Features, Modularity, and Variation Points

Don Batory
Dept. of Computer Science
University of Texas at Austin
batory@cs.utexas.edu

Peter Höfner
NICTA, Australia
University of New South Wales, Australia
peter.hoefner@nicta.com.au

Bernhard Möller, Andreas Zelend
Universität Augsburg, Germany
{bernhard.moeller,zelend}
@informatik.uni-augsburg.de

Abstract

A *feature interaction algebra (FIA)* is an abstract model of features, feature interactions, and their compositions. A *structured document algebra (SDA)* defines modules with variation points and how such modules compose. We present both FIA and SDA in this paper, and homomorphisms that relate FIA expressions to SDA expressions. This leads to mathematically precise formalizations of fundamental concepts used in software product lines, which can be used for improved FOSD tooling and teaching material.

Categories and Subject Descriptors D.2.10 [*SoftwareEngineering*]: Design; D.2.8 [*Software Engineering*]: Software Architectures

General Terms Theory

Keywords software product lines, features, FOSD

1. Introduction

Feature Oriented Software Development (FOSD) is a paradigm that is guided by semantic modularity, rather than structural modularity. Structural modularity is what today's programming languages and component-based tools provide: object-oriented classes and packages are the basic modules of modern software system construction. Features, in contrast, are increments in program functionality that arise from a collaboration or coordinated orchestration of multiple classes or packages. Adding a new feature to a program "cross-cuts" the contents of structural modules and thus requires fundamentally new ways to think about program modularity and system construction.

A distinguishing attribute of FOSD is compositionality: programs are constructed by "composing" sets of features. An end-user simply selects features from a feature model to specify a target program of interest. FOSD tools compose these features and translate this declarative specification into the target program (source, executable) automatically. This is FOSD's most attractive and powerful characteristic.

We are interested in algebraic models of program construction for FOSD. The reason is simple: features manipulate program structures in very regular, well-defined ways. Mathematics offers structures for precise definitions and calculations; it is well suited for for compositional descriptions of the universe, including software. But more pragmatically, our interests stem from a deep desire to expose the simplicity of the foundational concepts of FOSD. A better understanding of these concepts leads to the construction of simpler and more powerful tools for the FOSD community, as well as improved teaching material for community members of the future.

In this paper, we explore two very different algebraic models of FOSD construction and explain how they are related. The first, called the *Feature Interaction Algebra (FIA)*, axiomatizes the ideas of features, feature interaction, and feature compositions. It generalizes prior work [7] beyond classical feature models, and lays a foundation for others to build upon and to critique compositional models that are appropriate for FOSD "problem space" descriptions of *software product lines (SPLs)*. A program is defined in FIA as a feature expression (composition); FIA is general in that it does not specify how features are implemented.

Our second model, called the *Structured Document Algebra (SDA)*, is a general algebra for describing implementation techniques, modules, and their compositions; hence it belongs to the "solution space" of SPLs. SDA is based on *variation points (VPs)*. A VP is a location in a program where the content at that location can vary among different members of an SPL. A fragment is assigned to a VP to define its content; different programs of an SPL would assign different fragments. An SDA module is a collection of fragments, and compositions of SDA modules construct programs.

Lastly, we show how FIA and SDA algebras are related. We define how FIA expressions are mapped to corresponding SDA expressions to achieve FOSD program synthesis. More generally, when both the problem and solution space are defined by algebras (one for program specification, the other for program construction), mappings between algebras, called homomorphisms, become a foundational concept in FOSD tooling. We illustrate all of these ideas in this paper.

2. Feature Interaction Algebra

[7] presented an axiomatization of features, compositions, and interactions. The goal was to understand their fundamental relationships from an algebraic viewpoint, thereby abstracting away specifics of underlying languages, program representations, and hacks that originate from some implementation decision made in particular systems. Its axioms are listed in Figure 1. The basic operations are: $+$ the feature composition operation, # the feature interaction operation, and $\times$ the feature product operation.

To understand Figure 1, consider a feature model from telephony, where a customer can select the features he/she wants. Among the selectable features are *call forwarding (CF)* and *call waiting (CW)*. CF enables a customer to specify a secondary phone number to which additional calls are forwarded when the primary phone is busy. CW allows one call to be suspended while another

FOSD '13, October 26, 2013, Indianapolis, IN, USA.

http://dx.doi.org/10.1145/2528265.2528269

+ identity: $A + 0 = A$	# annihilate: $A \# 0 = 0$
+ commute: $A + B = B + A$	# commute: $A \# B = B \# A$
+ associate: $A + (B + C) = (A + B) + C$	# associate: $A \#(B \# C) = (A \# B) \# C$
distributivity: $A \#(B + C) = (A \# B) + (A \# C)$	$\times$ product: $A \times B = (A \# B) + A + B$
+ involution $A + A = 0$	# involution $A \# A = 0$

Figure 1. The Axioms of [7]

call is answered. If both features are selected by a customer and a call comes in while another is active, the phone system has to decide whether the call should be forwarded or the user should be notified that another call has arrived. CF and CW are said to *interact*. The resolution (problem fix) is provided by *another* feature CW # CF. Without a resolution, the phone system may behave unexpectedly or terminate erroneously.

So when a user or architect selects two (or more) features, they want their interaction resolutions to be included. This is the purpose of the *product* operation:

$$A \times B = (A \# B) + A + B\,.$$

That is, when architects want features A and B, they also want both features to work correctly together, which requires the addition (composition) of A and B with their interaction resolution A#B. This scales to the product of an arbitrary number of features; the product of 3 features may require the resolution of a 3-way interaction and three 2-way interactions:

$$A \times B \times C = A + B + C + (A \# B) + (B \# C) + (A \# C) + (A \# B \# C).$$

Recognize that each term of a summation is a placeholder: there may be no two-way interaction between B and C, or there may be no three-way interaction. In that case, these terms map to 0. The algebra exposes all possible interactions, as it should.

2.1 Feature Replication and Self-Composition

The core axioms of [7] are unshaded in Figure 1. The shaded axioms, + and # involution, match classical feature models where features could either be selected or not: feature replication was not permitted. Involution captures this constraint. It was proven in [7] that the axioms of Figure 1 are consistent and not contradictory.

It turns out we cannot just pick and choose axioms at will: not all combinations make sense. Consider the A # 0 axiom. There are only two possibilities: either $A \# 0 = 0$ (listed in Figure 1) or $A \# 0 = A$, meaning that the resolution of A and 0 should be A. Together with involution, the latter leads to inconsistency:

$$A = A\#0 = A\#(0+0) = A\#0 + A\#0 = A + A = 0.$$

That means that there is only one feature, the neutral element 0.

Many people have raised the question: what about feature replication? It does arise. How does this effect an axiomatization? We explored this too.

There are three possibilities for self-summation: $A + A = 0$ (involution of Figure 1), $A + A = A$ (idempotence), or we say nothing about the meaning and simplication of $A + A$. We explored $A + A = 0$ in [7]. Now let's consider idempotence $A + A = A$, as others have done, e.g. [1]. Given this, we have three possibilities for self-interaction: $A \# A = 0$, $A \# A = A$, or we can say nothing. Suppose $A \# A = A$. The following shows that the combination of $A + A = A$ and $A \# A = A$ leads to nonsense. Assume a feature, which is the sum of two features A and B, then:

$$\begin{aligned} A + B &= (A + B)\#(A + B) = A\#A + A\#B + B\#A + B\#B \\ &= A + A\#B + B = (A\#B) + A + B = A \times B \end{aligned}$$

The above means that the sum of any two features always equals their product, which we know is false. By enumerating all possible cases (there aren't many), idempotence-summation ($A + A = A$) is not compatible with our other axioms as it leads to inconsistencies or unwanted behaviour. We can rule out involution-summation $A + A = 0$, as it is incompatible with feature replication. Thus, we can say nothing about self-summation ($A + A$). For more details on this, see [8].

So where does this leave us? Mathematics tells us that at a deep level, feature replication does not make sense and the difficulties that we are having lie elsewhere.

Here is our proposed resolution: *All features are distinct*. If there are multiple replicants of A, they will be distinguished as $A_1, A_2, \ldots$ where $A_i \neq A_j$ when $i \neq j$. By doing so, we are able to distinguish resolutions among different instances (i.e., $A_1 \# A_2$ will be distinguishable from $A_2 \# A_3$.) Think of a replicatable feature A as a template or generic: it can be instantiated any number of times but with different parameters. Together a templatized feature and its parametric instantiations leads to a distinct, unparameterized features. This is, in fact, how feature replication was dealt with in an early example of software product lines [5]. It is consistent with the most recent proposal of the *Common Variability Language (CVL)* [10].

CVL goes further in permitting features to be products of other product lines. So if product line $\mathcal{P}$ has products $P_1 \ldots P_n$, it is possible in CVL for designers to select $k \geq 1$ distinct products from $\mathcal{P}$ as features of a larger product line. The products that are selected may internally use the same feature F, but again, these replicas would be distinct and distinguishable copies of F.

2.2 Recap

[7] presented an axiomatization of features, feature interactions and their compositions *for classical feature models, where feature replication was not permitted*. A generalization of this axiomatization to permit feature replication requires the removal of its self-summation and self-interaction axioms. Further, the axiomatization reveals that "pure" replication – where replicas are completely indistinguishable – is problematic. By insisting that feature replicas are distinguishable, ambiguity can be avoided. Our axioms remain consistent and not contradictory.

With this improvement, we now consider a rather different algebra that is useful at the implementation (or solution-space) level.

3. Structured Document Algebra

A classical concept in SPL construction is the variation point (VP). A VP is a labeled position in a program or document where contents can differ among programs in an SPL.

We now present a formal model of VPs, modules containing VPs, and compositions of such modules as the Structured Document Algebra (SDA). To keep SDA language-independent, we leave the exact nature of fragments open (e.g. text or AST) and view it as a parameter of the algebra. We do illustrate the core of SDA in our figures, and when we think it necessary, to explain how SDA could be implemented.

3.1 Variation Points and Fragments

The basic ingredients of SDA are:

- a set V of VPs at which fragments may be inserted;

- a set $\mathtt{F(V)}$ of *fragments* which may, among other things, contain VPs from $\mathtt{V}$.

We use a very broad notion of VPs and fragments. Until stated otherwise, what we present below is standard for coloring [16] and a VP interpretation of classical modularity.

Consider Figure 2a. It shows a Java file that defines class `A`. Three VPs and their associated fragments, indicated by bold left parentheses, are shown: $\mathtt{vp_a}$, $\mathtt{vp_b}$, and $\mathtt{vp_c}$. $\mathtt{vp_a}$ is a location in a directory at which a file can appear. (Such a VP is called a classpath). To us, this file is a fragment assigned to $\mathtt{vp_a}$. It is not the only fragment/file that could be assigned to $\mathtt{vp_a}$; another possibility is the file of Figure 2b. At most one of these two files/fragments can ever be assigned to $\mathtt{vp_a}$ at a time. This holds for all VPs—at most one fragment can be assigned to a VP at any one time. If there is no assignment, there is no file. We then say that the content of $\mathtt{vp_a}$ is *empty*. Emptiness may hold for all VPs.

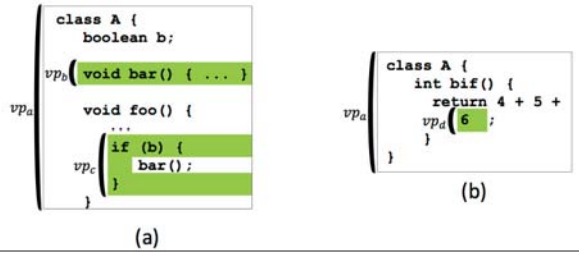

Figure 2. VPs and Fragments.

Now consider VPs $\mathtt{vp_b}$ and $\mathtt{vp_c}$ that are contained in the file fragment of Figure 2a. The fragment assigned to $\mathtt{vp_b}$ defines a method `bar`. The fragment at $\mathtt{vp_c}$ is a wrapper of the statement that calls `bar`. We say fragments that are internal to a file *fill* a VP and may wrap a VP [7, 16]. By wrapping we mean that a fragment `f` surrounds a VP, making `f` to appear that the VP is inside it.

There is one more possibility: *default fragments*. In general every VP needs a default fragment. All VPs we have seen so far had empty defaults, which is the normal case. But default values need not to be empty; consider $\mathtt{vp_d}$ in Figure 2b. The fragment containing the number `6` fills $\mathtt{vp_d}$. But this VP has a default (not shown) so that if the fragment `6` is removed, the empty fragment cannot be default. Reason: the resulting code would be syntactically incorrect. The default fragment should be a natural number, to make the expressions semantically meaningful. Upon module composition, default fragments can be replaced by non-default fragments, but not vice versa.

The above are standard ideas for modules with VPs. Our work generalizes these ideas. It is typical in coloring and the SPL literature that: (1) a VP occurs only once in a SPL program and (2) the set of fragments that can be assigned to that VP are unique to that VP. SDA imposes no such limitations. A VP can appear in multiple places in a document (or documents); when one instance is assigned, they are all assigned the same fragment. Similarly, a single fragment need not be assigned to a unique VP; a fragment can be assigned to multiple distinct VPs. Both of these possibilities should be familiar to readers: aspects in AOP have advice (in the form of fragments) that can be applied to different join points (VPs).

Finally, a word on our above-mentioned "VP interpretation of classical modularity". In classical modularity, a VP corresponds to an interface and a fragment implements that interface. Delaware et al. has shown that a formal (programming language) interface for a VP can be quite sophisticated, and so, too, can the fragment(s) that implement it [12]. And again, a VP has a default fragment (implementation) that can be overridden once by a non-default fragment (implementation).

3.2 SDA Basics

Modules. A *module* is a partial function $\mathtt{m} : \mathtt{V} \rightsquigarrow \mathtt{F(V)}$ such that its domain $\mathtt{dom(m)}$ is finite. VP $\mathtt{v}$ is *assigned* by $\mathtt{m}$ if $\mathtt{v} \in \mathtt{dom(m)}$, otherwise *unassigned* or *external*. Thus the domain $\mathtt{dom(m)}$ of a module is the set of VPs it "knows about" or that it administers.

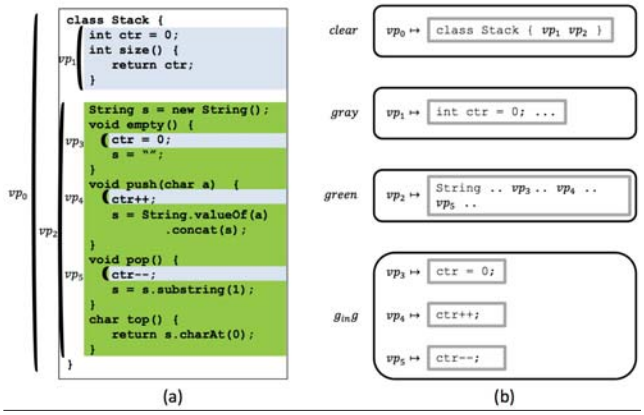

Figure 3. VPs, Fragments, and Modules.

A module $\mathtt{m}$ can be viewed in two ways:

- as a collection of fragments that instantiate the VPs of $\mathtt{dom(m)}$, i.e., a structured document;
- as filling certain VPs with contents (in term rewriting etc., it would be called a *substitution*).

EXAMPLE 3.1. *Figure 3a is a sample file (module) which is structured by the assignment of fragments to its VPs. Its partial function is given in Figure 3b. Here variation points (and their corresponding fragments) are also grouped.* □

By using partial functions rather than relations, a VP can be filled with at most one fragment (*uniqueness*).

A module should be cycle-free—no VP must depend directly or indirectly on itself. The simplest module is the *empty module* `0`, i.e., the empty partial map. Since $\mathtt{dom(0)} = \emptyset$, the empty module has no VPs.

Module Addition. We want to construct larger modules step by step by assigning more and more fragments to VPs. The central operation for this is module addition ($+$). Addition fuses two modules while maintaining uniqueness (and signaling an error upon a conflict). Desirable properties for $+$ are commutativity and associativity. If the modules to be combined have no VPs in common, the partial functions characterizing the modules can be easily combined. For example, $\mathtt{gray} + \mathtt{green}$ (Figure 3) is the partial function

$$\{\ \mathtt{vp_1} \mapsto \boxed{\texttt{int ctr= 0;...}}, \mathtt{vp_2} \mapsto \boxed{\texttt{String...}}\ \}$$

To make the handling of conflicts algebraically nicer we put more structure into the set of fragments that could be assigned to a VP. Besides normal or non-default fragments $\mathtt{f}, \mathtt{f_1}, \mathtt{f_2}, \ldots$ we have a default fragment □ and an error $\lightning$.[1] An error occurs when two or more non-default fragments are assigned to the same VP. The arrangement of these elements is the flat lattice of

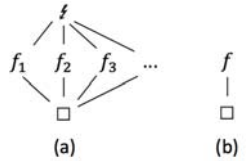

Figure 4. Lattice

[1] The $\lightning$ fragment has no VPs.

Figure 4a. This follows classical ideas in denotational semantics: $\square$ (corresponding to $\bot$) stands for absence of proper information, whereas $\lightning$ (corresponding to $\top$) stands for the error of overspecification.

> **Note**: Coloring, as currently defined in CIDE and other text coloring tools, is less general. The lattice for them allows only a default and non-default value, as shown in Figure 4b. SDA deals with a generalization that would be expected for a true modular approach to SPL development.

To prepare a convenient definition of $+$ on modules, we denote the supremum operator in this lattice again by $+$:

$$\begin{array}{ll} \square + \mathtt{x} = \mathtt{x} & \lightning + \mathtt{x} = \lightning \\ \mathtt{x} + \mathtt{x} = \mathtt{x} & \mathtt{f_i} + \mathtt{f_j} = \lightning \ \ (\mathtt{i} \neq \mathtt{j}) \,, \end{array}$$

where $\mathtt{x}$ is an arbitrary element, i.e. $\mathtt{x} \in \{\square, \mathtt{f_i}, \lightning\}$. By standard lattice theory this operation is commutative, associative and idempotent. Moreover, it has $\square$ as its neutral element.

The default fragment $\square$ is what makes our definition of modules $\mathtt{m}$ possible: every assigned VP $\mathtt{v} \in \mathtt{dom(m)}$ has at least (even in the lattice sense) the default fragment $\square$ assigned to it.

Addition of modules can now be defined as the lifting of $+$ on fragments to partial functions:

$$(\mathtt{m}+\mathtt{n})(\mathtt{v}) \ =_{df} \ \begin{cases} \mathtt{m(v)} & \text{if } \mathtt{v} \in \mathtt{dom(m)} - \mathtt{dom(n)} \\ \mathtt{n(v)} & \text{if } \mathtt{v} \in \mathtt{dom(n)} - \mathtt{dom(m)} \\ \mathtt{m(v)} + \mathtt{n(v)} & \text{if } \mathtt{v} \in \mathtt{dom(m)} \cap \mathtt{dom(n)} \\ \text{undefined} & \text{if } \mathtt{v} \notin \mathtt{dom(m)} \cup \mathtt{dom(n)} \end{cases}$$

If in the third case $\mathtt{m(v)} \neq \mathtt{n(v)}$ and $\mathtt{m(v)}, \mathtt{n(v)} \neq \square$ then $(\mathtt{m}+\mathtt{n})(\mathtt{v}) = \lightning$, thus signaling an error.[2]

By the above laws, the set of modules forms a commutative monoid under $+$. Therefore, for a finite family $\{\mathtt{m_i}\}_{\mathtt{i}\in\mathtt{I}}$ the sum $\sum_{\mathtt{i}\in\mathtt{I}} \mathtt{m_i}$ is well-defined. If $\mathtt{I} = \emptyset$ is the empty set of indices we get, as is standard, $\sum_{\mathtt{i}\in\emptyset} \mathtt{m_i} = \mathtt{0}$.

EXAMPLE 3.2. *Figure 3b shows four modules. The* `clear` *module contains a single fragment that is assigned to* $\mathtt{vp_0}$. *The* `gray` *module contains a single fragment that is assigned to* $\mathtt{vp_1}$. *The* `green` *module contains a single fragment that is assigned to* $\mathtt{vp_2}$. *And the* $\mathtt{g_{in}g}$ *(gray in green) module contains fragments that are assigned to* $\mathtt{vp_3}$, $\mathtt{vp_4}$, *and* $\mathtt{vp_5}$. *The module summation* `clear` $+$ `gray` $+$ `green` $+$ $\mathtt{g_{in}g}$ *is the module of Figure 3a.* $\square$

Assembling Fragments. We now describe how to assemble a structured document into a single fragment (while "forgetting" the structure). To define this formally we use an auxiliary function `single_fill(f,m)`. It takes a fragment `f` and a module `m` and yields the fragment that results from `f` by replacing, in parallel, all occurrences of every $\mathtt{w} \in \mathtt{VP(f)}$ by the corresponding fragment $\mathtt{m(w)}$ (if any). The precise definition of `single_fill` depends on the special type of fragments considered; as stated in the introduction we want to keep that parametric. For an acyclic module `m` and $\mathtt{v} \in \mathtt{dom(m)}$, the fragment $\mathtt{frag(v,m)}$ can be computed by iterating the `single_fill` function. By acyclicity of `m` this always terminates. To cope with the case of unassigned VPs we assume that for every VP `v` there is a trivial fragment $\mathtt{triv(v)}$ consisting only of `v`, to be used for possible later filling of `v`. With this, a corresponding assembly program looks as follows:

```
fragment frag (vp v, module m){
  fragment f = triv(v);
  while (VP(f) ∩ dom(m) != ∅)
    f = single_fill(f, m);
  return f; }
```

Once again, there are many ways to implement the above. Normally the target of `frag` is the VP of an entire file. By assigning a unique VP (such as $\mathtt{vp_0}$) for file fragments, the result of $\mathtt{frag}(\mathtt{vp_0}, \mathtt{m})$ for a (possibly composed) module `m` is the text of the entire file. A fast way to do this is to hash fragments on the VPs to which they can be assigned. From $\mathtt{vp_0}$ its fragment can be found quickly, and so too can each of its VPs and their assigned fragments, recursively. An error $\lightning$ is issued when two non-default fragments are assigned to the same VP.

3.3 Extended Example

The expression problem is a classical example of a product line [21]. Figure 5 shows three modules `base`, `print`, `eval`. Module `base` represents the shell of a program that can encode the sum and product of integers as operator trees. Module `print` enables operator trees to be printed and module `eval` enables operator trees to be evaluated.

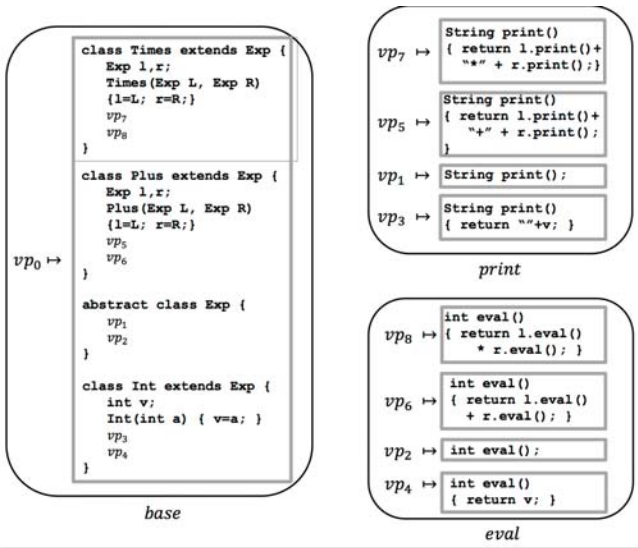

Figure 5. Three Modules.

Figure 6a shows the module sum `base` $+$ `print`, a program that can create and print sums and products of integers. Note that VPs $\mathtt{vp_2}, \mathtt{vp_4}, \mathtt{vp_6}, \mathtt{vp_8}$ have empty default fragments. VPs $\mathtt{vp_1}, \mathtt{vp_3}, \mathtt{vp_5}, \mathtt{vp_7}$ have been assigned their non-default fragments (and whose VP names are not shown). Figure 6b shows the module sum `base` $+$ `print` $+$ `eval`, a program that can create, print, and evaluate sums and products of integers.

3.4 Other Topics

Other Operations. It is possible to define both module subtraction and fragment deletion in SDA. We present the formal details of these operations in Appendix A.

Module Replication. SDA *does* permit replicated code fragments, as noted in Section 3.1. But replicated modules do not occur, as module sum is idempotent ($\mathtt{m} + \mathtt{m} = \mathtt{m}$). Again, we appeal to the same reasoning for FIA. Modules can have parameters, like templates. Different valuations of these parameters can lead to customized fragments (per different instantiation). Different instances produce distinct modules. Further, modules with customizable parameters would give SDA the power of classical preprocessors [15].

[2] This definition can be recoded in terms of total functions, which makes it easier to see that the $+$ operation indeed is commutative, associative and idempotent, hence induces a lattice, too. Moreover, it has the empty module `0` as its neutral element and satisfies $\mathtt{dom(m+n)} = \mathtt{dom(m)} \cup \mathtt{dom(n)}$.

Figure 6. Different Module Summations.

4. Combining FIA and SDA: Homomorphisms

FIA and SDA are distinct algebras: FIA deals with program specification while SDA deals with program construction. We now define the relationship between SDA and FIA (i.e. syntax and semantics).

Our vision of this relationship is displayed in Figure 7. A user selects features to specify a desired member of an SPL. The cross-product of selected features is taken to produce an FIA expression of the target program. This expression is then mapped to an SDA module expression. Evaluating the SDA module expression constructs the program.

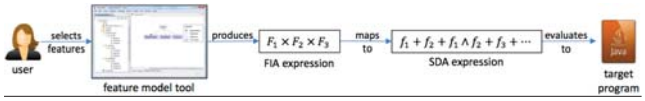

Figure 7. Feature Model Tools, FIA, and SDA.

The key to Figure 7 is the homomorphism μ : FIA $\rightarrow$ SDA that maps an FIA expression to an SDA expression. The simplest such μ uses the mechanism of colouring. In the remainder of this section we explore different versions of μ for different purposes.

4.1 The Coloring Homomorphism

Coloring is a popular way to encode SPLs (e.g. [16]). Briefly, it is the idea of painting programs with different colors: all code belonging to the BLUE feature is painted blue; all code belonging to the RED feature is painted red. Every fragment of code in a program P is painted by at least one color. Coloring also is a projection technology: if a code fragment is painted multiple colors (e.g. BLUE $\wedge$ RED), it appears *only* when all of its colors (BLUE and RED) are selected.

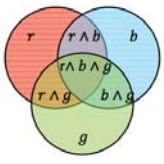

Figure 8. Venn Diagram

Consider the Venn diagram of Figure 8. The entire codebase of a program P is represented by the area within the rings for colors RED, BLUE, and GREEN. Every partition in this diagram represents the contents (code fragments) of a unique SDA module. There are seven SDA modules total: $\mathtt{r}, \mathtt{g}, \mathtt{b}, \mathtt{r} \wedge \mathtt{g}, \mathtt{g} \wedge \mathtt{b}, \mathtt{r} \wedge \mathtt{b}, \mathtt{r} \wedge \mathtt{b} \wedge \mathtt{g}$. The sum of these modules yields P:

$$\mathtt{P} \;=\; \mathtt{r} + \mathtt{g} + \mathtt{b} + \mathtt{r} \wedge \mathtt{g} + \mathtt{g} \wedge \mathtt{b} + \mathtt{r} \wedge \mathtt{b} + \mathtt{r} \wedge \mathtt{b} \wedge \mathtt{g}$$

Note that the token "$\wedge$" in the module names on the right-hand side is *not* an operator, but simply a character in a name. We return to this point in Section 4.2.

A characteristic of coloring is that each term of a feature expression (i.e. features and feature interactions) maps directly to a distinct SDA module. For Figure 8:

$$\begin{array}{rcl} \mu(\mathtt{RED}) & = & \mathtt{r} \\ \mu(\mathtt{GREEN}) & = & \mathtt{g} \\ \mu(\mathtt{BLUE}) & = & \mathtt{b} \\ \mu(\mathtt{RED\,\#\,GREEN}) & = & \mathtt{r} \wedge \mathtt{g} \\ \mu(\mathtt{RED\,\#\,BLUE}) & = & \mathtt{r} \wedge \mathtt{b} \\ \mu(\mathtt{BLUE\,\#\,GREEN}) & = & \mathtt{b} \wedge \mathtt{g} \\ \mu(\mathtt{RED\,\#\,BLUE\,\#\,GREEN}) & = & \mathtt{r} \wedge \mathtt{b} \wedge \mathtt{g} \end{array}$$

Here is the general mapping: let FS be the set of features and FIS be the set of feature interactions. *Coloring is the homomorphism that maps (feature) sums and interactions to sums of SDA modules*:

$$\mu(\mathtt{A} + \mathtt{B}) \;=\; \mu(\mathtt{A}) + \mu(\mathtt{B}) \quad // \,\mathtt{for\ all\ A,B} \in (\mathtt{FS} \cup \mathtt{FIS})$$

4.2 Interaction Homomorphism

There is one other way to relate FIA to SDA—the *interaction homomorphism*:

$$\mu(\mathtt{A\,\#\,B}) \;=\; \mu(\mathtt{A})\ \#_\mu\ \mu(\mathtt{B}) \quad // \,\mathtt{for\ all\ A,B} \in (\mathtt{FS} \cup \mathtt{FIS})$$

That is, given modules μ(A) and μ(B), one can compute (using a new SDA operation $\#_\mu$) the module μ(A # B) of their interaction.

A general algorithm for $\#_\mu$ does not exist; the task is not computable. There is not enough information within μ(A) and μ(B) to know what changes must be contained in μ(A # B) to lead to the desired program. Research on feature interactions can *detect* when μ(A # B) is non-empty (meaning that A and B interact), but such analyses cannot always compute the resolution (contents of μ(A # B)) [14]. Global information about the program is needed.

Coloring is no exception. It is impossible to compute μ(A # B) from μ(A) and μ(B). But coloring does the next best thing, the topic of the next section.

4.3 Virtual Modularity

VPs are implicit in coloring. At every point in a document where coloring changes, an implicit VP is created. Figure 9a shows an AST where the coloring of the fragment at $\mathtt{vp}_\alpha$ changes to BLUE at $\mathtt{vp}_\beta$. Figure 9b shows how this might be rendered by a colored text editor. Figure 9c shows an explicit encoding using a preprocessor.

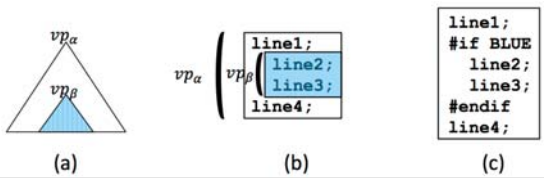

Figure 9. Coloring and VPs.

Because one colors the entire code base of a product line, it is possible to compute the contents of SDA modules and their VPs. *This is the essence of virtual modularity* [16]. Let F denote a feature and let f denote its SDA module. Let $\mathcal{F}$ denote the set of all code fragments (ASTs) that have the F color, and $\bar{\mathcal{F}}$ denote the set of all code fragments (ASTs) that do *not* have the F color.

The contents of module $\mathtt{f_i} \wedge \mathtt{f_j}$, where again $\wedge$ is *not an operation but simply a character in a composite name*, are computed by the formula:

$$\mathtt{f_i} \wedge \mathtt{f_j} \;=\; \mathcal{F}_\mathtt{i} \cap \mathcal{F}_\mathtt{j} \cap \bigcap_{\mathtt{r} \neq \mathtt{i},\mathtt{j}} \bar{\mathcal{F}}_\mathtt{r}$$

That is, the code fragments of module $\mathtt{f_i} \wedge \mathtt{f_j}$ are the intersection of the ASTs of $\mathcal{F}_\mathtt{i}$ and $\mathcal{F}_\mathtt{j}$ and the removal of all ASTs that belong to $\mathcal{F}_\mathtt{r}$ where $\mathtt{r} \notin \{\mathtt{i}, \mathtt{j}\}$. This formula generalizes in the obvious way to compute modules for individual features ($\mu(\mathtt{F}) = \mathtt{f}$) as well as modules for n-way interactions ($\mu(\mathtt{F_1 \# \ldots \# F_n}) = \mathtt{f_1} \wedge \ldots \wedge \mathtt{f_n}$).

> Note this does not contradict what we said in Section 4.1: to compute $\mathtt{f_i} \wedge \mathtt{f_j}$ one needs much more than modules $\mathtt{f_i}$ and $\mathtt{f_j}$—one needs knowledge of the coloring of the entire program P to determine the contents of module $\mathtt{f_i} \wedge \mathtt{f_j}$.

4.4 Other Homomorphisms

FIA defines the key terms (features and feature interactions) that are the semantic building blocks of SPLs. If colored modules are not used as an implementation, other homomorphisms are needed to map FIA terms to concrete representations. Here are some recent or well-known results with non-coloring or non-SDA implementations of features:

- Apel et al [2] showed how different program representations can be encoded as syntax-trees and feature composition maps to syntax-tree composition. Given the grammar of a language λ and rules for composing λ syntax-trees, FeatureHouse generates a tool that implements the homomorphism λ:

  $$\lambda(\mathtt{A} + \mathtt{B}) \;=\; \lambda(\mathtt{A}) \;+_\lambda\; \lambda(\mathtt{B})$$

  That is, a FeatureHouse-generated tool parses the λ modules for features A and B and composes them with the syntax-tree composition operation $+_\lambda$.

- Siegmund et al. [22, 23] showed how to compute a performance estimate π for a given workload for any program in an SPL. Procedures were given to estimate the performance delta that features and feature interactions contribute to a program. Assuming performance deltas of features are arithmetically added, their work relied on the homomorphism π:

  $$\pi(\mathtt{A} + \mathtt{B}) \;=\; \pi(\mathtt{A}) + \pi(\mathtt{B})$$

  Surprisingly accurate predictions were reported using this simple approach.

- The most sophisticated use to date of homomorphisms is by Delaware et al. [12], who showed how proofs of correctness of a program could be synthesized from its FIA expression. The SPL contained dialects of Featherweight Java. An integral part of any type system are the meta-theoretic proofs that show type soundness—the guarantee that the type system statically enforces the desired run-time behavior of a language, typically preservation and progress.[3] Four different representations of each feature—syntax, typing rules for preservation, evaluation rules for progress, and the proofs—were encoded as distinct modules in the Coq proof assistant [9]. Two homomorphisms were used: δ composed syntax, typing rule, and evaluation rule modules; ψ composed proof modules. Both δ and ψ were implemented as Coq libraries:

  $$\begin{aligned} \delta(\mathtt{A} + \mathtt{B}) \;&=\; \delta(\mathtt{A}) \;+_\delta\; \delta(\mathtt{B}) \\ \psi(\mathtt{A} + \mathtt{B}) \;&=\; \psi(\mathtt{A}) \;+_\psi\; \psi(\mathtt{B}) \end{aligned}$$

  Each distinct Coq module for feature syntax, feature typing rules, etc. is certified once by Coq (this is the expensive part) and reused as-is. Coq mechanically verifies the correctness of a composite proof by a simple interface check.

- When SDA modules permit fragment replacement, the actions that modules perform on programs are adding, deleting, and replacing fragments. In other words, modules become functions (program transformations). In Appendix A.2, we show that the homomorphism between FIA and SDA maps to the GenVoca model.

5. Related Work

Our work is a direct outgrowth of the Coloring Algebra [7] and differs in several important ways:

- we separate features from their implementations (i.e. the distinction of FIA and SDA),
- we use homomorphisms to map FIA expressions to (SDA) implementations,
- SDA presents a more general model of module composition via variation points, and
- we explored different and consistent sets of axioms to define feature algebras, of which [7] and our FIA are among the few reasonable possibilities.

The computation of SDA modules from coloring can be traced to [11] where elements of UML models could be tagged with feature predicates. Given a set of selected features, an element is removed from a model if its predicate is false. Modularizing elements that share the same predicate is the essence of coloring and SDA modularization.

Our work is a descendant of [17–19]. *Derivatives* were the first identified building blocks of feature modules. Unfortunately, the mathematics of derivatives was incomplete as composition of derivatives was not associative. This made it impossible to algebraically calculate the results of feature splitting (replacing T with $\mathtt{R} \times \mathtt{S}$ if T is split into features R and S) and feature merging (replacing $\mathtt{R} \times \mathtt{S}$ with T). CIDE [16] showed a simple way to visualize features and their interactions, resulting in the coloring algebra, which does support splitting and merging.

Some of the ideas of our basic SDA model can also be found in the calculus of traits presented in [13] (and in many papers prior to that). In particular, the idea of using a flat lattice is also employed there. However, the approach does not abstract from the case of classes and methods and hence is less general thachPn ours. Moreover, our definitions of the operators of module addition and subtraction lead to simpler equational laws than theirs.

Other algebras for feature-based composition, such as [3, 20], focus on the internal structure of color modules, rather than feature interactions. [3] is the first algebra (to our knowledge) that dealt with feature replication. It uses *distant idempotence* (a form of idempotence where adjacency of identical features is not required). Feature composition is not commutative and feature modules (called feature structure trees) have no inverses.

[3] *Preservation* says if expression e of type T evaluates to a value v then v also has type T. *Progress* says expression evaluation does not get "stuck", i.e. there are no expressions that cannot be evaluated.

The *Compositional Choice Calculus (CC)* [24] offers an interesting and alternative approach to our work. Our work and CC share the goal to integrate classical and virtual modularity; we do so using algebras, CC does so in the context of a formal programming language. Large-scale fragments can be placed in modules of their own, while small-scale fragments (suitable for annotations) can be embedded into other modules. Variation points and their contents are expressed as choice statements. The key difference between our work and CC is that the issues of classical and virtual modularity are not limited to a fixed set of programming languages. The ability to map an FIA expression to different representations (modular units of makefiles, HTML pages, performance models) other than traditional programming languages is basic to feature-oriented development. So too are the mappings that arise in MDE, and refactorings, and how they are connected to FOSD [6]. CC may be one of many good implementation targets for mapping FIA expressions.

Delta Oriented Programming (DOP) is another interesting language-based approach within our field of work. Delta modules are qualified to be composed into a product when the corresponding `where` clause is satisfied. Such a clause is a propositional formula over features, namely the conjunction of feature formulas that arise in coloring (and the coloring homomorphism of Section 4.1). Adding feature negation and disjunction seems more general. Disjunction allows a single module to be reused in different contexts (rather than requiring a module to be replicated for each context). Negation seems to offer a more general way for defining alternatives. Understanding this connection is a subject for future work.

Delta modules also have `after` clauses, which specify a partial ordering in which to compose them. Here is how our work implicitly encodes such ordering: When features can delete existing structures, the order in which features are composed matters (i.e. module summation no longer is commutative (Section A.2). The ordering of modules is then defined by feature products ($\mathtt{F} \times \mathtt{G} = \mathtt{F\#G} + \mathtt{F} + \mathtt{G}$): that is, interaction modules `F#G` are *always* composed *after* their base modules `F` and `G`. This recurses: 3rd-order interactions always are composed after 2nd-order interactions, and so on.

6. Conclusions and Outlook

FOSD is based on the composition and manipulation of structures. We want its tools and concepts to be based on formal models and rock-solid foundations. In this paper, we have contributed toward this goal.

FIA acts at the level of specifications to express features, feature interactions, and their compositions. We explored and explained how a prior algebra ([7]) could be generalized to admit feature replication. In contrast, SDA is a general model of modules with VPs, and how such modules can be added and subtracted. FIA deals with the 'semantics' of features and SDA deals more with the 'syntax' of modules. (Stated differently, FIA deals with the 'problem space' and SDA deals with the 'solution space'.) Projections of FIA to SDA via homomorphisms define the relationships between these two universes.

More generally, when both the problem and solution space are defined by algebras (one for program specification, the other for program construction), mappings between algebras, called homomorphisms, are a foundational concept in FOSD tooling. We have illustrated these ideas in this paper.

Acknowledgments We gratefully acknowledge support for this work by NSF grants CCF 0724979 and OCI-1148125, as well as by DFG grant MO 690/7-2. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebra for features and feature composition. In *AMAST*, 2008.

[2] S. Apel, C. Kästner, and C. Lengauer. Featurehouse: Language-independent, automated software composition. In *ICSE*, 2009.

[3] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming*, pages 1022–1047, 2010.

[4] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM TOSEM*, 1992.

[5] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. In *ACM SIGSOFT*, 1993.

[6] D. Batory, M. Azanza, and J. Saraiva. The Objects and Arrows of Computational Design. In *MODELS*, Oct. 2008.

[7] D. Batory, P. Höfner, and J. Kim. Feature Interactions, Products, and Composition. In *GPCE*, 2011.

[8] D. Batory, P. Höfner, B. Möller, and A. Zelend. Features, Modularity, and Variation Points. Technical Report TR-13-14, University of Texas at Austin, Dept. of CS, April 2013.

[9] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.

[10] CVL. Common variability language. `http://www.omgwiki.org/variability/doku.php`, 2013.

[11] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE*, 2005.

[12] B. Delaware, W. Cook, and D. Batory. Product lines of theorems. In *OOPSLA/SPLASH*, 2011.

[13] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.

[14] J. D. Hay and J. M. Atlee. Composing features and resolving interactions. In *SIGSOFT*, 2000.

[15] S. Jarzabek. *Effective Software Maintenance and Evolution: Reuse-based Approach*. CRC Press Taylor and Francis, 2007.

[16] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE*, 2008.

[17] C. H. P. Kim, C. Kästner, and D. Batory. On the modularity of feature interactions. In *GPCE*, 2008.

[18] J. Liu, D. Batory, and S. Nedunuri. Modeling interactions in feature oriented designs. In *ICFI*, 2005.

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

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

[21] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *SPLC*, 2010.

[22] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines. In *SPLC*, 2011.

[23] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. S. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *ICSE*, 2012.

[24] E. Walkingshaw and M. Erwig. A calculus for modeling and implementing variation. In *GPCE*, 2012.

A. SDA Extras

SDA has a wealth of useful capabilities beyond addition. We show some potentials and relationships to prior work.

A.1 Other Operations on Modules

It should be noted that the operations in this section are also definable for arbitrary finite maps.

Deletion and Subtraction. There are two ways of defining "inverses" to addition.

Variant I: We define the operation of *deletion* to shrink the domain of a partial map. For a module $\mathtt{m}$ and a set $\mathtt{U} \subseteq \mathtt{V}$ of VPs we define the module $\mathtt{m} \ominus \mathtt{U}$ by:

$$(\mathtt{m} \ominus \mathtt{U})(\mathtt{v}) =_{df} \begin{cases} \mathtt{m}(\mathtt{v}) & \text{if } \mathtt{v} \in (\mathtt{dom}(\mathtt{m}) - \mathtt{U}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Deletion satisfies the following laws, which are shown by straightforward calculation:

$$\begin{array}{rcl} \mathtt{dom}(\mathtt{m} \ominus \mathtt{U}) & = & \mathtt{dom}(\mathtt{m}) - \mathtt{U} \\ \emptyset \ominus \mathtt{U} & = & \emptyset \\ (\mathtt{m} + \mathtt{n}) \ominus \mathtt{U} & = & (\mathtt{m} \ominus \mathtt{U}) + (\mathtt{n} \ominus \mathtt{U}) \\ \multicolumn{3}{c}{\text{provided } \forall\, \mathtt{v} \in \mathtt{dom}(\mathtt{m}) \cap \mathtt{dom}(\mathtt{n}) : \mathtt{m}(\mathtt{v}) = \mathtt{n}(\mathtt{v})} \\ \mathtt{m} \ominus (\mathtt{U} \cup \mathtt{W}) & = & (\mathtt{m} \ominus \mathtt{U}) \ominus \mathtt{W} \\ \mathtt{m} \ominus \emptyset & = & \mathtt{m} \\ \mathtt{m} \ominus \mathtt{dom}(\mathtt{m}) & = & \emptyset \\ \mathtt{m} \ominus \mathtt{U} & \subseteq & \mathtt{m} \\ \mathtt{dom}(\mathtt{m}) \subseteq \mathtt{U} & \Leftrightarrow & (\mathtt{m} \ominus \mathtt{U}) = \emptyset \end{array}$$

A major drawback of this operation is its asymmetric type, i.e. $\ominus$ has arguments of different types.

Variant II: *Subtraction* is an operation with symmetric type. For modules $\mathtt{m}$ and $\mathtt{n}$ we define module $\mathtt{m} - \mathtt{n}$ as:

$$\mathtt{m} - \mathtt{n} =_{df} \mathtt{m} \ominus \mathtt{dom}(\mathtt{n})$$

This spells out to:

$$(\mathtt{m} - \mathtt{n})(\mathtt{v}) =_{df} \begin{cases} \mathtt{m}(\mathtt{v}) & \text{if } \mathtt{v} \in (\mathtt{dom}(\mathtt{m}) - \mathtt{dom}(\mathtt{n})) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that $\mathtt{m} - \mathtt{n}$ is *not* the set-theoretic difference of $\mathtt{m}$ and $\mathtt{n}$ considered as sets of argument-value pairs: let $\mathtt{f}_1, \mathtt{f}_2$ be different fragments and $\mathtt{u} \in \mathtt{V}$ be a VP. Set $\mathtt{m}_\mathtt{i} = \{(\mathtt{u}, \mathtt{f}_\mathtt{i})\}$, i.e.,

$$\mathtt{m}_\mathtt{i}(\mathtt{v}) =_{df} \begin{cases} \mathtt{f}_\mathtt{i} & \text{if } \mathtt{v} = \mathtt{u} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Then the set theoretic difference of $\mathtt{m}_1$ and $\mathtt{m}_2$ is $\mathtt{m}_1$. In contrast, $\mathtt{m}_1 - \mathtt{m}_2 = \mathtt{0}$ since $\mathtt{dom}(\mathtt{m}_1) = \mathtt{dom}(\mathtt{m}_2) = \{\mathtt{u}\}$.

Subtraction satisfies laws analogous to deletion; they can be found in [8]

Overriding. Ideally, modules that are composed have disjoint domains. And by using subtraction or deletion, modules can be customized. Still, object-oriented programmers are used to the notion of *overriding* or *replacing* definitions, an operation that can be defined in terms of subtraction and deletion. Module $\mathtt{m}$ overrides $\mathtt{n}$, written $\mathtt{m}\ \mathtt{onto}\ \mathtt{n}$:

$$\mathtt{m}\ \mathtt{onto}\ \mathtt{n} = \mathtt{m} + (\mathtt{n} \ominus \mathtt{dom}(\mathtt{m})) = \mathtt{m} + (\mathtt{n} - \mathtt{m})$$

This replaces all assignments in $\mathtt{n}$ for which $\mathtt{m}$ also provides a value. It may destroy acyclicity. `onto` is associative and idempotent with neutral element $\emptyset$, but not commutative.

EXAMPLE A.1. *Figure 10 shows two modules* $\mathtt{n}$ *and* $\mathtt{m}$ *with non-default fragments for* $\mathtt{vp}_1$. $\mathtt{m}\ \mathtt{onto}\ \mathtt{n}$ *replaces* $\mathtt{n}$*'s fragment at* $\mathtt{vp}_1$ *with* $\mathtt{m}$*'s fragment.* □

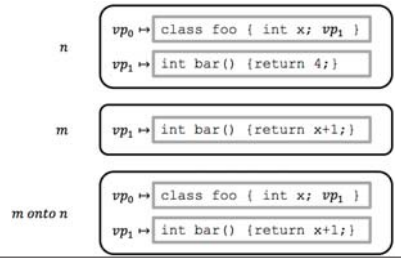

Figure 10. Onto Example.

A.2 The GenVoca Homomorphism

GenVoca is a model of SPLs where features are program transformations and feature composition is function composition. Features can add details to programs as well as override (replace) existing details. Let $m(\mathtt{x})$ denote the program transformation for feature $\mathtt{M}$ and let $\mathtt{m}$ be its SDA module. $m(\mathtt{x})$ is defined as:

$$m(\mathtt{x}) = \mathtt{m}\ \mathtt{onto}\ \mathtt{x}$$

If $\mathtt{M}$ is a base feature, m simplifies to:

$$m() = \mathtt{m}$$

GenVoca features were composed in a fixed order (see [4] for details). Further, every feature and feature interaction was encoded as a program transformation. Although FIA did not exist when GenVoca was created, an FIA explanation of GenVoca is simple: the cross-product of selected features was taken in a particular order and the resulting FIA expression was expanded in a fixed way to produce a sum of features and feature interactions.[4] Each of these terms was then mapped to a function that implemented that term. Again let FS be the set of features and FIS be the set of feature interactions. *GenVoca is the homomorphism* γ *that maps the sum of features and feature interactions to compositions of program transformations, where* $\cdot$ *is function composition:*

$$\begin{array}{lll} \gamma(\mathtt{A} + \mathtt{B}) & = \gamma(\mathtt{A}) \cdot \gamma(\mathtt{B}) & \text{// for all } \mathtt{A}, \mathtt{B} \in (\mathtt{FS} \cup \mathtt{FIS}) \\ & = a \cdot b & \end{array}$$

[4] By "fixed way" we mean that the FIA $\times$ and $+$ operators are not commutative read as associating to the right.