Nr.: FIN-006-2011

An Algebra for Refactoring and Feature-Oriented Programming

M. Kuhlemann, C. Kästner, S. Apel, and G. Saake

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Technical report

Nr.: FIN-006-2011

An Algebra for Refactoring and Feature-Oriented Programming

M. Kuhlemann, C. Kästner, S. Apel, and G. Saake

Arbeitsgruppe Datenbanken

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
M. Kuhlemann, C. Kästner, S. Apel, and G. Saake
Postfach 4120
39016 Magdeburg
E-Mail: martin.kuhlemann@ovgu.de

http://www.cs.uni-magdeburg.de/Technical_reports.html
Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 01.08.2011

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

An Algebra for Refactoring and Feature-Oriented Programming

Martin Kuhlemann[1], Christian Kästner[2], Sven Apel[3], and Gunter Saake[1]

[1] University of Magdeburg, Universitaetsplatz 2, 39016 Magdeburg, Germany
[2] University of Marburg, Hans-Meerwein Strasse, 35032 Marburg, Germany
[3] University of Passau, Innstrasse 33, 94032 Passau, Germany

Abstract. A *software product line (SPL)* is a set of programs that share features (i.e., user-visible program characteristics) and that differ in features. SPLs are commonly developed by reusing code from a shared code base. The code base of an SPL as well as the individual products are target of refactoring to maintain them or to integrate them with other programs. To develop proper refactoring tools, we must formalize the challenges faced and must distinguish stand-alone-program refactoring from SPL refactoring. We clarify the relationship between refactoring and SPLs, develop a formal, algebraic model of refactorings and SPL-implementation techniques, and prove important properties for the algebraic structure of refactorings in this setting. Developers of future refactoring tools of SPLs can now tackle the challenges in their tools one by one.

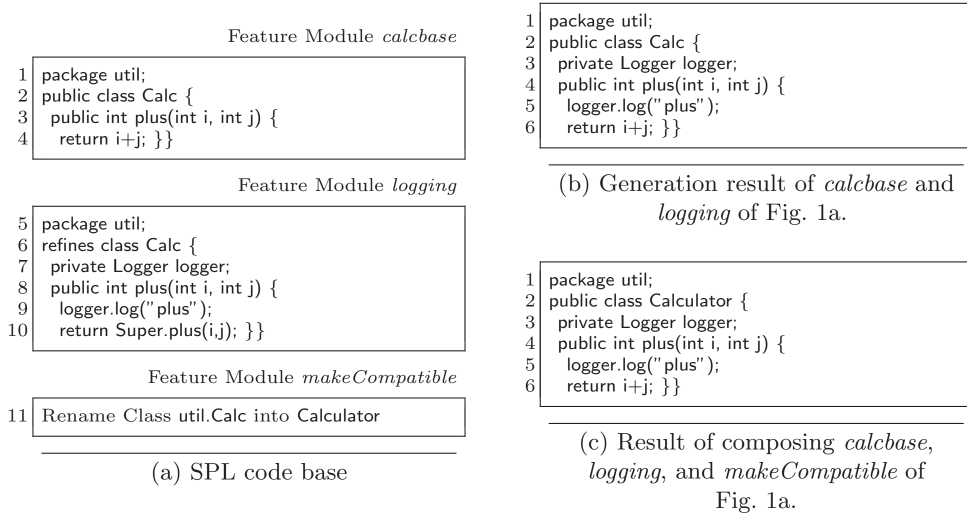
1 Introduction

A *software product line (SPL)* is a set of related programs of a domain [1]; each of these programs in turn is a *product* of the SPL. Products share features (i.e., user-visible program characteristics [2]) and differ in features. Here, we focus only on the SPL-implementation mechanism of program transformations each of which encapsulates the code of one feature (for others see [3]); users select features and, by doing this, they indirectly apply the associated transformations; the transformations then generate a product.

In practice, individual products as well as the entire code base of an SPL are often target to refactoring [4–6]. A *refactoring* is a program transformation that alters the structure of a program but does not alter the program's visible behavior [7]. For example, renaming a class and updating all references to that class is a Rename Class refactoring [6]; once this refactoring is successfully applied to the code base of an SPL, all products of that SPL contain the renamed class [8, 9].

A number of refactoring techniques exists for stand-alone programs [6, 10–12]. To use these tools for SPLs, researchers stated that certain algebraic properties are desirable for refactorings with respect to SPLs, e.g., the existence of identity operations [13, p.13][14, 8, 15]. We investigate whether the structure of refactorings has the desired properties with respect to SPLs, e.g., identity elements,

inverse elements, and distributivity over SPL transformations. To provide an answer, we conceptualize SPLs and refactorings in an algebra; then, with that algebra, we prove properties for the algebraic structure of refactorings with respect to SPLs. From the analyzed properties, we derive requirements for future (SPL-)refactoring techniques and sketch what kind of tools we can expect for SPLs in the future. Based on the analyzed properties, we finally distinguish refactorings from transformations of a sample SPL-implementation technique and show how both can interact.



(a) SPL code base

(b) Generation result of *calcbase* and *logging* of Fig. 1a.

(c) Result of composing *calcbase*, *logging*, and *makeCompatible* of Fig. 1a.

Fig. 1. SPL code base and SPL members.

The contributions of this paper are as follows: We formally prove that there are identity elements in the set of refactorings. We formally prove that there are inverse elements in the set of refactorings. We formally prove that there are no refactorings that distribute over module composition of an SPL in general. We use our results to position refactorings in existing algebras for *feature-oriented programming (FOP)*, to reason about optimization approaches for refactoring engines, and to reason about implementation techniques for SPL-refactoring tools. Though, we show our results for FOP, we argue that our results are general and cover a number of SPL approaches, like ifdefs [3].

2 Background

2.1 Feature-Oriented Programming

An SPL is a set of programs of one domain that share features (i.e., user-visible program characteristics [2]) and differ in features. One way to implement SPLs

is to separate the code of every feature in a distinct module and to compose those modules when selecting their associated features to generate products.

FOP is an approach to implement SPLs. In FOP, a feature is implemented by a *feature module*, which is composed with a program to synthesize a new program – ultimately, a product of the SPL [16]. A feature module is a program transformation that encapsulates classes and class extensions which all are composed with a program this feature module is composed with. A *class extension* adds members to classes and overrides methods of classes in the program.

In Fig. 1a, we show two feature modules *calcbase* and *logging* implemented using the FOP language Jak (please ignore the feature module *makeCompatible* for now). The feature modules are composed consecutively in top-down order: The feature module *calcbase* extends the program it is composed with (the empty program) and adds class Calc (Lines 1-4). The feature module *logging* extends the program it is composed with (the program generated by *calcbase*) by extending the class Calc, i.e., *logging* adds the field logger and overrides the Calc method plus (Jak-overriding keyword Super, Line 10). We show the result of composing the feature modules *logging* and *calcbase* with the empty program in Fig. 1b.

2.2 Refactoring and Refactoring-Feature Modules

Refactorings are transformations that alter the structure of a program but do not alter its visible behavior [7]. Refactorings commonly replace a piece of code in their input program by a behavior-equivalent piece of code. For example, a Rename Class refactoring replaces a class with a behavior-equivalent class that has a new name. Refactorings are used to maintain a program [17] and to integrate a program as a library with a bigger program [7].

A refactoring accepts parameters [12], which are fully-qualified, scoped names of pieces of code in general (*scoped names* for short) which reference the code the refactoring transforms. For instance, a Rename Class refactoring accepts two parameters: the scoped name of the class to rename and the new class name.

Refactoring-feature modules (RFMs) are special feature modules that integrate refactorings with FOP [18]. RFMs allow developers to alter the structure of a program at a high level of abstraction by selecting (refactoring-)features; selecting according features then applies RFMs. An applied RFM finally refactors the code of the program. We introduce RFMs here to ease the discussion on refactorings applied during the generation of a product.

In Fig. 1a, we list an RFM *makeCompatible*; the RFM implements a Rename Class refactoring that accepts the parameter scoped names util.Calc and Calculator to rename class util.Calc into Calculator. The program synthesized from feature modules *calcbase*, *logging*, and *makeCompatible* (see Fig. 1c) contains a class Calculator but no class Calc.

3 An Algebra for Refactorings and Feature Modules

Our algebra is a tupel of terms and operations. In this section, we define the terms (*code*, *error state*, *programs*) and operations (*code composition*, *program contrac-*

tion, *program extension*, and *program composition*) of this structure, which we use to formalize refactorings and feature modules later.

3.1 Terms

Code ($\mathbb{Q}$). We must represent the code of a program in order to make statements about it; but, we can use a simple representation. In our algebra, as a first step, we describe code as a set of scoped names because a number of refactorings alter scoped names, method bodies can be updated in general as needed, the order of scoped names does not matter in common mainstream languages such as in Java or C++, and scoped names are unique in the code of a feature module and of a program written in common mainstream languages.[4] For example, we describe the code of Fig. 1b by the set {util, util.Calc, util.Calc.plus(int,int), util.Calc.logger}. As a first step, we just concentrate on refactorings that reference scoped names and omit refactorings which reference method-local names or statements. In this paper, we use Q_x, Q_y, Q_z, Q_a, and Q_b to refer to sets of scoped names (to range over $\mathbb{Q}$).

Error state ($\mathbb{E} = \{\epsilon, \checkmark\}$). Users must be able to apply the transformation of each feature they select because users desire a program that provides the features they select; however, users do not desire programs that were generated erroneously (those programs do not provide the desired features) and so the application of a transformation should never result in an error [20, 21]. To record errors, we annotate pieces of code in our algebra with error states. Error state ϵ indicates that the piece was generated erroneously, whereas $\checkmark$ indicates that there was no error so far. We use e_1 and e_2 to describe states of programs (to range over $\mathbb{E}$).

Programs ($\mathbb{F} = \mathcal{P}(\mathbb{Q}) \times \mathbb{E}$). In our algebra, a program is a pair of code and error state. The error state of a program records whether the features which were composed to create the program were composed without error – if so, the error state becomes $\checkmark$, and ϵ otherwise. We define: A feature module is a program (code with error state) and the code of a feature module is free of error. For example, we describe the program of Fig. 1b by $\langle$ {util, util.Calc, util.Calc.plus(int, int), util.calc.Logger}; $\checkmark\rangle$.

In our algebra, we define that two programs are equal if the code *as well as* the error state match:

$$\begin{array}{rcl} ((Q_x \neq Q_y) \rightarrow (\langle Q_x; e_1\rangle & \neq & \langle Q_y; e_2\rangle)) \\ (\langle Q_x; \checkmark\rangle & \neq & \langle Q_y; \epsilon\rangle) \\ (\langle Q_x; \epsilon\rangle & = & \langle Q_x; \epsilon\rangle) \\ (\langle Q_x; \checkmark\rangle & = & \langle Q_x; \checkmark\rangle) \end{array} \quad (1)$$

[4] Textual order is important for scoped names of (static) field initializers and array initializers in Java [19, p.203]. We assume that our proofs hold in their presence too because field initializers cannot be parameters of the refactorings we focus on.

3.2 Operations

Code composition ($\cup : \mathcal{P}(\mathbb{Q}) \times \mathcal{P}(\mathbb{Q}) \to \mathcal{P}(\mathbb{Q})$). This operation composes members and classes from different pieces of code as described in Section 2.1 for FOP. In our algebra, code composition corresponds to the union of the sets of scoped names of the pieces of code to compose. We focus on refactorings in this paper, so we ignore errors of code composition.

Program contraction ($\ominus : F \times \mathcal{P}(\mathbb{Q}) \to F$). Refactorings replace code and this can be interpreted as the removal of pieces of code and the subsequent addition of pieces of code; we define program contraction to be the removal of code from the code of a program *during a refactoring.* To remove a piece of code A successfully from a program, A must exist in the program. Program contraction *fails* if the code to remove does not exist. In our algebra, removing code means to remove scoped names from the set of scoped names (when the program to remove code from was generated correctly):

$$(\langle Q_x; e_1\rangle \ominus Q_y) = \begin{cases} \langle (Q_x \backslash Q_y); \checkmark\rangle, & (Q_x \cap Q_y = Q_y) \wedge (e_1 = \checkmark) \\ \langle Q_x; \epsilon\rangle, & otherwise \end{cases} \tag{2}$$

Simplification note: Scoped names reference nested pieces of code, e.g., the scoped name util.Calc references a piece of code Calc that is nested in a piece of code util. So when an operation removes a piece of code in a program, it also removes all pieces nested in the removed piece, and their scoped names respectively. For simplicity, we define that removing a scoped name does not affect other scoped names than the removed one, e.g., $(\{\mathsf{util}, \mathsf{util.Calc}\} \backslash \{\mathsf{util}\}) = \{\mathsf{util.Calc}\}$ although util.Calc is nested in the removed util. As a result, we cannot show positive proofs for refactorings that remove classes or packages; negative proofs however still are general.

Program extension ($\oplus : F \times \mathcal{P}(\mathbb{Q}) \to F$). We define program extension to be the addition of code to the code of a program *during a refactoring.* To add a piece of code A successfully to a program, A is required not to exist in the program code. Program extension thus *fails* when the scoped names to add to a program already exist in the program code or when the program to extend is in error from beginning:

$$(\langle Q_x; e_1\rangle \oplus Q_y) = \begin{cases} \langle (Q_x \cup Q_y); \checkmark\rangle, & (Q_x \cap Q_y = \varnothing) \wedge (e_1 = \checkmark) \\ \langle Q_x; \epsilon\rangle, & otherwise \end{cases} \tag{3}$$

Program composition ($\bullet : F \times F \to F$). Two programs can be composed as described in Section 2.1; however, if one of them has the error state ϵ, the error state is propagated to the result to trace the failure. Code composition in our algebra corresponds to the union of the scoped-name sets of both programs to compose.

$$(\langle Q_x; e_1\rangle \bullet \langle Q_y; e_2\rangle) = \begin{cases} \langle (Q_x \cup Q_y); \checkmark\rangle, & (e_2 = e_1 = \checkmark) \\ \langle Q_x; \epsilon\rangle, & otherwise \end{cases} \tag{4}$$

Case#1 $((Q_x \cap Q_y = Q_y), (Q_x \cap Q_z = \varnothing))$**:**

$R_{Q_z \mapsto Q_y}(R_{Q_y \mapsto Q_z}(\langle Q_x; \checkmark \rangle))$
$\ldots (5)(5)(2)(3)(2)(3)$
$= \langle Q_x; \checkmark \rangle$
$\square$

Case#2 $((Q_x \cap Q_y = Q_y), (Q_x \cap Q_z \neq \varnothing), ((Q_x \backslash Q_y) \cap Q_z \neq \varnothing))$**:**

$R_{Q_z \mapsto Q_y}(R_{Q_y \mapsto Q_z}(\langle Q_x; \checkmark \rangle))$
$\ldots (5)(5)(2)(3)(2)(3)$
$\neq \langle Q_x; \checkmark \rangle$ (1)
$\lightning$

Case#3 $((Q_x \cap Q_y = Q_y), (Q_x \cap Q_z \neq \varnothing), ((Q_x \backslash Q_y) \cap Q_z = \varnothing))$**:**

$R_{Q_z \mapsto Q_y}(R_{Q_y \mapsto Q_z}(\langle Q_x; \checkmark \rangle))$
$\ldots (5)(5)(2)(3)(2)(3)$
$= \langle Q_x; \checkmark \rangle$
$\square$

Case#4 $((Q_x \cap Q_y \neq Q_y))$**:**

$R_{Q_z \mapsto Q_y}(R_{Q_y \mapsto Q_z}(\langle Q_x; \checkmark \rangle))$
$\ldots (5)(5)(2)(3)(2)(3)$
$\neq \langle Q_x; \checkmark \rangle$ (1)
$\lightning$

Fig. 2. Proof for inverse elements.

4 Properties of the Algebraic Structure of Refactorings

In the context of our algebra, refactoring a program can be described as follows: First, a refactoring removes all elements of the set of scoped names Q_x from the set of scoped names of a program; second, the refactoring joins the set of scoped names of the program with a set of new scoped names Q_y ($R : F \times Q \times Q \rightarrow F$). For example, a refactoring that renames class Calc into Calculator in a program, removes the elements of the set of scoped names $Q_x = \{\mathsf{Calc}\}$ from the set of scoped names of the program and joins the set of scoped names of the program with the set of new scoped names $Q_y = \{\mathsf{Calculator}\}$. Generally, all refactorings accept as parameters a set of scoped names Q_x to remove and a set of scoped names Q_y to generate, e.g., a refactoring "Rename Class Calc into Calculator" accepts the set $Q_x = \{\mathsf{Calc}\}$ as parameter and the set $Q_y = \{\mathsf{Calculator}\}$. Refactorings that affect more scoped names than those given to them in Q_x and Q_y, are considered in a *simplified* way to only affect the scoped names of Q_x and Q_y,

e.g., Rename Overridden Method[5] is considered in a simplified way. As the code to remove must exist and the code to create must not exist, we can describe a refactoring with a sequence of program-contraction and program-extension operations. We use this highly simplified representation of refactorings on purpose because it suffices for our purposes. Formally, we describe a refactoring R that removes the set of scoped names Q_y and generates the set of scoped names Q_z in a program F_1 by:

$$R_{Q_y \mapsto Q_z}(F_1) = ((F_1 \ominus Q_y) \oplus Q_z) \tag{5}$$

Next, we prove some properties for the algebraic structure we described for refactorings. We clarify challenges and opportunities for patching refactored products, for optimizing refactoring sequences, and for implementing tools that refactor SPL code bases. For every property, we discuss its potential benefits for SPL engineering and discuss the consequences of the corresponding proof result. If a property is found *not* to hold, we discuss ways to establish the property. If a property holds in a case with an erroneously generated product, we call this a failure because this product is undesired.

4.1 Theorem: An identity element exists in the algebraic structure of refactorings

The identity element in the algebraic structure of refactorings is a refactoring that removes the same set of scoped names which it adds back ($R_{Q_y \mapsto Q_y}(F_1) = F_1$). Refactorings that implement identity elements can be removed without effect on the generated product – this can reduce product-generation time.

In the case ($Q_x \cap Q_y = Q_y$) of our proof (omitted for brevity), we could apply the Equations 5, 2, 3 to the formula $R_{Q_y \mapsto Q_y}(\langle Q_x; \checkmark \rangle)$ to derive the original program $\langle Q_x; \checkmark \rangle$. We thus conclude that the hypothesis holds, i.e., that there are refactorings that implement an identity element.

Consequences. Identity elements *can* be removed to decrease product-generation time.

4.2 Theorem: Inverse elements exist in the algebraic structure of refactorings

If two refactorings, that execute consecutively, are no identity elements but together do not change the transformed program, then they invert each other and can be removed without affecting the generated product; this can reduce product-generation time. Inverting refactorings is also important for patching refactored or optimized products. Inverse elements thus exist if
$R_{Q_y \mapsto Q_x}(R_{Q_x \mapsto Q_y}(F)) = R_{Q_x \mapsto Q_x}(F) = F$.

[5] Renaming an overridden method A renames A along with all methods that override A and that are overridden by A. Scoped names of overriding and overridden methods, however, are not given to the refactoring as parameters.

There are inverse transformations for refactorings, see Fig. 2, Case #1. In Case #1, the refactoring that is to be inverted later, executed without error and so the inverse refactoring reestablishes the original program. In Cases #2, and #4, the preconditions of the first refactoring ($R_{Q_y \mapsto Q_z}(\langle Q_x; \checkmark \rangle)$) are not met and so the refactoring generates an erroneous program – thus, the inverting refactoring fails, as well (an erroneously generated product is undesirable). We conclude that, though there is an inverse refactoring operation for every refactoring, the operation of a particular refactoring can only be inverted by a second refactoring operation if the first refactoring succeeded.

Consequences. We can reduce the time that is required to execute a sequence of refactorings by removing refactorings that follow each other and invert each other. If approaches rely on that there is an inverse refactoring for a refactoring, then these approaches must ensure that every refactoring succeeds.

4.3 Theorem: Refactorings do not distribute over program extension

Different researchers stated their desire for refactorings being distributive over program composition [13, p.13][14, 8, 15]. If refactorings distribute over program composition, then we could refactor SPL code bases by refactoring every feature module independently. If refactorings distribute over program composition, then we could refactor products of an SPL with different approaches – either compose programs (feature modules) and refactorings (RFMs) successively (Fig. 4, left perimeter) or refactor the individual programs first and compose the refactored programs later (Fig. 4, right perimeter). If refactorings distribute over program composition, then we could reuse tools, which deal with the refactoring of stand-alone programs, to refactor SPL code bases. If refactorings distribute over program composition, then we could reorder refactorings and feature modules in mixed sequences of refactorings and feature modules, in order to re-group refactorings and feature modules (desired in [22, 13]).

We show representative cases of the proof in Fig. 3 and indicate the complete proof in Section 7. Case #1 justifies the desire of refactorings and program composition being distributive, i.e., there are situations in which refactoring distributes over program composition. In Case #1, refactoring distributes over program composition because the scoped names to refactor exist in every feature module of the SPL ($\langle Q_x; \checkmark \rangle$, $\langle Q_y; \checkmark \rangle$) and the scoped names, which the refactoring requires not to exist, do not exist in any feature module. However, with the counter examples in Case #2 and many cases in Section 7, we prove formally that refactoring *does not* distribute over program composition in general. Furthermore, from all cases of this proof (shown in Fig. 3 and Sec. 7), we can count that distributivity in theory is the exception rather than the rule – it holds in 3 *special* cases (cases #1, #5, #8; 4 preconditions in average) but it does not hold in 7 *general* cases (cases #2, #3, #4, #6, #7, #9, #10; 3 preconditions in average).

Case #1 $((Q_x \cap Q_a = Q_a), (Q_y \cap Q_a = Q_a), (Q_x \cap Q_b = \varnothing), (Q_y \cap Q_b = \varnothing))$**:**

$R_{Q_a \mapsto Q_b}(\langle (Q_x \cup Q_y); \checkmark \rangle)$
$= ((\langle (Q_x \cup Q_y); \checkmark \rangle \ominus Q_a) \oplus Q_b)$ (5)
$= (\langle ((Q_x \cup Q_y) \backslash Q_a); \checkmark \rangle \oplus Q_b)$ (2)
$= \langle (((Q_x \cup Q_y) \backslash Q_a) \cup Q_b); \checkmark \rangle$ (3)
$= \langle (((Q_x \backslash Q_a) \cup (Q_y \backslash Q_a)) \cup Q_b); \checkmark \rangle$
$= \langle (((Q_x \backslash Q_a) \cup Q_b) \cup ((Q_y \backslash Q_a) \cup Q_b)); \checkmark \rangle$
$= (\langle ((Q_x \backslash Q_a) \cup Q_b); \checkmark \rangle \bullet \langle ((Q_y \backslash Q_a) \cup Q_b); \checkmark \rangle)$ (4)
$= (\langle ((Q_x \backslash Q_a) \cup Q_b); \checkmark \rangle \bullet (\langle (Q_y \backslash Q_a); \checkmark \rangle \oplus Q_b))$ (3)
$= (\langle ((Q_x \backslash Q_a) \cup Q_b); \checkmark \rangle \bullet ((\langle Q_y; \checkmark \rangle \ominus Q_a) \oplus Q_b))$ (2)
$= (\langle ((Q_x \backslash Q_a) \cup Q_b); \checkmark \rangle \bullet R_{Q_a \mapsto Q_b}(\langle Q_y; \checkmark \rangle))$ (5)
$= ((\langle (Q_x \backslash Q_a); \checkmark \rangle \oplus Q_b) \bullet R_{Q_a \mapsto Q_b}(\langle Q_y; \checkmark \rangle))$ (3)
$= (((\langle Q_x; \checkmark \rangle \ominus Q_a) \oplus Q_b) \bullet R_{Q_a \mapsto Q_b}(\langle Q_y; \checkmark \rangle))$ (2)
$= (R_{Q_a \mapsto Q_b}(\langle Q_x; \checkmark \rangle) \bullet R_{Q_a \mapsto Q_b}(\langle Q_y; \checkmark \rangle))$ (5)
$\square$

Case #2 $((Q_x \cap Q_a = Q_a), (Q_y \cap Q_a \neq Q_a), (Q_x \cap Q_b = \varnothing), (Q_y \cap Q_b = \varnothing))$**:**

$R_{Q_a \mapsto Q_b}(\langle (Q_x \cup Q_y); \checkmark \rangle)$
$= ((\langle (Q_x \cup Q_y); \checkmark \rangle \ominus Q_a) \oplus Q_b)$ (5)
$= (\langle ((Q_x \cup Q_y) \backslash Q_a); \checkmark \rangle \oplus Q_b)$ (2)
$= \langle (((Q_x \cup Q_y) \backslash Q_a) \cup Q_b); \checkmark \rangle$ (3)
$\neq \langle ((Q_x \backslash Q_a) \cup Q_b); \epsilon \rangle$ (1)
$= (\langle ((Q_x \backslash Q_a) \cup Q_b); \checkmark \rangle \bullet \langle Q_y; \epsilon \rangle)$ (4)
$= (\langle ((Q_x \backslash Q_a) \cup Q_b); \checkmark \rangle \bullet (\langle Q_y; \epsilon \rangle \oplus Q_b))$ (3)
$= (\langle ((Q_x \backslash Q_a) \cup Q_b); \checkmark \rangle \bullet ((\langle Q_y; \checkmark \rangle \ominus Q_a) \oplus Q_b))$ (2)
$= (\langle ((Q_x \backslash Q_a) \cup Q_b); \checkmark \rangle \bullet R_{Q_a \mapsto Q_b}(\langle Q_y; \checkmark \rangle))$ (5)
$= ((\langle (Q_x \backslash Q_a); \checkmark \rangle \oplus Q_b) \bullet R_{Q_a \mapsto Q_b}(\langle Q_y; \checkmark \rangle))$ (3)
$= (((\langle Q_x; \checkmark \rangle \ominus Q_a) \oplus Q_b) \bullet R_{Q_a \mapsto Q_b}(\langle Q_y; \checkmark \rangle))$ (2)
$= (R_{Q_a \mapsto Q_b}(\langle Q_x; \checkmark \rangle) \bullet R_{Q_a \mapsto Q_b}(\langle Q_y; \checkmark \rangle))$ (5)
$\lightning$

Fig. 3. Proof of distributivity of refactorings and program composition.

Consequences. It is impossible in general that existing refactoring tools for standalone programs can be used to refactor the code base of an SPL; for that, we must change our expectations in this direction. It is impossible to re-group sequenced feature modules and RFMs in general as found desirable in models of FOP approaches, e.g., for having an additional (normalized) way of generating an SPL product [22, 13].

Solutions. We proved that a generally distributive refactoring operation does not exist though it is desirable. As a *trade-off*, tools can distribute *parts* of a refactoring operation over feature modules. For distributing parts of refactoring operations over feature modules (e.g., to refactor an SPL), tools must evaluate sophisticated error states at the level of an individual feature module, i.e., a global check remains and not the whole refactoring is distributed ($R_{Q_a \mapsto Q_b}((Q_x \cup$

$Q_y)) = R_\int((R_{Q_a \mapsto Q_b}(Q_x) \bullet R_{Q_a \mapsto Q_b}(Q_y))))$. However, the global test ($R_\int$) for success of refactorings at the feature-module level is complex because it must distinguish *non-critical* and *critical* error states from the feature-module level. Non-critical error states in feature-module-level refactorings do not harm the success of the global refactoring but critical do.

For illustration, we now discuss critical and non-critical errors for Rename Class: the refactoring at the feature-module level must return whether the class to rename existed in the transformed feature module and whether a class with the name to generate existed. The absence of the class to rename is non-critical when the class exists in *at least* one feature module; the absence of the class to rename is critical when the class does not exist *in any* feature module. The presence of the class-to-generate in a feature module can be alerted as critical error, or a tool could evaluate that this class will not cause composition errors in in any legal product of that SPL.

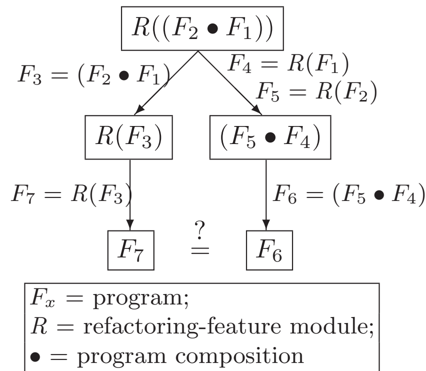


Fig. 4. Does refactoring composition results $(R((F_2 \bullet F_1)))$ and refactoring-feature modules before composition $((R(F_2) \bullet R(F_1)))$ yield equal programs?

To check in one feature module whether a field in one class extension can be renamed without accidentally overriding an existing field of a superclass is even harder. When all products share a common structure with respect to class names and class relations (advised in [8]) then we can decide whether two fields can override each other.

Inline Method refactoring cannot be distributed over current feature modules *at all* because the method must be composed first before its body can replace calls. So, new FOP mechanisms would be needed, like statement refinements.

Table 1. Comparision of feature modules with RFMs.

Algebraic Property	SPL Transform.‡	RFM
Identity element	✓	✓
Inverse element	✗	✓
Distribut.* (R & $\bullet$)	✗	

‡shown in [22]; ✓property holds; ✗ property does not hold; *can only hold for *multiple* operations

4.4 Discussion on Generality

Some of our proofs are limited in generality. We did not consider (static) field initializers and array initializers. We did not consider refactorings that accept statements or method-local names as parameters. We assumed that method-local names can be adapted when conflicts with scoped names occur. We assumed that removing a scoped name does not remove other scoped names. These simplifications still match the refactorings Rename Monomorphic Method, Move Method,

Rename Field, and Move Field, i.e., the positive proofs hold for these refactorings. However, we proved that distributivity does *not* hold in general for any refactoring in our simplified representation of code and refactorings – thus, distributivity will also not hold for full-fletched representations of Java programs, Jak programs, and refactorings.

4.5 Lessons learned

Refactorings are complex program transformations. However, by looking at scoped names of code we already could prove properties of refactorings. We learned from these properties that there are numbers of identity elements in the set of refactorings, that there are numbers of inverse operations in the set of refactorings, and that refactorings do not distribute over feature modules, in general. In Tab. 1, we summarize the properties of the algebraic structure of refactorings and compare them to algebraic properties of the structure of feature modules (proven before [22]).

Tools may implement a refactoring by adding feature modules that replace all feature modules with an obsolete structure. To implement the same functionality multiple times, however, is laborious and error-prone.

5 Related Work

Batory et al. listed desirable properties for the algebraic structure of refactorings and argued some of these properties to hold [13, 14, 8, 15]. We formally proved properties for the algebraic structure of refactorings in the context of SPLs and showed that some algebraic properties do not hold. Apel et al. formally proved algebraic properties of different FOP approaches [22], but not for refactorings.

Lynagh proposed an algebra of patches [23]. In this algebra, Lynagh resolves conflicts between *concurrent* patch sequences. We focus on refactorings which have different preconditions than patches; thus, the properties Lynagh showed do not hold all for refactorings.

Alves et al. refactor models of SPLs [24] but these models are far from being code. Different researchers refactored code of SPLs by extracting the code of features into new feature modules [25, 5, 26]; in contrast, we focus on object-oriented refactorings, e.g., the renaming or moving of classes in an SPL product.

Vittek applied refactorings to the code base of an ifdef-based SPL and demonstrated problems [27]. In contrast, we conceptualized refactorings and thus described challenges in refactoring SPLs, formally. The challenges we formalized can now be used to evaluate arbitrary refactoring engines.

6 Conclusions

We compared and distinguished program transformations used in software product lines (SPLs) versus refactorings. Specifically, we defined an algebra that integrates SPL transformations and refactorings and proved algebraic properties

that we and others desired. However, we also proved that certain intuitive assumptions about desirable properties for the algebraic structure of refactorings do *not* hold, e.g., distributivity of refactorings over SPL transformations. That is, we formally proved that we cannot refactor SPL transformations with the same techniques and tools that exist for programs. With the proof results, we also explained *why* expectations in refactoring tools for SPLs should be changed. We formalized the challenges that tools must overcome in order to refactor SPL code bases and we sketched solutions; in future work, refactoring tools for SPLs can be evaluated using the individual challenges that we describe.

7 Appendix: Remaining Cases of Distributivity-Proof

Case #3 $((Q_x \cap Q_a = Q_a), (Q_x \cap Q_b = \varnothing), (Q_y \cap Q_b \neq \varnothing))$**:**

$R_{Q_a \mapsto Q_b}(\langle (Q_x \cup Q_y); \checkmark \rangle)$
$\ldots (5)(2)$
$= \langle ((Q_x \cup Q_y) \backslash Q_a); \epsilon \rangle$ (3)
↯

Case #4 $((Q_y \cap Q_a = Q_a), (Q_x \cap Q_b \neq \varnothing), (Q_y \cap Q_b = \varnothing))$**:**

$R_{Q_a \mapsto Q_b}(\langle (Q_x \cup Q_y); \checkmark \rangle)$
$\ldots (5)(2)$
$= \langle ((Q_x \cup Q_y) \backslash Q_a); \epsilon \rangle$ (3)
↯

Case #5 $((Q_x \cap Q_a = Q_a), (Q_y \cap Q_a = Q_a), Q_a = Q_b)$**:**

$R_{Q_a \mapsto Q_b}(\langle (Q_x \cup Q_y); \checkmark \rangle)$
$\ldots (5)(2)(3)(4)(3)(2)(3)(2)$
$= (R_{Q_a \mapsto Q_b}(\langle Q_x; \checkmark \rangle) \bullet R_{Q_a \mapsto Q_b}(\langle Q_y; \checkmark \rangle))$ (5)
□

Case #6 $((Q_x \cap Q_a = Q_a), (Q_y \cap Q_a = Q_a), Q_a \neq Q_b, ((Q_x \backslash Q_a) \cap Q_b \neq \varnothing))$**:**

$(R_{Q_a \mapsto Q_b}(\langle Q_x; \checkmark \rangle) \bullet R_{Q_a \mapsto Q_b}(\langle Q_y; \checkmark \rangle))$
$\ldots (5)(2)(3)$
$= \langle (Q_x \backslash Q_a); \epsilon \rangle$ (4)
↯

Case #7 $((Q_x \cap Q_a = Q_a), (Q_y \cap Q_a = Q_a), Q_a \neq Q_b, ((Q_x \backslash Q_a) \cap Q_b = \varnothing), ((Q_y \backslash Q_a) \cap Q_b \neq \varnothing))$**:**

$(R_{Q_a \mapsto Q_b}(\langle Q_x; \checkmark \rangle) \bullet R_{Q_a \mapsto Q_b}(\langle Q_y; \checkmark \rangle))$
$\ldots (5)(2)(3)(5)(2)(3)$
$= \langle ((Q_x \backslash Q_a) \cup Q_b); \epsilon \rangle$ (4)
↯

Case #8 $((Q_x \cap Q_a = Q_a), (Q_y \cap Q_a = Q_a), Q_a \neq Q_b, ((Q_x \backslash Q_a) \cap Q_b = \varnothing), ((Q_y \backslash Q_a) \cap Q_b = \varnothing))$**:**

$(R_{Q_a \mapsto Q_b}(\langle Q_x; \checkmark \rangle) \bullet R_{Q_a \mapsto Q_b}(\langle Q_y; \checkmark \rangle))$
...(5)(2)(3)(5)(2)(3)(4)(3)(2)
$= R_{Q_a \mapsto Q_b}(\langle (Q_x \cup Q_y); \checkmark \rangle)$ (5)
□

Case #9 $((Q_y \cap Q_a \neq Q_a))$**:**

$(R_{Q_a \mapsto Q_b}(\langle Q_x; \checkmark \rangle) \bullet R_{Q_a \mapsto Q_b}(\langle Q_y; \checkmark \rangle))$
...(5)(2)(3)
$= \langle Q_z; \epsilon \rangle$ (4)
↯

Case #10 $((Q_x \cap Q_a \neq Q_a))$**:**

$(R_{Q_a \mapsto Q_b}(\langle Q_x; \checkmark \rangle) \bullet R_{Q_a \mapsto Q_b}(\langle Q_y; \checkmark \rangle))$
...(5)(2)(3)
$= \langle Q_x; \epsilon \rangle$ (4)
↯

References

1. C. Krueger, New methods in software product line practice, Communications of the ACM 49 (12) (2006) 37–40.
2. K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Tech. Rep. CMU/SEI-90-TR-21, Carnegie Mellon University Pittsburgh, USA (1990).
3. C. Kästner, S. Apel, M. Kuhlemann, Granularity in software product lines, in: Proceedings of the International Conference on Software Engineering, 2008, pp. 311–320.
4. M. Monteiro, J. Fernandes, Object-to-aspect refactorings for feature extraction, in: Proceedings of the International Conference on Aspect-Oriented Software Development, 2004.
5. J. Liu, D. Batory, C. Lengauer, Feature-oriented refactoring of legacy applications, in: Proceedings of the International Conference on Software Engineering, 2006, pp. 112–121.
6. M. Fowler, Refactoring: Improving the design of existing code, Addison-Wesley Longman Publishing Co., Inc., 1999.
7. W. Opdyke, Refactoring object-oriented frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign (1992).
8. D. Batory, A modeling language for program design and synthesis, in: Advances in Software Engineering, Vol. 5316 of Lecture Notes in Computer Science, 2007, pp. 39–58.
9. D. Batory, M. Azanza, J. Saraiva, The objects and arrows of computational design, in: Proceedings of the International Conference on Model Driven Engineering Languages and Systems, 2008, pp. 1–20.
10. M. Ó. Cinnéide, P. Nixon, Composite refactorings for Java programs, in: Workshop on Formal Techniques for Java Programs, 2000, pp. 129–135.
11. G. Kniesel, H. Koch, Static composition of refactorings, Science of Computer Programming 52 (1-3) (2004) 9–51.

12. T. Mens, N. Van Eetvelde, S. Demeyer, D. Janssens, Formalizing refactorings with graph transformations, Software Maintenance and Evolution: Research and Practice 17 (4) (2005) 247–276.
13. D. Batory, D. Smith, Finite map spaces and quarks: Algebras of program structure, Tech. Rep. TR-04-66, University of Texas at Austin, USA (2007).
14. D. Batory, Program refactoring, program synthesis, and model-driven development, in: Proceedings of the International Conference on Compiler Construction, Vol. 4420 of Lecture Notes in Computer Science, 2007, pp. 156–171.
15. D. Batory, On the importance and challenges of FOSD, keynote at International Workshop on Feature-Oriented Software Development [Available online: `http://www.infosun.fim.uni-passau.de/cl/staff/apel/FOSD2009/BatoryFOSDKeynote2.pdf`] (2009).
16. D. Batory, J. Sarvela, A. Rauschmayer, Scaling step-wise refinement, IEEE Transactions on Software Engineering 30 (6) (2004) 355–371.
17. D. Roberts, Practical analysis for refactoring, Ph.D. thesis, University of Illinois at Urbana-Champaign (1999).
18. M. Kuhlemann, D. Batory, S. Apel, Refactoring feature modules, in: Proceedings of the International Conference on Software Reuse, Vol. 5791 of Lecture Notes in Computer Science, 2009, pp. 106–115.
19. J. Gosling, B. Joy, G. Steele, G. Bracha, The Java language specification, 3rd Edition, Addison-Wesley Longman Publishing Co., Inc., 2005.
20. S. Thaker, D. Batory, D. Kitchin, W. Cook, Safe composition of product lines, in: Proceedings of the International Conference on Generative Programming and Component Engineering, 2007, pp. 95–104.
21. M. Kuhlemann, D. Batory, C. Kästner, Safe composition of non-monotonic features, in: Proceedings of the International Conference on Generative Programming and Component Engineering, Vol. 45 of ACM SIGPLAN Notices, 2009, pp. 177–186.
22. S. Apel, C. Lengauer, B. Möller, C. Kästner, An algebraic foundation for automatic feature-based program synthesis, Science of Computer Programming 75 (11) (2010) 1022–1047.
23. I. Lynagh, An algebra of patches, [Available online: `http://urchin.earth.li/~ian/conflictors/paper-2006-10-30.pdf`] (2006).
24. V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, C. Lucena, Refactoring product lines, in: Proceedings of the International Conference on Generative Programming and Component Engineering, 2006, pp. 201–210.
25. C. Kästner, M. Kuhlemann, D. Batory, Automating feature-oriented refactoring of legacy applications, in: Proceedings of the Workshop on Refactoring Tools, 2007, pp. 62–63, also presented as poster at ECOOP.
26. S. Trujillo, D. Batory, O. Diaz, Feature refactoring a multi-representation program into a product line, in: Proceedings of the International Conference on Generative Programming and Component Engineering, 2006, pp. 191–200.
27. M. Vittek, Refactoring browser with preprocessor, in: Proceedings of the European Conference on Software Maintenance and Reengineering, 2003, pp. 101–110.