The composition of global and full quarks does not have any of the above properties. This is due to the fact, that in a composition of full quarks with $\langle \mathbf{1}, \mathbf{0}, \mathbf{1} \rangle$, the global modifications are always applied.

4.4.2 Optional Configurations

In order to increase the flexibility of the composition of quarks, we have experimented with two optional axioms that limit the expressiveness of modifications. First, we explain the optional axioms and discuss subsequently which of their combinations cause which properties of quark composition, in particular, associativity, identity, and direct and distant idempotence.

Option 1 (Distant idempotence of $\circledast$)

$$(m_2 \circledast m_1 \circledast m_2) \odot i = (m_2 \circledast m_1) \odot i$$

The first optional axiom postulates the distant idempotence of modification product. That is, like with introduction sum, the repeated application of a single modification has no effect on the program.

Option 2 (Commutativity of $\circledast$)

$$(m_2 \circledast m_1) \odot i = (m_1 \circledast m_2) \odot i$$

The second optional axiom postulates commutativity of modification product. That is, the application order of modifications has no impact on the program behavior.
Table 1 shows how the two optional axioms affect the associativity, identity, and direct and distant idempotence of quark composition – compared to the standard configuration. Postulating distant idempotence of modification product makes global quark composition distantly idempotent. The combination of distant idempotence and commutativity makes global quark composition associative.

configuration	associativity				identity				direct idempotence				distant idempotence			
	simple	local	global	full	simple	local	global	full	simple	local	global	full	simple	local	global	full
standard	√	√			√	√			√				√			
distant idempotence	√	√			√	√			√				√		√	
commutativity	√	√			√	√			√				√			
distant idempotence & commutativity	√	√	√		√	√			√				√		√	

Table 1. Overview of the algebraic properties of quark composition assuming different optional axioms. The presence of a check mark indicates that the respective property has been proved. The absence of a check mark indicates that there is a counterexample to the respective property.

5 Discussion

We have presented a model of FOSD in which features are represented as FSFs and feature composition is expressed by tree superimposition and tree walks. This reflects the state of the art in programming languages and composition models that favor superimposition [35, 61, 64, 66, 73], quantification and weaving [18, 31, 46, 53], or a combination of both [12, 57, 68]. Our algebra describes precisely what the properties of the composition models are and how they can be integrated. This is not obvious from previous work, which has been based on specific implementations [12]. Feature algebra makes the similarity between composition by superimposition and composition by quantification and weaving explicit.

The quark model provides a compact and concise notation for feature composition comprising introduction and modification. A notable property of quarks is their compositionality. That is, the composition of two quarks results again in a quark. While this property is straightforward from the algebraic viewpoint, many contemporary programming languages and tools that support modification do not have it, e.g., aspect-oriented languages such as AspectJ, as discussed by Lopez-Herrejon et al. [52]. Our algebra and quark model shows how to constrain and integrate introduction and modification to attain a proper degree of compositionality. We believe that this is a valuable input for language and tool design.

The quark model allows us to choose between local and global modification. This variability is motivated by the presence of different lines of research. Local modification follows the paradigm of functions or transformations in stepwise development [17, 18]. Global modification by quantification is motivated by work on metaobject protocols and aspect-oriented programming. Again, feature algebra describes precisely the differences between both approaches and, provided we impose some disciplining restrictions, shows a way to integrate them.

A notable result is that our feature algebra forms a semimodule over a monoid, which is a weaker form of a vector space. The properties of this algebraic structure suggest that our decisions regarding the semantics of introductions and modifications and their operations are not arbitrary. With the standard configuration of our algebra, we achieve a high flexibility in feature composition, which is manifested in the associativity and distributivity laws. Specifically, we found that the following properties must hold to achieve minimal composition flexibility of basic features:

- Introduction sum must be distantly idempotent and associative.
- Modification product must satisfy the associativity axiom.
- Modification application must distribute over introduction sum.

In the standard configuration, the composition of simple quarks is most flexible. Local quark composition is at least associative and an identity element can be found. The composition of global and full quarks is least flexible.

In order to achieve a minimal flexibility of the composition of full features, additional properties must hold:

- Modification product must be distantly idempotent.
- Modification product must be commutative.

In order to guarantee these additional properties, we have postulated two optional

axioms (distant idempotence and commutativity of modifications) and found that, with combinations (optional configurations) of the three, we can attain associativity for all kinds of quark composition and distant idempotence for global quark composition. However, while this is a pleasing result, we still need to comment on the limitations that the optional axioms impose on the programming languages that support modification application.

First, distant idempotence for modifications can be implemented easily in a programming language. For example, the AspectJ compiler simply has to keep track of all pieces of advice (modifications) that are being applied and to remove the duplicates. With distantly idempotent modification application in AspectJ, the composition of AspectJ quarks becomes distantly idempotent, too.

Second, commutativity is more difficult to attain. It has been shown that two modifications are commutative when they do not refer to each other and when the changes they apply do not overlap [7]. Requiring this property in a language like AspectJ would limit the expressiveness of modifications dramatically. In order to ensure that two modifications are indeed commutative, an analysis of the entire program would be necessary. A way out of this dilemma is to exploit a property that we call *pseudo-commutativity* [7]:

$$m_2 \odot (m_1 \odot i) = m_1' \odot (m_2' \odot i)$$

The modifications m_1 and m_1' are not the same but implement similar changes to a program that result in an equivalent behavior – ditto for m_2 and m_2'. In prior work, we have shown that, for two modifications written in AspectJ, two pseudo-commutative counterparts can always be found whose swapped applications result in a behaviorally equivalent program [7]. Furthermore, we have shown that this process can be automated, thus, achieving a flexibility similar to the one provided by commutativity.

Our analysis has demonstrated that there is a trade-off between expressiveness and flexibility of feature composition. One advantage of an algebraic approach is that we can evaluate the effects of our own and possible alternative decisions directly by examining the properties of the resulting algebra, instead of implementing the mechanisms first in a programming language.

6 Perspectives

6.1 Higher-Order Modifications

A fundamental asymmetry in our algebra is the distinction between introductions and modifications. Since a modification applies to an introduction sum, it acts at a metalevel. In the literature, the addition of further levels on top of modifications has been proposed [8, 22, 70, 72]. In our algebra, this would require the inclusion of modifications that modify modifications, i.e., *higher-order modifications*. Higher-order modifications quantify over sums of lower-order modifications, much like modifications quantify over sums of introductions. Our introductions can be viewed

as modifications of order 0, modifications that apply to introductions have order 1, modifications that apply to modifications of order 1 are of order 2, and so on. In this view, introduction and modification become the same concept, in which a modification of a higher order affects a modification of the next lower order. This terminates when order 0, i.e., the level of introductions, has been reached. The operations of the algebra are then defined by induction, forming a hierarchy of one level per order; introduction sum and modification product become effectively the same operation (we use the operator $\circledast$ for consistency) with axioms analogous to Axiom 6:

$$
\begin{array}{l|l}
\circledast^{(0)} \quad : M^{(0)} \times M^{(0)} \to M^{(0)} & \\
\circledast^{(1)} \quad : M^{(1)} \times M^{(1)} \to M^{(1)} & \odot^{(1)} \quad : M^{(1)} \times M^{(0)} \to M^{(0)} \\
\circledast^{(2)} \quad : M^{(2)} \times M^{(2)} \to M^{(2)} & \odot^{(2)} \quad : M^{(2)} \times M^{(1)} \to M^{(1)} \\
\qquad\qquad \vdots & \qquad\qquad \vdots \\
\circledast^{(n-1)} : M^{(n-1)} \times M^{(n-1)} \to M^{(n-1)} & \odot^{(n-1)} : M^{(n-1)} \times M^{(n-2)} \to M^{(n-2)} \\
\circledast^{(n)} \quad : M^{(n)} \times M^{(n)} \to M^{(n)} & \odot^{(n)} \quad : M^{(n)} \times M^{(n-1)} \to M^{(n-1)}
\end{array}
$$

6.2 Programming Language Design

Interestingly, different kinds of quarks can be used to describe different kinds of languages. In Table 2, we illustrate that there are indeed differences between languages and tools for feature composition that are reflected by their corresponding quarks. The algebra and the quark model are formal means to explore and discuss the similarities and differences without being distracted by language-specific details. For example, it is interesting to note that, although FeatureHouse, CaesarJ, and FeatureC++ support both introduction sum and modification, FeatureHouse applies the modification locally but CaesarJ and FeatureC++ globally. This fundamental difference has not been considered before, even though it may influence program comprehension and lead to errors [52]. It would be interesting to explore the effects on practical software engineering. The algebra and the quark model revealed this issue apart from the other differences between the languages (e.g., CaesarJ is based on Java, FeatureC++ is based on C++, and FeatureHouse is a cross-language tool). Furthermore, the quark model offers a perspective for future languages to support both local and global modification. To the best of our knowledge, so far, there is no such language and, hence, it remains an open issue what such a languages would look like. The algebra with its mandatory and optional axioms already describes their properties. From the viewpoint of language design, it is open how to represent global and local application in program text (e.g., by keywords or modifiers) and whether it is useful to allow programmers this way to protect certain program elements from being changed.

Together with the results of Table 1, we can infer the properties of contemporary programming languages with respect to feature composition. For example, both

quark	tool or language
$\langle \mathbf{1}, i, \mathbf{1} \rangle$	Jak [17], Classbox/J [19], Jiazzi [54]
$\langle \mathbf{1}, \mathbf{0}, l \rangle$	ARJ* [7]
$\langle g, \mathbf{0}, \mathbf{1} \rangle$	AspectJ* [45], AspectC++* [67]
$\langle \mathbf{1}, i, l \rangle$	FeatureHouse [10]
$\langle g, i, \mathbf{1} \rangle$	CaesarJ [14], FeatureC++ [11]
$\langle g, i, l \rangle$	—

* Although inter-type declarations of aspect-oriented languages are related to introductions, we do not regard them as mechanisms for superimposition.

Table 2
Correspondence of quark types and composition tools and languages.

CaesarJ and FeatureC++ support global quarks. The composition of global quarks is associative when the application of modifications is distantly idempotent and commutative. When we look at the implementations of CaesarJ and FeatureC++, especially at how modifications are implemented, we can infer whether the overall feature composition is associative or has other properties – both are not associative. Usually, feature composition involves complex mechanisms (introduction and local and global modification), so that inferring overall properties of feature composition from properties of individual mechanisms is a feasible approach to understand the complexity involved.
Finally, it is interesting to use the algebra and the quark model to guide the design of future programming languages. It is often the case that a feature-oriented or aspect-oriented language is designed without the properties of feature composition in mind. Usually, after contemporary languages have been designed, researchers have analyzed which properties the actual language implementations have. We propose a different approach. Before the designers develop a feature composition language or tool, they should figure out which properties are desirable and how expressive the language should be. Based on these findings, they should include proper language mechanisms. As stated before, there is a trade-off between flexibility of composition (e.g., commutative feature composition) and expressiveness (e.g., global and local modification application), which has to be taken into account during language design. Tables 1 and 2 help language designers to make their choices.

6.3 Tool Development

We expect that the algebra will eventually have an impact on practical software engineering. So far, we have illustrated its use in theoretical explorations of the essential properties of features and feature composition and the relationship of the mechanisms involved. By capturing the essence of features and feature composition in a concrete representation of features and accompanying tools, we can build a tool infrastructure that can be reused for features written in different languages and for operations that reason about features in different ways.

To illustrate the potential of our approach for practical software engineering, we have been developing a tool suite, called *FeatureHouse*, that performs feature composition for a wide variety of software artifacts written, e.g., in Java, C#, C, Haskell, JavaCC, or XML, following the laws of the algebra [10, 25]. [15] FeatureHouse combines introduction sum and local modification application, as defined by our algebra. Technically, it relies on FSFs. In order to implement introduction sum, two FSFs are superimposed, as described in Section 2.3. In order to implement local modification application, we have developed an XML-based language for defining queries and definitions of change [25]. Modification application is implemented by pattern matching on nodes and subtrees in an FSF. In order to apply modifications locally, the FSFs corresponding to different features are superimposed in steps, and modifications are applied to the corresponding intermediate results of the superimposition.
We have used FeatureHouse for the feature-based composition of several small to medium-sized software systems written in various languages involving introduction sum and modification product and application, and we have demonstrated that the approach to software composition formalized by the algebra is indeed language independent [10, 25]. Besides gaining the insight that feature composition is largely language-independent, we have learned much about the properties of languages for feature composition (e.g., that the choice of the level of granularity of feature composition is crucial and shall be investigated further [10]), which complements the insights gained by our formal treatment. We believe that exploring feature composition both from a practical and a theoretical point of view is an appropriate way to strike a balance between formal precision and practicality.

6.4 Architectural Metaprogramming

The big picture of our endeavor is that the feature algebra serves as a formal foundation for the vision of automatic feature-based program synthesis and architectural metaprogramming [15, 16, 29]. The idea is to scale metaprogramming and generative programming techniques such as feature composition from the level of source code snippets to the level of software architecture.

6.4.1 Synthesis

Treating programs as values of metaprograms that manipulate them requires a formal theory that describes what is allowed and what is not. For example, a program transformation that simply deletes all elements of an input program is certainly not very useful in program synthesis. Metaprograms that apply arbitrary changes are even more dangerous since they can introduce subtle errors. Besides composition, we envision further operations on features, e.g., feature interaction analysis, type checking, visualization, decomposition, and refactoring.
Let us illustrate the role of the algebra for automatic feature-based program synthesis and architectural metaprogramming by means of an example. Suppose we have two Java programs, PROG_A and PROG_B, and we would like to merge them in the course

[15] `http://www.fosd.de/fh`

of composing two code bases (sets of features):

$$\text{PROG}_A = \langle\, \mathbf{1}, (F_1 :: p.C.m \oplus\ F_1 :: p.C \oplus\ F_1 :: p), \mathbf{1} \,\rangle$$
$$\text{PROG}_B = \langle\, \mathbf{1}, (F_2 :: p.C.n \oplus\ F_2 :: p.C \oplus\ F_2 :: p), \mathbf{1} \,\rangle$$

By analyzing the algebraic expressions that represent the code bases, we can infer that, in this particular case, feature composition is commutative (there is no terminal superimposition and the order of classes, methods, and fields is irrelevant). This information allows a tool to choose the composition order freely. Also, the information that a modification is local allows a tool to infer which features are definitely not affected by it. Swapping the composition order is useful when implementing incremental composition tools, such that some complex transformations can be done early and other, simpler ones later [7]. Similarly, the information of an overlap of FSFs (e.g., both $Prog_A$ and $Prog_B$ have a class `C` in package `p`) can reveal possible structural interactions between features. The disjointness of two FSFs indicates their structural independence, which is a useful information for refactoring tools that aim at increasing variability [44].

6.4.2 Abstraction

Furthermore, the algebra serves as a formalism for abstraction, which is essential to architectural metaprogramming. In fact, the feature algebra can be a means for reasoning about and manipulating software architecture. Metaprograms operate on feature algebra expressions to synthesize programs. At every step, a tool maintains the connection between the architectural and the implementation level. It guarantees that the operations transform the structures from one to another consistent state. One can think of algebraic expression manipulation as a way of *symbolic* optimization. Features and their constituents have names in the corresponding algebraic expression (symbols). If we can assign a meaning to a name, we can choose to replace features with other features and to alter the order of features in order to optimize the program behavior and quality attributes such as performance and resource consumption.
For illustration, suppose we have a simple database system consisting of a storage manager (STORE), a simple hash-based index structure (HEAP), a transaction management (TRANS), and a statistics feature (STATS):

$$\begin{aligned} \text{DB} &= \text{STATS} \blacklozenge \text{TRANS} \blacklozenge \text{HEAP} \blacklozenge \text{STORE} \\ \text{STORE} &= \langle \ldots \rangle \\ \text{HEAP} &= \langle \ldots \rangle \\ \text{TRANS} &= \langle \ldots \rangle \\ \text{STATS} &= \langle \ldots \rangle \end{aligned}$$

Furthermore, suppose we know from our customer that the database system is going to process huge amounts of data. From this knowledge, we can optimize our algebraic expression symbolically by replacing the term HEAP with BTREE, in which the former represents a simple heap and the latter an efficient B-Tree index and storage structure. Information of the inner structure of HEAP and BTREE,

represented by the corresponding quarks, helps to identify potential interactions and options to rearrange the overall expression.
Another example is the mutual interaction of the features STATS and TRANS. The former collects statistics on the database system, including information on committed and interrupted transactions. If the transaction feature is not selected (e.g., in a single-user database), we have to alter the implementation of the statistics feature in order to let it operate properly without depending on transactions. A tool can infer the points of interaction from the corresponding algebraic expressions.
The point is that we can make all these decisions and manipulations purely based on the algebraic expressions and based on the knowledge that, for example, B-trees are an efficient means to store and access large amounts of data. Of course, in this case, the domain knowledge has to be represented in algebraic terms, too, which is an interesting point of further research.

7 Related Work

7.1 Authors' Previous Work

Lopez-Herrejon, Batory, and Lengauer model features as functions and feature composition as function composition [52]. They distinguish between introductions and advice, which correspond roughly to our introductions and modifications. However, in their work, there is no semantic model that defines precisely what introductions and advice are. In our feature algebra, we define introductions in terms of FSFs and modifications in terms of tree walks. This enables us to bridge the gap between algebra and implementation.
Apel and Hutchins have developed a calculus, called *gDeep*, for features and feature composition independently of a particular language [4, 5]. The calculus includes an operational semantics and type system. The advantage of an algebra-based approach is that we can reason on an even more abstract level about features than gDeep, which is useful for domain-specific optimization [17,20,29] and architectural metaprogramming [15] (see Sec. 8). The advantage of a calculus is that it allows us to formulate a general logic-based type system for FOSD, into which the type systems of the artifact languages can be plugged. We believe that both abstraction levels (calculus and algebra) are equally important in exploring the principles of feature composition. A promising starting point is to connect the introduction inclusion relation of the algebra with the feature subtype relation of gDeep.
Höfner, Khedri, and Möller have developed an algebra for expressing software and hardware variabilities in the form of features [38]. This has recently been extended [39] to express a limited form of feature interaction. However, their algebra does not consider the structure and implementation of features.
Liu, Batory, and Lengauer have developed a model of feature interaction [51], in which interactions are made explicit in feature expressions. Making interaction explicit in our model would not incur any further algebraic operators; both features and their interaction code would be represented as quarks.

With FeatureHouse [10] we have been developing tool support for feature composition that builds on the insights and laws of the algebra.
Finally, in the shorter conference version of the present paper [13], we have laid the foundation for this journal paper. In the conference version, we have defined the operations of introduction sum, modification application, and modification composition (which corresponds to modification product) as well as the quark model including local and global modifications. In addition to this earlier work, we provide here a complete formal axiomatization with automatically generated proofs of all lemmas and theorems as well as a discussion of optional axioms and their impact on the flexibility of feature composition.

7.2 Work of Others

Ramalingam and Reps have proposed the use of algebraic methods for reasoning about program modifications [62, 63]. There are several interesting connections between our work and this early work, e.g., distant idempotence corresponds to extended idempotence. However, our algebra incorporates novel developments in the field of programming such as feature-oriented programming, multi-dimensional separation of concerns, and aspect-oriented programming, that led to the concepts of local and global modifications as well as to the quark model.
There are some calculi that support feature-like structures and composition by superimposition [9, 28, 30, 32, 33, 40, 58]. These calculi are typically tailored to Java-like languages and emphasize the type system. Instead, our feature algebra enables reasoning about feature composition at a more abstract level. We emphasize the structure of features and their static composition, independently of a particular language or execution semantics.
The notion of a feature is close to that of a component. Bosch [23] noted the possibility of superimposing the internal structures of components for adaptation purposes. However, many contemporary component calculi focus on concurrency and process-theoretic issues as well as on connector and composition languages [1, 65, 75]. We use superimposition and quantification and weaving to control composition. The selection of a set of features is equivalent to a specification in a composition language; modifications are equivalent to connectors. Our FSF model emphasizes the static structure of features, which enables us to model not only code artifacts, but any kind of artifact that provides a sufficient structure.
In the field of aspect-oriented programming, several approaches have been proposed to model and formalize quantification and weaving mechanisms [41, 50, 53, 72, 74]. However, their focus is on the operational semantics and on typing. Our feature algebra provides a static view of quantification and weaving, which is useful for feature composition that involves the introduction of new structures.
Several languages support features and their composition by superimposition [11, 17, 19, 54, 59]. Our algebra is a theoretical foundation that underlies and unifies all these languages. It reveals the properties that a language must have in order to be feature-ready. Several languages exploit the synergistic potential of superimposition and quantification and weaving [11, 12, 57, 68]. The feature algebra allows us to

study their relationship and integration, independently of a specific language.
As mentioned previously, features are implemented not only by source code. Several tools support the feature-based composition of non-source code artifacts [2,3,17,26]. Our algebra is general enough to describe a feature containing non-code artifacts as long as their representations can be mapped to FSFs.

8 Conclusion

We have presented a model of feature-oriented software development (FOSD) in which features are represented as feature structure forests (FSFs) and feature composition is expressed by tree superimposition (introduction sum) and tree walks (modification application). This reflects the state of the art in programming languages and composition models for feature composition. Our algebra describes precisely what their properties are and how FOSD concepts of contemporary languages, such as aspects, collaborations, or refinements, can be integrated.
The quark model ties the different elements of the algebra together in a concise notation. We have discussed several alternative instances of the algebra and their implications on the properties of quark composition. Furthermore, we have shown that alternatives in the structure and composition of quarks correspond to alternatives in programming paradigms, tools, and languages. We have illustrated how the algebra could and should have an impact on the theory of FOSD, on practical feature-based software engineering, and on upcoming software development paradigms such as automatic feature-based program synthesis and architectural metaprogramming.

Acknowledgments

We thank Don Batory, Tony Hoare, Peter Höfner and the anonymous AMAST'08 and SCP reviewers for helpful comments. The work has been funded in parts by the German Research Foundation (DFG), project numbers AP 206/2-1 and MO 690/7-1.

References

[1] F. Achermann, O. Nierstrasz, A Calculus for Reasoning About Software Composition, Theoretical Computer Science 331 (2–3) (2005) 367–396.

[2] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, C. Lucena, Refactoring Product Lines, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), ACM Press, 2006, pp. 201–210.

[3] F. Anfurrutia, O. Díaz, S. Trujillo, On Refining XML Artifacts, in: Proceedings of the International Conference on Web Engineering (ICWE), vol. 4607 of LNCS, Springer-Verlag, 2007, pp. 473–478.

[4] S. Apel, D. Hutchins, An Overview of the gDeep Calculus, Tech. Rep. MIP-0712, Department of Informatics and Mathematics, University of Passau (2007).

[5] S. Apel, D. Hutchins, A Calculus for Uniform Feature Composition, ACM Transactions on Programming Languages and Systems (TOPLAS) (2010).

[6] S. Apel, C. Kästner, An Overview of Feature-Oriented Software Development, Journal of Object Technology (JOT) 8 (5) (2009) 49–84.

[7] S. Apel, C. Kästner, D. Batory, Program Refactoring using Functional Aspects, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), ACM Press, 2008, pp. 161–170.

[8] S. Apel, C. Kästner, T. Leich, G. Saake, Aspect Refinement - Unifying AOP and Stepwise Refinement, Journal of Object Technology (JOT) – Special Issue: TOOLS EUROPE 2007 6 (9) (2007) 13–33.

[9] S. Apel, C. Kästner, C. Lengauer, Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), ACM Press, 2008, pp. 101–112.

[10] S. Apel, C. Kästner, C. Lengauer, FeatureHouse: Language-Independent, Automatic Software Composition, in: Proceedings of the International Conference on Software Engineering (ICSE), IEEE CS, 2009, pp. 221–231.

[11] S. Apel, T. Leich, M. Rosenmüller, G. Saake, FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), vol. 3676 of LNCS, Springer-Verlag, 2005, pp. 125–140.

[12] S. Apel, T. Leich, G. Saake, Aspectual Feature Modules, IEEE Transactions on Software Engineering (TSE) 34 (2) (2008) 162–180.

[13] S. Apel, C. Lengauer, B. Möller, C. Kästner, An Algebra for Features and Feature Composition, in: Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST), vol. 5140 of LNCS, Springer-Verlag, 2008, pp. 36–50.

[14] I. Aracic, V. Gasiunas, M. Mezini, K. Ostermann, An Overview of CaesarJ, Transactions on Aspect-Oriented Software Development (TAOSD) 1 (1) (2006) 135–173.

[15] D. Batory, From Implementation to Theory in Product Synthesis (Keynote), in: Proceedings of the International Symposium on Principles of Programming Languages (POPL), ACM Press, 2007, pp. 135–136.

[16] D. Batory, Program Refactorings, Program Synthesis, and Model-Driven Design (Keynote), in: Proceedings of the International Conference on Compiler Construction (CC), vol. 4420 of LNCS, Springer-Verlag, 2007, pp. 156–171.

[17] D. Batory, J. Sarvela, A. Rauschmayer, Scaling Step-Wise Refinement, IEEE Transactions on Software Engineering (TSE) 30 (6) (2004) 355–371.

[18] I. Baxter, Design Maintenance Systems, Communications of the ACM (CACM) 35 (4) (1992) 73–89.

[19] A. Bergel, S. Ducasse, O. Nierstrasz, Classbox/J: Controlling the Scope of Change in Java, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 2005, pp. 177–189.

[20] T. Biggerstaff, A Perspective of Generative Reuse, Annals of Software Engineering 5 (1) (1998) 169–226.

[21] S. Böcker, D. Bryant, A. Dress, M. Steel, Algorithmic Aspects of Tree Amalgamation, Journal on Algorithms 37 (2) (2000) 522–537.

[22] E. Bodden, F. Forster, F. Steimann, Avoiding Infinite Recursion with Stratified Aspects, in: Proceedings of the International Net.ObjectDays Conference, Gesellschaft für Informatik, 2006, pp. 49–64.

[23] J. Bosch, Super-Imposition: A Component Adaptation Technique, Information and Software Technology 41 (5) (1999) 257–273.

[24] L. Bouge, N. Francez, A Compositional Approach to Superimposition, in: Proceedings of the International Symposium on Principles of Programming Languages (POPL), ACM Press, 1988, pp. 240–249.

[25] S. Boxleitner, S. Apel, C. Kästner, Language-Independent Quantification and Weaving for Feature Composition, in: Proceedings of the International Conference on Software Composition (SC), No. 5634 in LNCS, Springer-Verlag, 2009, pp. 45–54.

[26] M. Bravenboer, E. Visser, Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation Without Restrictions, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 2004, pp. 365–383.

[27] M. Chandy, J. Misra, An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection, ACM Transactions on Programming Languages and Systems (TOPLAS) 8 (3) (1986) 326–343.

[28] D. Clarke, S. Drossopoulou, J. Noble, T. Wrigstad, Tribe: A Simple Virtual Class Calculus, in: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, 2007, pp. 121–134.

[29] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.

[30] E. Ernst, K. Ostermann, W. Cook, A Virtual Class Calculus, in: Proceedings of the International Symposium on Principles of Programming Languages (POPL), ACM Press, 2006, pp. 270–282.

[31] R. Filman, D. Friedman, Aspect-Oriented Programming Is Quantification and Obliviousness, in: Aspect-Oriented Software Development, Addison-Wesley, 2005, pp. 21–35.

[32] R. Findler, M. Flatt, Modular Object-Oriented Programming with Units and Mixins, in: Proceedings of the International Conference on Functional Programming (ICFP), ACM Press, 1998, pp. 94–104.

[33] M. Flatt, S. Krishnamurthi, M. Felleisen, Classes and Mixins, in: Proceedings of the International Symposium on Principles of Programming Languages (POPL), ACM Press, 1998, pp. 171–183.

[34] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[35] W. Harrison, H. Ossher, Subject-Oriented Programming: A Critique of Pure Objects, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 1993, pp. 411–428.

[36] W. Harrison, H. Ossher, P. Tarr, General Composition of Software Artifacts, in: Proceedings of the International Symposium on Software Composition (SC), vol. 4089 of LNCS, Springer-Verlag, 2006, pp. 194–210.

[37] U. Hebisch, H. Weinert, Semirings, World Scientific, 1998.

[38] P. Höfner, R. Khedri, B. Möller, Feature Algebra, in: Proceedings of the International Symposium on Formal Methods (FM), vol. 4085 of LNCS, Springer-Verlag, 2006, pp. 300–315.

[39] P. Höfner, R. Khedri, B. Möller, An Algebra of Product Families, Software and Systems Modeling. To appear.

[40] D. Hutchins, Eliminating Distinctions of Class: Using Prototypes to Model Virtual Classes, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 2006, pp. 1–19.

[41] R. Jagadeesan, A. Jeffrey, J. Riely, A Calculus of Untyped Aspect-Oriented Programs, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), vol. 2743 of LNCS, Springer-Verlag, 2003, pp. 54–73.

[42] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. Rep. CMU/SEI-90-TR-21, SEI, CMU (1990).

[43] C. Kästner, S. Apel, M. Kuhlemann, Granularity in Software Product Lines, in: Proceedings of the International Conference on Software Engineering (ICSE), ACM Press, 2008, pp. 311–320.

[44] C. Kästner, S. Apel, S. ur Rahman, M. Rosenmüller, D. Batory, G. Saake, On the Impact of the Optional Feature Problem: Analysis and Case Studies, in: Proceedings of the International Software Product Line Conference (SPLC), SEI, CMU, 2009, pp. 181–190.

[45] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An Overview of AspectJ, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), vol. 2072 of LNCS, Springer-Verlag, 2001, pp. 327–353.

[46] G. Kiczales, J. Rivieres, The Art of the Metaobject Protocol, MIT Press, 1991.

[47] M. Kuhlemann, D. Batory, C. Kästner, Safe Composition of Non-Monotonic Features, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), ACM Press, 2009, pp. 177–185.

[48] R. Lämmel, E. Visser, J. Visser, Strategic Programming Meets Adaptive Programming, in: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, 2003, pp. 168–177.

[49] K. Lieberherr, B. Patt-Shamir, D. Orleans, Traversals of Object Structures: Specification and Efficient Implementation, ACM Transactions on Programming Languages and Systems (TOPLAS) 26 (2) (2004) 370–412.

[50] J. Ligatti, D. Walker, S. Zdancewic, A Type-Theoretic Interpretation of Pointcuts and Advice, Science of Computer Programming (SCP) 63 (3) (2006) 240–266.

[51] J. Liu, D. Batory, C. Lengauer, Feature-Oriented Refactoring of Legacy Applications, in: Proceedings of the International Conference on Software Engineering (ICSE), ACM Press, 2006, pp. 112–121.

[52] R. Lopez-Herrejon, D. Batory, C. Lengauer, A Disciplined Approach to Aspect Composition, in: Proceedings of the International Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), ACM Press, 2006, pp. 68–77.

[53] H. Masuhara, G. Kiczales, Modeling Crosscutting in Aspect-Oriented Mechanisms, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), vol. 2743 of LNCS, Springer-Verlag, 2003, pp. 2–28.

[54] S. McDirmid, M. Flatt, W. Hsieh, Jiazzi: New-Age Components for Old-Fashioned Java, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 2001, pp. 211–222.

[55] S. McDirmid, W. Hsieh, Aspect-Oriented Programming with Jiazzi, in: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, 2003, pp. 70–79.

[56] M. Mezini, K. Ostermann, Conquering Aspects with Caesar, in: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, 2003, pp. 90–100.

[57] M. Mezini, K. Ostermann, Variability Management with Feature-Oriented Program ming and Aspects, in: Proceedings of the International Symposium on Foundations of Software Engineering (FSE), ACM Press, 2004, pp. 127–136.

[58] M. Odersky, V. Cremet, C. Röckl, M. Zenger, A Nominal Theory of Objects with Dependent Types, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), vol. 2743 of LNCS, Springer-Verlag, 2003, pp. 201–224.

[59] M. Odersky, M. Zenger, Scalable Component Abstractions, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 2005, pp. 41–57.

[60] H. Ossher, W. Harrison, Combination of Inheritance Hierarchies, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 1992, pp. 25–40.

[61] C. Prehofer, Feature-Oriented Programming: A Fresh Look at Objects, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), vol. 1241 of LNCS, Springer-Verlag, 1997, pp. 419–443.

[62] G. Ramalingam, T. Reps, A Theory of Program Modifications, in: Proceedings of the International Joint Conference on Theory and Practice of Software Development, on Advances in Distributed Computing (ADC), and the Colloquium on Combining Paradigms for Software Development (CCPSD), vol. 494 of LNCS, Springer-Verlag, 1991, pp. 137–152.

[63] G. Ramalingam, T. Reps, Modification Algebras, in: Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST), Springer-Verlag, 1991, pp. 547–558.

[64] T. Reenskaug, E. Andersen, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, P. Stenslet, OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems, Journal of Object-Oriented Programming (JOOP) 5 (6) (1992) 27–41.

[65] J. Seco, L. Caires, A Basic Model of Typed Components, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), vol. 1850 of LNCS, Springer-Verlag, 2000, pp. 108–128.

[66] Y. Smaragdakis, D. Batory, Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs, ACM Transactions on Software Engineering and Methodology (TOSEM) 11 (2) (2002) 215–255.

[67] O. Spinczyk, D. Lohmann, M. Urban, AspectC++: An AOP Extension for C++, Software Developer's Journal 1 (5) (2005) 68–74.

[68] P. Tarr, H. Ossher, W. Harrison, S. Sutton, Jr., N Degrees of Separation: Multi-Dimensional Separation of Concerns, in: Proceedings of the International Conference on Software Engineering (ICSE), IEEE CS, 1999, pp. 107–119.

[69] S. Thaker, D. Batory, D. Kitchin, W. Cook, Safe Composition of Product Lines, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), ACM Press, 2007, pp. 95–104.

[70] S. Trujillo, M. Azanza, O. Díaz, Generative Metaprogramming, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), ACM Press, 2007, pp. 105–114.

[71] S. Trujillo, D. Batory, O. Díaz, Feature Oriented Model Driven Development: A Case Study for Portlets, in: Proceedings of the International Conference on Software Engineering (ICSE), IEEE CS, 2007, pp. 44–53.

[72] D. Tucker, S. Krishnamurthi, Pointcuts and Advice in Higher-Order Languages, in: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, 2003, pp. 158–167.

[73] M. VanHilst, D. Notkin, Using Role Components in Implement Collaboration-based Designs, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 1996, pp. 359–369.

[74] D. Walker, S. Zdancewic, J. Ligatti, A Theory of Aspects, SIGPLAN Notices 38 (9) (2003) 127–139.

[75] M. Zenger, Type-Safe Prototype-Based Component Evolution, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), vol. 2374 of LNCS, Springer-Verlag, 2002, pp. 470–497.

A Implementation in Prover9

In this section, we list the Prover9 implementation of all axioms, lemmas, and theorems. A typical Prover9 script consists of a section for the definition of axioms (`formulas(assumptions)`) and a section for the lemmas and theorems to prove (`formulas(goals)`). We begin with a basic script covering axioms and lemmas for introductions (Sec. A.1). Then, we provide two scripts implementing the axioms and lemmas for modifications (Sec. A.2). Finally, we provide two scripts implementing the axioms and theorems for quarks (Sec. A.3).
Note that, in order to allow a smooth and efficient treatment in Prover9, we have used a simplified representation in which introductions are treated as "constant-valued" modifications, i.e., as modifications m with $m \odot j = i_m$ for all introductions j and some fixed introduction i_m. Otherwise, we would have needed to distinguish the "types" of introductions and modifications by a predicate and to endow all equational laws with preconditions expressing type correctness.

A.1 Introductions

For introductions and introduction sum, we define an operator and six axioms:

```
formulas(assumptions).

% introduction sum %
op(500, infix, "+").

% associativity %
x + (y + z) = (x + y) + z.
% identity I %
0 + x = x.
% identity II %
x + 0 = x.
% distant idempotence %
x + (y + x) = y + x.
% inclusion %
x <= y <-> x + y = y.
% equivalence %
eqv(x,y) <-> x <= y & y <= x.

end_of_list.
```

Using the six axioms, we prove the following eight lemmas: [16]

```
formulas(goals).

% direct idempotence %
x + x = x.
% least element %
```

[16] Note that Prover9 is not able to prove multiple goals of which some are not in normal form. In this case, each goal has to be proved separately.

```
0 <= i.
% least element is unique %
i <= 0 -> i = 0.
% upper bound I %
x <= x + y.
% upper bound II %
y <= x + y.
% reflexivity %
x <= x.
% transitivity %
x <= y & y <= z -> x <= z.
% quasicommutativity I %
eqv(x+y,y+x).

end_of_list.
```

A.2 Modifications

For modification product, we define three operators and eleven axioms, including the ones necessary for modifications which stem from introductions:

```
formulas(assumptions).

% introduction sum %
op(500, infix, "+").
% modification product %
op(510, infix, "/").
% modification application %
op(520, infix, "*").

%%%%%%%%%%%%%%%%%
% introductions %
%%%%%%%%%%%%%%%%%

% associativity %
(x + y) + z = x + (y + z).
% identity I %
0 + x = x.
% identity II %
x + 0 = x.
% distant idempotence %
x + (y + x) = y + x.
% inclusion %
x <= y <-> x + y = y.
% equivalence %
eqv(x,y) <-> x <= y & y <= x.

%%%%%%%%%%%%%%%%%
% modifications %
%%%%%%%%%%%%%%%%%

% distributivity %
x * (y + z) = (x * y) + (x * z).
% empty modification %
1 * x = x.
% iterative application %
(x / y) * z = x * (y * z).
% identity I %
1 / x = x.
% identity II %
x / 1 = x.

end_of_list.
```

Using the eleven axioms, we prove the following lemma:

```
formulas(goals).

% associativity %
((x / y) / z) * i = (x / (y / z)) * i.

end_of_list.
```

A.3 Quarks

For quark composition, we define an additional operator and an additional axiom. Furthermore, we include all optional axioms regarding modification product, which have to be commented out if quark composition in the standard configuration shall be examined:

```
formulas(assumptions).

%%%%%%%%%%%%%%%%%%
% introductions %
%%%%%%%%%%%%%%%%%%

% associativity %
(x + y) + z = x + (y + z).
% identity I %
0 + x = x.
% identity II %
x + 0 = x.
% distant idempotence %
x + (y + x) = y + x.
% inclusion %
x <= y <-> x + y = y.
% equivalence %
eqv(x,y) <-> x <= y & y <= x.

%%%%%%%%%%%%%%%%%%
% modifications %
%%%%%%%%%%%%%%%%%%

% empty modification %
1 * x = x.
% distributivity %
x * (y + z) = (x * y) + (x * z).
% associativity %
(x / y) / z = x / (y / z).
% identity I %
1 / x = x.
% identity II %
x / 1 = x.
% distant idempotence %
x / (y / x) = y / x.
% commutativity %
x / y = y / x.
% iterative application %
(x / y) * z = x * (y * z).

%%%%%%%%%%
% quarks %
%%%%%%%%%%

% quark composition %
[x1,y1,z1] @ [x2,y2,z2] = [x1 / x2, (x1 / x2) * (y1 + (z1 * y2)), z1 / z2].

```

```
end_of_list.

formulas(goals).
```

Using these axioms, we can prove several properties of the composition of simple, local, global, and full quarks:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% associativity of quark composition %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% associativity of composition of simple quarks %
[1,y1,1] @ ([1,y2,1] @ [1,y3,1]) = ([1,y1,1] @ [1,y2,1]) @ [1,y3,1].
% associativity of composition of local quarks %
[1,y1,z1] @ ([1,y2,z2] @ [1,y3,z3]) = ([1,y1,z1] @ [1,y2,z2]) @ [1,y3,z3].
% associativity of composition of global quarks %
[x1,y1,1] @ ([x2,y2,1] @ [x3,y3,1]) = ([x1,y1,1] @ [x2,y2,1]) @ [x3,y3,1].
% associativity of composition of full quarks %
[x1,y1,z1] @ ([x2,y2,z2] @ [x3,y3,z3]) = ([x1,y1,z1] @ [x2,y2,z2]) @ [x3,y3,z3].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% identity of quark composition %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

identity of simple quarks %
[1,0,1] @ [1,y2,1] = [1,y2,1].
identity of local quarks %
[1,0,1] @ [1,y2,z2] = [1,y2,z2].
identity of global quarks %
[1,0,1] @ [x2,y2,1] = [x2,y2,1].
identity of full quarks %
[1,0,1] @ [x2,y2,z2] = [x2,y2,z2].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% direct idempotence of quark composition %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

direct idempotence of simple quarks %
[1,y,1]@[1,y,1]=[1,y,1].
direct idempotence of local quarks %
[1,y,z]@[1,y,z]=[1,y,z].
direct idempotence of global quarks %
[x,y,1]@[x,y,1]=[x,y,1].
direct idempotence of full quarks %
[x,y,z]@[x,y,z]=[x,y,z].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% distant idempotence of quark composition %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

distant idempotence of simple quarks %
[1,y1,1]@([1,y2,1]@[1,y1,1])=[1,y2,1]@[1,y1,1].
distant idempotence of local quarks %
[1,y1,z1]@([1,y2,z2]@[1,y1,z1])=[1,y2,z2]@[1,y1,z1].
distant idempotence of global quarks %
[x1,y1,1]@([x2,y2,1]@[x1,y1,1])=[x2,y2,1]@[x1,y1,1].
distant idempotence of full quarks %
[x1,y1,z1]@([x2,y2,z2]@[x1,y1,z1])=[x2,y2,z2]@[x1,y1,z1].

end_of_list.
```