An Extension for Feature Algebra

Peter Höfner[a,c], Bernhard Möller[b]

[a]*NICTA, Sydney, Australia*
[b] *Institut für Informatik, Universität Augsburg, Germany*
[c]*School of Computer Science and Engineering, University of New South Wales, Australia*

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. The algebra captures major aspects of feature-orientation, such as 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. In other words, it does not reflect details of concrete implementations.

In this 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 properly reflect all aspects of feature-orientation from the level of directory structures down to the level of actual code. This gives 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*. 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

Dedication It is our pleasure to dedicate this paper to José Nuno Oliveira on the occasion of his 60th birthday. José has been very active in the field of formal methods and, particularly, in algebraic techniques for that, such as relation algebra and category theory. Therefore we think it adequate to contribute a study on an algebra for feature-oriented system development, not only for cross-fertilisation, but hopefully also for enjoyment. Best wishes, José, for yourself, your work and for fruitful further interaction, as in the past years!

1. Introduction

Over the last decade *feature-orientation* (FO) (e.g. [13, 14]) 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, FO summarises feature-oriented software development and feature-oriented programming. It has widespread applications from network protocols [13] and data structures [15] to software product lines [34]. 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.

Feature models are (compact and) structured representations for use in FO, first introduced in Feature-Oriented Domain Analysis (FODA) [32]. With respect to (software) product lines they describe all possible products in terms of features.

Over the years, FO has been increasingly supported by software tools. Examples are `FeatureHouse` [2] and the `AHEAD` Tool Suite [8]. The former is a general approach to the composition of software artefacts, using superimposition and three-way merge. It supports various input formats like source code, test cases, models, documentation and makefiles. `AHEAD` (Algebraic Hierarchical Equations for Application Design) is based on stepwise refinement. Each

Email address: `peter.hoefner@nicta.com.au` (Peter Höfner)

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. [2, Table 3], [34] and [33]).

There is, however, very little work on the mathematical structure and foundations of FO. Nevertheless, the purely algebraic approach of *Feature Algebra (FA)* was developed [4]. It is a formal framework that captures many of the common ideas of FO, such as introductions, refinements, or quantification, in an abstract, language- and tool-independent way and thus hides differences of minor importance. It is intended as a common basis for describing, evaluating and comparing existing notions, tools and other aspects in FO and not just as the starting point for the development of yet another tool. FA can also serve as a formal foundation of architectural metaprogramming [9] and automatic feature-based program synthesis [18].

In this paper, which is a substantially expanded version of [30], we build on this algebraic structure. The standard model of FA (cf. [4]) is based on *feature structure forests (FSFs)* [3]. An FSF captures the essential, hierarchical module structure of a given program; each node represents a structural element (such as a package or statement). In this paper we present two further models for FA: both are again based on FSFs, but use different representations of them. The first model uses prefix-closed sets of strings into which FSFs can be transformed, while the second one is based on prefix-free lists of strings. They are used to illustrate that FA does not capture all aspects of FO: if manipulation of code and not only of the overall program structure is to be included into the model some aspects of FO cannot be reflected properly in FA. In particular, we show that merging, overriding or extending bodies of methods leads to problems. Based on these findings we modify the axioms and operators of FA and introduce the concept of an Extended Feature Algebra (EFA). We explore some of its algebraic properties, in particular, operators for overriding.

The paper is organised as follows: after introducing the original model of FA (Section 2) we recapitulate the abstract notion of FA in Section 3. After that we present another model of FA in Section 4 and emphasise the commonalities and differences of the discussed 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. To overcome these deficiencies, we relax the axioms and introduce the concept of an extended feature algebra (EFA) in Section 6. To clarify the idea and to underpin the relaxation we extend the introduced models of FA to handle code explicitly in Section 7. Finally we discuss in Section 8 how additional operators can be introduced in FO formally. In particular, we present operators for merging, overriding and updating as used in code modification. The paper closes with a review of related work in Section 9, and a short summary as well as an outlook concerning future work in Section 10. Many proofs are deferred to an appendix.

2. A Model of Feature Algebra

Based on *feature structure forests (FSFs)*, we give a first concrete model for FA. The formal definition of abstract Feature Algebras will be given in the next section. FSFs capture the essential hierarchical module structure of a given system (e.g. [4]). Essentially, an FSF is a "stripped-down abstract syntax tree" [2], including information about directory structures. 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. 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) mainly 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 when appropriate. It is straightforward that hierarchical structures not always induce 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. 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 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 the

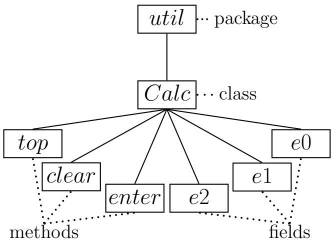


Figure 1: A simple JAVA-class as FSF ([9, 4])

descendant collection is empty then t is called a *leaf*. If each of the descendant collections in t is a set/list then t is *unordered*/*ordered*.

A *Feature Structure Forest* (FSF) is a forest (collection of trees, called *Feature Structure Trees* (FSTs)) 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 directories with modules like packages, classes and subclasses, while 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. For now we assume that the leaves are opaque and not further analysed. This will be changed in our extended algebra in Section 6.

It is well known that certain labelled forests can be represented using sets of strings of node labels (e.g., [7]). 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"; the tree nodes remain implicit.

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 of paths:

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

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—the names have to be unique. However, this is not a severe restriction, since we can always rename nodes.

Based on FSFs, feature combination can be modelled as superimposition of FSFs, i.e., as recursively merging their corresponding substructures [3]. This reflects a composition technique for software that has been successfully applied within several case studies covering various areas of computer science. In particular, it has been shown to play a crucial rôle in FO.

On $P\Sigma$-representations of FSFs, feature tree superimposition can be represented as simple set union on the elements of $P\Sigma$, which is well defined, since $P\Sigma$ is closed under set union. This operator is commutative and idempotent and hence in this model the order of combination does not matter.

Example 2.4. We return to the method part of Example 2.2. At the left of Figure 2 it is composed by superimposition, which corresponds with union on $P\Sigma$, with another FSF; this yields the FST at the right.
The left FST represents a simple implementation of a Class *Calc*, which is part of a package *util*, and offers functions for *enter*ing a number, *clear*ing and *add*ing two numbers. The second element is a feature forest, also offering *add* and a conversion *toStr*ing; it also provides a new package that implements some simple IO mechanisms.

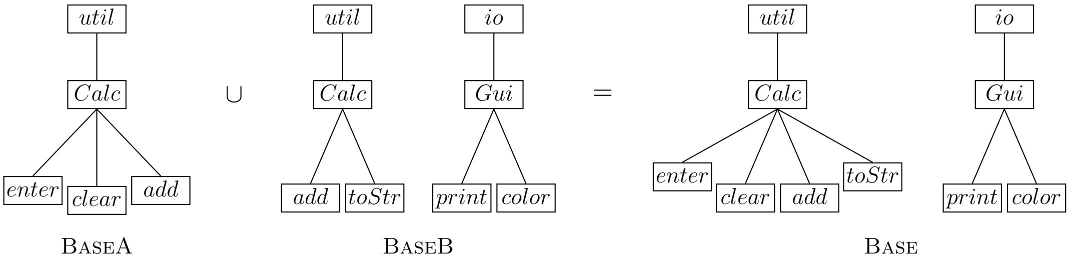


Figure 2: Superimposition of FSFs

The PΣ counterparts are

$$\begin{aligned}
\text{BaseA} &=_{df} \{util,\ util.Calc,\ util.Calc.enter,\ util.Calc.clear,\ util.Calc.add\}\,,\\
\text{BaseB} &=_{df} \{util,\ util.Calc,\ util.Calc.add, util.Calc.toStr,\\
&\qquad io, io.Gui, io.Gui.print, io.Gui.color\}\,,\\
\text{Base} &=_{df} \text{BaseA} \cup \text{BaseB}\\
&= \{util,\ util.Calc,\ util.Calc.enter,\ util.Calc.clear,\ util.Calc.add,\ util.Calc.toStr\,,\\
&\qquad io, io.Gui, io.Gui.print, io.Gui.color\}\,.
\end{aligned}$$

In particular, the common path *util.Calc.add* of BaseA and BaseB occurs only once in the result. This example illustrates a couple of interesting aspects: (i) New features, such as *toStr* can just be added. (ii) Features that are present in both forests have to be merged. Since the example does not contain any details about implementation, proper merging does not occur, and can be captured by simple set union. (iii) In an ordered model, the relative order between the subfeatures of the merged features is not disturbed. □

Of course in a more general setting, where leaves of FSFs contain more (language-dependent) information, the composition gets more complicated. For example one might need wrappers to merge nodes with the same name (cf. [3]). 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 modification) 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.

Proposition 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* $A \cup B \cup A = B \cup A$ *of* distant idempotence *holds for all* $A, B \in \mathrm{P}\Sigma$.

The law of distant idempotence expresses the fact that adding an already present feature has no effect. Since union $\cup$ is commutative, it is a trivial consequence of the direct idempotence $A \cup A = A$ in PΣ. However, in the abstract setting (cf. Section 3) we will generalise to non-commutative addition operators; for that case the law of distant idempotence does not follow from the direct one and hence needs to be postulated; normal idempotence then follows.

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 $\subseteq$ Base in Example 2.4. As known from simple set theory, the operator $\subseteq$ 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 FO. 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 [5], the two techniques are based on different ideas. Superimposition can be interpreted as *composing features* and hence is a binary inner operator on FSFs, sometimes called *introductions* in that context. 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. [3, 35, 43]).

3. Feature Algebra

We abstract from the concrete model of FSFs and introduce the structure of abstract Feature Algebra following [4]. That paper singles out a number of axioms that have to be satisfied by languages suitable for FO. Each of the axioms is derived from and illustrated by concrete examples from software development. 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 of this algebra described in [4]. However, we present a different model of it in the next section.

Definition 3.1. 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., $i + j + i = j + i$.
- The set of *modifications* M contains a special modification 1.
- The operators $\circ : M \times M \to M$ and $\cdot : M \times I \to I$ satisfy the following axioms:
  - $1 \cdot i = i$ (1 is a left-identity w.r.t. $\cdot$),
  - $m \cdot 0 = 0$ (0 is a right-annihilator w.r.t. $\cdot$),
  - $(m \circ n) \cdot i = m \cdot (n \cdot i)$ (pseudoassociativity of $\cdot/\circ$).
- The operator $\cdot$ distributes over $+$, i.e., $m \cdot (i + j) = (m \cdot i) + (m \cdot j)$.

Even 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 that the annihilation axiom could 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 operators are the *addition* operator $+$ 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 $P\Sigma$ from the previous section forms a Feature Algebra when a set of tree-rewriting functions is chosen as the set of modifications. The first item of Definition 3.1 is the abstract form of Proposition 2.5. The set of functions has to be chosen carefully, since otherwise one might, e.g., violate the condition that FSFs need to have unique names. In that model 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 $P\Sigma$ can be compared using the subset relation $\subseteq$ which is well known to be a partial order. The corresponding counterpart $\leq$ on abstract introductions is, in general, only a preorder, i.e., is reflexive and transitive. It induces an equivalence relation in a standard way.

Definition 3.2.

(a) The *natural preorder* or *introduction inclusion* is defined by $i \leq j \Leftrightarrow_{df} i + j = j$,

(b) 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, $i \leq j$ expresses that the introduction j has at least the same paths as the introduction i. That $\leq$ need not form an order 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 [31]. That calculus was developed to provide a type-safe support for virtual classes. Hence the abstract algebra not only provides a mathematical basis for FSFs, but also for Deep. The introduction equivalence means that two introductions contain the same features up to re-ordering; hence it might also be called *permutation equivalence*. The following properties are consequences of Definition 3.1 and are first listed in [4].

Lemma 3.3. *Assume i, j, k to be introductions 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.,* $i + j \sim j + i$.

All proofs are straightforward and hence skipped. The meanings of Parts (a)–(c) are obvious. 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 up to $\sim$. Part (g) shows that $+$ is commutative up to introduction equivalence. In the concrete model of FSFs that means that superimposition commutes up to the order of the subtrees.

Next, 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 result when applied to the same introduction.

Definition 3.4. 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.

Lemma 3.5. *Assume m, n, o to be modifications of a Feature Algebra.*

(a) $m \circ (n \circ o) \approx (m \circ n) \circ o$.

(b) $m \circ 1 \approx 1 \circ m \approx m$.

Parts (a) and (b) respectively mean that, up to application equivalence, $\circ$ is associative and 1 is its neutral element.

Up to now, we have presented the mathematical basics of the original Feature Algebra (FA). The above definitions will be the basis for our extension that reflects certain aspects of the code level.

The above two lemmas also show a particular advantage of the abstract algebraic approach. Since it uses 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 this paper we have encoded Feature Algebra in Waldmeister [26][1]and Isabelle/HOL [39]. Waldmeister has been used to prove the abstract laws of the algebra; Isabelle/HOL was used to verify those laws as well, but also to show that the presented models $L\Sigma$ and $P\Sigma$ are instances of the algebra, a concept that cannot be proved using automated theorem provers. The input files for Waldmeister and Isabelle/HOL can be found at `hoefner-online.de/jlamp15/`.

[1] In contrast to [5], we use Waldmeister instead of Prover9 since it can handle multiple sorts. For Feature Algebra we use the two sorts M and I.

4. Another Model of Feature Algebra

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 from roots to leaves by forming the prefix closure. It should be noted that the maximal paths can be viewed as atomic introductions[2] in the sense of [4], 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 would be 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 any 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.

To keep the model simple, we allow that paths from one tree may occur interspersed with ones from other trees. All that matters is the relative order of the paths belonging to the same tree and the relative order of the roots of the trees. Although in this way one and the same forest may be represented by different prefix-free lists, uniqueness could be easily achieved by grouping all paths from a single tree together. However, then the superimposition operator would become more complicated.

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

$$[\mathit{util.Calc.top},\ \mathit{util.Calc.clear},\ \mathit{util.Calc.enter},\ \mathit{util.Calc.e}0,\ \mathit{util.Calc.e}1,\ \mathit{util.Calc.e}2]\,. \qquad \Box$$

Superimposition + of prefix-free lists is now defined inductively over the structure of the first list:

- The empty list [ ] does not affect any list l of paths: $[\,] + l \ =_{df}\ l$.
- A singleton list $[p]$, for a path p, is added to an existing list l as follows.
  - If p is a prefix of some element of l then p is simply skipped and not added to l.
  - If l contains a proper prefix q of p then q is removed from l and p is prefixed to the resulting list. 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 prefixed to l.

  This procedure is described by the following definition:

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

- For a longer first argument list we define recursively $[p_1, \ldots, p_m, p_{m+1}] + l \ =_{df}\ [p_1, \ldots, p_m] + ([p_{m+1}] + l)$.

Now structural induction over prefix-free lists shows that + preserves prefix-freeness and indeed satisfies associativity and distant idempotence.

Definition 4.2. We define $\mathrm{L}\Sigma$ as the set of all prefix-free lists of elements of Σ^+.

Lemma 4.3.

(a) The structure $\mathrm{L}\Sigma = (\mathrm{L}\Sigma, +, [\,])$ *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 operators as before, the whole structure forms a Feature Algebra.

[2] An introduction i is *atomic* iff $i \neq 0 \ \wedge\ (\forall j.\ j \leq i \Rightarrow\ j = 0 \vee j = i)$.

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 FO. However, in the next section we will show that this axiom faces problems in a model that considers more details.

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

Example 4.4. Assume the following two lists:

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

By definition of + it 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 (or introduction equivalent), see Definition 3.2, which is algebraically expressed as $L_1 \sim L_2$. □

It is easy to see that there is a close relationship between this model and the original model of FA, as presented in [5].

5. The Lost Idempotence

As mentioned in the previous sections, distant idempotence (and hence idempotence), i.e., the fact that adding an already present feature has no effect, is of central interest for Feature Algebra. In [5], it is stated that languages and tools for feature combination usually have that idempotence property.

FA was introduced as a formal treatment of FO. 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.

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 operator. The composition may be 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 are usually merged. In Section 7 we will give some possible definitions for such an operator. In this 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 example already illustrates the crucial points.

First we have to enrich FSFs with (abstract) code. Later, we will also see how this affects PΣ and LΣ. To integrate code into an FST, each leaf is extended with a code fragment, which may, e.g., just be an interface, but possibly also contain concrete methods and field definitions.

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

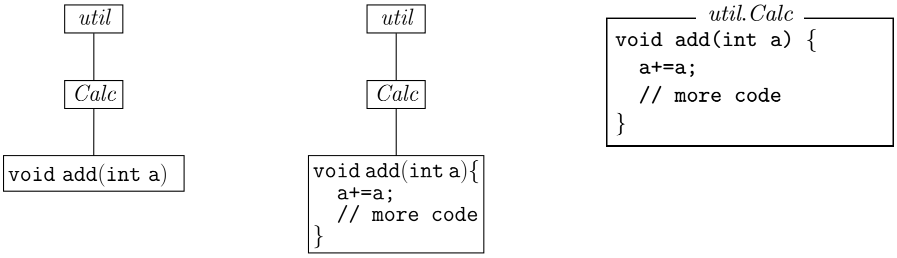
```
void add(int a) {
  a+=a;
  // more code
}
```

As in Example 2.4, we assume that the method *add* belongs to a class *Calc*, which is part of a package *util*. The FSF at the left of Figure 3 illustrates the original abstraction, which only provided the name *add*. The tree in the middle depicts the full specification, including code, while at the right we represent the tree in a compressed form, that we will use in the remainder of the paper. □

When code fragments for the same introduction occur in two or more FSFs that are composed via superimposition, a mechanism for merging them is needed. Most of the time merging by replacement, concatenation, or overriding suffices, but the approach is open to more sophisticated mechanisms, occurring in multi-dimensional separation of concerns [40] or software merging [37].

In `FeatureHouse`, for example, upon a merge code parts stored in the same leaf of an FSF are overridden or updated. Places where this occurs are marked by the keyword `original`.[3] We skip most of the details and only

[3] Its necessity in the context of `FeatureHouse` is discussed in [2].



```
void add(int a){
  a+=a;
  // more code
}
```

```
util.Calc
void add(int a) {
  a+=a;
  // more code
}
```

Figure 3: FST, FST equipped with code, and compressed FST

discuss one particular situation. Assume a method with body *m* that is updated/overridden by a body *n* which contains the keyword `original`. If these two bodies are merged then 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 a "merging-point" and disappears after composition.

We illustrate the situation by an example. To emphasise the difference to the case of FA we denote the merge operator now by ⊕.

Example 5.2. Assume the following two FSTs:

T_1:

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

T_2:

```
util.Calc
void add(int a) {
  a+=a;
  // more code
}
```

Both programs are merged by superimposition, since they provide implementations for the same leaf in the corresponding FSF; this mechanism is used, for instance, in `FeatureHouse`. The result of $T_1 \oplus T_2$ is

```
util.Calc
void add(int a) {
  a++;
  a+=a;
  // more code
}
```

As shown in the example, the keyword `original` disappears during the merge. To merge the generated code fragment $T_1 \oplus T_2$ with another element T_3, the element T_2 has to contain another occurrence of `original`. If no `original` call exists the old element is overwritten by the new one. □

Code merging does not only occur in `FeatureHouse` and Java, but also in other approaches like AspectJ. There, the method `proceed` is used to navigate to previously defined join points where code is to be merged; this task is then performed by the weaver of AspectJ. Most other approaches and tools for FO behave similarly. Therefore it is mandatory for formal methods capturing FO to treat this phenomenon.

To illustrate why the law of distant idempotence is not valid any longer, we compose two identical methods containing the keyword `original`.

Example 5.3.

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

⊕

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

=

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

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

6. Extended Feature Algebra

We have shown that the axiom of distant idempotence does 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). 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 *util.Calc* have already been given in Figure 3. The corresponding algebraic expressions over PΣ or LΣ are

$$\{\mathit{util},\ \mathit{util.Calc},\ \mathit{util.Calc}.\texttt{void add(int a)}\} \quad \text{and} \quad [\mathit{util.Calc}.\texttt{void add(int a)}],$$

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

$$[(\mathit{util.Calc}.\texttt{void add(int a)},\ \mathit{code}_{\mathit{add}})]\,.$$

Here $\mathit{code}_{\mathit{add}}$ denotes 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 $(M, I, C, E, \oplus, \circ, \cdot, \triangleright, \mathbf{0}, \mathbf{1})$ such that the following properties hold for all $m, n \in M$, $i, j \in I$, $x \in E$ and $a, b \in C$.

- The set C of *code fragments* offers an associative *update* or *override* operator $\triangleright$, i.e., $(C, \triangleright)$ is a semigroup;
- $(E, \oplus, \mathbf{0})$ is a monoid of *extended introductions* with $I[C] \subseteq E$ satisfying
  - $i[a] \oplus i[b] = i[a \triangleright b]$ (pseudodistributivity of $\oplus$ over $\triangleright$),
  - $i \neq j \Rightarrow i[a] \oplus j[b] = j[b] \oplus i[a]$ (restricted commutativity),
  - $i[a] = j[b] \Rightarrow i = j \wedge a = b$ (injectivity).
- The operators $\circ : M \times M \to M$ and $\cdot : M \times E \to E$ satisfy the following axioms:
  - $\mathbf{1} \cdot x = x$ ($\mathbf{1}$ is left-identity w.r.t. $\cdot$),
  - $m \cdot \mathbf{0} = \mathbf{0}$ ($\mathbf{0}$ is a right-annihilator),
  - $(m \circ n) \cdot x = m \cdot (n \cdot x)$ (pseudoassociativity of $\cdot/\circ$).
- $\cdot$ distributes over $+$.

The definitions of EFA and FA are basically the same. The only difference is that introductions I (of FA) are replaced by a new type E, containing the introductions of I, which are considered as atomic now, and the corresponding code (elements of the new type C). As a consequence the new operator $\triangleright$ needs to be defined. In particular, if C consists of a single element a only, and hence $a \triangleright a = a$, the this instance of an EFA is equivalent to an FA.

The set E of extended introductions does not only contain atomic introductions, but also arbitrary ones; in case of PΣ and LΣ these introductions can be generated from atomic ones (singleton sets/lists) using addition.

The operator $\triangleright$ may be an arbitrary associative function, but will often characterise an update. The merge operator discussed in the previous section would be an instance of such an operator. In the next section we discuss further instances of this operator 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.

If an idempotent update operator is chosen, i.e., an operator $\triangleright$ with $a \triangleright a = a$, then the above axiom yields the special case $i[a] \oplus i[a] = i[a]$, i.e., direct idempotence on $I[C]$. This is automatic when C is taken as a set containing only one single element (behaving like the empty or hidden 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; even worse, a neutral element yields problems when defining the operator $\triangleright$ as complete overriding.

Since application equivalence of modifications (Definition 3.4) does not depend on the subsumption order, we can use exactly the same definition in EFA, and hence Lemma 3.5 holds for EFA as well.

The presented models for FA 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, an analogous extension of the models PΣ and LΣ is straightforward. The operators for superimposition are adapted in a straightforward way, too.

Example 6.3. 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

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

from the previous example now yields

$$\begin{aligned} &[(util.Calc.\texttt{void add(int a)}, code_{add})] \oplus [(util.Calc.\texttt{void add(int a)}, code_{add})] \\ &= [(util.Calc.\texttt{void add(int a)}, code_{add} \triangleright code_{add})]\,, \end{aligned}$$

where $code_{add} \triangleright code_{add}$ is given by

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

□

We have seen in earlier laws that the natural preorder $\leq$ of an FA is useful for expressing substructure relationships. Unfortunately, the subsumption relation of an EFA need not be a preorder. While it is still transitive, reflexivity can fail. This is due to the lack of (distant) idempotence. Hence many of the laws from Lemma 3.3 do not hold in EFA. To overcome this problem we can introduce two alternative comparison relations.

Definition 6.4. In an EFA we define the *right natural preorder* as

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

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

If $\oplus$ 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 6.5. Assume the two FSTs 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 implementation T of *util.Calc*.`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 6.6. *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) $\mathbf{0} \leq_{lr} x$.

(b) *$\leq_{lr}$ are preorders, i.e., $x \leq_{lr} x$ and $x \leq_{lr} y \wedge y \leq_{lr} z \Rightarrow x \leq_{lr} z$. If $\oplus$ is left/right-cancellative and $\mathbf{0}$ is indecomposable, i.e., $x \oplus y = \mathbf{0} \Rightarrow x = \mathbf{0} = y$, then $\leq_l / \leq_r$ is a partial order.*

(c) *$x \leq_r x \oplus y$ and $y \leq_l x \oplus y$.*

(d) *$\oplus$ is $\leq_{lr}$-isotone.*

The properties ($i \le_{lr} \mathbf{0} \Rightarrow i = \mathbf{0}$), $x \le_l x \oplus y$ and $y \le_r x \oplus y$ do not hold.

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

Lemma 6.7. *Assume an EFA that additionally satisfies the axiom of distant idempotence. For arbitrary extended introductions x, y the following holds.*

(a) $\le$ coincides with $\le_r$, i.e., $x \le y \Leftrightarrow x \le_r y$.

(b) $\le_l$ refines $\le_r$, i.e., $x \le_l y \Rightarrow x \le_r y$ (and therefore by (a) *also $\le$).*

(c) $\le_l$ coincides with $\le_r$ if and only if $\oplus$ is commutative.

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

7. Interfaces, Restriction and Update

In this section, we discuss the update operator $\rhd$ in concrete situations. In particular, we present some consequences when $\rhd$ satisfies additional conditions. In the simplest case the update operator $\rhd$ overrides everything. Informally, if a feature is added that is already there then the content of the feature (e.g. code) of that present feature is lost and only the details of the newly added feature are preserved.

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

util.Calc

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

$\rhd$

util.Calc

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

}
```

$=$

util.Calc

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

□

Mathematically, such an update operator can be defined by

$$a \rhd b \;=_{df}\; a\,. \tag{1}$$

In FO overriding occurs quite often; e.g., `FeatureHouse` is mainly based on overriding when performing superimposition [2]. In AspectJ overriding can occur if the advice `around` is used. In the case of overriding the update operator $\rhd$ is idempotent.

This override operator also shows that we cannot assume a neutral element for $\rhd$ in general. If we were to assume $(C, \rhd, 1_\rhd)$ to be a monoid rather than a semigroup then the law $1_\rhd \rhd a = a$ would hold. From that we would obtain by Eq. (1) $a = 1_\rhd \rhd a = 1_\rhd$. Hence C would only contain the single element $1_\rhd$. Since this contradicts the intention of an extended Feature Algebra we only assume $(C, \rhd)$ to be a semigroup in the definition. However, in some situations there might be a neutral element, as we will see later.

Dual to complete overriding one can specify complete preservation by $a \rhd b \;=_{df}\; b$. Instead of overriding the "old" code, it is now entirely 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 discuss the case where the operator $\rhd$ is a proper update. It is to override those parts that are already in the feature and to add those parts that 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, `FeatureHouse` usually overrides declarations and functions as long as the keyword `original` does not occur in the code.

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 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 overridden 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.

```
<CATALOG>
 <CD>
  <TITLE>Best Recordings 1927-39</TITLE>
  <ARTIST>Comedian Harmonists</ARTIST>
  <COUNTRY>Germany</COUNTRY>
  ...
 </CD>
</CATALOG>
```

```
<CATALOG>
 <CD>
  <TITLE></TITLE>
  <ARTIST></ARTIST>
  <COUNTRY></COUNTRY>
  ...
 </CD>
</CATALOG>
```

□

Now we look at a 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 $\mathit{face} : C \to T$. Next we define two functions

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

The restriction operator $\downharpoonleft$ 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.

To define the update function $\triangleright$ we need to lift the function face to sets $X \subseteq C$ of code fragments by $\mathit{face}(X) =_{df} \{\mathit{face}(x) \, : \, x \in X\}$. Then the update operator $\triangleright$ can be defined by

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

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 [31].

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.

Proposition 7.4. *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 operators $\lfloor$ 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 $X - (U \cup V) = (X - U) - V$ as well as $X\lfloor(U - V) = (X - V)\lfloor U = (X\lfloor U) - V$ 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. The first two laws of Part (d) provide the analogue of Part (a) for the removal operator; the last law there holds, since $Y - W$ can also be read as the restriction of Y to the complement of W. 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 10).

8. Algebraic Interfaces and Updates

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 EFA. 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.

First, we characterise the restriction operator $\lfloor$ and the removal operator $-$ purely algebraically. The definition is motivated by Proposition 7.4.

We assume the set C of code fragments to be a semilattice L (e.g. [20]) 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. The *restriction* $\lfloor : L \times N \to L$ and the *removal* $- : L \times N \to L$ have to satisfy the following axioms:

$$(a \sqcup b)\lfloor p = (a\lfloor p) \sqcup (b\lfloor p) \tag{2}$$

$$a\lfloor(p \sqcup q) = (a\lfloor p) \sqcup (a\lfloor q) \tag{3}$$

$$(a \sqcup b) - p = (a - p) \sqcup (b - p) \tag{4}$$

$$a - (p \sqcup q) = (a - p) - q \tag{5}$$

$$a\lfloor 0_C = 0_C \tag{6}$$

$$a = (a\lfloor p) \sqcup (a - p) \tag{7}$$

$$a\lfloor(q - p) = (a - p)\lfloor q = (a\lfloor q) - p \tag{8}$$

$$p - p = 0_C \tag{9}$$

$$p\lfloor q \sqsubseteq q \tag{10}$$

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 counterexample search (Prover9/Mace4 [36]) 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 Proposition 7.4, and therefore we deem the axiomatisation quite intuitive.

In the sequel we assume that $\downharpoonright$ and $-$ bind more tightly than $\sqcup$. We note that Equations (7) and (9) imply

$$p \downharpoonright p = p\,. \tag{11}$$

Further properties will be shown below. In our proofs we can employ some useful properties induced by the semilattice; examples are $a \sqsubseteq 0_C \;\Rightarrow\; a = 0_C$ and

$$a \sqcup b \sqsubseteq c \;\Leftrightarrow\; a \sqsubseteq c \;\wedge\; b \sqsubseteq c\,. \tag{12}$$

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

We show some important properties describing the interplay between both operators.

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

(a) $(a - p) \downharpoonright p = 0_C$ *and* $(a \downharpoonright p) - p = 0_C$.

(b) $a \sqsubseteq b - p \;\Leftrightarrow\; a \sqsubseteq b \;\wedge\; a \downharpoonright p = 0_C$.

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

(d) $p \sqsubseteq q \;\Leftrightarrow\; p \sqsubseteq q \downharpoonright p \;\Leftrightarrow\; p \;=\; q \downharpoonright p$.

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.
Based on this we can show additional properties.

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

(a) $a \downharpoonright p \sqsubseteq a$. *In particular* $0_C \downharpoonright p = 0_C$.

(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., $a \sqsubseteq b \;\Rightarrow\; a \downharpoonright p \sqsubseteq b \downharpoonright p$ *and* $p \sqsubseteq q \;\Rightarrow\; a \downharpoonright p \sqsubseteq a \downharpoonright q$.

(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., $a \downharpoonright (p \downharpoonright q) = (a \downharpoonright p) \downharpoonright q$.

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

Part (a) means that $\downharpoonright$ 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.

Lemma 8.4. *Assume that N has a greatest element 1_C.*

(a) $1_C \downharpoonright p = p$.

(b) $1_C = p \sqcup 1_C - p$.

(c) $p = p - (1_C - p)$.

(d) $1_C - (1_C - p) = p$.

(e) $(1_C - p) \downharpoonright p = 0_C$.

The properties given so far show that the elements of N almost have the structure of a Boolean algebra. In fact, we have the following result.

Theorem 8.5. *If N is closed under $\sqcup$ and has a greatest element 1_C then it is a Boolean algebra with $\overline{p} = 1_C - p$ and $p \sqcap q = \overline{\overline{p} \sqcup \overline{q}} = p \downharpoonright q = q \downharpoonright p$.*

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 operator $\downharpoonright$:

$$\mathit{face}(X) \subseteq U \;\Leftrightarrow\; X = X \downharpoonright U\,,$$

where X is a code fragment and U an abstract interface as in Section 7. In fact, since $X \downharpoonright U \subseteq U$ holds anyway by definition, this can be relaxed to $X \subseteq X \downharpoonright U \;\Leftrightarrow\; \mathit{face}(X) \subseteq U$. 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 $X = X \downharpoonright U$ 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 8.6. The *abstract interface* operator F on L is axiomatised by $\mathrm{F}(a) \sqsubseteq p \;\Leftrightarrow_{df}\; a \sqsubseteq a \downharpoonright p$, where $a \in L$, $p \in N$.

The function F behaves in many respects like the abstract codomain operator introduced in [22]. This correspondence allows us to re-use a large body of well-known theory—another advantage of an abstract algebraic view.

Corollary 8.7. *For code fragments $a, b \in L$ and $p, q \in N$ we have $a \sqsubseteq a \downharpoonright \mathrm{F}(a)$ and $\mathrm{F}(a \downharpoonright p) \sqsubseteq p$.*
In particular, $\mathrm{F}(0_C) = 0_C$.

Lemma 8.8. *Assume arbitrary code fragments $a, b \in L$ and $p, q \in N$.*

(a) The abstract interface distributes through join, 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) $a = a \downharpoonright \mathrm{F}(a)$.

(d) $\mathrm{F}(a - p) = \mathrm{F}(a) - p$ *and* $\mathrm{F}(a \downharpoonright p) = \mathrm{F}(a) \downharpoonright p$.

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) gives 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 8.9. The *abstract update operator* can now be defined by $a \triangleright b \;=_{df}\; (b - \mathrm{F}(a)) \sqcup a$. We call a and b *compatible* if they agree on the common part of their interfaces, i.e., if $a \downharpoonright (\mathrm{F}(a) \sqcap \mathrm{F}(b)) = b \downharpoonright (\mathrm{F}(a) \sqcap \mathrm{F}(b))$.

As mentioned before, the abstract interface function behaves in many respects like the abstract codomain operator of [22]. Moreover, the update operator is identical to the one of Ehm introduced for pointer structures [24] (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 [38, 24].

Corollary 8.10. *If a, b, c are arbitrary code fragments, then*

(a) $a = a \triangleright 0_C = 0_C \triangleright a$.

(b) $(a \triangleright b) \triangleright c = a \triangleright (b \triangleright c)$.

(c) The following properties are equivalent:

- *a and b are compatible.*
- $a \lfloor \mathrm{F}(b) = b \lfloor \mathrm{F}(a)$.
- $a \triangleright b = a \sqcup b = b \triangleright a$.
- $a \triangleright b = b \triangleright a$.

(d) If a and b are compatible then $(a \sqcup b) \triangleright c = a \triangleright (b \triangleright c)$.

(e) If a and b are compatible and $b \triangleright c = c$ then $(a \sqcup b) \triangleright c = a \triangleright c$.

(f) If $\mathrm{F}(a) \lfloor \mathrm{F}(b) = 0_C$ *then* $a \triangleright (b \sqcup c) = b \sqcup (a \triangleright c)$.

The first two items show that $(L, \triangleright, 0_C)$ 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 overridings 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; this can be viewed as a weaker form of (distant) idempotence. Part (f) states a 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.

9. Related Work

Some of the authors' previous work on Feature Algebra and FSFs has already been discussed throughout the paper. Together with Khedri, the authors also developed an algebra for expressing variability of software product lines w.r.t. features [27] and parts of feature interaction [28]. Originally that algebraic approach only focussed on "syntactic" models. This has been extended to encompass semantics as, e.g., provided by implementation details and code in [29]. 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 FSTComposer [3] and `FeatureHouse` [2]. The former tool is an implementation of the AHEAD [14] model. As discussed above, the latter tool is a general approach to the composition of software artifacts.

There are only few algebraic approaches to FO by other researchers. Ramalingam and Reps developed modification algebras [42, 41]. These 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 postulated. However, our algebras (EFA and FA) are abstracted to a level where they not only cover revision control [6], but also state-of-the-art programming concepts like FO.

Other algebraic approaches are based on finite maps and simple lattice theory. In [23] 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. Other approaches, closely related to our approach and the one in [23], are feature interaction algebra [10] and Structured Document Algebra [12]. Feature interaction algebra [10] presents an axiomatisation of features, compositions and interactions, and discusses the fundamental relationships between them from an algebraic viewpoint, thereby abstracting away specifics of underlying languages or program representations. Document Algebra [12] defines modules with variation points and shows how such modules can be composed. It was used to deal algebraically with Software Product Lines [19, 11].

In [16], 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 (FMs); the second and more general algebra uses quarks, a concept that represents both FMs and their transformations, to express concepts in aspect-orientation. Both approaches use flat lattices 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. This means that rules for merging have 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 [21].

Non-algebraic approaches for FO include the Deep [31] calculus as well as its generalisation gDeep [1]. 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 structures including composition techniques (e.g., [17, 25, 31]). It has been shown that these techniques are closely related to superimposition [3, 5] 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 are theoretical foundations for FO. This has been demonstrated within this section and in previous papers. Due to its abstract and uniform character it forms a basis for treating nearly all languages, tools and mechanisms for FO. Therefore one could mention much more related work. However we restricted ourselves to a short overview, pointing into different directions.

10. Conclusion and Outlook

The present paper is based on [4] 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 FO (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 two models for Feature Algebra. These models demonstrate that there are more than one (meaningful) interpretation of the axioms of Feature Algebra. Moreover the list-based model may be used to encode and to perform superimposition 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 FO, this additional level has to be under consideration, since the goal should be to argue about concrete feature-oriented programs, not just about the overall directory structure containing the programs. We have shown that at that level the axiom of distant idempotence is not valid and has to be modified. To achieve this, we have introduced the structure of an Extended Feature Algebra to model feature-oriented programming at code level in an abstract manner. In particular the 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 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, which 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 FO. 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 particularly needed, since in this paper we only presented fundamental concepts for modelling that 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 Han-Hing Dang, Rob van Glabbeek, Roland Glück and Sven Mentl for helpful comments. We also thank the anonymous referees for their comments. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program. The research in Augsburg was partially funded by DFG grants MO 690/7-1 and MO 690/7-1 on *Algebra-Based Feature-Oriented Program Synthesis*.

References

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

[2] S. Apel, C. Kästner, and C. Lengauer. Language-independent and automated software composition: The FeatureHouse experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] D. Batory, P. Höfner, and J. Kim. Feature interactions, products, and composition. In E. Denney and U. Schultz, editors, *Generative Programming And Component Engineering (GPCE'11)*, ACM Press, pages 13–22, 2011.

[11] D. Batory, P. Höfner, D. Köppl, B. Möller, and A. Zelend. Structured document algebra in action. In R. De Nicola and R. Hennicker, editors, *Software, Services, and Systems*, volume 8950 of *Lecture Notes in Computer Science*, pages 291–311. Springer, 2015.

[12] D. Batory, P. Höfner, B. Möller, and A. Zelend. Features, modularity, and variation points. In A. Classen and N. Siegmund, editors, *Feature-Oriented Software Development (FOSD '13)*, pages 9–16. ACM Press, 2013.

[13] 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.

[14] 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.

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

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

[17] 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.

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

[19] H.-H. Dang, R. Glück, B. Möller, P. Roocks, and A. Zelend. Exploring modal worlds. *J. Log. Algebr. Meth. Program.*, 83(2):135–153, 2014.

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

[21] 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.

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

[23] 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.

[24] 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.

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

[26] T. Hillenbrand, A. Buch, and R. Vogt. Waldmeister: High performance equational deduction. *Journal of Automated Reasoning*, 18:265–270, 1997.

[27] 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.

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

[29] P. Höfner, R. Khédri, and B. Möller. Supplementing product families with behaviour. *Int. J. Software and Informatics*, 5(1-2):245–266, 2011.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] W. W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9`. (accessed May 26, 2017).

[37] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

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

[39] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[40] H. Ossher and P. L. Tarr. Hyper/J: Multi-dimensional separation of concerns for java. In C. Ghezzi, M. Jazayeri, and A. L. Wolf, editors, *International Conference on on Software Engineering (ICSE '00)*, pages 734–737. ACM Press, 2000.

[41] 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.

[42] 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.

[43] 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. Deferred Proofs and Additional Properties

In this appendix we list the deferred proofs.

Proof of Lemma 6.6. We only show the properties for $\leq_r$; the proofs for $\leq_l$ are similar.

(a) Immediate by definition when choosing $z = \mathbf{0}$.

(b) Reflexivity ($x \leq_r x$) follows from definition by $x \oplus \mathbf{0} = x \;\Rightarrow\; \exists\, z \in E : x \oplus z = x \;\Leftrightarrow\; x \leq_r x$.
Transitivity can be shown as follows:

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

(c) Immediate from the definition.

(d) From the assumption $x \leq_r y$ we get $\exists\, z' \in E : x \oplus z' = y$. Using this fact and distant idempotence, we get

$$y \oplus z \;=\; x \oplus z' \oplus z \;=\; x \oplus z \oplus z' \oplus z\,.$$

Therefore $\exists\, \tilde{z} \in E\ (\tilde{z} = z' \oplus z) : y \oplus z = x \oplus z \oplus \tilde{z}$, which is equivalent to $x \oplus z \leq_r y \oplus z$.
Left isotony is somewhat simpler: since $x \oplus z' = y$, we get for arbitrary u that $u \oplus x \oplus z' = u \oplus y$, which implies $u \oplus x \leq_r u \oplus y$. □

Proof of Lemma 6.7.

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

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

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

$$x \oplus y \;=\; x \oplus x \oplus z \;=\; x \oplus z \;=\; y\,.$$

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

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

(c) ($\Leftarrow$) is obvious, using Part (b). For ($\Rightarrow$) we use $x \leq_r x \oplus y$ (Lemma 6.6(c)). By the assumption this is equivalent to $x \leq_l x \oplus y$ and hence there is a $z \in E$ with $z \oplus x = x \oplus y$. By this and (distant) idempotence we now get the claim:

$$y \oplus x \;=\; x \oplus y \oplus x \;=\; z \oplus x \oplus x \;=\; z \oplus x \;=\; x \oplus y\,.$$

□

Proof of Lemma 8.2.

(a) By Axioms (8), (9) and (6) we get $(a - p)\lfloor p \;=\; a\lfloor(p - p) \;=\; a\lfloor 0_C \;=\; 0_C$. Similarly, we can show the claim $(a\lfloor p) - p = 0_C$.

(b) ($\Rightarrow$) The conjunct $a \sqsubseteq b$ follows from Axiom (7) and transitivity of $\sqsubseteq$, the second conjunct from Part (a), isotony and the assumption:

$$a\lfloor p \sqsubseteq (b - p)\lfloor p \;=\; 0_C\,.$$

($\Leftarrow$)We first calculate $a \;=\; a\lfloor p \sqcup a - p \;=\; a - p$. This is done using Axiom (7) and the assumption $a\lfloor p \sqsubseteq 0_C$. The claim then follows by isotony and the other assumption.

(c) Similar to Part (b).

(d) By (10) we have $p = q\lfloor p \;\Leftrightarrow\; p \sqsubseteq q\lfloor p$. Hence the claim is immediate from Part (c) and Axiom (9). □

Proof of Lemma 8.3.

(a) The claim follows from Axiom (7) using (12).

(b) The first assertion is shown analogously to Part (a). The specialisations $0_C - p = 0_C$ and $a - 0_C \sqsubseteq a$ are immediate from that and Equation (11). The remaining claim $a \sqsubseteq a - 0_C$ follows again from splitting (7) and Axiom (6):

$$a \;=\; a\lfloor 0_C \sqcup a - 0_C \;=\; 0_C \sqcup a - 0_C \;=\; a - 0_C\,.$$

(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 (2) and (3):

$$a\lfloor p \sqcup b\lfloor p \;=\; (a \sqcup b)\lfloor p \;=\; b\lfloor p\,,$$
$$a\lfloor p \sqcup a\lfloor q \;=\; a\lfloor(p \sqcup q) \;=\; a\lfloor q\,.$$

(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 the Distributivity Axiom (4):

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

For the second claim we get by the assumption, Axiom (5) and Part (b):

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

(e) We first derive an auxiliary property. By the Splitting Axiom (7), Exchange Axiom (8), Annihilation Axiom (9), and Part (a),

$$q \;=\; q\lfloor(p-q) \,\sqcup\, q-(p-q) \;=\; (q-q)\lfloor p \,\sqcup\, q-(p-q) \;=\; 0_C\lfloor p \,\sqcup\, q-(p-q) \;=\; q-(p-q)\,. \qquad (*)$$

Now we can prove the quasi-associativity property. By Axioms (7) and (8), we first get

$$a\lfloor p \;=\; (a\lfloor p)\lfloor q \sqcup (a\lfloor p) - q \;=\; (a\lfloor p)\lfloor q \,\sqcup\, a\lfloor(p-q)\,.$$

Similarly, we obtain $a\lfloor p = a\lfloor(p\lfloor q \sqcup p - q) = a\lfloor(p\lfloor q) \,\sqcup\, a\lfloor(p-q)$ and hence

$$(a\lfloor p)\lfloor q \,\sqcup\, a\lfloor(p-q) \;=\; a\lfloor(p\lfloor q) \,\sqcup\, a\lfloor(p-q)\,.$$

We now "subtract" $p - q$ on both sides and get, using distributivity (4),

$$((a\lfloor p)\lfloor q) - (p-q) \,\sqcup\, (a\lfloor(p-q)) - (p-q) \;=\; (a\lfloor(p\lfloor q)) - (p-q) \,\sqcup\, (a\lfloor(p-q)) - (p-q)\,.$$

Now we can apply Lemma 8.2(a). This yields

$$((a\lfloor p)\lfloor q) - (p-q) \;=\; (a\lfloor(p\lfloor q)) - (p-q)\,.$$

To show the claim, we simplify both sides of this equation. By Axiom (8) and Equation $(*)$ we get

$$((a\lfloor p)\lfloor q) - (p-q) \;=\; (a\lfloor p)\lfloor(q-(p-q)) \;=\; (a\lfloor p)\lfloor q\,.$$

Similarly one can show that $(a\lfloor(p\lfloor q)) - (p-q) = a\lfloor(p\lfloor q)$, which finally shows the claim.

(f) $(\Rightarrow)$ By Axiom (7), the assumption, isotony of $\sqcup$, and Axiom (10) we get

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

$(\Leftarrow)$ By isotony and the assumption, Axiom (5), annihilation (9) 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 (7) and (10) we have $p\lfloor q \sqsubseteq p$ and $p\lfloor q \sqsubseteq q$, i.e., $p\lfloor q$ 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 $\lfloor$ and Equation (11) we get $r = r\lfloor r \sqsubseteq p\lfloor q$, and hence $p\lfloor q$ is the greatest lower bound. □

Proof of Lemma 8.4.

(a) Immediate from Lemma 8.2 (d) and the assumption that 1_C is the greatest element of N, hence $p \sqsubseteq 1_C$.

(b) By Axiom (7), and Part (a):

$$1_C = 1_C \lfloor p \sqcup (1_C - p) = p \sqcup (1_C - p)\,.$$

(c) First, by Part (b), we obtain

$$1_C \;=\; (1_C - p) \sqcup (1_C - (1_C - p))\,. \tag{$\triangle$}$$

Now, by Part (a), ($\triangle$) and Axiom (2), Axiom (8), Part (a) again, Axiom (9) and neutrality:

$$p \;=\; 1_C \lfloor p \;=\; (1_C{-}p)\lfloor p \sqcup (1_C{-}(1_C{-}p))\lfloor p) \;=\; (1_C \lfloor p){-}p \sqcup (1_C \lfloor p){-}(1_C{-}p) \;=\; p{-}p \sqcup p{-}(1_C{-}p) \;=\; p{-}(1_C{-}p)\,.$$

(d) By Part (b), Axiom (2), Axiom (9) and neutrality, and Part (c):

$$1_C - (1_C - p) = (p \sqcup (1_C - p)) - (1_C - p) = p - (1_C - p) \sqcup (1_C - p) - (1_C - p) = p - (1_C - p) = p\,.$$

(e) By Axiom (8), Part (a) and (9): $(1_C - p)\lfloor p \;=\; (1_C \lfloor p) - p \;=\; p - p \;=\; 0\,.$ □

Proof of Theorem 8.5. We first show that the definition $\overline{p} \;=_{df}\; 1_C - p$ satisfies Huntington's Axiom

$$p \;=\; \overline{\overline{p} \sqcup q} \sqcup \overline{\overline{p} \sqcup \overline{q}}\,.$$

We calculate

$\quad \overline{\overline{p} \sqcup q} \sqcup \overline{\overline{p} \sqcup \overline{q}}$
$=\quad$ ⦃ definition of $\overline{}$ ⦄
$\quad (1_C - ((1_C - p) \sqcup q)) \sqcup (1_C - ((1_C - p) \sqcup (1_C - q)))$
$=\quad$ ⦃ commutativity of $\sqcup$ (twice) ⦄
$\quad (1_C - (q \sqcup (1_C - p))) \sqcup (1_C - ((1_C - q) \sqcup (1_C - p)))$
$=\quad$ ⦃ Axiom (5) (twice) ⦄
$\quad ((1_C - q) - (1_C - p)) \sqcup ((1_C - (1_C - q)) - (1_C - p))$
$=\quad$ ⦃ Axiom (4) ⦄
$\quad ((1_C - q) \sqcup (1_C - (1_C - q))) - (1_C - p)$
$=\quad$ ⦃ Lemma 8.4 (b) ⦄
$\quad 1_C - (1_C - p)$
$=\quad$ ⦃ Lemma 8.4 (d) ⦄
$\quad p\,.$

Next we show $p\lfloor q = \overline{\overline{p} \sqcup \overline{q}}$.

$\quad p\lfloor q$
$=\quad$ ⦃ Axiom (7) ⦄
$\quad (p\lfloor(1_C - q) \sqcup p - (1_C - q))\lfloor q$
$=\quad$ ⦃ Axiom (2) ⦄
$\quad (p\lfloor(1_C - q))\lfloor q \sqcup (p - (1_C - q))\lfloor q$
$=\quad$ ⦃ since by Axiom (10), isotony and Lemma 8.4 (e) $(p\lfloor(1_C - q))\lfloor q \sqsubseteq (1_C - q)\lfloor q = 0$ ⦄
$\quad (p - (1_C - q))\lfloor q$
$=\quad$ ⦃ since by isotony and Lemma 8.4 (d) $p - (1_C - q) \sqsubseteq 1_C - (1_C - q) = q$, and hence by Lemma 8.4 (d) ⦄

$$
\begin{array}{ll}
 & p - (1_C - q) \\
= & \quad \{\!\![\ \text{Lemma 8.4 (d)}\ ]\!\!\} \\
 & (1_C - (1_C - p)) - (1_C - q) \\
= & \quad \{\!\![\ \text{Axiom (5)}\ ]\!\!\} \\
 & (1_C - ((1_C - p) \sqcup (1_C - q)) \\
= & \quad \{\!\![\ \text{definition of}\ ^{-}\ ]\!\!\} \\
 & \overline{\overline{p} \sqcup \overline{q}}\,.
\end{array}
$$

Proof of Lemma 8.8.

(a) From Lemma 8.2(c) we get $a \sqsubseteq a \downharpoonleft p \;\Leftrightarrow\; a - p = 0_C$ and hence

$$\mathbf{F}(a) \sqsubseteq p \;\Leftrightarrow\; a - p = 0_C. \qquad (\circ)$$

By $(\circ)$, Axiom (4), lattice theory, $(\circ)$ twice, and lattice theory:

$$
\begin{aligned}
\mathbf{F}(a \sqcup b) \sqsubseteq p \Leftrightarrow (a \sqcup b) - p = 0_C \Leftrightarrow (a - p) \sqcup (b - p) = 0_C \Leftrightarrow a - p = 0_C \;\wedge\; b - p = 0_C \\
\Leftrightarrow \mathbf{F}(a) \sqsubseteq p \;\wedge\; \mathbf{F}(b) \sqsubseteq p \Leftrightarrow \mathbf{F}(a) \sqcup \mathbf{F}(b) \sqsubseteq p\,,
\end{aligned}
$$

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 Lemma 8.2(d):

$$\mathbf{F}(p) \sqsubseteq q \quad\Leftrightarrow\quad p \sqsubseteq p \downharpoonleft q \quad\Leftrightarrow\quad p \sqsubseteq q\,.$$

(c) $a \sqsubseteq a \downharpoonleft \mathbf{F}(a)$ follows directly from the definition of abstract interfaces, whereas the other inequality follows from Lemma 8.3(a).

(d) For the first claim we use again the principle of indirect equality. For arbitrary q we reason as follows. By $(\circ)$, Axiom (5), $(\circ)$, and shunting (Lemma 8.3(f)):

$$\mathbf{F}(a - p) \sqsubseteq q \Leftrightarrow (a - p) - q = 0_C \Leftrightarrow a - (p \sqcup q) = 0_C \Leftrightarrow \mathbf{F}(a) \sqsubseteq p \sqcup q \Leftrightarrow \mathbf{F}(a) - p \sqsubseteq q\,.$$

We now show the second claim ($\mathbf{F}(a \downharpoonleft p) \sqsubseteq \mathbf{F}(a) \downharpoonleft p$). With Lemma 8.2(c), isotony and shunting we get

$$
\begin{array}{l}
\quad \mathbf{F}(a \downharpoonleft p) \sqsubseteq \mathbf{F}(a) \downharpoonleft p \\
\Leftrightarrow \mathbf{F}(a \downharpoonleft p) \sqsubseteq \mathbf{F}(a) \;\wedge\; \mathbf{F}(a \downharpoonleft p) - p \sqsubseteq 0_C \\
\Leftarrow a \downharpoonleft p \sqsubseteq a \;\wedge\; \mathbf{F}(a \downharpoonleft p) \sqsubseteq p\,.
\end{array}
$$

The left part holds by Lemma 8.3 (a); the right one by Corollary 8.7. □