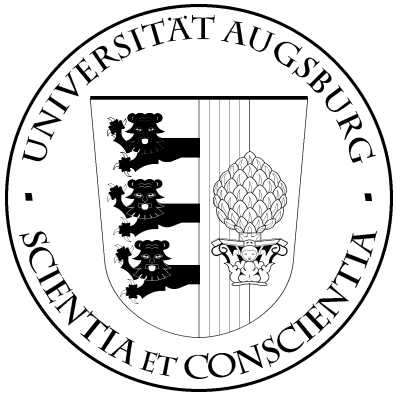
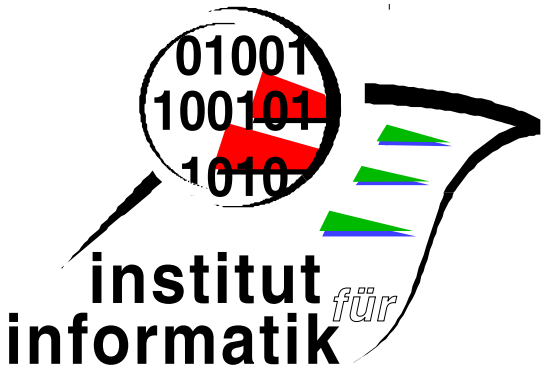
Universität Augsburg



An Extension for Feature Algebra

Peter Höfner **Bernhard Möller**

Report 2010-09 October 2010



Institut für Informatik
D-86135 Augsburg

An Extension for Feature Algebra

Peter Höfner, Bernhard Möller

Universität Augsburg, 86135 Augsburg, Germany

Abstract

Feature algebra was introduced as an abstract framework for feature oriented software development. One goal is to provide a common, clearly defined basis for the key ideas of feature orientation. So far, feature algebra captures major aspects of feature orientation, like the hierarchical structure of features and feature composition. However, as we will show, it is not able to model aspects at the level of code, i.e., situations where code fragments of different features have to be merged. With other words, it does not reflect details of concrete implementations.

In the paper we first present concrete models for the original axioms of feature algebra which represent the main concepts of feature oriented programs. This shows that the abstract feature algebra can be interpreted in different ways. We then use these models to show that the axioms of feature algebra do not reflect all aspects of feature orientation properly. This gives the motivation to extend the abstract algebra — which is the second main contribution of the paper. We modify the axioms and introduce the concept of an *extended feature algebra*. The original algebra can be retrieved by a single additional axiom. As third contribution we introduce more operators to cover concepts like overriding in the abstract setting.

Keywords: Feature orientation, feature algebra, algebraic characterisation of FOSD

1. Introduction

Over the last years *feature orientation* (e.g. [12, 13, 1]) has been established in computer science as a general programming paradigm that provides formalisms, methods, languages, and tools for building variable, customisable, and extensible software. In particular, feature orientation summarises feature-oriented software development and feature oriented programming. It has widespread applications from network protocols [12] and data structures [14] to software product lines [38]. It arose from the idea of level-based designs, i.e., the idea that each program (design) can be successively built up by adding more and more levels (features). Later, this idea was generalised to the abstract concept of features. A *feature* reflects an increment in functionality or in software development.

Over the years, feature oriented software development was more and more supported by software tools. Examples are `FeatureHouse` [3] and the `AHEAD` Tool Suite [10]. The former is a general approach to the composition of software artefacts. It supports various input formats like source code, test cases, models, documentation and makefiles. In `FeatureHouse`, features and software artefacts are represented as (forests of) *feature structure trees* (FSTs) that capture the essence of an artefact's modular structure in the form of a tree (cf. Section 2). `AHEAD` (Algebraic Hierarchical Equations for Application Design) is an approach to feature orientation that is based on stepwise refinement. Each refinement step is modelled by simple algebraic expressions that specify changes to a given program. The corresponding Tool Suite directly supports that approach. As shown in several case studies, `FeatureHouse` and `AHEAD` can be used for large-scale program synthesis (e.g. [3, Table 4], [38] and [37]).

Although the progress over the last time in the area of feature orientation was quite impressive, the mathematical structure and the mathematical foundations were studied less intensively. Steps towards a structural description and analysis are done with the help of feature models first introduced in the Feature-Oriented Domain Analysis method

Email addresses: `hoefner@informatik.uni-augsburg.de` (Peter Höfner), `moeller@informatik.uni-augsburg.de` (Bernhard Möller)

(FODA) [36]. Since then, feature modelling has been widely adopted by the software product line community and a number of extensions have been proposed. A further step towards an abstract mathematical description of feature orientation is AHEAD [13]. It expresses hierarchical structures as nested sets of equations. Most recently, a purely algebraic approach was developed [6]. *Feature algebra* is a formal framework that captures many of the common ideas of feature orientation, such as introductions, refinements, or quantification, in an abstract, language- and tool-independent way and thus masks differences of minor importance. It is intended as a common basis for describing, evaluating and comparing existing notions, tools and other aspects in feature orientation and not just as the starting point for the development of yet another tool. In addition, feature algebra can serve as a formal foundation of architectural metaprogramming [11] and automatic feature-based program synthesis [21]. Both paradigms emerged from feature orientation and facilitate the treatment of programs as values manipulated by metaprograms, e.g., in order to add a feature to a program system. This requires a formal theory that precisely describes which manipulations are allowed.

In the present paper we will build on this algebraic structure; we will show that certain aspects are not captured by the original feature algebra (OFA) and will extend that to cover most of the missing concepts.

The standard model of OFA (cf. [6]) is based on *feature structure forests (FSFs)* [5]. An FSF captures the essential, hierarchical module structure of a given program. Each node of an FSF represents a structural element (such as a package etc.). More precisely, a node has a name that corresponds to the name of the structural element and a type that corresponds to its syntactic category. Inner nodes denote modules like classes and packages and the leaves store the modules' contents like method declarations. Based on FSFs, feature combination can be modelled as superimposition of FSFs, i.e., as recursively merging their corresponding substructures.

We present two further concrete models for OFA in this paper; both are based on FSFs, too, but use different representations for them. The first model (see Section 2) uses prefix-closed sets of strings into which FSFs can be transformed. After the introduction of that model, we recapitulate the abstract notion of OFA in Section 3. After that we present the second model which is based on prefix-free lists of strings. For the sake of completeness we also recapitulate the standard model of OFA. In particular we will emphasise the commonalities and differences of the models. In Section 5 we then show that the models are adequate as long as one does not consider feature oriented programming at code level. If manipulation of code and not only of the overall program structure is to be included into the model some aspects of feature orientation cannot be reflected properly in OFA. In particular, we show that merging, overriding or extending bodies of methods leads to problems. To overcome this deficiency, we relax the axioms and introduce the concept of an extended feature algebra (EFA). To clarify the idea and to underpin the relaxation we extend the introduced models of OFA to handle code explicitly in Section 7. Finally we discuss how additional operators can be introduced to capture even more properties of feature orientation formally. In particular, we present operators for merging, overriding and updating as they arise in code modification. The paper closes with a short summary and an outlook concerning future work.

2. A Model of Feature Algebra

Based on *feature structure forests* (FSFs), we give a first concrete model for OFA. The formal definition of feature algebras will be given in the next section. FSFs capture the essential hierarchical module structure of a given system (e.g. [6]). Essentially, an FSF is a "stripped-down abstract syntax tree" [3]. An example is given in Figure 1, where a simple Java class *Calc* is described as an FSF consisting of a single tree. For the present paper we restrict ourselves (and the given examples) to Java, since it is a well-known and widely used programming language. Examples from other feature oriented programming languages can easily be described in a similar way.

Each node of an FSF has a name and a type. For example, the class *Calc* is represented by the node *Calc* of type class in Figure 1. For the present paper we are mostly not interested in the type information, hence we will skip this information if it is appropriate. Moreover it is straightforward to see that a hierarchical structure not always yields single trees. This is for example the case if we deal with several classes in several packages. Hence also proper forests consisting of several trees occur. This motivates the following definition.

Definition 2.1 (feature structure forest). A *forest* over a finite set Σ of labels and a finite set N of nodes is a collection (i.e., a set or a list) of mutually node-disjoint trees over Σ. A tree t over Σ consists of a *root node* $r \in N$ with a label from

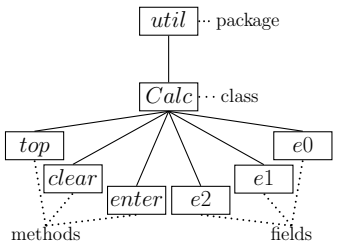


Figure 1: A simple JAVA-class as FSF ([11, 6])

Σ and a forest f over Σ and N in which r does not occur again. The trees in f are called the *immediate descendants* of the root. If each of the descendant collections in t is a set then t is *unordered*, otherwise *ordered*. If the descendant collection is empty then t is called a *leaf*.

A *Feature Structure Forest* (FSF) is a forest (collection of trees) over a set of labels that correspond to, e.g., directory names in large structured systems.

The tree structure in an FSF expresses the hierarchical interdependence of the system components. In Java and in most other languages, the non-leaf nodes correspond to modules like packages, classes and subclasses and the leaves store the contents of the modules. This is captured in a language-independent way by the branching structure of the trees in a corresponding FSF. Contrarily, often the leaves store language-dependent data like source code etc. As in earlier work by Apel et al. [7], we ignore the language-dependent parts for the moment. Instead we assume that the leaves are opaque and not further analysed. This will be changed in our extended model in Section 6.

It is well known that certain labelled forests can be represented using sets of strings of node labels (e.g., [9]). Let Σ be an alphabet of node labels and, as usual, Σ^+ the set of all non-empty finite strings over Σ. Every word from Σ^+ can be thought of as the sequence of node labels along a path in some FST. In the sequel we will just write "path" instead of the lengthy "sequence of labels along a path". Mostly, the tree nodes remain implicit.

A first model now represents a forest by all possible paths from roots to nodes. Since every prefix of the path leading to a node x corresponds to a path from the respective root to an ancestor of x, with a path also all its non-empty prefixes are paths in the forest. Therefore the set of all possible paths is prefix-closed.

Conversely, one can (uniquely up to the order of branches) reconstruct a forest from the prefix-closed set of its paths.

Example 2.2. For the FSF of Figure 1 the alphabet Σ has to be a superset of $\{util, Calc, top, clear, enter, e0, e1, e2\}$. The tree itself can be encoded as the following prefix-closed set:

$$\{util,\ util.Calc,\ util.Calc.top,\\ util.Calc.clear,\ util.Calc.enter,\\ util.Calc.e0,\ util.Calc.e1,\\ util.Calc.e2\}\ ,$$

where . is used to separate the elements of Σ in a path. □

Definition 2.3. We define $P\Sigma$ as the set of all prefix-closed subsets of Σ^+.

Using a *set* of paths forgets about the relative order of child nodes of a node, i.e., this model is suitable only for unordered trees.

Note that this approach does not allow different roots with identical labels and no identical labels on the immediate descendants of a node, hence only certain forests can be encoded. Hence the names have to be unique. However, this is not a severe restriction, since we can always rename nodes.

In [5], *(feature tree) superimposition* is presented by Apel and Lengauer. It is a composition technique for software that has successfully applied within several case studies covering various areas of computer science. In particular, it is shown to play a crucial rôle in feature orientation. If we abstract from the code-level and use FSFs where nodes

contain only abstract data, feature tree superimposition can be represented as simple set union on the elements of PΣ. This is well defined, since PΣ is closed under set union. Moreover, this operation allows simple algebraic calculations. In particular, this operator is commutative and idempotent and hence in this model the order of combination does not matter.

Example 2.4. The following example is a simplified version of an example given in [7]. Composing the two FSTs at the left of Figure 2 by superimposition (on PΣ) yields the right FST. In the figure, we use the same notation as for the algebraic operation to symbolise superimposition.

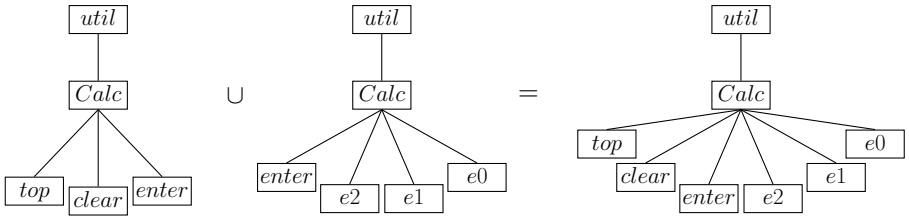


Figure 2: Superimposition of FSFs ([7])

The algebraic counterparts are

$$\begin{aligned}
\textsc{BaseA} &=_{df} \{\mathit{util},\ \mathit{util.Calc},\ \mathit{util.Calc.top},\ \mathit{util.Calc.clear},\ \mathit{util.Calc.enter}\}\,,\\
\textsc{BaseB} &=_{df} \{\mathit{util},\ \mathit{util.Calc},\ \mathit{util.Calc.enter},\ \mathit{util.Calc.e2},\ \mathit{util.Calc.e1},\ \mathit{util.Calc.e0}\}\,,\\
\textsc{Base} &=_{df} \textsc{BaseA} \cup \textsc{BaseB}\\
&= \{\mathit{util},\ \mathit{util.Calc},\ \mathit{util.Calc.top},\ \mathit{util.Calc.clear},\ \mathit{util.Calc.enter}\,,\\
&\quad\ \mathit{util.Calc.e2},\ \mathit{util.Calc.e1},\ \mathit{util.Calc.e0}\}\,.
\end{aligned}$$

In particular the path *util.Calc.enter* occurs only once. □

Of course in a more general setting, where leaves of FSFs contain more (language-dependent) information, the composition is getting more complicated. For example one might need wrappers to merge nodes with the same name (cf. [5]). We will revisit this crucial point of "code-level-merging" later. As we will see, altering the contents of a leaf node (using either superimposition or modifications) may lead to problems. Examples for altering the content of a leaf are overriding or extending a method body (if present).

From the definitions the following property is immediate.

Lemma 2.5. *The structure* $\mathrm{P}\Sigma = (\mathrm{P}\Sigma, \cup, \emptyset)$ *forms a commutative monoid, i.e.,* $\cup$ *is associative and commutative with* $\emptyset$ *as its neutral element. Due to commutativity and idempotence of its addition operator* $\cup$*, the law* $A \cup B \cup A = B \cup A$ *for* $A, B \in \mathrm{P}\Sigma$ *holds.*

The law $A \cup B \cup A = B \cup A$ is called *the axiom of distant idempotence*. It expresses the fact that adding an already present feature has no effect. This law will play a crucial rôle in the abstract setting (cf. Section 3).

Since the algebra is set-based, we can use set inclusion to describe that an FSF has the same or more features, nodes or leaves as another. For example we have BaseA ⊆ Base in Example 2.4. As known from simple set theory, the operator ⊆ can be defined by the union operator as $A \subseteq B \Leftrightarrow_{df} A \cup B = B$. In the next section we will use a similar construction to define an abstract version of inclusion.

In addition to feature superimposition, a feature algebra also comprises modifications. Modifications can change the content of the nodes of FSFs. In the concrete model modifications correspond to tree rewriting functions. It is easy to see that such functions can be used to model many different aspects of feature orientation. With respect to FSFs a modification might be the action of adding a new child node (adding a method to a class), of deleting a node (removing a class or method) or of renaming a node (renaming a class).

The difference between superimposition and modification lies in their types. Moreover, as discussed in [7], the two techniques are based on different ideas. Superimposition can be interpreted as *composing features* and hence is a binary inner operation on introductions. In contrast to that, modifications can be interpreted as descriptions of FSF-transformations. A particular modification might be to add some fixed FSF using superimposition; however, we do not want to limit ourselves to that special case. Both concepts have been intensively studied (e.g. [5, 39, 45]).

A concrete tool for performing the operations of a FSFs is `FeatureHouse` [3, 5]. For various languages, such as Java, C#, C, Haskell, and JavaCC, it allows composing features written in the respective language using FSF superimposition. This is not restricted to manipulation of the overall hierarchical directory structure of such features; `FeatureHouse` also composes features regarding the code in the leaves of the directory FSF. In Section 5 we will show that this may violate the axioms of the original feature algebra, which was developed without regard of the code level.

3. Feature Algebra

We now abstract from the concrete model of FSFs and introduce the structure of feature algebra. It was first presented by Apel, Lengauer, Möller and Kästner in [6]. There a number of axioms is distinguished that have to be satisfied by languages suitable for feature orientation. Each of the axioms is derived from and illustrated by concrete examples from software development. To underpin the usability of this algebraic approach a concrete model is introduced. The model is also based on FSFs. However next to nodes each tree has a unique name. We will recapitulate this model in the next section.

For the present paper we compact the axioms in the following definition. To focus on the main aspects we omit a discussion of the variants and alternatives described in [6].

Definition 3.1 (feature algebra). A *feature algebra* is a tuple $(M, I, +, \circ, \cdot, 0, 1)$ such that for all $m, n \in M$ and all $i, j \in I$

- $(I, +, 0)$ is a monoid of *introductions* satisfying the additional axiom of distant idempotence, i.e., $i + j + i = j + i$.
- $(M, \circ, 1)$ is a groupoid of *modifications* operating via $\cdot$ on I, i.e., $\circ$ is a binary inner operation on M and 1 is an element of M such that furthermore
  - $\cdot$ is an external binary operation from $M \times I$ to I,
  - $(m \circ n) \cdot i = m \cdot (n \cdot i)$,
  - $1 \cdot i = i$.
- 0 is a right-annihilator for $\cdot$, i.e., $m \cdot 0 = 0$.
- $\cdot$ distributes over $+$, i.e., $m \cdot (i + j) = (m \cdot i) + (m \cdot j)$.

In case of a non-commutative interpretation of $+$, a term $i + j$ means that the features of i are added to the ones of j, ignoring the ones already present, as expressed by the axiom of distant idempotence.

One might think whether the annihilation axiom can be dropped. This would allow modifications to add features to the empty introduction. However, that is not the intention with modifications: they rather should be applied uniformly to all "branches" of an FSF, as expressed by the distributivity law. Since the empty introduction 0 has no branches at all, it is reasonable that a modification should not change it. There is also an algebraic counterpart of this: for arbitrary $i \in I$ we have

$$m \cdot i = m \cdot (i + 0) = m \cdot i + m \cdot 0$$

and similarly, $m \cdot i = m \cdot 0 + m \cdot i$. This means that $m \cdot 0$ is a neutral element for all results of the modification m anyway. So it is reasonable to stipulate by the annihilation axiom that $m \cdot 0$ coincides with the global neutral element 0.

In a feature algebra the set I is the abstract counterpart of FSFs or, more precisely, the sets of strings representing the FSFs. The modifications in the set M allow changing the introductions and hence correspond to term rewriting functions. The central operations are the *addition* $+$ that abstractly models feature tree superimposition, the operator $\cdot$ that allows *application* of a modification to an introduction and the modification *composition* operator $\circ$.

The model PΣ from the previous section forms a feature algebra when a suitable set of tree rewriting functions is chosen as the set of modifications. The first item of Definition 3.1 is the abstract form of Lemma 2.5. The set of functions has to be chosen carefully, since otherwise one might, e.g., violate the uniqueness condition that different nodes need to have different names. The operator $\cdot$ coincides with function application and $\circ$ with function composition. The axiom $(m \circ n) \cdot i = m \cdot (n \cdot i)$ is satisfied by the usual definition of function composition: applying a composed function is equivalent to applying the single functions in sequence.

In the concrete model the elements of PΣ can be compared using the subset relation $\subseteq$ which is well known to be a partial order. The corresponding counterpart $\leq$ on general introductions is only a preorder, i.e., is reflexive and transitive. It induces an equivalence relation in a standard way.

Definition 3.2 (natural preorder/introduction equivalence).

(1) The *natural preorder* or *introduction inclusion* is defined by $i \leq j \Leftrightarrow_{df} i + j = j$.
(2) The *introduction equivalence* by $i \sim j \Leftrightarrow_{df} i \leq j \wedge j \leq i$.

In the literature the natural preorder is sometimes called *subsumption preorder*. As in the concrete model, $i \leq j$ describes that the introduction j has at least the same paths as the introduction i. That $\leq$ need not form an ordering is demonstrated in the next section with the help of another concrete example. Moreover, introduction inclusion is closely related to the subtyping relation $<:$ in the Deep calculus of Hutchins [34]. This calculus was developed to provide a type-safe support for virtual classes. Hence the abstract algebra not only provides a mathematical base for FSFs, but also for Deep and shows certain commonalities. The introduction equivalence means that two introductions contain the same features up to re-ordering and hence might also be called *permutation equivalence*.

Finally, we define an equivalence relation on modifications that expresses equality under application, i.e., two modifications are equivalent if and only if they produce the same if applied to the same introduction.

Definition 3.3 (application equivalence). The *application equivalence* $\approx$ of two modifications m, n is defined by $m \approx n \Leftrightarrow_{df} \forall\, i : m \cdot i = n \cdot i$. This is clearly an equivalence relation.

Example 3.4. Assume two different modifications acting on an FSF F and let *new* be a new name that does not occur in any tree in F. The first modification m_1 adds the feature $\{util.Calc.tmp\}$. In the case that this feature is already present, it does nothing. After that m_1 renames the node *tmp* to *new*. The second modification m_2 first creates a feature $\{util.Calc.new\}$ and then copies the structure "below the node *tmp*" (all descendant of *tmp*) to node *new*. Last, m_2 deletes $\{util.Calc.tmp\}$ and all its descendants. Then m_1 and m_2 are different, but application-equivalent, modifications. □

Up to now, we have presented the mathematical basics of the original feature algebra. The above definition will be the base for our extension that reflects certain aspects of the code level. Before that, we recapitulate some basic properties of feature algebra that are needed afterwards.

Lemma 3.5. *Assume i, j, k to be introductions and m, n, o to be modifications of a feature algebra.*

(a) $0 \leq i$ *and* $(i \leq 0 \Rightarrow i = 0)$.
(b) $+$ *is idempotent; i.e.,* $i + i = i$.
(c) $\leq$ *is a preorder, i.e.,* $i \leq i$ *and* $i \leq j \wedge j \leq k \Rightarrow i \leq k$.
(d) $i \leq i + j$ *and* $j \leq i + j$.
(e) $i \leq k \wedge j \leq k \Leftrightarrow i + j \leq k$.
(f) $+$ *is isotone in both arguments:* $i \leq j \Rightarrow i + k \leq j + k \wedge k + i \leq k + j$.
(g) $+$ *is quasi-commutative w.r.t.* $\sim$*, i.e.,* $i + j \sim j + i$.
(h) $(m \circ (n \circ o)) \cdot i = ((m \circ n) \circ o) \cdot i$.
(i) $(m \circ 1) \cdot i = (1 \circ m) \cdot i = m \cdot i$.

Meanings of Parts (a)–(c) are straightforward. Part (d) says that addition determines an upper bound with respect to the natural preorder. Part (e) shows that the sum is even a least upper bound. Part (g) shows that + is commutative up to introduction equivalence. In the concrete model of FSFs that means that super imposition commutes up to the

tree ordering. Parts (h) and (i) show that, up to application equivalence, $\circ$ is associative and 1 is its neutral element, i.e., $(m \circ (n \circ o)) \approx ((m \circ n) \circ o)$ and $(m \circ 1) \approx (1 \circ m) \approx m$.

This lemma shows a particular advantage of the abstract algebraic approach. Since it contains only first-order equational axioms, it is predestined for automatic theorem proving and therefore we can skip most of the proofs since they can be automated in off-the-shelf theorem provers. For the purpose of that paper we have encoded feature algebra in Waldmeister [18][1]. A generic input file for Waldmeister is presented in Appendix A. The input files for particular properties proved in this paper as well as the corresponding output files can be found at a website [29]. In Appendix B, we will only present proofs for important properties.

4. Two More Examples

Since in certain applications the relative order of the immediate successor nodes in a tree matters, we now present a second model that reflects forests of ordered labelled trees. It uses the fact that all paths in a tree can be recovered from the maximal ones that lead from roots to leaves by forming their prefix closure. It should be noted here that the maximal paths can be viewed as atomic introductions in the sense of [6], since they uniquely determine the leaf nodes of the forest in which the code etc. resides (the subcomponents of which are not considered in the present view).

This could have been exploited already in the previous model, but would have led to a much more complicated definition of tree superimposition. While a finite unordered forest can be represented as a finite *set* of paths, for an ordered one we use finite *lists* of paths. The representation can be made unique by using prefix-closed lists; however, this both cumbersome and wasteful. Therefore we instead use lists that are prefix-free, i.e., lists l that with a path p do not contain a proper or improper prefix of p elsewhere in l. In particular, such lists are repetition-free. Like the previous model, this does not admit different roots with identical labels and no identical labels on immediate descendants of a node.

Example 4.1. The FSF of Figure 1 is encoded as the following prefix-free list:

$$[util.Calc.top,\ util.Calc.clear,\\ util.Calc.enter,\ util.Calc.e2,\\ util.Calc.e1,\ util.Calc.e0]\ .$$

□

Superimposition + is now defined inductively over the structure of the first list:

- The empty list [ ] does not affect another list of paths l :

$$[\,] + l =_{df} l\ .$$

- A singleton list $[p]$ is added to an existing list l as follows.
  - If l is empty then p is simply added to it:

$$[p] + [\,] = [p]\ .$$

  - If p is a prefix of some element of l then it is simply skipped and not added to l. If l contains a proper prefix q of p then q is replaced by p. Note that there is at most one such prefix, since l is assumed to be prefix-free and for any two prefixes of p one must be a prefix of the other. Hence if l contained two different prefixes of p it would not be prefix-free. If none of the above cases applies then p is added to the front of l. This procedure is described by the following definition:

$$[p] + [q_1, \ldots, q_n] =_{df} \begin{cases} [q_1, \ldots, q_n] & \text{if } p \text{ is a prefix of some } q_i \\ [q_1, \ldots q_{i-1}, p, q_{i+1}, \ldots, q_n] & \text{if } q_i \text{ is a prefix of } p \\ [p, q_1, \ldots, q_n] & \text{otherwise .} \end{cases}$$

[1] In contrast to [7], we use Waldmeister instead of Prover9 since it can handle multiple sorts. For feature algebra we use the two sorts M and I.

- For a longer first argument list we set

$$[p_1, \ldots, p_m, p_{m+1}] + l =_{df} [p_1, \ldots, p_m] + ([p_{m+1}] + l) .$$

Definition 4.2. We define LΣ as the set of all prefix-free lists of elements of Σ^+.

Lemma 4.3.

(a) *The structure* LΣ = (LΣ, +, [ ])*, where + is extended pointwise to sets of prefix-free lists, forms a (non-commutative) monoid that additionally satisfies the axiom of distant idempotence; its natural preorder reflects list inclusion and the associated equivalence relation is permutation equivalence, i.e., equality up to a permutation of the list elements.*

(b) *If modifications are again rewriting functions with the same operations as before, the whole structure forms a feature algebra.*

In both algebras PΣ and LΣ the law of distant idempotence models the fact that adding an already present feature has no effect. This reflects that the law of (distant) idempotence seems of central interest in feature orientation. However, in the next section we will show that the axiom of distant idempotence yields problems in a model that considers more details.

The model also shows that the preorder need not be not an order.

Example 4.4. Assume the following two lists:

$$L_1 =_{df} [util.Calc.clear,\ util.Calc.enter] \quad \text{and} \quad L_2 =_{df} [util.Calc.enter,\ util.Calc.clear] .$$

By definition of + is is easy to show that $L_1 + L_2 = L_2$ and $L_2 + L_1 = L_1$; hence $L_1 \leq L_2$ and $L_2 \leq L_1$, but $L_1 \neq L_2$. However L_1 and L_2 are permutation equivalent, which is algebraically expressed as $L_1 \sim L_2$. □

As a third model NΣ we recapitulate the one given in the original papers concerning feature algebra [6, 7]. As we will show it is closely related to the previous one. In the original model of feature algebra, nodes in FSFs not only contain a name and a type, but each path contains also the name of "source" of the particular feature. Instead of a complicated description and definition we just give a simplified example from [7].

Example 4.5. Assume again the FSTs on the left hand side of Figure 2. Using the above model LΣ the first FST would be encoded as

$$[util.Calc.top,\ util.Calc.clear,\ util.Calc.enter] .$$

This model may also be used to endow each maximal path with an additional name that indicates the source to which the path belongs. For simplicity we assume that all features of the first two FSFs stem from the same source, namely BaseA and BaseB. Therefore the two terms are now encoded as

$$\begin{aligned} \text{BaseA} &=_{df} [\text{FeatA} :: util.Calc.top,\ \text{FeatA} :: util.Calc.clear,\ \text{FeatA} :: util.Calc.enter] , \\ \text{BaseB} &=_{df} [\text{FeatB} :: util.Calc.enter,\ \text{FeatB} :: util.Calc.e2,\ \text{FeatB} :: util.Calc.e1,\ \text{FeatB} :: util.Calc.e0] . \end{aligned}$$

Now, the composition operator + for superimposition has to be slightly revised: two paths are only merged if both, the path and the source, coincide. In our particular example therefore two occurrences of *enter* appear in the superimposition.

$$\begin{aligned} \text{Base} &=_{df} \text{BaseA} + \text{BaseB} \\ &= [\text{FeatA} :: util.Calc.top,\ \text{FeatA} :: util.Calc.clear,\ \text{FeatA} :: util.Calc.enter, \\ &\quad \text{FeatB} :: util.Calc.enter,\ \text{FeatB} :: util.Calc.e2,\ \text{FeatB} :: util.Calc.e1,\ \text{FeatB} :: util.Calc.e0] \\ &= [\text{FeatA} :: util.Calc.top] + [\text{FeatA} :: util.Calc.clear] + [\text{FeatA} :: util.Calc.enter] + \\ &\quad [\text{FeatB} :: util.Calc.enter] + [\text{FeatB} :: util.Calc.e2] + [\text{FeatB} :: util.Calc.e1] + [\text{FeatB} :: util.Calc.e0] \end{aligned}$$

The term where the composed element is represented by a single list is the generic one w.r.t. the models introduced so far. The last algebraic expression where the element is split into its "basic" components is given since this is the way how features are presented by Apel, Lengauer, Möller and Kästner [7]. □

Note that we do not have a one-to-one relationship between feature structure forests and algebraic expressions in that model: while each algebraic term (list of maximal paths) can be transformed into an FSF in a unique way; the converse direction does not hold any longer.

Of course the model NΣ can be embedded into LΣ. The trick is to add the feature names as roots in the corresponding forest.

Example 4.6. Instead of using the list [FeatA :: *util.Calc.top*, FeatA :: *util.Calc.clear*, FeatA :: *util.Calc.enter*] to describe BaseA, we use [FeatA.*util.Calc.top*, FeatA.*util.Calc.clear*, FeatA.*util.Calc.enter*]. □

Vice versa, each FSF of LΣ can be embedded into the model NΣ if one introduces one single fixed source name, meaning that all features stem from the same source. Hence both models are equivalent. Using this correspondence, it is straightforward that NΣ also forms a feature algebra. Since it can be embedded into LΣ, we will only use the earlier two models PΣ and LΣ in the remainder of the paper.

5. The Lost Idempotence

As mentioned in the previous sections, distant idempotence (and hence idempotence), i.e., the fact that adding a feature that is already present has no effect, was of central interest for feature algebra. In [7], it is stated that languages and tools for feature combination usually have the idempotence property.

This works fine as long as a feature only contains the name and not its implementation. At the code level this property need not hold any longer. If code is considered, the composition of code pieces has to be defined. Obviously, this definition depends on the specific programming language under consideration. From a programmer's perspective, adding implementation details can be seen as incorporating additional information into the FSF which is then used to define a more sophisticated composition operation. The composition may consist in overriding, replacement, concatenation or merging. Moreover it may act differently on different types of nodes of the corresponding FSF, i.e., while fields and declarations are, for example, replaced, bodies of methods usually are merged. In Section 7 we will give some possible definitions for such a composition operator. In the present section we will look at a simple Java program and discuss some consequences when code fragments are considered in the calculations. Of course we could have used a much more intricate example, but this small foo-example already illustrates the crucial points.

First we have to insert code into FSFs. Later we will also have to model this using extensions PΣ and LΣ. To integrate code into an FST, each terminal node is extended with a code fragment, which may, e.g., just be an interface, but also possibly contain concrete method and field definitions.

Example 5.1. Consider a Java method *add* given by

```
void add(int a) {
  a+=a;

}
```

and the FST at the left of Figure 3. In contrast to Figure 1, where we just used the method's name to identify the node in the FST, we now use the complete method declaration. This allows us to handle overriding in Java. Moreover it makes the model more adequate if also the code level is under consideration. In particular, this FST describes the method `void add(int a)` to be an element of a class *Bar* which itself is part of package *foo*. On the right hand side the extended FST is presented. In detail, the leaf now contains the code. □

To simplify matters we depict the trees in compressed form such as

foo.Bar

```
void add(int a) {
  a+=a;

}
```

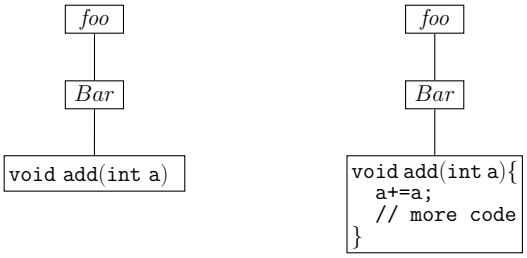


Figure 3: FST and FST equipped with code

As mentioned before, OFA was introduced as a formal treatment of feature orientation. There, if a feature that is already present is added to a program the code parts have to be merged in some way. At the level where the tree leaves are considered as opaque, so that their inner structure does not play any rôle, this can be done by simply ignoring the repeated addition. This is adequately expressed by the axiom of distant idempotence as discussed in the previous section.

Things change as soon one tries to incorporate the inner structure of the leaf nodes into the formal treatment. We will show in this section that then the axiom of distant idempotence in its original form is no longer valid and hence has to be modified. Let us explain the reasons for this.

For instance, in `FeatureHouse` (e.g. [3]), upon a merge duplicate code parts are overwritten or updated. We skip most of the details and only discuss one situation where the special keyword `original` occurs. It can be used in bodies of methods to realise updating and indicates where the bodies are merged. More precisely, assume a method with body m that is updated/overwritten by a body n which contains the keyword `original`. If these two bodies are merged the result is the body of n in which `original` is replaced by the whole body of m. Due to this, `original` is not a new keyword of Java; it only marks the "merging-point" and disappears after composition. We illustrate the situation by an example.

Example 5.2. Assume the following two FSTs:

T_1:

foo.Bar
```
void add(int a) {
  a++;
  original(a);
}
```

T_2:

foo.Bar
```
void add(int a) {
  a+=a;

}
```

Both programs are merged by superimposition, since they are implementations for the same leaf in the corresponding FSF. `FeatureHouse` for example uses this mechanism to merge code. The result of $T_1 \oplus T_2$ is

foo.Bar
```
void add(int a) {
  a++;
  a+=a;
}
```

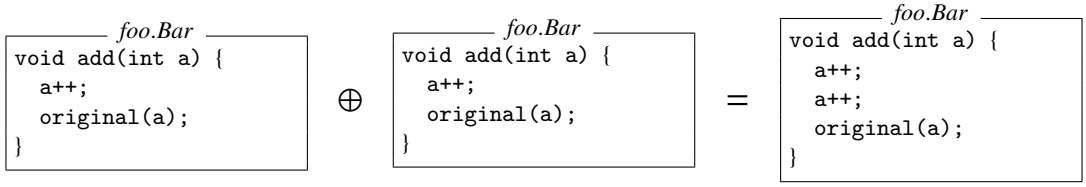
where $\oplus$ denotes the superimposition operator of FSFs by which code fragments are merged as described above. □

This small example already suggests the usefulness of `original`. The necessity in the context of `FeatureHouse` is discussed in [3]. Code merging does not only occur in `FeatureHouse` and Java, but also in other approaches like AspectJ. Here, the keyword `proceed` merges code at previously defined join points. The weaver of AspectJ then merges the respective code fragements.

Most other approaches and tools for feature orientation behave similarly. Therefore it is mandatory for formal methods capturing feature orientation to treat this phenomenon. In this section we present some conclusions from handling code merging. In particular we show that the axiom of distant idempotence is not valid any longer.

To this end, we compose two identical methods containing the keyword `original`.

Example 5.3. Similar to the previous example we get



In particular, the result is not the same as either of the elements on the left, which shows that ⊕ need not be idempotent. □

This short example shows that idempotence is not satisfied in general in the setting of feature orientation. This section provided only a brief description and focussed on some particular parts of `FeatureHouse` and feature algebra. It was not the intention to explain every fact of `FeatureHouse` and feature algebra in detail. The interested reader is referred to [3, 6].

6. Extended Feature Algebra

We have shown that the axiom of distant idempotence and hence also standard idempotence may not hold when arguing at code level. In the remainder of the paper we present a possible solution for this problem. To model code-level behaviour at an abstract level we extend feature algebra by a third sort C of *code fragments* (next to introductions and modifications). This yields an extended structure. The main idea is to consider pairs (i, c), where i is an atomic introduction (corresponding to a maximal path in the forest under consideration) and c is the code fragment contained in the leaf at the tip of that path. We denote (i, c) by $i[c]$. The set of all these pairs is denoted by $I[C]$.

Example 6.1. The FST and the extended FST for the method *add* of *foo.Bar* have already been given in Section 3. The corresponding algebraic expressions over PΣ or LΣ are

$$\{foo,\ foo.Bar,\ foo.Bar.\texttt{void add(int a)}\} \quad \text{and} \quad [foo.Bar.\texttt{void add(int a)}]\ ,$$

respectively. Based on the latter list interpretation, the representation in the extended structure is now the following singleton list with an introduction/code pair:

$$([foo.Bar.\texttt{void add(int a)}],\ code_{add})\ .$$

Here $code_{add}$ is the complete code for `add`. □

These extended structures motivate an extension of feature algebra.

Definition 6.2. An *extended feature algebra (EFA)* is a tuple $(M, I, C, E, +, \circ, \cdot, \triangleright, \mathbf{0}, \mathbf{1})$ such that the following properties hold for all $m, n \in M$, $i, j \in I$, $x \in E$ and $a, b \in C$.

- $(C, \triangleright)$ is a semigroup of *code fragments* in which $\triangleright$ is an update or override operation;
- $(E, +, \mathbf{0})$ is a monoid of *extended introductions* with $I[C] \subseteq E$ satisfying $i[a] + x + i[b] = x + i[a \triangleright b]$;
- $(M, \circ, \mathbf{1})$ is a groupoid of *modifications* operating via $\cdot$ on $I[C]$;
- $\mathbf{0}$ is a right-annihilator for $\cdot$ and
- $\cdot$ distributes over $+$.

The operation $\triangleright$ can be seen as an update. In the previous section, $\triangleright$ merged code fragments. In the next section we will discuss this operation in our concrete models. Note that we have modified the axiom of distant idempotence: adding a feature a second time updates the earlier instance of that feature rather than just ignoring it. The overall structure remains the same. In particular, the original definition of a feature algebra can be retrieved by choosing an idempotent update operator, i.e., an operator $\triangleright$ with $a \triangleright a = a$. This is automatic when C is taken C as a set containing only one single element (behaving like the empty code fragment). One might ask why the code fragments are only required to form a semigroup and not a monoid. As we will show later, the existence of a neutral element cannot be guaranteed at code level; even worse, a neutral element creates problems when defining the operator $\triangleright$ as complete overriding.

Lemma 6.3. *Assume an extended feature algebra in which every element of E is a finite sum of elements from $I[C]$. If the update operator satisfies $a \triangleright a = a$ then the structure $(M, E, +, \circ, \cdot, \mathbf{0}, \mathbf{1})$ is a feature algebra.*

Proof. All but the axiom of distant idempotence are immediate from the definition. For the distant idempotence we get $i[a] + x + i[a] = x + i[a \triangleright a] = j[c] + i[a]$. □

The presented models for OFA immediately give models for EFA. They can either be used in their original form or be equipped with implementation details. Since we already discussed how FSFs are extended by code, the analogous extension of the models PΣ and LΣ is straightforward. Again we use pairs (i, c), where i is an introduction corresponding to a maximal path in the forest under consideration (as before) and c is the code fragment. Now the code is not contained in the leaf, it is appended at the end of each maximal string of names. The operations for superimposition are modified in a straightforward way.

Example 6.4. Assume that $a \triangleright b$ represents overriding when the keyword `original` does not occur and merging otherwise. Then we are able to express Examples 5.2 and 5.3 algebraically using EFA. For example, using the element $([foo.Bar.\texttt{void add(int a)}], code_{add})$ of the previous example now yields

$$\begin{aligned}
&([foo.Bar.\texttt{void add(int a)}], code_{add}) + ([foo.Bar.\texttt{void add(int a)}], code_{add}) \\
&= ([foo.Bar.\texttt{void add(int a)}], code_{add} \triangleright code_{add}) \\
&= ([foo.Bar.\texttt{void add(int a)}], code'_{add}) \,,
\end{aligned}$$

where $code'_{add}$ is given by

```
void add(int a) {
  a++;
  a++;
  original(a);
}
```

□

Let us now discuss some consequences of EFA. Unfortunately, we cannot define a natural preorder on an extended feature algebra. This is due to the lack of (distant) idempotence. Hence the counterpart of Lemma 3.5 reduces to

Lemma 6.5. *Assume x to be an extended introduction and m, n, o to be modifications of an extended feature algebra. Then*

(1) $(m \circ (n \circ o)) \cdot x = ((m \circ n) \circ o) \cdot x$,
(2) $(m \circ 1) \cdot x = (1 \circ m) \cdot x = m \cdot x$.

The proofs similar to the one of Lemma 3.5. To overcome the deficiency of the missing preorder we can define two different relations.

Definition 6.6 (left/right natural preorder). On an extended feature algebra we define the *right natural preorder* as

$$x \leq_r y \;\Leftrightarrow_{df}\; \exists z \in E : x + z = y \,.$$

Similarly, the *left natural preorder* is defined as $x \leq_l y \;\Leftrightarrow_{df}\; \exists z \in E : z + x = y$.

If + is interpreted as superimposition, both preorders have different meanings. Informally, the right natural preorder requires the existence of a "base element" (z in the definition) to which x is added, that means the atomic introductions in the base element are updated by the corresponding ones in x (if any), whereas the additional features of x are just added. Conversely, the left natural preorder describes the situation where x can be transformed into y by a suitable "update".

Example 6.7. Assume the two element T_1 and T_2 of Example 5.2. From $T_2 \oplus T_1 = T_2$, we get $T_1 \leq_l T_2$. Moreover it is easy to see that for an arbitrary implementations T of $foo.Bar.$`void add(int a)`, the merged code $T_1 \oplus T$ contains `a++` as first line. Therefore $T_1 \not\leq_r T_2$. □

Lemma 6.8. *Assume extended introductions x, y, z and let $\leq_{lr}$, $lr \in \{l, r\}$ be an abbreviation for $\leq_l$ and $\leq_r$. ($\leq_{lr}$ stands for the same preorder if it occurs more than once in a formula.)*

(a) $0 \leq_{lr} x$.
(b) $\leq_{lr}$ *are preorders, i.e.,* $x \leq_{lr} x$ *and* $x \leq_{lr} y \wedge y \leq_{lr} z \Rightarrow x \leq_{lr} z$.
(c) $x \leq_r x + y$ *and* $y \leq_l x + y$.

The properties ($i \leq_{lr} 0 \Rightarrow i = 0$), $x \leq_l x + y$ and $y \leq_r x + y$ do not hold in general.

To get a better understanding we now have a look at the introduced preorder in the setting of EFAs where the axiom of distant idempotence holds (for example in OFAs).

Lemma 6.9. *Assume an EFA that satisfies the axiom of distant idempotence. For arbitrary extended introductions x, y the following holds.*

(a) $\leq$ *coincides with* $\leq_r$*, i.e.,* $x \leq y \Leftrightarrow x \leq_r y$.
(b) $\leq_l$ *refines* $\leq_r$*, i.e.,* $x \leq_l y \Rightarrow x \leq_r y$ *(and therefore by* (a) *also* $\leq$*).*
(c) $\leq_l$ *coincides with* $\leq_r$ *if and only if* $+$ *is commutative.*

The proof is given in a more general setting of an arbitrary monoid in the appendix (see Lemma 6.9).

This lemma points out the interaction of both preorders. In particular it shows that $\leq_r$ is the proper generalisation of the subsumption order of the introduction inclusion.

7. The Update Operator

In this section, we discuss the update operator $\triangleright$ in concrete situations. In particular, we present some consequences if $\triangleright$ satisfies certain additional conditions.

In the most easiest case the update operator $\triangleright$ overrides everything. Informally, that means that if a feature is added that is already there, the content of the feature (e.g. code) of that present feature is lost and only the details of the newly added feature is preserved.

Example 7.1. Using the code fragments of Example 5.2, we have

```
void add(int a) {
  a++;
  original(a);
}
```
$\triangleright$
```
void add(int a) {
  a+=a;

}
```
$=$
```
void add(int a) {
  a++;
  original(a);
}
```
□

Mathematically, that means such an update operator can be defined as

$$a \triangleright b =_{df} a\,.$$

In feature orientation overriding occurs quite often. For example, `FeatureHouse` is mainly based on overriding when performing superimposition [3]. In AspectJ overriding can occur if the advice `around` is used. In the case of overriding the update operator $\triangleright$ is idempotent. Hence, by Lemma 6.3 both extended models, PΣ and LΣ (with additional code fragments), form feature algebras if the underlying alphabet Σ is finite.

As a consequence of overriding we see that the update operator need not have a neutral element. If we would assume $(C, \triangleright, 1_\triangleright)$ to be a monoid rather than a semigroup then the law $1_\triangleright \triangleright a = 1_\triangleright$ would hold. From that we would have $a = 1_\triangleright \triangleright a = 1_\triangleright$. Hence C would only contain a single element. Since this contradicts the intention of an extended feature algebra we only assume $(C, \triangleright)$ to be a semigroup in the definition. However in some situations there might be a neutral element, as we will see later.

Similar to overriding one can define preservation by $a \triangleright b =_{df} b$. Instead of overriding the "old" code, it is now preserved and the new one is ignored. While overriding forbids a left neutral element, this definition forbids the existence of a right neutral element by similar arguments.

Next, we will discuss the case where the operator $\triangleright$ is a real update operator. It shall overwrite those parts which are already in the feature and adds those parts which are new. To identify the parts that have to be overridden, we need to identify the "common part" of two given implementations of the same feature oriented program. Based on the common part one can determine which part of a method body has to be overridden and which part has to be preserved. Of course these calculations highly depend on the respective language and have to follow exact rules. In Java, for example, `FeatureHouse` simply overrides declarations and functions as long as the keyword `original` does not occur in the code.[2] For a detailed description we refer again to [3].

To model this kind of behaviour we define *abstract interfaces* for each code fragment such as a method in Java. Whereas a general Java element may contain arbitrary (legal) programming constructs, abstract interfaces contain only the types of the corresponding Java parts and "forget" the remaining bodies, initialisations etc. We illustrate this by an example.

Example 7.2. On the left hand side there is a simple Java method while its abstract interface appears on the right hand side.

```
int min5(int a) {
  int b=5;
  if(a<b) return a;
  else return b;
}
```

```
int min5(int a) {
  int b;
}
```

The typing of the local variable `b` appears, since its declaration may be overwritten during a feature combination. □

A precise definition of the abstract interface will need to reflect also nested scopes etc. Note that the use of abstract interfaces may yield invalid Java code (e.g., `return` statements are omitted). This does not matter, though, since the notion will only be employed to identify the "common parts" upon updates.

Abstract interfaces can be defined quite similarly for other feature-oriented languages like Scala, FeatureC++, Lightweight Feature Java or even XML.

Example 7.3. In Scala the simple program of Example 7.2 and its corresponding interface read

```
def min5(a:Int):Int = {
  val b = 5
  if(a<b) a else b
}
```

```
def min5(a:Int):Int = {
  val b : Int
}
```

A particular concept of Scala is type inference. This means that Scala can determine the type of an expression like `val b=5`. It realises that this line is equivalent to `val b : Int = 5`.

XML is even simpler. Since the tags are nodes in the corresponding FSF, the leaf information is just plain text that does not contain any further information. Hence forming the abstract interface can be seen as "forgetting the content". Only the overall structure is kept.

[2] There are some exceptions.

```
<CATALOG>
  <CD>
    <TITLE>
        Best Recordings 1927-1939
    </TITLE>
    <ARTIST>Comedian Harmonists</ARTIST>
    <COUNTRY>Germany</COUNTRY>
    <COMPANY>Gp Serie</COMPANY>
    <PRICE>10.98</PRICE>
    <YEAR>2008</YEAR>
  </CD>
</CATALOG>
```

```
<CATALOG>
  <CD>
    <TITLE></TITLE>
    <ARTIST></ARTIST>
    <COUNTRY></COUNTRY>
    <COMPANY></COMPANY>
    <PRICE></PRICE>
    <YEAR></YEAR>
  </CD>
</CATALOG>
```

□

Now we look at an characterisation of the abstract interface in the concrete model. Let again C be the set of possible code fragments and $T \subseteq C$ the set of the corresponding abstract interfaces. The function that determines the abstract interface for a given Java code is denoted by $face : C \to T$. Next we define two functions

$$\begin{aligned} \lfloor, - &: \mathcal{P}(C) \times \mathcal{P}(T) \to \mathcal{P}(C) \\ X \lfloor U &= \{x \in X \,:\, face(x) \in U\} \\ X - U &= \{x \in X \,:\, face(x) \notin U\} \,. \end{aligned}$$

The restriction operator $\lfloor$ determines for a set X of code fragments the ones whose corresponding abstract interfaces lie in the given subset $U \subseteq T$, while the operator $-$ removes all these.

Example 7.4. Looking at Java, the set C of all possible code fragments is nearly infinite. However, all code parts presented so far, are elements of C. The abstract interface given in Example 7.2 is obviously an element of T. □

To define the update function $\triangleright$ we need to lift the function $face$ to sets of code fragments by

$$face(X) =_{df} \{face(x) \,:\, x \in X\} \,.$$

Then the update operator $\triangleright$ can be defined by

$$X \triangleright Y =_{df} (Y - face(X)) \cup X \,.$$

This means that all "old" definitions of elements in Y that are redefined in X are discarded and replaced by the ones in X; moreover, all elements of X not mentioned in Y are added. It should be noted that $face$ and $\triangleright$ are closely related to the interface operator $\uparrow$ and the asymmetric composition $\&*$ in the DEEP calculus [34].

From the definitions, it is straightforward to prove the following properties. We will use them for an abstract characterisation of the update operator in the next section.

Lemma 7.5. *Let X, Y be sets of code fragments and U, V abstract interfaces.*

(a) $\lfloor$ *distributes over* $\cup$*, that is* $(X \cup Y) \lfloor U = X \lfloor U \cup Y \lfloor U$ *and* $X \lfloor (U \cup V) = X \lfloor U \cup X \lfloor V$.
(b) *The operations* $\lfloor$ *and* $face$ *are isotone w.r.t.* $\subseteq$.
(c) $\emptyset$ *is an annihilator w.r.t.* $\lfloor$*, i.e.,* $X \lfloor \emptyset = \emptyset$ *and* $\emptyset \lfloor U = \emptyset$.
(d) *The distributivity and exchange laws* $(X \cup Z) - U = (X - U) \cup (Z - U)$ *and* $X - (U \cup V) = (X - U) - V$ *hold.*
(e) $\emptyset$ *is a right neutral element and a left annihilator of* $-$*, i.e.,* $X - \emptyset = X$ *and* $\emptyset - X = \emptyset$.

These properties can intuitively be understood as follows. The first part of (a) means that restriction works modularly by inspecting both parts of a union separately and uniting the results. The second part of (a) is a dual modularity property for reducing more complex selections to simpler ones. Part (b) states that an increase in code leads to an increase in selected code and interface. Part (c) expresses that restricting to nothing or selecting from nothing leaves nothing. Part (d) is the analogue of Part (a) for the removal operation. Part (e) expresses that removing nothing preserves the respective code while removing anything from nothing leaves nothing.

Depending on the concrete model it is also possible to combine different update operators. One can for example use the operator that models overriding for declarations, fields etc. and the update, we just discussed, for merging methods. However this would require the type of nodes in the corresponding FSF which we have not considered for the present paper. Hence interaction of different update operators for different types is part of our future work, but we do not see any difficulties at the moment (see Section 11).

8. An Algebraic Characterisation of Interfaces

It turns out that the rather concrete definition of an abstract interface $face$ presented in the last section can be lifted to the same completely language-independent level of abstraction as that of our (extended) feature algebra. Again, this opens the possibility for automated verification. We will give this abstract definition and discuss several consequences. It is straightforward to "translate" the abstract properties back to the more concrete level of the previous section.

We first characterise the restriction operator $\lfloor$ and the operator $-$ purely algebraically. The definition is mainly motivated by Lemma 7.5 where essential properties for these operators are given.

We now assume the set C of code fragments to be a semilattice L (e.g. [22]) with least element 0_C. A semilattice has a binary operator $\sqcup$, called *join*, and, based on that, an order $\sqsubseteq$, defined like the subsumption relation by $a \sqsubseteq b \Leftrightarrow_{df} a \sqcup b = b$. The operator $\sqcup$ forms the supremum (or least upper bound) in L and can be viewed as yielding the union of two code fragments. The order $\sqsubseteq$ corresponds to the inclusion order on sets. The least element 0_C can be thought of as the empty code fragment. The set T of abstract interfaces is abstracted to a sublattice N of L with $0_C \in N$.

Definition 8.1 (restriction and complement). The *restriction* $\lfloor : L \times N \to L$ and the *complement* $- : L \times N \to L$ have to satisfy the following axioms:

$$(a \sqcup b)\lfloor p = (a\lfloor p) \sqcup (b\lfloor p) \quad (1) \qquad a = (a\lfloor p) \sqcup (a - p) \quad (6)$$

$$a\lfloor(p \sqcup q) = (a\lfloor p) \sqcup (a\lfloor q) \quad (2) \qquad a\lfloor(q - p) = (a - p)\lfloor q = (a\lfloor q) - p \quad (7)$$

$$(a \sqcup b) - p = (a - p) \sqcup (b - p) \quad (3) \qquad p - p = 0_C \quad (8)$$

$$a - (p \sqcup q) = (a - p) - q \quad (4) \qquad p\lfloor q \sqsubseteq q \quad (9)$$

$$a\lfloor 0_C = 0_C \quad (5)$$

where $a, b \in L$, $p, q \in N$. Moreover we assume that N is closed under $\lfloor$ and $-$, i.e.,if $p, q \in N$ then also $p\lfloor q, p - q \in N$.

Using first-order automated theorem provers and counter example search (e.g. Prover9/Mace4 [40]) we have shown that the axioms are minimal. This means that none of the axioms follows from the others. Equivalent characterisations may exist (maybe smaller in the number of axioms); however, the meaning of the above axioms is easy to understand on the basis of the explanations following Lemma 7.5, and therefore we deem the axiomatisation quite intuitive.

We list a couple of properties for the newly introduced operators. For example Equations (6) and (8) imply $p\lfloor p = p$. We only present the most important properties; many more can be found in [33]. Proofs by hand are given in Appendix B. Note that we can employ some useful properties induced by the semilattice; examples are $a \sqsubseteq 0_C \Rightarrow a = 0_C$ and

$$a \sqcup b \sqsubseteq c \Leftrightarrow a \sqsubseteq c \wedge b \sqsubseteq c \quad (10)$$

A further one is that each equation can be split into inequalities, i.e., $a = b \Leftrightarrow (a \sqsubseteq b \wedge b \sqsubseteq a)$.

Lemma 8.2. *Assume arbitrary code fragments $a, b \in L$ and $p, q \in N$.*

(a) $a\lfloor p \sqsubseteq a$. *In particular* $0_C\lfloor p = 0_C$.
(b) $a - p \sqsubseteq a$. *In particular* $0_C - p = 0_C$. *Moreover* $a - 0_C = a$.
(c) *Restriction is isotone in both arguments, i.e.,* $a \sqsubseteq b \Rightarrow a\lfloor p \sqsubseteq b\lfloor p$ *and* $p \sqsubseteq q \Rightarrow a\lfloor p \sqsubseteq a\lfloor q$.
(d) *The operator $-$ is isotone in its left and antitone in its right argument, i.e.,* $a \sqsubseteq b \Rightarrow a - p \sqsubseteq b - p$ *and* $p \sqsubseteq q \Rightarrow a - q \sqsubseteq a - p$,
(e) *Restriction is quasi-associative, i.e.,* $a\lfloor(p\lfloor q) = (a\lfloor p)\lfloor q$.
(f) *On interfaces the operator $-$ behaves like relative complementation (or set difference), and hence satisfies the shunting rule* $p - q \sqsubseteq r \Leftrightarrow p \sqsubseteq q \sqcup r$.

(g) *The restriction $p \lfloor q$ is the greatest lower bound (i.e., the largest common part) of p and q in N. Hence also $p \lfloor q = q \lfloor p$.*

Part (a) means that $\lfloor$ really restricts an element a, i.e., the restriction of a to p is contained in a. The same holds for the operator $-$ which is formalised in Part (b). The following two items show that quite natural monotonicity properties hold. Parts (e) and (f) are useful properties that will be used to derive properties presented in the sequel of the paper.

Next we show some important formulas describing the interplay between the both operators.

Lemma 8.3. *We assume again arbitrary code fragments $a, b \in L$ and $p, q \in N$.*

(a) $(a - p) \lfloor p = 0_C$ *and* $(a \lfloor p) - p = 0_C$.
(b) $a \sqsubseteq b - p \;\Leftrightarrow\; a \sqsubseteq b \wedge a \lfloor p = 0_C$.
(c) $a \sqsubseteq b \lfloor p \;\Leftrightarrow\; a \sqsubseteq b \wedge a - p = 0_C$.

Part (a) presents annihilation laws. If we subtract from a anything related to p and then restrict exactly to these parts, nothing will remain. In the second equation, a is first restricted. Parts (b) and (c) express that same phenomenon in a way that is more suitable for further proofs.

Finally, as a simple consequence of Lemma 8.3 (c) and the shunting rule (Lemma 8.2)(f) we get

$$p \sqsubseteq q \;\Leftrightarrow\; p \sqsubseteq p \lfloor q \;\Leftrightarrow\; p = p \lfloor q \,. \tag{11}$$

Now we come to the axiomatic characterisation of the abstract interface function $face$. The key is the observation that $face(X)$ is the least set that leaves X unchanged under the restriction operation $\lfloor$:

$$face(X) \subseteq U \;\Leftrightarrow\; X = X \lfloor U \,,$$

where X is a code fragment and U an abstract interface as in Section 7. In fact, since $X \lfloor U \subseteq U$ holds anyway by definition, this can be relaxed to $X \subseteq X \lfloor U \;\Leftrightarrow\; face(X) \subseteq U$. Informally, "least" can be interpreted as the fact that the interface has to contain "enough" or "all necessary" information (e.g. all variables of X should be have a counterpart in $face(X)$) but not more (e.g., declarations of variables that do not occur in X should not be in the interface). In that sense the condition $X = X \lfloor U$ can also be viewed as a typing assertion saying that X implements the interface U faithfully. This leads to the following algebraic characterisation of the abstract interface.

Definition 8.4 (abstract interface on L). The *abstract interface* F on L is axiomatised by

$$\mathrm{F}(a) \sqsubseteq p \;\Leftrightarrow_{df}\; a \sqsubseteq a \lfloor p \,,$$

where $a \in L$, $p \in N$.

The function F behaves in many respects like the abstract codomain operator introduced by Desharnais, Möller and Struth [24]. This correspondence allows us to re-use a large body of well-known theory — another advantage of an abstract algebraic view.

As before we present some useful and meaningful consequences of this definition; further properties can again be found in [33]. The first two are immediate from the definition.

Corollary 8.5. *For code fragments $a, b \in L$ and $p, q \in N$ we have $a \sqsubseteq a \lfloor \mathrm{F}(a)$ and $\mathrm{F}(a \lfloor p) \sqsubseteq p$. In particular $\mathrm{F}(0_C) = 0_C$.*

Lemma 8.6. *Assume arbitrary code fragments $a, b \in L$ and $p, q \in N$.*

(a) *The abstract interface is additive, i.e, $\mathrm{F}(a) \sqcup \mathrm{F}(b) = \mathrm{F}(a \sqcup b)$. In particular F is isotone.*
(b) $\mathrm{F}(p) = p$. *In particular,* $\mathrm{F}(\mathrm{F}(a)) = \mathrm{F}(a)$.
(c) $a = a \lfloor \mathrm{F}(a)$.
(d) $\mathrm{F}(a - p) = \mathrm{F}(a) - p$ *and* $\mathrm{F}(a \lfloor p) = \mathrm{F}(a) \lfloor p$.

Part (a) again is a distributivity property. Part (b) means that an abstract interface cannot be abstracted further, since it is abstract already. Part (c) means that the abstract interface of a is no more abstract than necessary: the full a is preserved when restricting it to its abstract interface. Part (d) are import/export laws for bringing extra removals/restrictions in and out of the abstract interface function.

Finally we can characterise the update operator in this abstract setting as follows.

Definition 8.7 (update operator). The *abstract update operator* can now be defined by

$$a \rhd b \;=_{df}\; (b - \mathrm{F}(a)) \sqcup a \;.$$

We call a and b *compatible* if they agree on the common part of their interfaces, i.e., if

$$a \lfloor (\mathrm{F}(a) \lfloor \mathrm{F}(b)) = b \lfloor (\mathrm{F}(a) \lfloor \mathrm{F}(b)) \;.$$

As mentioned before, the abstract interface function behaves in many respects like the abstract codomain operator of [24]. Moreover, the update operator is identical to the one of Ehm introduced for pointer structures [26] (except that we replace the codomain operator by our interface). Hence we can re-use the theory and get, among others, the following properties of update for free. For the proofs and further laws we refer to earlier work of Möller and Ehm [41, 26].

Corollary 8.8. *If a, b, c are arbitrary code fragments, then*

(a) $a = a \rhd 0_C = 0_C \rhd a$.
(b) $(a \rhd b) \rhd c = a \rhd (b \rhd c)$
(c) *The following properties are equivalent:*
- *a and b are compatible.*
- $a \lfloor \mathrm{F}(b) = b \lfloor \mathrm{F}(a)$.
- $a \rhd b = a \sqcup b$,
- $a \rhd b = b \rhd a$.

(d) *If a and b are compatible then* $(a \sqcup b) \rhd c = a \rhd (b \rhd c)$.
(e) *If a and b are compatible and $a \rhd b = b$ then* $(a \sqcup b) \rhd c = a \rhd c$.
(f) *If* $\mathrm{F}(a) \lfloor \mathrm{F}(b) = 0_C$ *then* $a \rhd (b \sqcup c) = b \sqcup (a \rhd c)$.

The first two items show that $(L, \rhd, 0_C)$ forms a monoid. In particular there is a neutral element w.r.t. the update operator. In the concrete model this neutral element corresponds to some code fragment without any content. Part (c) means that compatibility is equivalent to the fact that update and join coincide. Part (d) is a sequentialisation property and says that a complex update may also be done by two simpler overwritings if the updates are compatible. Part (e) says that part of an update may be skipped if it would add something that is already present.Part (f) states localisation property, viz. that an update need only be applied to that part of a composition it actually affects.

In this section we have shown how the abstract interface and the update operator, first defined for a concrete model of EFA, can be lifted to an abstract, purely algebraic setting. We only gave the main ideas and derived some few meaningful properties. More examples as well as a further analysis of the underlying algebraic structure can be found in [33].

9. Discussion

We have presented an extension of feature algebra. The original algebra was introduced by Apel et al. in [6]. There the axioms of feature algebra were given and the standard model based on FSFs are defined. Essentially, an FSF is a "stripped-down abstract syntax tree" [3]. In the leaves usually the features' implementations are given. Although tools for feature orientation work at code level, the algebra does not reflect this part. This is the reason why the present paper extends the algebra. A first step in that direction was already done in preliminary work [32]. In that paper we identified the problem of the lost idempotence and gave a possible solution, which is enhanced in the present paper.

The axioms are justified in a concrete model, and many more consequences are discussed. In particular, the idea of abstract interfaces is investigated. Finally, the present work discusses the relationship between the presented algebra and feature orientation in more detail.

Instead of an extension, one might consider other possibilities. For example one could use complete abstract syntax trees (ASTs) instead of FSFs. These trees contain all details of the implementation. In particular, nodes there would be single statements or even the single components of a statement. For example, `a=a+b` would yield a (sub)tree with 5 nodes. ASTs describe the whole structure of a given program. However we are not aware of any update or merging mechanism like the ones of FSFs. Such a merging operation would be hard to characterise, since first the matching and merge points have to be identified. On FSFs this task is assigned to the programmer. A programmer adds additional information so that when an FST is superimposed with another FST, that information "drives" the composition. As we have seen, adding information (e.g. the keyword `original`) yields the described behaviour. Since we are not aware of operations operating on abstract syntax trees, we are also not aware of any algebraic counterparts.

Another solution for the identified problem could be the restriction of superimposition rather than the extension of the algebra. One can restrict the superimposition only to those cases where no merging occurs. Then superimposition could only insert new code (new nodes). Changing the nodes' contents always leads to the risk and the possibility of merging operations. However, modifications could change code. Using this approach it would be possible to work with EFA. From a programmer's perspective this is not practicable. A user has to compose two features. The reason why features can be composed are various and well known. For example different features can be developed independently and then merged into a final product; (multi-dimensional) separation of concerns also yields a separation of features that have to be matched at the end. Composition is usually achieved by superimposition and not by modification, since modification only changes an existing feature. Moreover it has been shown in many case studies that merging is a necessary and useful concept: as we have seen, merging can occur on FSFs. The example of Figure 2 can be extended to present inlining which is essentially merging [7]. This example is recapitulated by Figure 4. Next to inlining, other

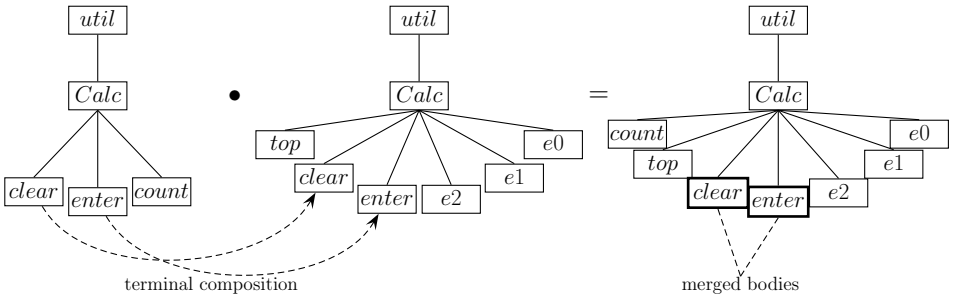


Figure 4: Superimposing Java Methods ([7])

superimposition rules for terminals, such as wrapping, are proposed (e.g. [13, 28]). In other languages similar rules for updating code can be defined (e.g. [3]). It has been shown that superimposition can also be used for a component adaptation technique [17]. There, concepts like wrapping are used and therefore lead again to the discussed problem.

Although all examples of the paper use Java, the algebra still remains language independent. It has been shown and discussed that the original algebra is language independent [6, 7] and that the original feature algebra is a uniform foundation for several languages supporting feature orientation and feature composition [42, 16, 4]. Our extension introduces a further sort C with a corresponding update operator into the algebra; this new sort is not restricted at all. Therefore it can be Java code, like in the paper, an arbitrary other programming language or plain text like in XML-tags. The language dependence is encapsulated into the abstract-interface operation F.

The problems at code level are solved in the following way. When a feature (method) is updated by a feature with the same name, and the code fragments are merged, then the two features interact. Therefore the update operator is closely related to feature interaction (e.g. [19]). The proposed algebra is not intended to solve the feature interaction problem. It covers the problem only to the extent what FSFs in combination with composition techniques like weaving,

inlining etc. can achieve.

10. Related Work

The authors' previous work on feature algebra and FSFs has already been discussed in the previous section and throughout the paper. Together with Khedri, the authors also developed an algebra for expressing variability of software product lines w.r.t. features [30] and parts of feature interaction [31]. Originally that algebraic approach only focussed on "syntactic" models. This is currently extended to encompass semantics as, e.g., provided by implementation details and code. It is possible to combine that approach with feature algebra to construct a feature-based product family algebra; this is the subject of forthcoming papers.

Tools that mainly build on the original axioms of feature algebra are the FSTComposer [5] and `FeatureHouse` [3]. The former tool is an implementation of the AHEAD [13] model. As discussed above, the latter tool is a general approach to the composition of software artifacts.

There are only few algebraic approaches to feature orientation by other researchers. Ramalingam and Reps developed modification algebras [44, 43]. These algebras allow reasoning about program changes. The motivation of that work was the formalisation of version control systems. Of course, this is closely related to our work. Updating a repository can be seen as a modification in the sense of feature algebra; merging two independently developed branches of the same file corresponds to merging (feature composition). The modification algebra behaves similar to feature algebra, since axioms like associativity and distant idempotence are also postulated. However, our algebras (EFA and OFA) are abstracted to a level where they not only cover revision control [8], but also state-of-the-art programming languages like feature orientation.

Other algebraic approaches are based on finite maps and simple lattice theory. In [25] a formal model of traits has been developed. In that model traits can be composed, either to form other traits, or to form classes. That algebra unifies the existing schemes based on single inheritance, multiple inheritance, or mixins; all pose numerous problems for reuse in an abstract way. In [15], Batory and Smith introduce two algebras that unify different software composition models of contemporary concern-based tools and programming languages. The first equates concerns with finite maps; the second and more general algebra uses quarks, a construct that represents both FM expressions and their transformations, to express concepts in aspect-orientation. Both approaches uses a flat lattice to characterise inconsistency and error states occurring in the composition of two features. This mechanism allows the composition of code. However, the merging mechanism is usually not lifted to an abstract level. That means that rules for merging has to be developed for different tools and languages separately. In contrast to that, the update operator in the present paper abstracts and subsumes even merging. A related definition of an update-like operator and a corresponding abstract interface for Lightweight Feature Java can be found in [23].

Non-algebraic approaches for feature orientation include the DEEP [34] calculus as well as its generalisation gDEEP [2]. Both calculi provide mechanisms for feature composition. The connection to feature algebra has already been presented in Section 3. Moreover there exist many calculi supporting features or feature-like structure including composition techniques (e.g., [20, 27, 35]). It has been shown that these techniques are closely related to superimposition [5, 7] and hence also to feature algebra and the update operator. However, these calculi are usually developed w.r.t. a specific programming language (most often Java-like).

In sum, feature algebra and hence also its extension is a theoretical foundation for feature foundation. This has been demonstrated within this section and in previous papers. Due to its abstract and uniform character it forms the base for treating nearly all languages, tools and mechanisms for feature orientation. Therefore one could mention much more related work. However we restricted ourselves to a short overview, pointing in different directions.

11. Conclusion and Outlook

The present paper is based on [6] where the original version of feature algebra was introduced. There a formal model — the feature algebra — was introduced that is intended to capture the commonalities of feature oriented software development such as introductions, refinements and quantification. Moreover it abstracts from differences of minor importance.

A main concept in feature orientation (feature oriented software development and feature oriented programming) are feature structure forests. Such forests capture the essence of an artefact's modular structure in the form of a

tree. From a mathematical point of view feature structure forests are just labelled forests. Based on these forests, we presented three models for feature algebra. These models demonstrate that there are more than one (meaningful) interpretation of the axiom of feature algebra. Moreover the list-based model may be used to encode and to perform super imposition in an efficient way.

Based on that we showed that these concrete models and therefore also the abstract algebra are fine as long as one does not consider feature oriented programming at code level. However, when working on concrete problems in the area of feature orientation, this additional level has to be under consideration since the goal should be to argue about feature oriented programs. If the level of code is considered, not all aspects of feature orientation can be modelled. In the present paper we have demonstrated it with the help of a simple "foo-bar"-example. Obviously, this example does not respect real feature programs, but it focuses on some crucial problems. In particular we have shown that the axiom of distant idempotence is not valid and has to be modified. To remedy this, we have introduced the structure of an extended feature algebra. This structure can model feature oriented programming at code level in an abstract manner. In particular an extended feature algebra generalises the original feature algebra. To clarify the idea we have also extended the introduced models correspondingly.

Finally we presented some possibilities how an update operator may be characterised. In one of the easiest cases, this operator models overriding. In a more complicated setting it should be model "real" update, i.e., bodies should for example be merged. The update operator was then lifted to the same abstract and algebraic level as the feature algebra. These operations show that the introduced algebra is able to handle nearly all aspects of feature oriented software development. In particular it underlines the strength of an abstract algebraic approach is a further step towards a complete algebraic description .

As future work we will apply the extended feature algebra to some real case studies. This will show whether the extension adequately covers the essential properties of feature orientation. To achieve this goal further investigations of the update operator are needed. First, more properties need to be derived for the introduced operators. For example, the merging of code fragments has to be investigated in much more detail. This is particular needed since in this paper we only presented fundamental concepts for modelling this operator. The main topic was the introduction of the extended algebraic structure. Second, one might need more update operators. So far we have presented one operator that models overriding, one for preserving and another for updating. Analysing real case studies might yield another type of operator.

Acknowledgement. We are grateful to Sven Mentl, Han-Hing Dang, Roland Glück and the anonymous reviewers for helpful comments.

References

[1] S. Apel. *The Role of Features and Aspects in Software Development*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, 2007.

[2] S. Apel and D. Hutchins. A calculus for uniform feature composition. *ACM Transactions on Programming Languages and Systems*, 32(5):1–33, 2010.

[3] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-independent, automated software composition. In *31th International Conerence on Software Engineering(ICSE)*, pages 221–231. IEEE Press, 2009.

[4] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In R. Glück and M. Lowry, editors, *Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2005.

[5] S. Apel and C. Lengauer. Superimposition: A language-independent approach to software composition. In C. Pautasso and É. Tanter, editors, *Software Composition*, volume 4954 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2008.

[6] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebra for features and feature composition. In *AMAST 2008: Proceedings of the 12th international conference on Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2008.

[7] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming*, 75(11):1022–1047, 2010.

[8] S. Apel, J. Liebig, C. Lengauer, C. Kästner, and W. Cook. Semistructured merge in revision control systems. In D. Benavides, D. Batory, and P. Grünbacher, editors, *Workshop on Variability Modelling of Software-Intensive Systems*, volume 37 of *ICB-Research Report*, pages 13–19. Universität Duisburg-Essen, 2010.

[9] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

[10] D. Batory. Feature-oriented programming and the AHEAD tool suite. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 702–703. IEEE Press, 2004.

[11] D. Batory. From implementation to theory in product synthesis. *ACM SIGPLAN Notices*, 42(1):135–136, 2007.

[12] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions Software Engineering and Methodology*, 1(4):355–398, 1992.

[13] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197. Proceedings of the IEEE, 2003.

[14] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. *ACM SIGSOFT Software Engineering Notes*, 18(5):191–199, 1993.

[15] D. Batory and D. Smith. Finite map spaces and quarks: Algebras of programme structure. Technical Report TR07-66, University of Texas, 2007.

[16] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/j: Controlling the scope of change in java. *ACM SIGPLAN Notices*, 40(10):177–189, 2005.

[17] J. Bosch. Superimposition: A component adaptation technique. *Information & Software Technology*, 41(5):257–273, 1999.

[18] A. Buch, T. Hillenbrand, and R. Fettig. Waldmeister: High Performance Equational Theorem Proving. In J. Calmet and C. Limongelli, editors, *Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems*, number 1128 in Lecture Notes in Computer Science, pages 63–64. Springer, 1996.

[19] M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 41(1):115–141, 2003.

[20] D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: A simple virtual class calculus. In *Aspect-oriented Software Development*, pages 121–134. ACM Press, 2007.

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

[22] B. A. Davey and H. A. Priestley. *Introduction to lattices and order,*. Cambridge University Press, 2nd edition, 2002.

[23] B. Delaware, W. Cook, and D. Batory. Fitting the pieces together: A machine-checked model of safe composition. In *Proc. ESEC/FSE '09*, pages 243–252. ACM Press, 2009.

[24] J. Desharnais, B. Möller, and G. Struth. Kleene algebra with domain. *ACM Transactions on Computational Logic*, 7(4):798–833, 2006.

[25] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, 2006.

[26] T. Ehm. Pointer Kleene algebra. In R. Berghammer, B. Möller, and G. Struth, editors, *RelMiCS*, volume 3051 of *Lecture Notes in Computer Science*, pages 99–111. Springer, 2004.

[27] E. Ernst, K. Ostermann, and W. Cook. A virtual class calculus. *ACM SIGPLAN Notices*, 41(1):270–282, 2006.

[28] W. Harrison, H. Ossher, and P. Tarr. General composition of software artifacts. In W. Löwe and M. Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 194–210. Springer, 2006.

[29] P. Höfner. Database for automated proofs of Kleene algebra. `http://www.kleenealgebra.de` (accessed October 5, 2010).

[30] P. Höfner, R. Khédri, and B. Möller. Feature algebra. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 300–315. Springer, 2006.

[31] P. Höfner, R. Khedri, and B. Möller. Algebraic view reconciliation. *Software Engineering and Formal Methods, SEFM '08.*, pages 85–94, 2008.

[32] P. Höfner and B. Möller. An extension for feature algebra — extended abstract. In *FOSD '09: Proceedings of the First International Workshop on Feature-Oriented Software Development*, pages 75–80. ACM Press, 2009.

[33] P. Höfner and B. Möller. An algebra for abstract interfaces. Technical Report 2010-2, Institut für Informatik, Universität Augsburg, 2010.

[34] D. Hutchins. Eliminating distinctions of class: Using prototypes to model virtual classes. In P. L. Tarr and W. R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*, pages 1–20. ACM Press, 2006.

[35] D. Hutchins. Eliminating distinctions of class: Using prototypes to model virtual classes. In *Object-oriented Programming Systems, Languages, and Applications*, pages 1–20. ACM Press, 2006.

[36] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute, 1990.

[37] C. Kästner, S. Apel, and D. Batory. A case study implementing features using AspectJ. In *Software Product Lines, 11th International Conference (SPLC)*, pages 223–232. IEEE Computer Society, 2007.

[38] R. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In J. Bosch, editor, *GCSE '01: Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, volume 2186 of *Lecture Notes in Computer Science*, pages 10–24. Springer, 2001.

[39] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *European Conf. on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 2–28. Springer, 2003.

[40] W. W. McCune. Prover9 and Mace4.
<`http://www.cs.unm.edu/~mccune/prover9`>. (accessed October 5, 2010).

[41] B. Möller. Towards pointer algebra. *Science of Computer Programming*, 21(1):57–90, 1993.

[42] M. Odersky and M. Zenger. Scalable component abstractions. In *Object-oriented programming, systems, languages, and applications*, pages 41–57. ACM Press, 2005.

[43] G. Ramalingam and T. Reps. A theory of program modifications. In T. Maibaum, editor, *Theory and Practice of Software Development, Vol.2*, volume 494 of *Lecture Notes in Computer Science*, pages 137–152. Springer, 1991.

[44] G. Ramalingam and T. Reps. Modification algebras. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Methodology and Software Technology*, pages 547–558. Springer, 1992.

[45] P. Tarr, H. Ossher, W. Harrison, and S. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software engineering (ICSE '99)*, pages 107–119. ACM Press, 1999.

Appendix

A. Feature Algebra in Waldmeister

We provide a proof template for Waldmeister for extended feature algebra. Waldmeister is a theorem prover for equational logic. It tries to prove a given conjecture fully automatically from given axioms. Since the structure of the axioms and the theorems of OFA and EFA are quite simple, Waldmeister finds proofs very fast. We have used Waldmeister to verify the lemmas and theorems presented throughout the paper. To use Waldmeister one has to add a conclusion (the conjectured lemma/theorem etc.) to the proof template and start the automated tool (the standard way is to execute the command

```
./waldmeister -f <inputfile>
```

in a shell. Another possibility is to copy the template to the Waldmeister's homepage [3] and to click the start button.

```
NAME            FEATURE_ALGERBA
MODE            PROOF

SORTS           MOD INTRO
SIGNATURE
                add: INTRO INTRO -> INTRO
                mul: MOD MOD -> MOD
                app: MOD INTRO -> INTRO

                0: -> INTRO
                1: -> MOD
                i1: -> INTRO
                i2: -> INTRO
                i3: -> INTRO
                m1: -> MOD
                m2: -> MOD
                m3: -> MOD

ORDERING        LPO
                mul > app > add > i1 > i2 > i3 > m1 > m2 > m3 > 1 > 0

VARIABLES
                i,j,k : INTRO
                m,n,o : MOD

EQUATIONS       add(i,0) = i
                add(i,add(j,k)) = add(add(i,j),k)
                add(i,add(j,i)) =  add(j,i)

                app(mul(m,n),i)=app(m,app(n,i))
                app(m,0) = 0
                app(1,i) = i

                app(m,add(i,i))=add(app(m,i),app(m,i))

CONCLUSION      %goal to be shown, e.g.
               add(i,0) = i
```

[3] http://www.mpi-inf.mpg.de/ hillen/waldmeister/tryout.htm

B. Deferred Proofs and Additional Properties

In this appendix we list the deferred proofs.

Proof of Lemma 6.8. We only show the properties for $\leq_r$; the proofs for $\leq_l$ are similar.

(1) Immediate by definition when choosing $z = 0$.

(2) Reflexivity ($x \leq_r x$) follows from definition by $x + 0 = x \Rightarrow \exists z \in E : x + z = x \Leftrightarrow x \leq_l x$.
Transitivity can be shown as follows

$$\begin{aligned}
& x \leq_r y \wedge y \leq_r z \\
\Leftrightarrow\ & \exists z', z'' \in E : x + z' = y \wedge y + z'' = z \\
\Rightarrow\ & \exists z', z'' \in E : (x + z') + z'' = z \\
\Leftrightarrow\ & \exists z', z'' \in E : x + (z' + z'') = z \\
\Leftrightarrow\ & \exists \tilde{z} \in E : x + \tilde{z} = z \\
\Leftrightarrow\ & x \leq_r z\ .
\end{aligned}$$

(3) Immediate from the definition. □

Proof of Lemma 6.9.

(a) ($\Rightarrow$) Setting $z = y$ in the definition of $\leq$ gives

$$x \leq y \Leftrightarrow x + y = y \Rightarrow \exists z.\, x + z = y \Leftrightarrow x \leq_r y\ .$$

($\Leftarrow$) From the assumption $x \leq_r y \Leftrightarrow \exists z.\, x + z = y$ we have to show that $x + y = y$. By the assumption, (distant) idempotence and assumption again we get

$$x + y = x + x + z = x + z = y\ .$$

(b) The goal can be shown using the definition, distant idempotence, choosing $\tilde{z} = z + a$ and the definition again:

$$x \leq_l y \Leftrightarrow \exists z : z + x = y \Leftrightarrow \exists z : x + z + x = y \Rightarrow \exists \tilde{z} : x + \tilde{z} = y \Leftrightarrow x \leq_r y\ .$$

(c) ($\Leftarrow$) is obvious. For ($\Rightarrow$) we use $x \leq_r x + y$ (Lemma 6.8(3)). By the assumption this is equivalent to $x \leq_l x + y$ and hence there is a $z \in E$ with $z + x = x + y$. By this and (distant) idempotence we now get the claim:

$$y + x = x + y + x = z + x + x = z + x = x + y\ .$$

□

Proof of Lemma 8.2. In this proof we assume that $\lfloor$ and $-$ bind tighter than $\sqcup$.

(a) The claim follows from Axiom (6) by (10).

(b) The first assertion is shown analogously to Part (a). The remaining claim $a \sqsubseteq a - 0_C$ follows again from splitting (6) and Axiom (5)

$$a = (a \lfloor 0_C) \sqcup (a - 0_C) = 0_C \sqcup (a - 0_C) = a - 0_C\ .$$

(c) By definition of $\sqsubseteq$, the assumptions are $a \sqcup b = b$ and $p \sqcup q = q$. The claims transform into $(a \lfloor p) \sqcup (b \lfloor p) = b \lfloor p$ and $(a \lfloor p) \sqcup (a \lfloor q) = (a \lfloor q)$, resp., and follow from the distributivity axioms (1) and (2).

$$\begin{aligned}
(a \lfloor p) \sqcup (b \lfloor p) &= (a \sqcup b) \lfloor p = b \lfloor p\ , \\
(a \lfloor p) \sqcup (a \lfloor q) &= a \lfloor (p \sqcup q) = a \lfloor q\ .
\end{aligned}$$

(d) By definition of $\sqsubseteq$, the assumptions are $a \sqcup b = b$ and $p \sqcup q = q$. The claim for isotony transforms into $(a - p) \sqcup (b - p) = b - p$ which follows from distributivity (3):

$$(a - p) \sqcup (b - p) = (a \sqcup b) - p = b - p\ .$$

For the second claim we get by the assumption, Axiom (4) and Part (2)

$$a - q = a - (p \sqcup q) = (a - p) - q \sqsubseteq a - p\ .$$

(e) Before showing quasi associativity we derive an auxiliary property. By the splitting axiom (6), exchange law (7), annihilation (8), Axiom (7) and Part (a),

$$q \;=\; q\mathbin{\rfloor}(p-q) \sqcup q-(p-q) \;=\; (q-q)\mathbin{\rfloor} p \sqcup q-(p-q) \;=\; 0_C\mathbin{\rfloor} p \sqcup q-(p-q) \;=\; q-(p-q)\,. \qquad (*)$$

Now we can prove the associativity property. By the splitting axiom (6) and Equation (4), we first get

$$a\mathbin{\rfloor} p \;=\; (a\mathbin{\rfloor} p)\mathbin{\rfloor} q \sqcup (a\mathbin{\rfloor} p)-q \;=\; a\mathbin{\rfloor} p)\mathbin{\rfloor} q \sqcup a\mathbin{\rfloor}(p-q)\,.$$

Similarly, we get $a\mathbin{\rfloor} p = a\mathbin{\rfloor}(p\mathbin{\rfloor} q \sqcup p-q) = a\mathbin{\rfloor}(p\mathbin{\rfloor} q) \sqcup a\mathbin{\rfloor}(p-q)$ and hence

$$(a\mathbin{\rfloor} p)\mathbin{\rfloor} q \sqcup a\mathbin{\rfloor}(p-q) \;=\; a\mathbin{\rfloor}(p\mathbin{\rfloor} q) \sqcup a\mathbin{\rfloor}(p-q)\,.$$

We now "subtract" $p-q$ on both sides and get, using distributivity (3),

$$(((a\mathbin{\rfloor} p)\mathbin{\rfloor} q)-(p-q)) \sqcup ((a\mathbin{\rfloor}(p-q))-(p-q)) \;=\; ((a\mathbin{\rfloor}(p\mathbin{\rfloor} q))-(p-q)) \sqcup ((a\mathbin{\rfloor}(p-q))-(p-q))\,.$$

Now we can apply Lemma 8.3(a) (the proof is given below). This yields

$$((a\mathbin{\rfloor} p)\mathbin{\rfloor} q)-(p-q) \;=\; (a\mathbin{\rfloor}(p\mathbin{\rfloor} q))-(p-q)\,.$$

To show the claim, we simplify both sides of this equation. By Equations (7) and (*) we get

$$((a\mathbin{\rfloor} p)\mathbin{\rfloor} q)-(p-q) \;=\; (a\mathbin{\rfloor} p)\mathbin{\rfloor}(q-(p-q)) \;=\; (a\mathbin{\rfloor} p)\mathbin{\rfloor} q\,.$$

Similarly one can show that $(a\mathbin{\rfloor}(p\mathbin{\rfloor} q))-(p-q) = a\mathbin{\rfloor}(p\mathbin{\rfloor} q)$ which finally shows the claim.

(f) ($\Rightarrow$) By Axiom (6), the assumption, isotony of $\sqcup$, and Axiom (9) we get

$$p \;=\; p\mathbin{\rfloor} q \sqcup p-q \sqsubseteq p\mathbin{\rfloor} q \sqcup r \sqsubseteq q \sqcup r\,.$$

($\Leftarrow$) By isotony and the assumption, Axiom (4), annihilation (8) and Parts (a) and (b), we get

$$p-q \sqsubseteq (q\sqcup r)-q \;=\; q-q \sqcup r-q \;=\; r-q \sqsubseteq r\,.$$

(g) By Axioms (6) and (9) we have $p\mathbin{\rfloor} q \sqsubseteq p$ and $p\mathbin{\rfloor} q \sqsubseteq q$, i.e., $p\mathbin{\rfloor} q$ is a common lower bound of p and q in N. Let now $r \sqsubseteq p, q$ be another common lower bound in N. Then by isotony of $\mathbin{\rfloor}$ we get $r = r\mathbin{\rfloor} r \sqsubseteq p\mathbin{\rfloor} q$, and hence $p\mathbin{\rfloor} q$ is the greatest lower bound. □

Proof of Lemma 8.3.

(a) By Axioms (7), (8) and (5) we get $(a-p)\mathbin{\rfloor} p \;=\; a\mathbin{\rfloor}(p-p) \;=\; a\mathbin{\rfloor} 0_C \;=\; 0_C$. Similarly, we can show the claim $(a\mathbin{\rfloor} p)-p = 0_C$. The remaining claim follows from Axioms (7), (8) and (5)

$$(a\mathbin{\rfloor} p)-p \;=\; a\mathbin{\rfloor}(p-p) \;=\; a\mathbin{\rfloor} 0_C \;=\; 0_C\,.$$

(b) ($\Rightarrow$) The conjunct $a \sqsubseteq b$ follows by isotony of $-$ and transitivity of $\sqsubseteq$, the second conjunct from Part (a), isotony and the assumption

$$a\mathbin{\rfloor} p \sqsubseteq (b-p)\mathbin{\rfloor} p \;=\; 0_C\,.$$

($\Leftarrow$)We first calculate $a \;=\; (a\mathbin{\rfloor} p)\sqcup(a-p) \;=\; a-p$. This is done using Axiom (6) and the assumption $a\mathbin{\rfloor} p \sqsubseteq 0_C$. The claim then follows by isotony and the other assumption.

(c) Similar to Part (b). □

Proof of Lemma 8.6.

(a) From Lemma 8.3(a) we get $a \sqsubseteq a\mathbin{\rfloor} p \;\Leftrightarrow\; a-p \sqsubseteq 0_C$ and hence

$$\textsc{f}(a) \sqsubseteq p \;\Leftrightarrow\; a-p = 0_C\,.$$

By this, Axiom (3) lattice theory and the shunting rule from Lemma 8.2(f), we get, for arbitrary p,

$$
\begin{array}{ll}
& \mathrm{F}(a \sqcup b) \sqsubseteq p \\
\Leftrightarrow & (a \sqcup b) - p \sqsubseteq 0_C \\
\Leftrightarrow & (a - p) \sqcup (b - p) \sqsubseteq 0_C \\
\Leftrightarrow & a - p \sqsubseteq 0_C \ \ \wedge \ \ b - p \sqsubseteq 0_C \\
\Leftrightarrow & \mathrm{F}(a) \sqsubseteq p \ \ \wedge \ \ \mathrm{F}(b) \sqsubseteq p \\
\Leftrightarrow & \mathrm{F}(a) \sqcup \mathrm{F}(b) \sqsubseteq p \ ,
\end{array}
$$

which by the principle of indirect equality, viz.

$$c = d \ \Leftrightarrow \ (\forall\, e : c \sqsubseteq e \ \Leftrightarrow \ d \sqsubseteq e) \ ,$$

implies the claim.

(b) Again we use the principle of indirect equality. The claim then follows by definition and Axiom (11):

$$\mathrm{F}(p) \sqsubseteq q \ \ \Leftrightarrow \ \ p \sqsubseteq p \mathbin{\lfloor} q \ \ \Leftrightarrow \ \ p \sqsubseteq q \ .$$

(c) $a \sqsubseteq a \mathbin{\lfloor} \mathrm{F}(a)$ follows directly from the definition of abstract interfaces, whereas the other inequality follows from Lemma 8.2(a).

(d) We first use the splitting axiom (6) and get by Part (a) and Corollary 8.5

$$\mathrm{F}(a) \ = \ \mathrm{F}((a \mathbin{\lfloor} p) \sqcup (a - p)) \ = \ \mathrm{F}(a \mathbin{\lfloor} p) \sqcup \mathrm{F}(a - p) \sqsubseteq p \sqcup \mathrm{F}(a - p) \ .$$

This implies by shunting (Lemma 8.2(f)) $\mathrm{F}(a) - p \sqsubseteq \mathrm{F}(a - p)$. To show $\mathrm{F}(a - p) \sqsubseteq \mathrm{F}(a) - p$ we use shunting (Lemma 8.2(f)), the definition of f, distributivity (2), isotony, distributivity (3), Lemma 8.3(a), Axiom (7) and the definition again, to calculate, for arbitrary p,

$$
\begin{array}{ll}
& \mathrm{F}(a) - p \sqsubseteq q \\
\Leftrightarrow & \mathrm{F}(a) \sqsubseteq q \sqcup p \\
\Leftrightarrow & a \sqsubseteq a \mathbin{\lfloor} (q \sqcup p) \\
\Leftrightarrow & a \sqsubseteq (a \mathbin{\lfloor} q) \sqcup (a \mathbin{\lfloor} p) \\
\Rightarrow & a - p \sqsubseteq ((a \mathbin{\lfloor} q) \sqcup (a \mathbin{\lfloor} p)) - p \\
\Leftrightarrow & a - p \sqsubseteq ((a \mathbin{\lfloor} q) - p) \sqcup ((a \mathbin{\lfloor} p) - p) \\
\Leftrightarrow & a - p \sqsubseteq (a \mathbin{\lfloor} q) - p \\
\Leftrightarrow & a - p \sqsubseteq (a - p) \mathbin{\lfloor} q \\
\Leftrightarrow & \mathrm{F}(a - p) \sqsubseteq q \ .
\end{array}
$$

By the principle of indirect inequality

$$c \sqsubseteq d \ \Leftrightarrow \ (\forall\, e : c \sqsubseteq e \ \Rightarrow \ d \sqsubseteq e) \ ,$$

this implies $\mathrm{F}(a - p) \sqsubseteq \mathrm{F}(a) - p$.

Next we show $\mathrm{F}(a) \mathbin{\lfloor} p \sqsubseteq \mathrm{F}(a \mathbin{\lfloor} p)$ As above we split a by Axiom (6), use Part (a), distributivity (1), the previous result and Lemma 8.2(a)

$$
\begin{array}{ll}
& \mathrm{F}(a) \mathbin{\lfloor} p \\
= & \mathrm{F}((a \mathbin{\lfloor} p) \sqcup (a - p)) \mathbin{\lfloor} p \\
= & (\mathrm{F}(a \mathbin{\lfloor} p) \sqcup \mathrm{F}(a - p)) \mathbin{\lfloor} p \\
= & (\mathrm{F}(a \mathbin{\lfloor} p) \mathbin{\lfloor} p) \sqcup (\mathrm{F}(a - p) \mathbin{\lfloor} p) \\
= & (\mathrm{F}(a \mathbin{\lfloor} p) \mathbin{\lfloor} p) \sqcup ((\mathrm{F}(a) - p) \mathbin{\lfloor} p) \\
= & \mathrm{F}(a \mathbin{\lfloor} p) \mathbin{\lfloor} p \\
\sqsubseteq & \mathrm{F}(a \mathbin{\lfloor} p)
\end{array}
$$

Finally we have to prove that $\mathrm{F}(a \mathbin{\lfloor} p) \sqsubseteq \mathrm{F}(a) \mathbin{\lfloor} p$. With Lemma 8.3(c), isotony and shunting we get

$$
\begin{array}{ll}
& \mathrm{F}(a \mathbin{\lfloor} p) \sqsubseteq \mathrm{F}(a) \mathbin{\lfloor} p \\
\Leftrightarrow & \mathrm{F}(a \mathbin{\lfloor} p) \sqsubseteq \mathrm{F}(a) \ \ \wedge \ \ \mathrm{F}(a \mathbin{\lfloor} p) - p \sqsubseteq 0_C \\
\Leftarrow & a \mathbin{\lfloor} p \sqsubseteq a \ \ \wedge \ \ \mathrm{F}(a \mathbin{\lfloor} p) \sqsubseteq p \ .
\end{array}
$$

The left part holds by Lemma 8.2(a); the right one by Corollary 8.5. □